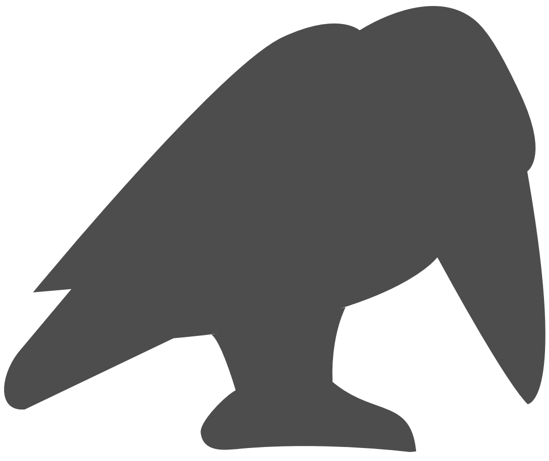
F. Drewes
A. Habel
B. Hoffmann
D. Plump (eds.)

Manipulation of Graphs, Algebras and Pictures

Essays Dedicated to Hans-Jörg Kreowski
on the Occasion of His 60th Birthday



Editors

Frank Drewes
Umeå universitet, Institutionen för datavetenskap
S-90187 Umeå, Sweden
drewes@cs.umu.se

Annegret Habel
Carl-von-Ossietzky-Universität Oldenburg, Fachbereich Informatik
D-26111 Oldenburg, Germany
Annegret.Habel@informatik.uni-oldenburg.de

Berthold Hoffmann
Universität Bremen, Fachbereich Mathematik und Informatik
D-28334 Bremen, Germany
hof@informatik.uni-bremen.de

Detlef Plump
The University of York, Department of Computer Science
York YO10 5DD, United Kingdom
det@cs.york.ac.uk

Cover Design

Caroline von Totth
Universität Bremen, Fachbereich Mathematik und Informatik
D-28334 Bremen, Germany
caro@informatik.uni-bremen.de

Illustrations by Caroline von Totth and Frank Drewes

Formatting: LaTeX style LLNCS by Springer, with modifications made by Frank Drewes

Printed and bound in August 2009 by
Hohnholt Reprografischer Betrieb GmbH
Buchtstrasse 9-10
D-28195 Bremen, Germany
book@hohnholt.com
http://www.hohnholt.com

Hans-Jörg Kreowski

Preface

This Festschrift is dedicated to Hans-Jörg Kreowski on the occasion of his 60th birthday on August 10, 2009. We invited Hans-Jörg's coauthors, the past and present members of his group at Universität Bremen, and the speakers of the colloquium on September 5, 2009 to contribute to this book. In response, we received 18 essays and seven personal greetings. Most essays were refereed three times. We thank all authors for their contribution to the Festschrift and we are grateful to the referees for their constructive comments.

Many of the essays reflect Hans-Jörg's main research interests: graph transformation, algebraic specification, and syntactic picture generation. Hans-Jörg's academic career started in 1974 when, after graduating in mathematics, he became a research associate at Technische Universität Berlin. His first papers addressed the application of category theory to automata, but soon his interests moved towards the emerging fields of *graph grammars* and *algebraic specification*. Graph grammars, in particular, have fascinated Hans-Jörg ever since, and he has made numerous important contributions to the field which nowadays is called *graph transformation*. In Berlin, he and Hartmut Ehrig developed the basic theory of the double-pushout approach, the most successful theoretical foundation for graph transformation. Lines of research associated with Hans-Jörg's name include canonical derivation sequences, the relation between Petri nets and graph grammars, and context-free graph languages generated by edge- and hyperedge replacement grammars.

Most of the work in the latter area was done at Universität Bremen, where Hans-Jörg has been a professor since 1982. More recently, Hans-Jörg and Sabine Kuske developed graph transformation units as a structuring concept for graph transformation systems and general rule-based systems. Another research area started by Hans-Jörg in Bremen are collage grammars for generating pretty pictures—unsurprisingly, given his inclination for the arts. (The reader is invited to browse this book to find that several of the pictures separating contributions have been generated by collage grammars!)

Beyond research and teaching in theoretical computer science, Hans-Jörg has been committing himself to critically analyse how computer applications affect society and to warn of their potential harm—especially in a military context. He has been an active member of the German Forum of Computer Professionals for Peace and Social Responsibility (FIfF) since its foundation in 1984, and is its chairman today. More information on this branch of Hans-Jörg's activities can be found in Ralf E. Streibl's essay in this book.

Finally, we want to mention an aspect of Hans-Jörg's artistic talent which the participants of social dinners at graph-transformation events will surely remember: his performances of *sound poetry* in the tradition of Dada artists such as Kurt Schwitters. Small appetizers of this art (necessarily only in written form) can be found in this book on two separating pages, in the greeting of Andy Schürr, and in the title of the contribution by Paolo Baldan, Andrea Corradini, Fabio Gadducci and Ugo Montanari.

This Festschrift was presented to Hans-Jörg on September 5, 2009 during a one-day colloquium celebrating his 60th birthday at Haus der Wissenschaft in Bremen. We asked Wolfgang Coy (Humboldt-Universität zu Berlin), Hartmut Ehrig (Technische Universität Berlin), Klaus-Peter Löhr (Freie Universität Berlin), Till Mossakowski (Universität Bremen), Grzegorz Rozenberg (Universiteit Leiden and University of Colorado at Boulder) and Marie-Theres Tinnefeld (Hochschule München) to give a talk at the colloquium. We are very pleased that all of them accepted.

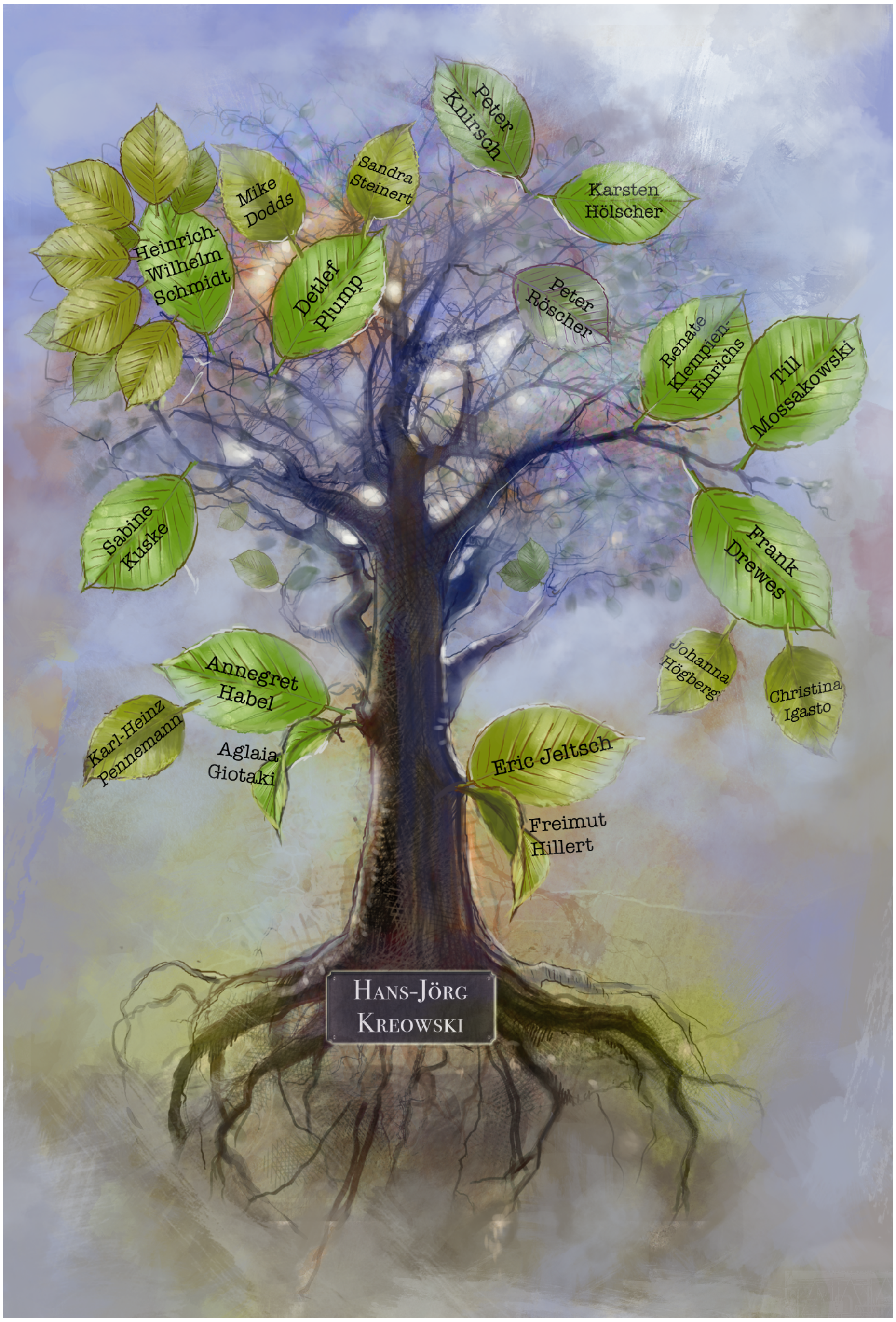
We gratefully acknowledge the financial support for the colloquium by the University of Bremen, in particular by the Department of Mathematics and Computer Science and the Collaborative Research Centre Autonomous Cooperating Logistic Processes (SFB 637), and by the German Research Center for Artificial Intelligence Bremen (DFKI). Many people have helped with the organisation of the colloquium and the production of this book. In particular, we wish to thank Karsten Hölscher, Renate Klempien-Hinrichs, Dorle Kreowski, Kai Kreowski, Sabine Kuske, Melanie Luderer, Frieder Nake, Jan Peleska, Sylvie Rauer, Ralf E. Streibl, Hauke Tönnies and Caro von Totth, who shared our enthusiasm and fun in preparing the event and this book.

Collectively we conclude by: *Congratulations, Hans-Jörg!*

September 2009

Frank Drewes
Annegret Habel
Berthold Hoffmann
Detlef Plump

Table of Contents



Hans-Jörg's scientific family tree

Publications of Hans-Jörg Kreowski
collected by Sabine Kuske

Books

[1] Hans-Jörg Kreowski. *Logische Grundlagen der Informatik – Handbuch der Informatik 1.1.* Oldenbourg-Verlag, München, 1991.

[2] Hans-Jörg Kreowski and Heinz-Wilhelm Schmidt. *Some Algebraic Concepts of the Specification Language SEGRAS and Their Initial Semantics*, volume 93 of *GMD-Studien.* Gesellschaft für Mathematik und Datenverabeitung MBH, 1984.

[3] Hartmut Ehrig, Klaus-Dieter Kiermeier, Hans-Jörg Kreowski, and Wolfgang Kühnel. *Universal Theory of Automata: A Categorical Approach.* Teubner, Stuttgart, 1974.

Edited Books

[4] Hans-Jörg Kreowski, editor. *Informatik und Gesellschaft – Verflechtungen und Perspektiven.* LIT Verlag, Berlin, 2008.

[5] Hans-Dietrich Haasis, Hans-Jörg Kreowski, and Bernd Scholz-Reiter, editors. *Proc. 1st International Conference on Dynamics in Logistics (LDIC).* Springer, 2008.

[6] Hans-Jörg Kreowski, Ugo Montanari, Fernado Orejas, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Formal Methods in Software and System Modeling*, volume 3393 of *Lecture Notes in Computer Science.* Springer, 2005.

[7] Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. 1st International Conference on Graph Transformation (ICGT 2002)*, volume 2505 of *Lecture Notes in Computer Science.* Springer, 2002.

[8] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. 6th International Workshop on Theory and Application of Graph Transformations (TAGT'98)*, volume 1764 of *Lecture Notes in Computer Science.* Springer, 2000.

[9] Egidio Astesiano, Hans-Jörg Kreowski, and Bernd Krieg-Brückner, editors. *Algebraic Foundations of Systems Specification.* Springer, 1999.

[10] Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, 1999.

[11] Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 3: Concurrency, Parallelism, and Distribution.* World Scientific, 1999.

[12] Hans-Jörg Kreowski, Thomas Risse, Andreas Spillner, Ralf Streibl, and Karin Vosseberg, editors. *Realität und Utopien der Informatik.* Agenda Verlag, 1995.

[13] Hans-Jörg Kreowski, editor. *Informatik zwischen Wissenschaft und Gesellschaft, Zur Erinnerung an Reinhold Franck*, volume 309 of *Informatik-Fachberichte.* Springer, 1992.

[14] Michel Bidoit, Hans-Jörg Kreowski, Pierre Lescanne, Fernando Orejas, and Donald Sannella, editors. *Algebraic System Specification and Development — A Survey and Annotated Bibliography*, volume 501 of *Lecture Notes in Computer Science.* Springer, 1991.

[15] Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science.* Springer, 1991.

[16] Hartmut Ehrig, Horst Herrlich, Hans-Jörg Kreowski, and Gerhard Preuß, editors. *Categorical Methods in Computer Science*, volume 393 of *Lecture Notes in Computer Science.* Springer, 1989.

[17] Hans-Jörg Kreowski, editor. *Recent Trends in Data Type Specification – Selected Papers of the 3rd Workshop on Theory and Applications on Abstract Data Types*, volume 116 of *Informatik-Fachberichte.* Springer, 1985.

Journal Articles

[18] Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units to model interacting sequential and parallel processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.

[19] Mehrdad Babazadeh, Hans-Jörg Kreowski, and Walter Lang. Selective predictors of environmental parameters in wireless sensor networks. *International Journal of Mathematical Models and Methods in Applied Sciences*, 2:355–363, 2008.

[20] Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Paul Ziemann. Towards an integrated graph-based semantics for uml. *Software and Systems Modeling*, 2008.

[21] Giorgio Busatto, Hans-Jörg Kreowski, and Sabine Kuske. Abstract hierarchical graph transformation. *Mathematical Structures in Computer Science*, 15:773–819, 2005.

[22] Frank Drewes, Sigrid Ewert, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Computing raster images from grid picture grammars. *Journal of Automata, Languages and Combinatorics*, 8(3):499–519, 2003.

[23] Frank Drewes, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Table-driven and context-sensitive collage languages. *Journal of Automata, Languages and Combinatorics*, 8(1):5–24, 2003.

[24] Frank Drewes, Hans-Jörg Kreowski, and Denis Lapoire. Criteria to disprove context-freeness of collage languages. *Theoretical Computer Science*, 290(3):1445–1458, 2003.

[25] Marc Andries, Gregor Engels, Annegret Habel, Berthold Hoffmann, Hans-Jörg Kreowski, Sabine Kuske, Detlef Plump, Andy Schürr, and Gabriele Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, 1999.

[26] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[27] Hartmut Ehrig, Hans-Jörg Kreowski, and Fernando Orejas. Correctness of horizontal and vertical composition for implementation concepts based on constructors and abstractors. *Revista Matematica*, 10:365–387, 1997.

[28] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.

[29] Frank Drewes and Hans-Jörg Kreowski. (Un-)decidability of geometric properties of pictures generated by collage grammars. *Fundamenta Informaticae*, 25(3):295–325, 1996.

[30] Frank Drewes, Hans-Jörg Kreowski, and Nils Schwabe. COLLAGE-ONE: A system for evaluation and visualisation of collage grammars. *Machine Graphics & Vision*, 5(1/2):393–402, 1996.

[31] Frank Drewes, Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Generating self-affine fractals by collage grammars. *Theoretical Computer Science*, 145(1&2):159–187, 1995.

[32] Hans-Jörg Kreowski and Till Mossakowski. Equivalence and difference of institutions: Simulating Horn clause logic with based algebras. *Mathematical Structures in Computer Science*, 5(2):189–215, 1995.

[33] Annegret Habel, Hans-Jörg Kreowski, and Clemens Lautemann. A comparison of compatible, finite and inductive graph properties. *Theoretical Computer Science*, 110(1):145–168, 1993.

[34] Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Collages and patterns generated by hyperedge replacement. *Languages of Design*, 1:125–145, 1993.

[35] Hartmut Ehrig, Annegret Habel, and Hans-Jörg Kreowski. Introduction to graph grammars with application to semantic networks. *Computers and Mathematics with Applications*, 23(6-9):557–572, 1992.

[36] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. Parallelism and concurrency in high-level replacement systems. *Journal of Mathematical Structures in Computer Science*, 1(3):361–404, 1991.

[37] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. *Fundamenta Informaticae*, 15(1):37–60, 1991.

[38] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. Decidable boundedness problems for sets of graphs generated by hyperedge-replacement. *Theoretical Computer Science*, 89(1):33–62, 1991.

[39] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars – Part I. *Information Sciences*, 52(2):185–210, 1990.

[40] Hans-Jörg Kreowski and Grzegorz Rozenberg. On structured graph grammars – Part II. *Information Sciences*, 52(3):221–246, 1990.

[41] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. Metatheorems for decision problems on hyperedge replacement graph languages. *Acta Informatica*, 26(7):657–677, 1989.

[42] Annegret Habel and Hans-Jörg Kreowski. Characteristics of graph languages generated by edge replacement. *Theoretical Computer Science*, 51(1/2):81–115, 1987.

[43] Hans-Jörg Kreowski and Anne Wilharm. Net processes correspond to derivation processes in graph grammars. *Theoretical Computer Science*, 44:275–305, 1986.

[44] Hartmut Ehrig, Hans-Jörg Kreowski, James W. Thatcher, Eric G. Wagner, and Jesse B. Wright. Parameter passing in algebraic specification languages. *Theoretical Compututer Science*, 28:45–81, 1984.

[45] Hans-Jörg Kreowski and Grzegorz Rozenberg. Note on node-rewriting graph grammars. *Information Processing Letters*, 18(1):21–24, 1984.

[46] Hartmut Ehrig and Hans-Jörg Kreowski. Compatibility of parameter passing and implementation of parameterized data types. *Theoretical Computer Science*, 27:255–286, 1983.

[47] Hartmut Ehrig, Hans-Jörg Kreowski, Bernd Mahr, and Peter Padawitz. Algebraic implementation of abstract data types. *Theoretical Computer Science*, 20(3):209–263, 1982.

[48] Hartmut Ehrig, Hans-Jörg Kreowski, Andrea Maggiolo-Schettini, Barry K. Rosen, and Jozef Winkowski. Transformations of structures: An algebraic approach. *Mathematical Systems Theory*, 14:305–334, 1981.

[49] Hartmut Ehrig and Hans-Jörg Kreowski. Applications of graph grammar theory to consistency, synchronization and scheduling in data base systems. *Information Systems*, 5(3):225–238, 1980.

[50] Hartmut Ehrig and Hans-Jörg Kreowski. The skeleton of minimal realization. *Studien zur Algebra und ihre Anwendungen*, 7:137–154, 1979.

[51] Hartmut Ehrig and Hans-Jörg Kreowski. Pushout-properties: An analysis of gluing constructions for graphs. *Mathematische Nachrichten*, 91:135–149, 1979.

[52] Hartmut Ehrig and Hans-Jörg Kreowski. Systematic approach of reduction and minimization in automata and systems theory. *Computer Systems Science*, 12(3):269–304, 1976.

[53] Hartmut Ehrig, Hans-Jörg Kreowski, and Michael Pfender. Kategorielle Theorie der Reduktion, Minimierung und Äquivalenz von Automaten. *Mathematische Nachrichten*, 59:105–124, 1974.

Contributions to Proceedings and Books

[54] Hans-Jörg Kreowski and Sabine Kuske. Graph multiset transformation as a framework for massively parallel computation. In H. Ehrig, R. Heckel, G. Rozenberg, and G. Taentzer, editors, *Proc. 4th International Conference on Graph Transformations (ICGT 2008)*, volume 5214 of *Lecture Notes in Computer Science*, pages 351–365, 2008.

[55] Hans-Jörg Kreowski and Sabine Kuske. Communities of autonomous units for pickup and delivery vehicle routing. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd International Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 281–296, 2008.

[56] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – an overview. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models. Essays Dedicated to Ugo Montanari on the Occasion of His 65th Birthday*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. 2008.

[57] Karsten Hölscher, Renate Klempien-Hinrichs, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units: Basic concepts and semantic foundation. In M. Hülsmann and K. Windt, editors, *Understanding Autonomous Cooperation and Control in Logistics. The Impact on Management, Information and Communication and Material Flow*, pages 103–120, 2007.

[58] Hans-Jörg Kreowski and Sabine Kuske. Autonomous units and their semantics – the parallel case. In J.L. Fiadeiro and P.Y. Schobbens, editors, *Proc. 18th International Workshop on Recent Trends in Algebraic Development Techniques (WADT 2006)*, volume 4409 of *Lecture Notes in Computer Science*, pages 56–73, 2007.

[59] Ingo Timm, Hans-Jörg Kreowski, Peter Knirsch, and Andreas Timm-Giel. Autonomy in software systems. In M. Hülsmann and K. Windt, editors, *Understanding Autonomous Cooperation and Control in Logistics. The Impact on Management, Information and Communication and Material Flow*, pages 255–274, 2007.

[60] Adrian Horia Dediu, Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Benedek Nagy. Contextual hypergraph grammars – a new approach to the generation of hypergraph languages. In O. H. Ibarra and Z. Dang, editors, *Proc. 10th International Conference on Developments in Language Theory*, volume 4036 of *Lecture Notes in Computer Science*, pages 327–338, 2006.

[61] Karsten Hölscher, Renate Klempien-Hinrichs, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Autonome Transformationseinheiten zur regelbasierten Modellierung vernetzter logistischer Prozesse. In E. Müller, editor, *Vernetzt planen und produzieren (VPP 2006)*, pages 113–118, 2006. Wissenschaftliche Schriftenreihe des Institutes für Betriebswissenschaften und Fabriksysteme, Technische Universität Chemnitz.

[62] Karsten Hölscher, Peter Knirsch, and Hans-Jörg Kreowski. Modelling transport networks by means of autonomous units. In H.-D. Haasis, H. Kopfer, and J. Schönberger, editors, *Proc. Operations Research 2005*, pages 399–404, 2006.

[63] Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units and their semantics — the sequential case. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. 3rd International Conference on Graph Transformations (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 245–259, 2006.

[64] Hans-Jörg Kreowski, Karsten Hölscher, and Peter Knirsch. Semantics of visual models in a rule-based setting. In R. Heckel, editor, *Proc. School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 75–88, 2006.

[65] Hans-Jörg Kreowski, Renate Klempien-Hinrichs, and Sabine Kuske. Some essentials of graph transformation. In Z. Ésik, C. Martin-Vide, and V. Mitrana, editors, *Recent Advances in Formal Languages and Applications*, volume 25 of *Studies in Computational Intelligence*, pages 229–254. 2006.

[66] Dirk Janssens, Hans-Jörg Kreowski, and Grzegorz Rozenberg. Main concepts of networks of transformation units with interlinking semantics. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and System Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 325–342. 2005.

[67] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Rule-based transformation of graphs and the product type. In P. van Bommel, editor, *Transformation of Knowledge, Information, and Data: Theory and Applications*, pages 29–51. 2005.

[68] Hans-Jörg Kreowski. Autonomous units to model cooperating logistic processes: Basic features. In K.S. Palwar, editor, *Proc. 10th International Symposium on Logistics (ISL 2005)*, pages 377–380, 2005.

[69] Julia Padberg and Hans-Jörg Kreowski. Loose semantics of Petri nets. In H.-J. Kreowski, U. Montanari, F. Orejas, G. Rozenberg, and G. Taentzer, editors, *Formal Methods in Software and System Modeling*, volume 3393 of *Lecture Notes in Computer Science*, pages 370–384. 2005.

[70] Björn Cordes, Karsten Hölscher, and Hans-Jörg Kreowski. UML interaction diagrams: Correct translation of sequence diagrams into collaboration diagrams. In M. Nagl, J. Pfaltz, and B. Böhlen, editors, *Proc. 2nd International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*, volume 3062 of *Lecture Notes in Computer Science*, pages 275–291, 2004.

[71] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Sabine Kuske. Typing of graph transformation units. In H. Ehrig, G. Engels, F. Parisi-Presicce, and G. Rozenberg, editors, *Proc. 2nd International Conference on Graph Transformations (ICGT 2004)*, volume 3256 of *Lecture Notes in Computer Science*, pages 112–127, 2004.

[72] Hans-Jörg Kreowski. Syntax, Semantik und ... In Karl-Heinz Rödiger, editor, *Algorithmik – Kunst – Semiotik – Hommage für Frieder Nake*, pages 75–88. 2003.

[73] Hans-Jörg Kreowski and Sabine Kuske. Approach-independent structuring concepts for rule-based systems. In M. Wirsing, D. Pattison, and R. Hennicker, editors, *Proc. 16th International Workshop on Recent Trends in Algebraic Development Techniques (WADT 2002)*, volume 2755 of *Lecture Notes in Computer Science*, pages 299–311, 2003.

[74] Hans-Jörg Kreowski. A sight-seeing tour of the computational landscape of graph transformation. In W. Brauer, H. Ehrig, J. Karhumäki, and A. Salomaa, editors, *Formal and Natural Computing. Essays Dedicated to Grzegorz Rozenberg*, volume 2300 of *Lecture Notes in Computer Science*, pages 119–137. 2002.

[75] Hans-Jörg Kreowski, Giorgio Busatto, Renate Klempien-Hinrichs, Peter Knirsch, and Sabine Kuske. Structured modeling with GRACE. In M. Bauderon and A. Corradini, editors, *Proc. GETGRATS Closing Workshop*, volume 51 of *Electronic Notes in Theoretical Computer Science*, pages 233–245, 2002.

[76] Sabine Kuske, Martin Gogolla, Hans-Jörg Kreowski, and Ralf Kollmann. An integrated semantics for UML class, object and state diagrams based on graph transformation. In M. Butler, L. Petre, and K. Sere, editors, *Proc. 3rd International Conference on Integrated Formal Methods (IFM 2002)*, volume 2335 of *Lecture Notes in Computer Science*, pages 11–28, 2002.

[77] Andrea Corradini and Hans-Jörg Kreowski. GETGRATS and APPLIGRAPH: Theory and applications of graph transformation. In G. Păun, G. Rozenberg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science*, pages 164–170. 2001.

[78] Frank Drewes, Sigrid Ewert, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Computing raster images from grid picture grammars. In S. Yu and A. Păun, editors, *Proc. 5th International Conference on Implementation and Application of Automata (CIAA 2000)*, volume 2088 of *Lecture Notes in Computer Science*, pages 113–121, 2001.

[79] Frank Drewes and Hans-Jörg Kreowski. Reading words in graphs generated by hyperedge replacement. In C. Martin-Vide and V. Mitrana, editors, *Where Mathematics, Computer Science, Linguistics and Biology Meet*, chapter 22, pages 243–252. 2001.

[80] Hans-Jörg Kreowski, Giorgio Busatto, and Sabine Kuske. GRACE as a unifying approach to graph-transformation-based specification. In H. Ehrig, C. Ermel, and J. Padberg, editors, *Proc. Uniform Approaches to Graphical Process Specification Techniques*, volume 44/4 of *Electronic Notes in Theoretical Computer Science*, 2001. 15 pages.

[81] Frank Drewes, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Table-driven and context-sensitive collage languages. In G. Rozenberg and W. Thomas, editors, *Proc. Developments in Language Theory (DLT'99)*, pages 326–337, 2000.

[82] Frank Drewes, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Graph transformation modules and their composition. In M. Nagl, A. Schürr, and M. Münch, editors, *Proc. International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *Lecture Notes in Computer Science*, pages 15–30, 2000.

[83] Peter Knirsch and Hans-Jörg Kreowski. A note on modeling agent systems by graph transformation. In M. Nagl, A. Schürr, and M. Münch, editors, *Proc. International Workshop on Applications of Graph Transformations with Industrial Relevance (AGTIVE'99)*, volume 1779 of *Lecture Notes in Computer Science*, pages 79–86, 2000.

[84] Hans-Jörg Kreowski and Sabine Kuske. Note on approach-independent structuring concepts for rule-based systems. In H. Ehrig and G. Taentzer, editors, *Proc. Joint Appligraph and GETGRATS Workshop on Graph Transformation Systems*, Technical Report Nr. 2000-2, pages 41–49, Technische Universität Berlin, 2000.

[85] Hans-Jörg Kreowski and Sabine Kuske. Suggestions on the modularization of rule-based systems. In M. Wirsing, M. Gogolla, H.-J. Kreowski, T. Nipkow, and W. Reif, editors, *Proc. Rigorose Entwicklung software-intensiver Systeme*, Technical Report 0005, pages 73–82, Universität München, 2000.

[86] Hans-Jörg Kreowski and Gabriel Valiente. Redundancy and subsumption in high-level replacement systems. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 6th International Workshop on Theory and Application of Graph Transformations*, volume 1764 of *Lecture Notes in Computer Science*, pages 215–227, 2000.

[87] Frank Drewes and Hans-Jörg Kreowski. Picture generation by collage grammars. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 11, pages 397–457. 1999.

[88] Hartmut Ehrig and Hans-Jörg Kreowski. Refinement and implementation. In E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner, editors, *Algebraic Foundations of Systems Specification*, pages 201–242. 1999.

[89] Renate Klempien-Hinrichs, Hans-Jörg Kreowski, and Stefan Taubenberger. Correct translation of mutually recursive function systems into TOL collage grammars. In G. Ciobanu and Gh. Păun, editors, *Proc. 12th International Symposium on Fundamentals of Computation Theory (FCT'99)*, volume 1684 of *Lecture Notes in Computer Science*, pages 350–361, 1999.

[90] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, pages 607–638. 1999.

[91] Hans-Jörg Kreowski. Künstliche Intelligenz und Gehirn. In H.J. Sandkühler, editor, *Repräsentation, Denken und Selbstbewusstsein*, vol-

ume 20 of *Schriftenreihe des Zentrums Philosophische Grundlagen der Wissenschaften*, pages 193–204, Universität Bremen, 1998.

[92] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. 1997.

[93] Frank Drewes, Hans-Jörg Kreowski, and Denis Lapoire. Criteria to disprove context-freeness of collage languages. In B.S. Chlebus and L. Czaja, editors, *Proc. 11th International Symposium on Fundamentals of Computation Theory*, volume 1279 of *Lecture Notes in Computer Science*, pages 169–178, 1997.

[94] Hans-Jörg Kreowski, Veronika Oechtering, and Ingrid Rügge. Frauen auf dem Weg, das Image der Informatik zu verändern. In J. Jarke, K. Pasedach, and K. Pohl, editors, *Informatik '97 - Informatik als Innovationsmotor*, pages 345–354, 1997.

[95] Hans-Jörg Kreowski and Sabine Kuske. On the interleaving semantics of transformation units — a step into GRACE. In J. E. Cuny, H. Ehrig, G. Engels, and Grzegorz Rozenberg, editors, *Proc. 5th International Workhop on Graph Grammars and Their Application to Computer Science*, volume 1073 of *Lecture Notes in Computer Science*, pages 89–108, 1996.

[96] Hans-Jörg Kreowski. Specification and programming (by graph transformation). In A. Corradini and U. Montanari, editors, *Proc. Joint Workshop on Graph Rewriting and Computation COMPUGRAPH/SEMAGRAPH*, volume 2 of *Electronic Notes in Computer Science*, pages 187–190, 1995.

[97] Hans-Jörg Kreowski. Graph grammars for software specification and programming: An eulogy in praise of GRACE. In F. Rosselló Llompart and G. Valiente, editors, *Colloquium on Graph Transformation and its Application in Computer Science*, Technical Report, pages 55–61, Palma de Mallorca, 1995.

[98] Frank Drewes, Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Generating self-affine fractals by collage grammars. In G. Rozenberg and A. Salomaa, editors, *Proc. Developments in Language Theory 93. At the Crossroads of Mathematics, Computer Science and Biology*, pages 278–289, 1994.

[99] Hartmut Ehrig, Hans-Jörg Kreowski, and Gabriele Taentzer. Canonical derivations for high-level replacement systems. In H. J. Schneider and H. Ehrig, editors, *Dagstuhl Seminar on Graph Transformations in Computer Science*, volume 776 of *Lecture Notes in Computer Science*, pages 153–169, 1994.

[100] Hans-Jörg Kreowski. An axiomatic approach to canonical derivations. In *Proc. IFIP World Computer Congress*, volume A-51 of *IFIP-Transactions*, pages 348–353, 1994.

[101] Gnanamalar David, Frank Drewes, and Hans-Jörg Kreowski. Hyperedge replacement with rendezvous. In J.P. Jouannaud, editor, *Proc. Collo-*

quium on Trees in Algebra and Programming (CAAP'93), volume 668 of *Lecture Notes in Computer Science*, pages 167–181, 1993.

[102] Hans-Jörg Kreowski. Five facets of hyperedge replacement beyond context-freeness. In Z. Ésik, editor, *Proc. 9th International Conference on Fundamentals of Computation Theory*, volume 710 of *Lecture Notes in Computer Science*, pages 69–86, 1993.

[103] Hans-Jörg Kreowski. Translations into the graph grammar machine. In R. M. Sleep, R. Plasmeijer, and M. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 13, pages 171–183. 1993.

[104] Hans-Jörg Kreowski. Some initial sections of the algebraic specification tale. In G. Rozenberg and A. Salomaa, editors, *Current Trends in Theoretical Computer Science – Essays and Tutorials*, pages 54–75. 1993.

[105] Hans-Jörg Kreowski. Eine konkrete Utopie von korrekter Software. In *Informatik zwischen Wissenschaft und Gesellschaft, Zur Erinnerung an Reinhold Franck*, volume 309 of *Informatik-Fachberichte*, pages 108–124, 1992.

[106] Hans-Jörg Kreowski. Ein Vorschlag zum Testen strukturierter algebraischer Spezifikationen. In P. Liggesmeyer, H. M. Sneed, and A. Spillner, editors, *Testen, Analysieren und Verifizieren von Software*, Informatik Aktuell, pages 130–142, 1992.

[107] Frank Drewes and H.-J. Kreowski. A note on hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 1–11, 1991.

[108] Hartmut Ehrig, Annegret Habel, Hans-Jörg Kreowski, and Francesco Parisi-Presicce. From graph grammars to high-level replacement systems. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 532, pages 269–291, 1991.

[109] Annegret Habel and Hans-Jörg Kreowski. Collage grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 411–429, 1991.

[110] Eric Jeltsch and Hans-Jörg Kreowski. Grammatical inference based on hyperedge replacement. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th International Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 461–474, 1991.

[111] Annegret Habel and Hans-Jörg Kreowski. Filtering hyperedge-replacement languages through compatible properties. In M. Nagl, editor, *Proc. 15th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'89)*, volume 411 of *Lecture Notes in Computer Science*, pages 107–120, 1990.

[112] Hans-Jörg Kreowski and Zhenyu Quian. Relation-sorted specifications with built-in coercers: Basic notions and results. In C. Choffrut and T. Lengauer, editors, *Proc. 7th Annual Symposium on Theoretical Aspects of Computer Science (STACS'90)*, volume 415 of *Lecture Notes in Computer Science*, pages 165–175, 1990.

[113] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. Decidable boundedness problems for hyperedge replacement graph grammars. In J. Díaz and F. Orejas, editors, *Proc. Joint Conference on Theory and Practice of Software Development (TAPSOFT'89), Vol. 1*, volume 351 of *Lecture Notes in Computer Science*, pages 275–289, 1989.

[114] Hans-Jörg Kreowski. Colimits as parameterized data types. In H. Ehrig, H. Herrlich, H.-J. Kreowski, and G. Preuß, editors, *Proc. Workshop on Categorial Methods in Computer Science – with Aspects from Topology*, volume 393 of *Lecture Notes in Computer Science*, pages 36–49, 1989.

[115] Hans-Jörg Kreowski. Informationstechnische Grundbildung für alle ist Unfug. In Felix Rauner and Julie K. Ruth, editors, *Informationstechnische Grundbildung zwischen Affirmation und Gestaltungskompetenz*, pages 27–38. 1989.

[116] Annegret Habel and Hans-Jörg Kreowski. Pretty patterns produced by hyperedge replacement. In H. Göttler and H.J. Schneider, editors, *Proc. International Workshop on Graph-Theoretic Concepts in Computer Science (WG'87)*, volume 314 of *Lecture Notes in Computer Science*, pages 32–45, 1988.

[117] Annegret Habel, Hans-Jörg Kreowski, and Detlef Plump. Jungle evaluation. In D. Sannella and A. Tarlecki, editors, *Recent Trends in Data Type Specification, Proc. 5th Workshop on Specification of Abstract Data Types*, volume 332 of *Lecture Notes in Computer Science*, pages 92–112, 1988.

[118] Hans-Jörg Kreowski. Complexity in algebraic specifications: An upper bound result. In H. Ehrig, editor, *Proc. 6th Workshop on Abstract Data Types*, Technical Report, Technische Universität Berlin, 1988.

[119] Annegret Habel and Hans-Jörg Kreowski. May we introduce to you: Hyperedge replacement. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proc. 3rd International Workshop on Graph Grammars and Their Application to Computer Science (GRAGRA'86)*, volume 291 of *Lecture Notes in Computer Science*, pages 15–26, 1987.

[120] Annegret Habel and Hans-Jörg Kreowski. Some structural aspects of hypergraph languages generated by hyperedge replacement. In F.J. Brandenburg, G. Vidal-Naquet, and M. Wirsing, editors, *Proc. 4th Annual Symposium on Theoretical Aspects of Computer Science (STACS'87)*, volume 247 of *Lecture Notes in Computer Science*, pages 207–219, 1987.

[121] Hans-Jörg Kreowski. Partial algebras flow from algebraic specification. In T. Ottmann, editor, *Proc. 4th International Colloquium on Automata, Languages and Programming (ICALP'87)*, volume 267 of *Lecture Notes in Computer Science*, pages 521–530, 1987.

[122] Hans-Jörg Kreowski. Is parallelism already concurrency? Part 1: Derivations in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proc. 3rd International Workshop on Graph Grammars and Their Application to Computer Science (GRAGRA'86)*, volume 291 of *Lecture Notes in Computer Science*, pages 343–360, 1987.

[123] Hans-Jörg Kreowski. Informatik und Militär: Zusammen in den Abgrund. In M. Löwe, M. Schmidt, and R. Wilhelm, editors, *Umdenken in der Informatik*, pages 37–42. 1987.

[124] Hans-Jörg Kreowski and Anne Wilharm. Is parallelism already concurrency? Part 2: Non-sequential processes in graph grammars. In H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld, editors, *Proc. 3rd International Workshop on Graph Grammars and Their Application to Computer Science GRAGRA'86*, volume 291 of *Lecture Notes in Computer Science*, pages 361–377, 1987.

[125] Hans-Jörg Kreowski. Rule trees represent derivations in edge replacement systems. In G. Rozenberg and A. Salomaa, editors, *The book of L*, pages 217–232. 1986.

[126] Hans-Jörg Kreowski. Based algebras. In K. Drosten, H.-D. Ehrich, M. Gogolla, and U. W. Lipeck, editors, *Proc. 4th Workshop on Abstract Data Types*, Informatik-Bericht Nr. 86-09, Universität Braunschweig, 1986.

[127] Hans-Jörg Kreowski and Anne Wilharm. Solving conflicts in graph grammars derivation processes. In H. Noltemeier, editor, *Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG'85)*, pages 161–179, 1985.

[128] Hans-Jörg Kreowski and Anne Wilharm. Processes on Petri nets and graph grammars: A summary. In U. Pape, editor, *Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG'84)*, pages 189–200, 1984.

[129] Annegret Habel and Hans-Jörg Kreowski. On context-free graph languages generated by edge replacement. In H. Ehrig, M. Nagl, and G. Rozenberg, editors, *Proc. 2nd International Workshop on Graph Grammars and Their Application to Computer Science*, volume 153 of *Lecture Notes in Computer Science*, pages 143–158, 1983.

[130] Hans-Jörg Kreowski. Graph grammar derivation processes. In M. Nagl and J. Perl, editors, *Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG'83)*, pages 136–150, 1983.

[131] Hans-Jörg Kreowski. Specification of partial functions – only a tentative suggestion. In M. Broy and M. Wirsing, editors, *Proc. 2nd Workshop on Abstract Data Types*, Technical Report, Universität Passau, 1983.

[132] Klaus-Peter Hasler, Hans-Jörg Kreowski, Michael Löwe, and Michaela Reisin. Suggestions on the interpretation of algebraic specifications. In M. Broy and M. Wirsing, editors, *Proc. 2nd Workshop on Abstract Data Types*, Technical Report, Universität Passau, 1983.

[133] Hartmut Ehrig and Hans-Jörg Kreowski. Keywords in context: An algebraic specification. In J. Staunstrup, editor, *Proc. Workshop on Program*

Specification, volume 134 of *Lecture Notes in Computer Science*, pages 73–83, 1982.

[134] Hartmut Ehrig and Hans-Jörg Kreowski. Parameter passing commutes with implementation of parameterized data types. In M. Nielsen and E.M. Schmidt, editors, *Proc. International Colloquium on Automata, Languages and Programming (ICALP'82)*, volume 140 of *Lecture Notes in Computer Science*, pages 197–211, 1982.

[135] Hartmut Ehrig and Hans-Jörg Kreowski. Example 2: Kwic-index generation. In Jørgen Staunstrup, editor, *Program Specification*, volume 134 of *Lecture Notes in Computer Science*, pages 78–83, 1982.

[136] Hartmut Ehrig, Hans-Jörg Kreowski, James Thatcher, Eric Wagner, and Jesse Wright. Parameter passing in algebraic specification languages. In J. Staunstrup, editor, *Proc. Workshop on Program Specification*, volume 134 of *Lecture Notes in Computer Science*, pages 322–369, 1982.

[137] Hans-Jörg Kreowski. An algebraic implementation concept for abstract data types. In H.-D. Ehrich and U. W. Lipeck, editors, *Proc. 1st Workshop on Abstract Data Types*, Technical Report, Universität Dortmund, 1982.

[138] Hartmut Ehrig, Werner Fey, and Hans-Jörg Kreowski. Algebraische Spezifikation eines Stücklistensystems – eine Fallstudie. In C. Floyd and H. Kopetz, editors, *Proc. GACM-Konferenz Software Engineering – Entwurf und Spezifikation*, volume 5 of *Berichte des German Chapter of the ACM*, pages 75–90, 1981.

[139] Hans-Jörg Kreowski. A comparison between Petri-nets and graph grammars. In H. Noltemeier, editor, *Proc. International Workshop on Graph-theoretic Concepts in Computer Science (WG'80)*, volume 100 of *Lecture Notes in Computer Science*, pages 306–317, 1981.

[140] Hans-Jörg Kreowski. Algebraische Spezifikation von Softwaresystemen. In C. Floyd and H. Kopetz, editors, *Proc. GACM-Konferenz Software Engineering – Entwurf und Spezifikation*, volume 5 of *Berichte des German Chapter of the ACM*, pages 46–74, 1981.

[141] Hans-Jörg Kreowski. Wo liegen die Grenzen der praktischen Anwendung formaler Methoden für die Entwurfsspezifikation? In C. Floyd and H. Kopetz, editors, *Software Engineering*, volume 5 of *Berichte des German Chapter of the ACM*, pages 281–283, 1981.

[142] Hans-Jörg Kreowski and Grzegorz Rozenberg. On the constructive description of graph languages accepted by finite automata. In J. Gruska and M. Chytil, editors, *Proc. 10th Symposium on Mathematical Foundations of Computer Science (MFCS'81)*, volume 118 of *Lecture Notes in Computer Science*, pages 398–409, 1981.

[143] Hartmut Ehrig and Hans-Jörg Kreowski. A graph grammar approach to optimal and consistent schedules in data base systems. In U. Pape, editor, *Discrete Structures and Algorithms, Proc. Workshop Graphtheoretic Concepts in Computer Science (WG'79)*, pages 223–240, 1980.

[144] Hartmut Ehrig, Hans-Jörg Kreowski, Bernd Mahr, and Peter Padawitz. Compound algebraic implementations: An approach to stepwise refine-

ment of software systems. In P. Dembinski, editor, *Proc. 9th Symposium on Mathematical Foundations of Computer Science*, volume 88 of *Lecture Notes in Computer Science*, pages 231–245, 1980.

[145] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Algebraic implementation of abstract data types: Concept, syntax, semantics and correctness. In J.W. de Bakker and J. van Leeuwen, editors, *Proc. 7th Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 142–156, 1980.

[146] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. A case study of abstract implementations and their correctness. In B. Robinet, editor, *Proc. International Symposium on Programming*, volume 83 of *Lecture Notes in Computer Science*, pages 108–122, 1980.

[147] Hartmut Ehrig, Hans-Jörg Kreowski, James W. Thatcher, Eric Wagner, and Jesse Wright. Parameterized data types in algebraic specification languages. In J.W. de Bakker and J. van Leeuwen, editors, *Proc. 7th Colloquium on Automata, Languages and Programming*, volume 85 of *Lecture Notes in Computer Science*, pages 157–168, 1980.

[148] Hartmut Ehrig and Hans-Jörg Kreowski. Algebraic theory of graph grammars applied to consistency and synchronization in data base systems. In M. Nagl and H.-J. Schneider, editors, *Graphs, Data Structures, Algorithms, Proc. Workshop on Graphtheoretic Concepts in Computer Science (WG'78)*, volume 13 of *Applied Computer Science*, pages 227–244, 1979.

[149] Hartmut Ehrig, Hans-Jörg Kreowski, and Herbert Weber. Neue Aspekte algebraischer Spezifikationsschemata für Datenbanksysteme. In H.C. Mayr and B.E. Meyer, editors, *Proc. GI-Fachtagung Formale Modelle für Informationssysteme*, volume 21 of *Informatik-Fachberichte*, pages 181–198, 1979.

[150] Hans-Jörg Kreowski. A pumping lemma for context-free graph languages. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. International Workshop on Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 270–283, 1979.

[151] Hartmut Ehrig, Hans-Jörg Kreowski, Andrea Maggiolo-Schettini, Barry K. Rosen, and Jozef Winkowski. Deriving structures from structures. In J. Winkowski, editor, *Proc. 7th Symposium on Mathematical Foundations of Computer Science*, volume 64 of *Lecture Notes in Computer Science*, pages 177–190, 1978.

[152] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Stepwise specification and implementation of abstract data types. In G. Ausiello and C. Böhm, editors, *Proc. 5th Colloquium on Automata, Languages and Programming*, volume 62 of *Lecture Notes in Computer Science*, pages 205–226, 1978.

[153] Hartmut Ehrig, Hans-Jörg Kreowski, and Herbert Weber. Algebraic specification schemes for data base systems. In B. S. Yao, editor, *Proc.*

4th International Conference on Very Large Data Bases, pages 427–440, 1978.

[154] Hans-Jörg Kreowski. Transformations of derivation sequences in graph grammars. In M. Karpiński, editor, *Proc. International Conference on Fundamentals of Computation Theory*, volume 56 of *Lecture Notes in Computer Science*, pages 275–286, 1977.

[155] Hartmut Ehrig and Hans-Jörg Kreowski. Parallel graph grammars. In A. Lindenmayer and G. Rozenberg, editors, *Proc. Automata, Languages, Development*, pages 425–442, 1976.

[156] Hartmut Ehrig and Hans-Jörg Kreowski. Categorial approach to graphic systems and graph grammars. In *Proc. Algebraic System Theory*, volume 131 of *Lecture Notes in Economics and Mathematical Systems*, pages 323–351, 1976.

[157] Hartmut Ehrig and Hans-Jörg Kreowski. Minimization concepts of automata in pseudoclosed categories. In *Proc. Algebraic System Theory*, volume 131 of *Lecture Notes in Economics and Mathematical Systems*, pages 359–374, 1976.

[158] Hartmut Ehrig and Hans-Jörg Kreowski. Parallelism of manipulations in multidimensional information structures. In A. Mazurkiewicz, editor, *Proc. 5th Symposium on Mathematical Foundations of Computer Science*, volume 45 of *Lecture Notes in Computer Science*, pages 284–293, 1976.

[159] Hartmut Ehrig and Hans-Jörg Kreowski. Power and initial automata in pseudoclosed categories. In E. G. Manes, editor, *Proc. 1st International Symposium on Category Theory Applied to Computation and Control*, volume 25 of *Lecture Notes in Computer Science*, pages 144–150, 1974.

Further Publications

[160] Karsten Hölscher, Renate Klempien-Hinrichs, Peter Knirsch, Hans-Jörg Kreowski, and Sabine Kuske. Regelbasierte Modellierung mit autonomen Transformationseinheiten. Technical Report 1/06, Universität Bremen, Fachbereich Mathematik & Informatik, 2006.

[161] Hans-Jörg Kreowski and Peter Knirsch, editors. *Proc. Applied Graph Transformation (AGT'02)*, 2002. Satellite Event of ETAPS 2002.

[162] Renate Klempien-Hinrichs and Hans-Jörg Kreowski. Algebraic specification goes multimedia – a few tentative steps. *Bulletin of the EATCS*, 75:224–227, 2001.

[163] Martin Wirsing, Martin Gogolla, Hans-Jörg Kreowski, Tobias Nipkow, and Wolfgang Reif, editors. *Proc. Rigorose Entwicklung software-intensiver Systeme*, Technical Report 0005, Universität München, 2000.

[164] Hartmut Ehrig, Hans-Jörg Kreowski, and Fernando Orejas. Correctness of actualization for parameterized implementation concepts based on constructors and abstractors. *Bulletin of the EATCS*, 56, 1995.

[165] Frank Drewes, Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. A sketch of collage grammars. *Bulletin of the EATCS*, 50:209–219, 1993.

[166] Frank Drewes, Hans-Jörg Kreowski, and Sabine Kuske. Hyperedge replacement: A basis for efficient graph algorithms. In M. Beyer, H. Ehrig, and M. Löwe, editors, *Computing by Graph Transformation (COMPUGRAPH) — Survey, Results, and Applications*, 1992. Project brochure.

[167] Hans-Jörg Kreowski. Aspects of systems of logic programming. *Bulletin of the EATCS*, 44:144–146, 1991.

[168] Reinhold Franck, Brian J. Gerloff, Roland Hardt, Rainer Isle, Hans-Jörg Kreowski, Inger Kuhlmann, Klaus-Peter Löhr, Richard Voet, and Anne Wilharm. Informatik 2000, Kampf um Märkte und Vorherrschaft. Informatik-Bericht 5, Universität Bremen, 1987.

[169] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. Compatible graph properties are decidable for hypergraph replacement graph languages. *Bulletin of the EATCS*, 33:55–61, 1987.

[170] Hartmut Ehrig, Hans-Jörg Kreowski, and Peter Padawitz. Algebraic implementation of abstract data types: An announcement. *SIGACT News*, 11(2):25–29, 1979.

[171] Hans-Jörg Kreowski. *Manipulationen von Graphmanipulationen.* Doctoral Dissertation. Technische Universität Berlin, 1978.

Part I

Greetings

Betriebssysteme und Algebraische Spezifikation?

Klaus-Peter Löhr

Wenn ich über Hans-Jörg Kreowski als *Kollegen* spreche, so kann ich bis in die 70er Jahre zurückblicken: Wir waren damals Kollegen an der Technischen Universität Berlin – damals noch als *Assistenten* (heute *Wissenschaftliche Mitarbeiter*) und dann als *Assistenzprofessoren* (heute *Juniorprofessoren*). In den 80er Jahren waren wir dann beide als Professoren am Aufbau der Informatik an der Universität Bremen beteiligt, bis ich 1985 an die Freie Universität Berlin wechselte.

In den Jahren nach 1968 herrschte an den (West-)Berliner Universitäten eine fieberhafte Umbruchstimmung, wie man sie sich heute kaum noch vorstellen kann. 1969 wurde durch ein *neues Hochschulgesetz* die Ordinarienuniversität durch die Gruppenuniversität abgelöst, und die großen Fakultäten wurden durch kleinere, überschaubare Fachbereiche ersetzt. An der Technischen Universität fielen diese Änderungen mit der Einrichtung des aus Bundesmitteln (2. DV-Programm) geförderten *neuen Studiengangs Informatik* zusammen. Ein neugegründeter *Fachbereich Kybernetik* – später in Informatik umbenannt – wurde zum Schauplatz lebhafter, bisweilen erbitterter Auseinandersetzungen: Studenten, Assistenten und Professoren stritten über fachliche, hochschulpolitische und allgemeinpolitische Fragen. Und häufig verbarg sich hinter einer fachlichen eine politische Kontroverse (und umgekehrt). Die Beteiligten waren natürlich keine Informatiker (denn solche gab es ja noch nicht), sondern kamen meist aus der Elektrotechnik oder aus der Mathematik.

Aus heutiger Sicht mutet bizarr an, dass bei einigen Forschungsgruppen vor Besetzung der Professorenstelle die zugehörigen Assistentenstellen besetzt wurden (von einer Einstellungskommission des Fachbereichsrats), was in der bundesdeutschen Informatik bald als "Berliner bottom-up-Methode" kritisiert wurde. Ohne fachliche Führung, und aus anderen Fächern kommend, mussten die Assistenten sich in Eigenverantwortung zu Informatikern fortbilden und häufig auch gleich eigenverantwortlich Lehrveranstaltungen durchführen. Man kann sich vorstellen, wie die Begeisterung für das neue Fach sowie die größere Rolle, die das neue Hochschulgesetz den Assistenten in den Gremien zubilligte, einen starken Zusammenhalt in der Assistentenschaft zur Folge hatte. Kontroversen zwischen den verschiedenen politischen Hochschulgruppen gab es trotzdem, und daran war auch die Gruppe beteiligt, in der Hans-Jörg und ich aktiv waren, die ADSen – Aktionsgemeinschaft von Demokraten und Sozialisten.

Wir waren in diesen Jahren Weggefährten bei der Ausgestaltung eines qualitativ anspruchsvollen Informatik-Studiengangs, sowohl hinsichtlich der "kerninformatischen" Qualität als auch unter Berücksichtigung der Anwendungen und Auswirkungen. Unser beider fachliche Ausrichtung war (und ist) allerdings denkbar unterschiedlich. Hans-Jörg kam von der kategorientheoretischen Behandlung von Automaten zur Theoretischen Informatik, ich von der Numerischen Mathematik zur Systemsoftware und Softwaretechnik. Wir haben nie fachlich zusammengearbeitet. Berührungspunkte gab es zwar in Gesprächen über formale Spezifikation. Ich konnte mich aber für Hans-Jörgs Arbeitsrichtung, die damals mehr auf die mathematische Fundierung der algebraischen Spezifikation als auf softwaretechnische Anwendung abzielte, nicht erwärmen. Gut erinnere ich mich an meine Ratlosigkeit bei einem Kolloquiumsvortrag von Joseph Goguen über algebraische Spezifikation, bei dem ich nichts Weitergehendes erkennen konnte als die alternative Formulierung einer bereits bekannten Technik. Auch dass die Begriffe *Algebraische Spezifikation* und *Abstrakte Datentypen* quasi als Synonyme verwendet wurden, fand ich nicht richtig. Kurz und gut, die theoretischen Grundlagen der algebraischen Spezifikation schienen mir für die praktische Softwaretechnik wenig relevant.

Politisch allerdings lagen wir auf gleicher Wellenlänge, und das mit vielen anderen Assistenten im Fachbereich. Dieses grundsätzliche Einvernehmen war die Basis für viele gemeinsame Aktivitäten in der Hochschule und darüber hinaus. Ein prominentes Beispiel war die *Kampagne gegen Berufverbote* in den Jahren 1973-76. Im Gefolge des *Radikalenerlasses* der Ministerpräsidenten von 1972 wurde 1975 einigen Assistenten der Universität (damals als Beamte auf Widerruf eingestellt) die Verlängerung ihrer Verträge verweigert mit der Begründung, dass sie nicht die Gewähr dafür böten, "jederzeit für die freiheitliche demokratische Grundordnung einzutreten". Die Empörung darüber war im Fachbereich so groß, dass viele Assistenten im Januar 1976 einem Appell zur Rücknahme der Entlassungen mit einem einwöchigen *Lehrboykott* Nachdruck verliehen: Wir ließen – unter Verletzung unserer Dienstpflichten – unsere Lehrveranstaltungen ausfallen. Den Entlassenen half das leider nichts, und unsere Aktion blieb nicht ungeahndet und führte zu einem Eintrag in die Personalakten. Unsere Klage dagegen blieb natürlich erfolglos. – So etwas schweißt zusammen.

1979 wechselte ich auf eine Professorenstelle in den neu eingerichteten Studiengang Informatik im Fachbereich Mathematik der Universität Bremen, in den auch Wolfgang Coy und Hermann Gehring berufen wurden. Wir waren sehr froh, dass wir bald danach auch Hans-Jörg für Bremen gewinnen konnten und dass es gelang, Frieder Nake von der Elektrotechnik zur Informatik zu holen. Bremen war damals als *rote Kaderschmiede* verrufen; allerdings wurde im Fachbereich Mathematik alles nicht so heiß gegessen. Wir fanden eine reformfreudige Umgebung vor und konnten einen Studiengang entwickeln, der sich durch ein projektorientiertes Hauptstudium sowie durch eine starke Berücksichtigung gesellschaftsbezogener Inhalte auszeichnete. Hans-Jörgs Engagement beschränkte sich dann auch nicht auf die Theoretische Informatik. Um eines von vielen Beispielen zu nennen: Ich weiß nicht, wann er seine Weihnachtsvorlesung zum ersten Mal durchführte; aber sie findet immer noch statt, ist heute so beliebt wie damals und ist ein typisches Element der Bremer Informatik.

Auf dem Höhepunkt der durch den NATO-Doppelbeschluss von 1979 ausgelösten Friedensbewegung wurde 1984 das FIfF gegründet – *Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung.* An der Gründung waren Mitglieder und ehemalige Mitglieder der TU Berlin maßgeblich beteiligt. Hans-Jörg war von Anfang an mit dabei und hat sich im Laufe der Jahre immer wieder stark im FIfF engagiert, im Vorstand, bei den Jahrestagungen, in der Bremer Regionalgruppe und – gerade auch jetzt wieder im 25. Jahr des FIfF – als FIfF-Vorsitzender. Dieses Engagement ist charakteristisch für Hans-Jörg: Als akademischer Lehrer lebt er den Studierenden vor, wie man die Tätigkeit als Informatiker mit gesellschaftlichem Engagement verbinden kann. Es gibt in der deutschen Informatik nur wenige Hochschullehrer, die in dieser Weise über ihr Fach hinaus wirken.

Nicht nur Hans-Jörg und das FIfF, auch die Bremer Informatik feiert ein Jubiläum – sie wird 30 Jahre alt. Ich bin froh, dass sie heute zu den in jeder Hinsicht vorzeigbaren Informatiken in Deutschland gehört, und weiß, dass Hans-Jörg einen wesentlichen Anteil daran hat. Ich beglückwünsche beide und hoffe, dass die Bremer Informatik-Studierenden zu schätzen wissen, was sie an ihrem Studiengang haben.

Und für die von Hans-Jörg geleiteten Gremiensitzungen hoffe ich, dass er die Teilnehmer immer noch so schön mit einem *Obstteller* erquickt, wie er das während meiner Zeit in Bremen getan hat.

Prof. Dr. Klaus-Peter Löhr (emeritus)
Institut für Informatik
Fachbereich Mathematik und Informatik
Freie Universität Berlin
D-14195 Berlin (Germany)
lohr@inf.fu-berlin.de
http://page.mi.fu-berlin.de/lohr

Klaus-Peter Löhr and Hans-Jörg Kreowski were colleagues during two periods in their careers. First, until 1978, as research associates at the Department of Computer Science at TU Berlin and then, from 1982 to 1985, as professors at the Department of Computer Science at the University of Bremen.

Graphs are Everywhere

Giorgio Busatto and Peter Knirsch

Bremen, 2009

We would like to take the occasion of Hans-Jörg's 60$^{\text{th}}$ anniversary to thank him for the scientific and professional experience that we could gather during our stay in his group in Bremen. As a founder of many concepts our research was based on he was for us a guide and a steady source of inspiration.

After being colleagues in Hans-Jörg's group, we both took a different direction and worked in the industry in Germany and Italy, respectively. For some strange coincidence, since September 2008 we are again colleagues in a software company in Bremen.

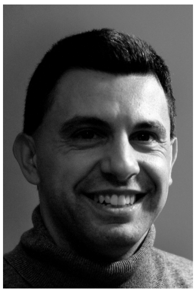
We often remember one of Hans-Jörg's favourite quotes: "Graphs are everywhere!" Indeed, graphs often offer us the right level of abstraction to reason about problems we encounter in our daily work: "Think of it as a graph together with its possible transformation." So even if we have not found a straightforward bridge between theory and practice yet—if there is a bridge it surely is a graph.

Giorgio Busatto and Peter Knirsch

Graph grammar riddle

H	K	T	G	X	L	B	J	T	G
S	P	O	F	D	P	O	A	R	C
G	E	T	G	R	A	T	S	E	C
E	R	V	P	R	O	G	R	E	S
A	E	O	G	R	A	G	R	A	S
A	G	T	I	V	E	T	R	U	T
E	D	G	E	L	E	L	V	I	S
U	M	L	O	M	O	D	N	A	C
S	U	O	C	S	X	U	O	Y	H
R	T	W	M	Q	M	Z	H	N	T

„Solution: You can try to find as many Gragra related terms as possible but try to mark the vowels first."



Dr. Giorgio Busatto
MeVis Medical Solutions
Universitätsallee 29
D-28359 Bremen (Germany)
giorgioxyzb@hotmail.com

Giorgio Busatto was a guest researcher in Hans-Jörg Kreowski's team with a GetGraTS grant in 2000 and 2001. They co-authored several papers in the field of graph transformation, and Hans-Jörg Kreowski was external examiner of his doctoral thesis.

Dr. Peter Knirsch
MeVis BreastCare Solutions
Universitätsallee 29
D-28359 Bremen (Germany)
peter@knirsch.info

Peter Knirsch was a doctoral student supervised by Hans-Jörg Kreowski. He was a research associate in Hans-Jörg's team from 1997 to 2007 and received his doctoral degree in 2007.

Alter schützt vor Theoretisieren nicht

Sofie Czernik

Lieber Hans Jörg,

ich hoffe, dass dieser modifizierte Spruch stimmt – nicht nur für Dich. Denn was wäre das Leben ohne zeitweilige Flucht in die abstrakte Welt des Theoretisierens? Langeweile, nichts als Langeweile! Der Reiz des Theoretisierens besteht doch gerade darin, die ausgetretenen Gedankenpfade zu verlassen, dem Alltagstrott zu entfliehen und ganz neue Wege zu bestreiten. Dazu sind wir hoffentlich nie zu alt.

Ich wünsche Dir zu Deinem 60. Geburtstag weiter hin den Mut zu vielen, vielen neuen Wegen in der (Graph-)Theorie!

Gleichzeitig erinnere ich mich gerne an manche interessante Stunden des Theoretisierens (und nicht nur), die ich bereits vor mehr als zehn Jahren dank Dir in der Arbeitsgruppe Theoretische Informatik an der Universität Bremen mit vielen Gleichgesinnten verbringen durfte. Auch die Irrwege waren herrlich!

Nun freue ich mich auf das Wiedersehen im September, um an diesem Ehrentag Dich hochleben und mit Dir die schöne Zeit noch mal aufleben lassen zu können,

Sofie

P.S. Ebenso schöne Geburtstagsgrüße und -wünsche von Dennis Chong.

Prof. Dr. Sofie Czernik
Fachbereich 2
Hochschule Bremerhaven
27568 Bremerhaven (Germany)
s.czernik@hs-bremerhaven.de
http://www.hs-bremerhaven.de/Sofie_Czernik.html

When Sofie Czernik was a doctoral student at the University of Bremen during the years 1993–1997, Hans-Jörg Kreowski was her second examiner. From 1997 to 1999 she was a member of his team, and is now a professor at Bremerhaven University of Applied Sciences. Together with Hans-Jörg Kreowski, she is currently supervising Dennis Chong's doctoral studies.

Ein FIfFiger Informatiker

Stefan Hügel

Bei den vielen Würdigungen von Hans-Jörg Kreowski in diesem Band darf natürlich eine nicht fehlen: die des FIfF, das er durch seine Arbeit über Jahre hinweg wesentlich geprägt hat. Typischerweise fällt diese Aufgabe dem Vorsitzenden zu – da er sich nicht gut selbst würdigen kann, darf nun also ich diese erfreuliche Aufgabe übernehmen.

Selbst bin ich erst 1993 ins FIfF eingetreten (und habe danach jahrelang ein Leben als „FIfF-Kommunikation-lesende Karteileiche" geführt); Hans-Jörgs Aktivitäten begannen also lange vor meiner Zeit, auch schon vor der Gründung des FIfF. Hervorheben will ich vor allem seine Stellungnahme zur *Star-Wars-Initiative* (SDI) des damaligen US-Präsidenten Reagan, gegen die er sich zusammen mit vielen anderen prominenten FIfF-Mitgliedern öffentlich ausgesprochen hat.

Die Inhalte des Themengebiets „Informatik und Gesellschaft" – und damit die Inhalte des FIfF – haben sich seitdem gewandelt. Manche haben an Bedeutung verloren, einige sind hinzugekommen, viele sind geblieben. Das Thema Frieden – Gründungsimpuls und Kernthema das FIfF – hat heute eine andere, aber kaum geringere Bedeutung als damals im kalten Krieg. Datensammelwut und Überwachungswahn – damals bereits wichtige Themen – sind trotz des Rechts auf informationelle Selbstbestimmung heute noch bedeutsamer geworden, auch deswegen, weil viele ihr Recht auf informationelle Selbstbestimmung nicht in dem Maße wahrnehmen, wie sie es könnten – und vielleicht sollten.

Nicht vorauszusehen war zu dieser Zeit die Entwicklung des Internet. Kommuniziert wurde über Mailboxen und Bildschirmtext – das Netz in der Form, die heute für uns selbstverständlich ist, steckte bestenfalls in den Kinderschuhen. Neben großen Chancen, die wir alle gerne wahrnehmen, birgt es auch Risiken – das FIfF sorgt bis heute, zusammen mit vielen anderen, dafür, dass sie in der Euphorie nicht übersehen werden.

Anders als manche Aktive der achtziger und frühen neunziger Jahre ist Hans-Jörg Kreowski dem FIfF und seinen Inhalten treu geblieben. Er hat in einer kritischen Phase des Vereins 2003 den Vorsitz übernommen, den er bis heute inne hat. Das FIfF hat ihm nicht zuletzt deswegen viel zu verdanken.

Für die demnächst anstehenden Wahlen hat er angekündigt, nicht mehr für den Vorsitz zu kandidieren. Das wird eine nur schwer zu füllende Lücke hinterlassen. Gerne nehmen wir zur Kenntnis, dass dies aber nicht das Ende seines Engagements im FIfF sein wird.

Zu seinem 60. Geburtstag und für seinen weiteren beruflichen und privaten Weg wünschen wir vom Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF e.V.) Hans-Jörg Kreowski alles Gute!

Bemerkung. Ein Beitrag von Ralf E. Streibl zur Geschichte des FIfF findet sich auf Seite 341 dieser Festschrift.

Stefan Hügel
c/o FIfF e.V.
Goetheplatz 4
D-28203 Bremen (Germany)
sh@fiff.de

Hans-Jörg Kreowski and Stefan Hügel are both members of the FIfF-Board – Hans-Jörg Kreowski being the Chairman and Stefan Hügel the Vice Chairman. Stefan studied Computer Science at the Universities of Karlsruhe and Freiburg. He currently lives in Munich; in his professional life he works for a Software and Consulting Company.

Stomach Cramps, Dadaism, and Marinated Truts

Andy Schürr

Dear Hans-Jörg,

Unfortunately, I can't recall the moment when we met for the first time. Probably, it was around the time of the 4th Int. Workshop on Graph Grammars in Bremen. At that time I was a greenhorn in the gragra research community with my first dreams of a graph transformation programming environment and you a renowned graph grammar veteran. Nevertheless, you were the first professor in my life that I was allowed to address informally ("duzen").

Afterwards we met each other again and again in Bremen and other glorious places around the world as the "International Committee of German Tourists"; especially when you started the successful initiative to create closer contacts and bonds between the different graph grammar subschools in Aachen, Berlin, Bremen, and other places. During that time I've learned to spell difficult words like"pushout" or "hypergraph", and to appreciate dadaism and the related poems of Hans Arp, Kurt Schwitters, Hans-Jörg Kreowski, and other famous artists.

We even wrote papers together including one, where we introduced a fixpoint theory for nested *Truts*. But alas, soon afterwards we found out that "Trut" is not only the abbreviation for *Nested Graph Transformation Units*, but also for a very special kind of human beings in Dutch and the name for a sort of demon that causes nightmares. Maybe this was the reason why I did always have nasty stomach cramps, when I visited your research group in Bremen for writing these papers (although it is more likely that your very old and rarely used coffee machine has to be blamed, since I was the only one who was drinking coffee and not tea during our meetings).

Today I even found out that "Trut" is a synonym for "eel" and I googled the following instructions how to handle truts appropriatley: *"Truts zu mariniren. Man reisset, salzet und kerbet dieselben, und bratet sie auf dem Rost, wobei sie stets mit Baumöl zu beschmieren sind. . . . "*

Please keep these instructions always in mind when you are writing new papers about further extensions of this powerful graph transformation composition mechanism. By the way: Merging these instructions with a description of the essential ideas of our joint fixpoint semantics paper and feeding the result as input to a Dada poetry generator produces about the following result, my special gift for your 60th birthday:

Vermittelst sodenn endlich Truts reisset,
daß sie werden category-based Fixpoints.
Unten fortzufahren wird Pushout auf dem
gänzlich Graphdefinition dieselben,
sodenn pullback semantics darauf!

My special greetings for this very special event!

Andy

Prof. Dr. Andy Schürr
Real-Time Systems Lab (FG EchtzeitSysteme)
Darmstadt University of Technology
Andy.Schuerr@es.tu-darmstadt.de
http://www.es.tu-darmstadt.de

Similar to Hans-Jörg Kreowski's research interests, one of the main research fields of Andy Schürr is graph transformation. Since the early 1990s, they have met numerous times on conferences, workshops, and informal research meetings. They took part in the EC projects CompuGraph, AppliGraph, and SeGraVis, and have several joint publications.

Erinnerungen

Nils Schwabe

So kurz erst habt ihr euch vom wilden Stammbaum abgelöst, so eng seid ihr noch mit den Lemuren und Halbaffen verwandt, daß ihr, nach Abstraktion strebend, der Anschaulichkeit nicht entbehren könnt, so daß ein Vortrag, der nicht auf praller Sinnlichkeit beruht, der voll von Formeln ist, die über einen Stein mehr sagen, als euch das Betrachten, Belecken und Betasten dieses Steins verraten können, euch langweilt und abstößt oder doch ein Gefühl der Unbefriedigung zurückläßt, das selbst den hohen Theoretikern, den Abstraktoren eurer höchsten Klasse, nicht fremd ist, wovon zahllose Beispiele aus den vertraulichen Geständnissen von Wissenschaftlern Zeugnis geben, denn sie bekennen sich in überwältigender Mehrheit dazu, sich beim Entwickeln abstrakter Argumente ganz auf sinnlich faßbare Dinge stützen zu müssen.
– Stanislaw Lem, Also sprach GOLEM. Suhrkamp, Frankfurt a.M., 1981

Lieber Hans-Jörg,

zunächst einmal meinen herzlichsten Glückwunsch zu Deinem 60. Geburtstag!

Die Gelegenheit, zu diesem festlichen Anlaß ein Grußwort an Dich richten zu können und in diesem einmal kurz die gemeinsam verbrachte Zeit zu reflektieren, nehme ich sehr gerne wahr.

Es ist eine etwas irritierende Erkenntnis, dass mittlerweile schon fast 12 Jahre vergangen sind, seit ich die Universität verlassen habe, um in der freien Software-Wirtschaft mein Heil zu suchen. Also einmal in alten Unterlagen geblättert: das BiZarR2-Projekt, (Map-) L-Systeme, IFS, Collagen-Grammatiken, Bilder und Filme daraus machen. Das waren schon außergewöhnliche Themen, in jedem Fall interessant und – bizarr. Für mich persönlich war es ein guter Weg, die theoretische Informatik kennen und schätzen zu lernen. So kam es zu einer Diplomarbeit über kontextsensitive Collagen-Grammatiken und einer direkt anschließenden Mitarbeit in der Arbeitsgruppe Theoretische Informatik mit praktischen und theoretischen Anteilen. Mir ist dies als eine sehr schöne und lehrreiche Zeit in Erinnerung, die ich mit sympathischen und kompetenten Kollegen verbringen durfte.

Umso mehr freue ich mich, nun nach dieser langen Zeit zumindest für einen Tag zurückkehren zu dürfen, um gemeinsam mit alten Bekannten noch einmal einzutauchen in die theoretische Ursuppe, aus der die Bits und Bytes – und die bizarren Bilder – hervorgegangen sind.

In diesem Sinne, alles Gute und auf ein Wiedersehen im September in Bremen,

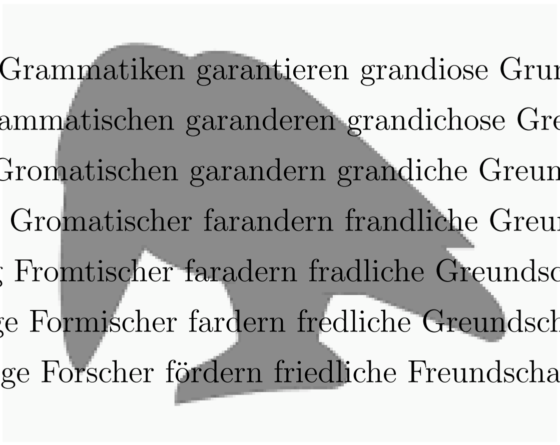
Dein Nils

Nils Schwabe
GEBIT Solutions GmbH
Hammer Straße 19
D-40219 Düsseldorf (Germany)
nils.schwabe@gebit.de

Nils Schwabe studied Computer Science at the University of Bremen. Hans-Jörg Kreowski was his teacher in several courses, and led the students' project BiZarR2. After receiving his diploma degree, Nils was a research associate in Hans-Jörg Kreowski's team from 1996 to 1997, before he decided to accept a job offer from the software industry.

Verbindungen schaffen

Karin Vosseberg und Andreas Spillner



Graph Grammatiken garantieren grandiose Grundlagen
Griph Grammatischen garanderen grandichose Greundlagen
Griff Gromatischen garandern grandiche Greundagen
Griffig Gromatischer farandern frandliche Greundafen
Friffig Fromtischer faradern fradliche Greundschafen
Friffige Formischer fardern fredliche Greundschaften
Fiffige Forscher fördern friedliche Freundschaften

Karin & Andreas möchten sich bei Dir für die langjährige Freundschaft und fortwährende Unterstützung recht herzlich bedanken.

Prof. Dr. Karin Vosseberg
Fachbereich 2
Hochschule Bremerhaven
D-27568 Bremerhaven (Germany)
Karin.Vosseberg@web.de

Karin Vosseberg studied Computer Science at the University of Bremen. Afterwards, she became a doctoral student supervised by Hans-Jörg Kreowski's colleague and friend Prof. Dr. Reinhold Franck. After Reinhold's accidental death, Hans-Jörg Kreowski became a major support, helping her to complete her doctoral studies. From 1997 to 2000, Karin was a member of his team, in the project *Informatica Feminale.* Another professional link is their engagement in the *Forum Computer Professionals for Peace and Social Responsibility* (FIfF). Over the years, Hans-Jörg Kreowski has turned from a teacher and mentor into a dear friend.

Prof. Dr. Andreas Spillner
Fakultät für Elektrotechnik und Informatik
Hochschule Bremen
D-28199 Bremen
andreas.spillner@hs-bremen.de
http://www.informatik.hs-bremen.de/spillner

Hans-Jörg Kreowski was the second supervisor of Andreas Spillner's doctoral thesis (first supervisor Prof. Dr. Reinhold Franck). After Reinhold Franck's accidental death in 1990, Hans-Jörg Kreowski acted as the provivional chair of the group. During these years, a vocational relation has turned into a friendship that is going to last.

Part II

Essays

Frm Ptr Nts t Grph Trnsfrmtn Sstms*

A Contribution in Honour of Hans-Jörg Kreowski

Paolo Baldan, Andrea Corradini, Fabio Gadducci, and Ugo Montanari

Abstract. Hans-Jörg Kreowski was among the first researchers to point out that P/T Petri nets can be interpreted as instances of Graph Transformation Systems, a fact now considered folklore. We elaborate on this observation, discussing how several different models of Petri nets can be encoded faithfully into Graph Transformation Systems. The key idea we pursue is that the net encoding is uniquely determined, and distinct net models are mapped to alternative approaches to graph transformation.

1 Introduction

The success of Petri nets as specification formalism for concurrent or distributed systems is due (among other things) to the fact that they can describe in a natural way the evolution of systems whose states have a distributed nature. For example, in a Place/Transition net like the one depicted in Fig. 1, a state of the system is represented by a marking, i.e., a set of tokens distributed among a set of places. Hence the state is intrinsically distributed, thus allowing for an easy explicit representation of phenomena like *mutual exclusion*, *concurrency*, *causality*, and *non-determinism*. Nets and their semantics are therefore a reference point for any formalism intended to describe concurrent and distributed systems, and thus also for Graph Transformation Systems (GTSs).

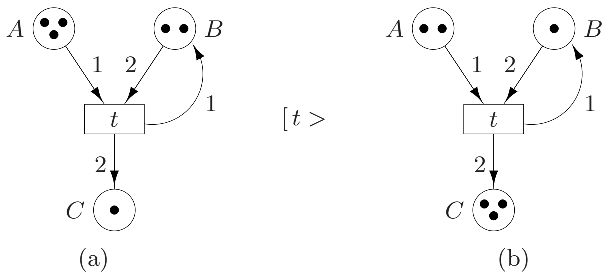


Fig. 1. (a) A marked P/T net. (b) The marking after the firing of transition t.

Indeed, it belongs to the folklore that Graph Transformation Systems can be seen as a generalisation of Petri nets. The first formalization of this intuition, to our knowledge, was proposed by Hans-Jörg Kreowski in [1] using the double-pushout (DPO) approach, and it is illustrated in Fig. 2. The marked net

* Research partially supported by the EU FP6-IST IP 16004 SENSORIA.

of Fig. 1(a) is represented in Fig. 2 by the graph $Kr(M_0)$ having three kinds of nodes (for transitions, places, and tokens, respectively) and where edges connect either places and transitions (modelling the causal dependency relation) or tokens and places (determining the place where a token lies). Transition t is represented by rule $Kr(t)$ (the top row of the figure): The rule does not modify the topological structure of the net (nodes and edges corresponding to places, transitions and causal dependency relation are also in the interface), but only deletes and creates the nodes representing tokens together with the edges connecting them to places. It is easy to check that the rule is applicable to graph $Kr(M_0)$ (the gluing conditions are satisfied), and since the two squares in the figure are pushouts, that $Kr(M_0) \overset{Kr(t)}{\Longrightarrow} Kr(M_1)$; moreover, the derived graph $Kr(M_1)$ represents the marking M_1, such that $M_0\,[t\rangle\,M_1$.

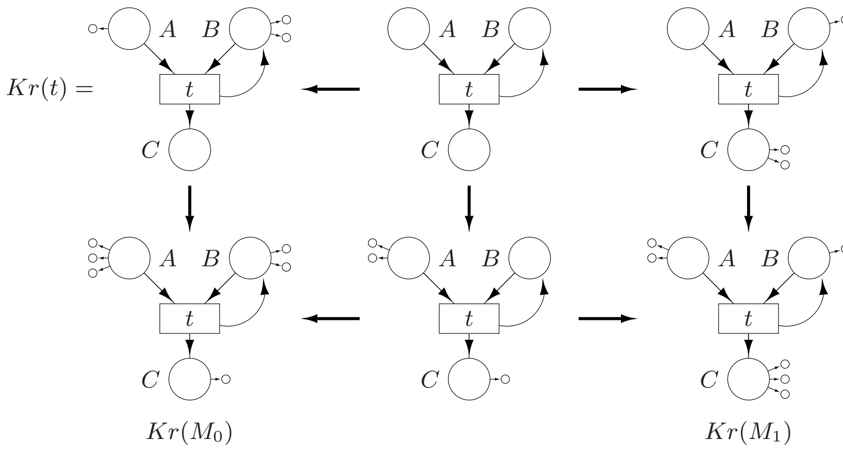


Fig. 2. Encoding of nets as grammars according to Kreowski.

Several encodings of Petri nets as GTSs have been proposed since then, and it is impossible even to summarize them here: for some of the earliest, see [2] and the references therein. In this paper we elaborate on this idea, starting from the observation that P/T nets are only one (a noticeable one) among the alternative models of Petri net which have been proposed along the years. Sticking to “low level” Petri nets, other models of nets may allow at most one token at a time in a place, as for *Condition/Event (C/E) nets* [3] or *Elementary Net Systems (*ENS*)* [4], and correspondingly a transition can fire only if the post-conditions are empty. In the so-called *Consume-Produce-Read (*CPR*) nets* [5], more permissively, the transition can fire anyhow, but the token produced on a place is “coalesced” with a possibly pre-existing token [5]. Orthogonally, nets of all kinds can be equipped with *read* or *inhibitor arcs*, specifying that the presence or the absence of a token on a place is necessary for firing, but it does not affect the result [6–10]. Another type of arcs, called *reset arcs* [11], allows to specify that the firing of a transition deletes all the tokens, if any, from a given place.

What about representing these models of nets as GTSs? In principle, all of them can be encoded using DPO rewriting, because the latter is Turing complete [12]. We prefer to follow a different approach, which on the one hand allows us to keep the encoding very simple for all the models of nets mentioned above, and on the other hand exploits the fact that also for GTSs alternative formalisms have been proposed. From the GTS side we shall stick to the family of algebraic approaches, among which we consider the classical single- and double-pushout approaches [13, 14], and the less known Subobject Transformation Systems [15]. The latter basically consists of rewriting in the lattice of subgraphs of a given graph, and it turns out to be the natural framework for encoding net models which allow at most one token on a place (where a state is a subset of places).

We encode nets using a very simple kind of graphs, containing nodes and unary edges only. A marking of a net is represented by a set of edges, one for each token, each attached to a node representing a place. It is thus reminiscent of the encoding by Kreowski discussed above, even if the transitions are not represented explicitly in the states: They are encoded only as rules of the GTS. Interestingly, inhibitor and reset arcs can be encoded exactly in the same way: The different behaviour is determined by the choice of the GTS approach.

The following table summarizes the results we shall present. For each of the three basic net models, we indicate the GTS approach that can be used to encode it in presence of read, inhibitor and/or reset arcs: note that we do not allow for nets which include both inhibitor and reset arcs.

	Read arcs	Read + Inhibitor	Read + Reset
P/T nets	DPO or SPO	DPO	SPO
ENS	STS or $\text{STS}^{\sqsubseteq}$	STS	$\text{STS}^{\sqsubseteq}$
CPR nets	STS_m or $\text{STS}_m^{\sqsubseteq}$	STS_m	$\text{STS}_m^{\sqsubseteq}$

Table 1. Summary of the proposed encodings.

The few variants of the STS approach referred to in the table will be introduced later on. The encodings of P/T Petri nets with read, inhibitor and reset arcs as GTSs were originally discussed in [16]. The present paper provides a systematic view of such encodings, viewing them in a much more general framework which recomprises Elementary Net Systems and CPR nets.

The paper is structured as follows. Section 2 presents the three GTS approaches we deal with in our work, and it is complemented in Section 3 by the kinds of nets for which we present an encoding. Section 4 discusses these encodings, and the correspondence between alternative net models and GTS approaches. Section 5 draws some conclusions and offers pointers to future works.

2 Algebraic approaches to graph transformation

This section introduces some basic notions concerning the algebraic formalisms for graph rewriting considered in the paper. We concentrate on *typed Graph Transformations Systems* (GTSs), both in the *single-pushout* (SPO) [13, 17] and the *double-pushout* (DPO) [14, 18] approach, and on *Subobject Transformation Systems* (STSs) [15]. Typed rewriting is a well-established variant of the classical proposals where rewriting takes place on so-called typed graphs, i.e., graphs labelled over a structure which is itself a graph [19, 20].

2.1 Graphs and graph morphisms

We introduce here the basic concepts concerning graphs and their morphisms. For the sake of simplicity, our introduction to GTSs will deal with unary hypergraphs only, since they are just what is needed for the encoding of Petri nets that we are going to present. Indeed, all the remarks in this section could be generalized to any kind of (hyper-)graphs or, albeit with some additional care, to any *adhesive* category [21]. Similarly, the encodings presented later would work in standard categories of (hyper-)graphs.

Given a partial function $f : A \rightarrowtail B$ we denote by $dom(f)$ its *domain*, i.e., the set $\{a \in A \mid f(a) \text{ is defined}\}$. Let $f, g : A \rightarrowtail B$ be two partial functions. We write $f \leq g$ when $dom(f) \subseteq dom(g)$ and $f(x) = g(x)$ for all $x \in dom(f)$.

Definition 1 (graph and graph morphism). *A* (unary) graph *G is a triple $G = (V_G, E_G, c_G)$, where V_G is a set of nodes, E_G is a set of edges and $c_G : E_G \to V_G$ is a function mapping each edge to the node it is connected to.*

A partial graph morphism *$f : G \rightarrowtail H$ is a pair of partial functions $f = \langle f_N : N_G \rightarrowtail N_H, f_E : E_G \rightarrowtail E_H \rangle$ such that $c_H \circ f_E \leq f_N \circ c_G$ (see Fig. 3.(a))*

We denote by **PGraph** *the category of (unlabelled) graphs and partial graph morphisms. A morphism is called* total *if both components are total, and the corresponding subcategory of* **PGraph** *is denoted by* **Graph**.

Notice that if a partial graph morphism f is defined over an edge, then it must be defined on the node the edge is connected to: This ensures that the domain of f is a well-formed graph.

Definition 2 (subgraph lattice). *A graph G is a* subgraph *of H, written $G \subseteq H$, if $N_G \subseteq N_H$, $E_G \subseteq E_H$, and the inclusions form a graph morphism. The set of subgraphs of H ordered by inclusion form a distributive lattice, denoted* $\mathbf{Sub}(H)$, *where the meet $\cap$ and the join $\cup$ are defined as component-wise intersection and union, respectively.*

Given graphs H and $G \subseteq H$, we will write, a bit informally, $H \setminus G$ to denote the set of items (nodes and edges) of H which do not belong to G.

Given a graph T, a *typed graph G* over T is a graph $|G|$, together with a total morphism $t_G : |G| \to T$. A *partial morphism* between T-typed graphs $f : G_1 \rightarrowtail G_2$ is a partial graph morphisms $f : |G_1| \rightarrowtail |G_2|$ consistent with the typing, i.e., such that $t_{G_1} \geq t_{G_2} \circ f$ (see Fig. 3.(b)). A typed graph G is called *injective* if the typing morphism t_G is injective. The category of T-typed graphs and partial typed graph morphisms is denoted by T-**PGraph**.

$$\begin{tikzcd}
E_G \arrow[r, "f_E"] \arrow[d, "c_G"'] \arrow[dr, phantom, "\geq"] & E_H \arrow[d, "c_H"] \\
N_G \arrow[r, "f_N"'] & N_H
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
{|G_1|} \arrow[rr, "f"] \arrow[dr, "t_{G_1}"'] & \geq & {|G_2|} \arrow[dl, "t_{G_2}"] \\
& T &
\end{tikzcd}$$

(a) (b)

Fig. 3. Diagrams for partial graph and typed graph morphisms.

Given a partial typed graph morphism $f : G_1 \rightarrowtail G_2$, we denote by $dom(f)$ the domain of f typed in the obvious way. Given a subgraph G of T, i.e., an element of $\mathbf{Sub}(T)$, we often consider it as a graph typed over T by the inclusion. Since we work only with typed notions, we usually omit the qualification "typed".

2.2 Double-pushout rewriting

Chosen a type graph T, a *(T-typed)* DPO *rule* $q = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ is a pair of *injective* (total, T-typed) graph morphisms $l : K \hookrightarrow L$ and $r : K \hookrightarrow R$, where $|L|$, $|K|$ and $|R|$ are finite graphs. The graphs L, K, and R are called the *left-hand side*, the *interface*, and the *right-hand side* of the rule, respectively.

Definition 3 (DPO direct derivation). *Given a graph G, a* DPO *rule q, and a* match *(i.e., a total graph morphism) $g : L \to G$, a* DPO direct derivation from G to H using q (based on g) *exists, written $G \Rightarrow_q^{\mathrm{DPO}} H$, if the diagram*

$$\begin{tikzcd}
q : L \arrow[d, "g"'] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d, "k"] & R \arrow[d, "h"] \\
G & D \arrow[l, "b"] \arrow[r, "d"'] & H
\end{tikzcd}$$

*can be constructed, where both squares are pushouts in T-***Graph**.

Given an injective morphism $l : K \hookrightarrow L$ and a match $g : L \to G$ as in the above diagram, their *pushout complement* (i.e., a graph D with morphisms k and b such that the left square is a pushout) exists if and only if the *gluing condition* is satisfied. This consists of two parts:

– the *identification condition*, requiring that if two distinct nodes or edges of L are mapped by g to the same image, then both are in the image of l;

– the *dangling condition*, stating that no edge in $G \setminus g(L)$ should be connected to a node in $g(L \setminus l(K))$ (because otherwise the application of the rule would leave such an edge "dangling").

2.3 Single-pushout rewriting

Chosen a type graph T, a *(T-typed)* SPO *rule* $q = (L \overset{r}{\rightarrowtail} R)$ is an injective partial typed graph morphism $r : L \rightarrowtail R$. The graphs L and R are called the *left-hand side* and the *right-hand side* of the rule, respectively.

Definition 4 (SPO direct derivation). *Given a graph G, an* SPO *rule q, and a* match *(i.e., a total graph morphism) $g : L \to G$, we say that there is an* SPO direct derivation from G to H using q (based on g)*, written $G \Rightarrow_q^{\mathrm{SPO}} H$, if the following is a pushout square in T-***PGraph***.*

$$\begin{tikzcd}
L \arrow[r, tail, "r"] \arrow[d, "g"'] & R \arrow[d, tail, "h"] \\
G \arrow[r, tail, "d"'] & H
\end{tikzcd}$$

Roughly speaking, the rewriting step removes from the graph G the image of the items of the left-hand side which are not in the domain of r, namely $g(L \setminus dom(r))$, adding the items of the right-hand side which are not in the image of r, namely $R \setminus r(dom(r))$. The items in the image of $dom(r)$ are "preserved" by the rewriting step (intuitively, they are accessed in a "read-only" manner).

A relevant difference with respect to the DPO approach is that here there is no *dangling condition* preventing a rule to be applied whenever its application would leave dangling edges. In fact, as a consequence of the way pushouts are constructed in T-**PGraph**, when a node is deleted by the application of a rule also all the edges connected to such node are deleted by the rewriting step, as a kind of side-effect. For instance, rule q in the top row of Fig. 4, which consumes node B, can be applied to the graph G in the same figure. As a result both node B and edge L are removed.

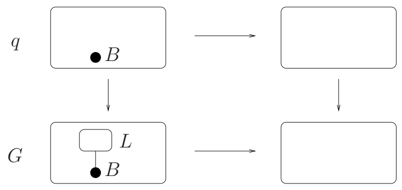


Fig. 4. Side-effects in SPO rewriting.

Even if the category **PGraph** has all pushouts, still we will consider a condition which corresponds to the *identification condition* of the DPO approach.

Definition 5 (valid match). *A match $g : L \to G$ is called* valid *when for any $x, y \in |L|$, if $g(x) = g(y)$ then $x, y \in dom(r)$.*

Conceptually, a match is not valid if it requires a single resource to be consumed twice, or to be consumed and preserved at the same time. In the paper we consider only *valid* derivations: This is needed to have a resource-conscious interpretation for derivations, i.e., where a resource is consumed at most once.

We close this section noting that for each DPO rule we can easily construct an SPO rule, which behaves like the original one when the dangling condition is satisfied. Clearly, the converse construction is possible as well.

Definition 6 (from DPO to SPO rules, and vice versa). *Let $q = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ be a T-typed DPO rule. Then, the associated T-typed SPO rule, denoted by $\mathcal{S}(q)$, is given by the partial graph morphism $r \circ l^* : L \rightarrowtail R$, where $l^* : L \rightarrowtail K$ is the partial inverse of l, defined in the obvious way.*

Vice versa, for a T-typed SPO rule $q = (L \overset{r}{\rightarrowtail} R)$, the associated DPO rule is defined as $\mathcal{D}(q) = (L \hookleftarrow dom(r) \overset{r}{\hookrightarrow} R)$.

2.4 Subgraph Transformation Systems

In the typed approaches to graph transformation, the type graph plays a role analogous to the set of places in Petri nets. In particular, the constraint that a place can contain at most one token can be translated into the requirement that the typing morphism is injective. Both the DPO and the SPO approaches can be equipped with side conditions that guarantee that only injectively typed graphs are generated during rewriting, but this condition is built-in in the instance of the *Subobject Transformation System* approach [15] that we present here.

In the original formulation, the framework where rewriting is defined is the distributive lattice of subobjects of a fixed object of an adhesive category. Such generality is unnecessary here, and we instantiate the definitions to the case where the category of concern is **Graph**, which is indeed adhesive. As a consequence, in the following we read "STS" as *Subgraph* Transformation Systems.

Chosen a type graph T, a *(T-typed)* STS *rule* q is a triple $q = \langle L, K, R\rangle$, where $L, K, R \in \mathbf{Sub}(T)$, $K \subseteq L$ and $K \subseteq R$. The graphs L, K and R are called the *left-hand side*, the *interface* and the *right-hand side* of the rule, respectively.

Definition 7 (STS direct derivation). *Given a graph G in $\mathbf{Sub}(T)$ and an* STS *rule $q = \langle L, K, R\rangle$, there is an* STS direct derivation from G to H using q, *written $G \Rightarrow_q^{\text{STS}} H$, if $H \in \mathbf{Sub}(T)$ and there exists $D \in \mathbf{Sub}(T)$ such that*

$$\begin{array}{ll} (i)\ \ L \cup D = G; & (iii)\ D \cup R = H; \\ (ii)\ L \cap D = K; & (iv)\ \ D \cap R = K. \end{array}$$

If such a graph D exists, we shall refer to it as the *context* of the direct derivation $G \Rightarrow_q^{\text{STS}} H$.

It is instructive to consider the relationship between an STS direct derivation and a DPO direct derivation as introduced above. First observe that $\mathbf{Sub}(T)$ can be seen as a category where the arrows are the inclusions, and a rule $\langle L, K, R\rangle$ can be seen as a span $q = (L \supseteq K \subseteq R)$, i.e., a pair of arrows in $\mathbf{Sub}(T)$. Next, we shall say that there is a *contact situation* for a rule $\langle L, K, R\rangle$ at a subgraph $G \supseteq L \in \mathbf{Sub}(T)$ if $G \cap R \not\subseteq L$. Intuitively, this means that some items of the subgraph G are created but not deleted by the rule: If we were allowed to apply the rule at this match via a DPO direct derivation, the resulting object would contain the common part twice and consequently the resulting morphism to T would not be injective; i.e., the result would not be a subgraph of T. The next result, presented in [15], shows that an STS direct derivation is also a DPO direct derivation if no contact occurs.

Proposition 1 (STS derivations are contact-free double pushouts). *Let G and H be graphs in $\mathbf{Sub}(T)$ and $q = \langle L, K, R\rangle$ be an STS rule. Then $G \Rightarrow_q^{\mathrm{STS}} H$ if and only if $L \subseteq G$, $G \cap R \subseteq L$, and $G \Rightarrow_q^{\mathrm{DPO}} H$, i.e., if there is a graph $D \in T$-$\mathbf{Graph}$ such that the diagram below forms two pushouts in T-$\mathbf{Graph}$.*

$$\begin{tikzcd}
L \arrow[d, "\subseteq"'] & K \arrow[l, "\supseteq"'] \arrow[r, "\subseteq"] \arrow[d] \arrow[l, phantom, "(1)", near start, shift left=0] & R \arrow[d] \\
G & D \arrow[l] \arrow[r] & H
\end{tikzcd}$$

In the last result we used the fact that an STS rule can be considered as a T-typed DPO rule, considering the inclusions as arrows in $\mathbf{Graph}$. Conversely, a T-typed DPO rule $q = (L \overset{l}{\hookleftarrow} K \overset{r}{\hookrightarrow} R)$ induces an STS rule $\mathcal{I}(q)$ obtained by considering the images of $|L|$, $|K|$ and $|R|$ in the type graph, i.e., $\mathcal{I}(q) = \langle t_L(|L|), t_K(|K|), t_R(|R|)\rangle$.

2.5 Other kinds of STSs

We introduce here three variations of the definition of STS direct derivation, obtained by slightly changing the properties verified by the context graph D.

The first definition is reminiscent of the *sesqui-pushout* approach [22], and it leads to an SPO-like approach for STS, where rules can be applied regardless of the dangling condition, removing, as a side-effect, those edges which would remain dangling.

Definition 8 (STS$^{\sqsubseteq}$ direct derivation). *Given a graph G in $\mathbf{Sub}(T)$ and an STS rule $q = \langle L, K, R\rangle$, there is an STS$^{\sqsubseteq}$* direct derivation from G to H using q, *written $G \Rightarrow_q^{\mathrm{STS}^{\sqsubseteq}} H$, if $H \in \mathbf{Sub}(T)$ and*

(ii)′ D is the largest subgraph of G such that $L \cap D = K$;
(iii) $D \cup R = H$;
(iv) $D \cap R = K$.

Dropping the first condition of Definition 7 and imposing the "largest subgraph" requirement in (ii) implies that some items of $G \setminus L$ may not occur in D, as when deleting a node forces the deletion of incident edges in the SPO approach.

The next variants drop the requirement $D \cap R = K$. This allows for some overlap between the items preserved in the context D and those newly introduced by R: The injectivity of the typing forces these items to be coalesced, similarly to what happens in CPR nets. This is done for STSs both in DPO and SPO style.

Definition 9 ($\textsc{sts}_m$ **and** $\textsc{sts}_m^{\sqsubseteq}$ **direct derivations).** *Given a graph G in* $\mathbf{Sub}(T)$ *and an* STS *rule* $q = \langle L, K, R\rangle$*, there is an* $\textsc{sts}_m$ direct derivation from G to H using q, *written* $G \Rightarrow_q^{\mathrm{STS}_m} H$*, if* $H \in \mathbf{Sub}(T)$ *and there exists* $D \in \mathbf{Sub}(T)$ *such that*

$$(i)\ \ L \cup D = G; \qquad (iii)\ D \cup R = H.$$
$$(ii)\ L \cap D = K;$$

Analogously, there is an $\textsc{sts}_m^{\sqsubseteq}$ direct derivation from G to H using q, *written* $G \Rightarrow_q^{\mathrm{STS}_m^{\sqsubseteq}} H$*, if* $H \in \mathbf{Sub}(T)$ *and*

(ii)′ D is the largest subgraph of G such that $L \cap D = K$;
(iii) $D \cup R = H$.

Figure 5 shows the differences among the various kinds of STS direct derivations introduced in Definitions 7, 8 and 9. The type graph T contains two nodes, $\circ$ and $\bullet$, and one edge connected to $\circ$; all the elements of $\mathbf{Sub}(T)$ (the subgraphs of T) are depicted, with the obvious inclusions. The arrows show all the possible direct derivations using the STS rule $q = \langle\{\circ\}, \emptyset, \{\bullet\}\rangle$ and the approaches introduced in Definitions 7, 8 and 9.

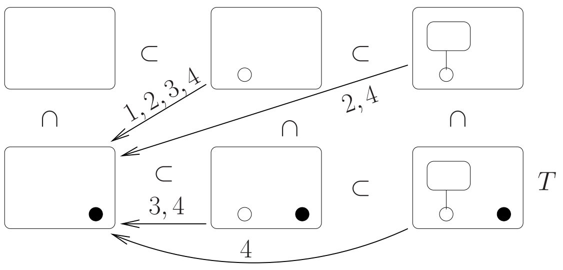


Fig. 5. Examples of the various kinds of STS direct derivations. Arrows represent direct derivations among elements of $\mathbf{Sub}(T)$ using rule $q = \langle\{\circ\}, \emptyset, \{\bullet\}\rangle$ and the following approaches: $1 = \textsc{sts}, 2 = \textsc{sts}^{\sqsubseteq}, 3 = \textsc{sts}_m, 4 = \textsc{sts}_m^{\sqsubseteq}$.

2.6 Graph grammars

In the previous sections we presented six different definitions of direct derivation, each of which determines a different algebraic approach to graph transformation. For each one of those approaches, a graph grammar contains a type graph, a start graph, a set of rule names, and a mapping from rule names to corresponding rules. Clearly, the precise definition of start graph and of rule depends on the chosen approach.

Definition 10 (graph grammar). *A* KND graph grammar*, where* KND $\in \{$DPO, SPO, STS, $\text{STS}^{\sqsubseteq}$, STS_m, $\text{STS}^{\sqsubseteq}_m\}$*, is a tuple* $\mathcal{G} = \langle T, G_s, P, \pi \rangle$*, where* $T \in \mathbf{Graph}$ *is the* type graph*,* P *is a set of* rule names*,* π *is a function which associates a* KND *rule*[1] *to each rule name in* P*, and* G_s *is the* start graph*, which has to be consistent with* KND*. That is,* G_s *is a* T*-typed graph if* KND $\in \{$DPO, SPO$\}$*, and* $G_s \in \mathbf{Sub}(T)$ *in all other cases.*

A *derivation* over a KND grammar $\mathcal{G}$ is a sequence of KND direct derivations using rules in P, starting from the start graph, namely $\rho = \{G_{i-1} \Rightarrow^{\text{KND}}_{p_{i-1}} G_i\}_{i \in \{1,\ldots,n\}}$, with $G_0 = G_s$.

3 Enriched Petri nets

In this section we introduce some basic extensions of Petri nets, namely, nets with read, inhibitor and reset arcs. A study of the expressiveness of these kinds of arcs, along with a comparison with other extensions proposed in the literature, like priorities, exclusive-or transitions and switches, is carried out in [23, 24].

To give the formal definition of these generalised nets we need some notation for sets and multisets. Given a set X we write $\mathbf{2}^X$ for the powerset of X and $X^{\oplus}$ for the free commutative monoid over X, with monoid operation $\oplus$, whose elements will be referred to as *multisets* over X. Given a multiset $M \in X^{\oplus}$, with $M = \bigoplus_{x \in X} M_x \cdot x$, for $x \in X$ we will write $M(x)$ to denote the coefficient M_x. Moreover, we denote by $[\![M]\!]$ the underlying subset of X, defined as $[\![M]\!] = \{x \in X \mid M(x) > 0\}$. With little abuse of notation, we will write $x \in M$ iff $x \in [\![M]\!]$.

Given $M, M' \in X^{\oplus}$ we write $M \leq M'$ when $M(x) \leq M'(x)$ for all $x \in X$. In this case the *multiset difference* $M' \ominus M$ is the multiset M'' such that $M \oplus M'' = M'$. For $Y \subseteq X$ and $M \in X^{\oplus}$, we denote by $M[Y]$ the restriction of M to Y, i.e., $M[Y](x) = M(x)$ if $x \in Y$, and $M[Y](x) = 0$ otherwise. Finally, the symbol $\emptyset$ denotes the empty multiset.

3.1 Place/Transition nets

We are now ready to define the enriched P/T nets considered in the paper. Besides ordinary flow arcs and read arcs, the nets are endowed with so-called "distinguished arcs" (represented by the $^{\circledcirc}(.)$ function below), which will be interpreted either as inhibitor or reset arcs in the token game.

[1] To be precise, for KND $\in \{\text{STS}^{\sqsubseteq}, \text{STS}_m, \text{STS}^{\sqsubseteq}_m\}$, a KND rule is an STS rule.

Definition 11 (enriched P/T nets). *An* enriched (marked) P/T Petri net *is a tuple* $N = \langle S, Tr, {}^\bullet(.), (.)^\bullet, \underline{(.)}, {}^{\circledcirc}(.), m\rangle$*, where*

- S *is a set of* places*;*
- Tr *is a set of* transitions*;*
- ${}^\bullet(.), (.)^\bullet : Tr \to S^\oplus$ *are functions mapping each transition to its pre-set and post-set, respectively;*
- $\underline{(.)} : Tr \to \mathbf{2}^S$ *is a function mapping each transition to its* context*;*
- ${}^{\circledcirc}(.) : Tr \to \mathbf{2}^S$ *is a function mapping each transition to its* distinguished set *of places, such that for all* $t \in Tr$*,* $({}^\bullet t \oplus \underline{t} \oplus t^\bullet)[{}^{\circledcirc}t] = \emptyset$ *(i.e., no token in* ${}^{\circledcirc}t$ *can be either read, consumed or produced by* t*);*
- $m \in S^\oplus$ *is a multiset called the* initial marking*.*

We assume, as usual, that $S \cap Tr = \emptyset$. We shall denote with ${}^\bullet(.), (.)^\bullet, \underline{(.)}$ and ${}^{\circledcirc}(.)$ also the functions from S to $\mathbf{2}^{Tr}$ defined as, for $s \in S$, ${}^\bullet s = \{t \in Tr \mid s \in t^\bullet\}$, $s^\bullet = \{t \in Tr \mid s \in {}^\bullet t\}$, $\underline{s} = \{t \in Tr \mid s \in \underline{t}\}$, and ${}^{\circledcirc}s = \{t \in Tr \mid s \in {}^{\circledcirc}t\}$.

A state of a P/T net is defined as a *marking*, that is, a set of tokens distributed over the places. Formally, a marking M is a multiset of places, i.e., $M \in S^\oplus$. The *token game* determines when a transition t is *enabled* at a given marking, and, if enabled, what marking is reached after firing the transition. For a transition t to be enabled at a marking M, it is necessary for M to contain the pre-set of t and an additional set of tokens which covers the context of t. Additional conditions for enabledness, as well as the result of firing, depend on the interpretation given to the distinguished arcs: As anticipated, we interpret them either as *inhibitor arcs* or as *reset arcs*, obtaining the classes of nets below.

Definition 12 (inhibitor and reset P/T nets). *An* inhibitor P/T net *is an enriched P/T net* $\langle S, Tr, {}^\bullet(.), (.)^\bullet, \underline{(.)}, {}^{\circledcirc}(.), m\rangle$ *where the distinguished arcs are interpreted as inhibitor arcs. Given a marking* $M \in S^\oplus$ *and a transition* $t \in Tr$*,* t *is* i-enabled *if* ${}^\bullet t \oplus \underline{t} \leq M$ *and* $M[{}^{\circledcirc}t] = \emptyset$ *(i.e.,* M *contains no token in any place of* ${}^{\circledcirc}t$*). The* inhibitor transition relation *between markings is defined as*

$$M\,[t\rangle_i\,M' \qquad \textit{if} \qquad t \textit{ is i-enabled at } M \textit{ and } M' = (M \ominus {}^\bullet t) \oplus t^\bullet.$$

A reset P/T net *is an enriched P/T net where the distinguished arcs are interpreted as reset arcs. Given* $M \in S^\oplus$ *and* $t \in Tr$*,* t *is* r-enabled *if* ${}^\bullet t \oplus \underline{t} \leq M$*. The* reset transition relation *is defined as*

$$M\,[t\rangle_r\,M' \qquad \textit{if} \qquad t \textit{ is r-enabled at } M \textit{ and } M' = ((M \ominus {}^\bullet t) \oplus t^\bullet) \ominus M[{}^{\circledcirc}t]$$

(i.e., the firing of t *deletes all the tokens from places in* ${}^{\circledcirc}t$*: Such places are certainly empty after the firing, because they cannot belong to the post-set of* t*).*

For a transition t, if the distinguished set ${}^{\circledcirc}t$ is empty the two alternative enabling conditions coincide, as well as the induced transition relations on markings. In the following, we call *contextual Petri nets* the class of nets such that all its transitions have the distinguished set empty.

Firing sequences and reachable markings are defined in the usual way.

Example 1. An example of an enriched P/T net N can be found in the left part of Fig. 6. Graphically, transitions are connected to context places by undirected arcs and to distinguished places by dotted undirected arcs.

Starting from the initial marking $s_0 \oplus s_1 \oplus s_2 \oplus s_4$, a possible firing sequence is $t_1; t_2$ leading to the marking $s_2 \oplus s_3 \oplus 2s_4 \oplus s$.

If we first fire t_2, the net reaches the marking $s_0 \oplus s_2 \oplus s_4 \oplus s$. Now, if N is seen as an inhibitor P/T net, the presence of a token in s inhibits t_1 which cannot fire. If, instead, N is seen as a reset P/T net, transition t_1 can fire and, as a consequence, place s is emptied, producing the marking $s_2 \oplus s_3 \oplus 2s_4$.

3.2 Elementary nets

Let us call *elementary* a net where the states are defined as *(sub)sets* of places, rather than *multisets* of places as for P/T nets. Thus elementary nets comprise several net models proposed in the literature, including C/E nets [3], Elementary Net Systems [4], Consume-Produce-Read nets [5] and others.

An *enriched elementary (marked) net* $\langle S, Tr, {}^\bullet(.), (.)^\bullet, \underline{(.)}, {}^{\circledcirc}(.), m\rangle$ is defined as an enriched P/T net in Definition 11, requiring ${}^\bullet(.), (.)^\bullet : Tr \to \mathbf{2}^S$ and $m \in \mathbf{2}^S$ (i.e., ${}^\bullet t$ and $t^\bullet$ for all $t \in Tr$, as well as the initial marking m, are sets rather than multisets). Furthermore, besides the disjointness condition on the distinguished places, that is formulated as $({}^\bullet t \cup \underline{t} \cup t^\bullet) \cap {}^{\circledcirc}t = \emptyset$, it is required that no token in $\underline{t}$ is consumed or produced, i.e., $({}^\bullet t \cup t^\bullet) \cap \underline{t} = \emptyset$ for all $t \in Tr$.

Both inhibitor and reset elementary nets are easily defined, interpreting the distinguished arcs as expected. However, since the states are subsets of places, the enabling condition and the transition relation must ensure that the marking reached by firing a transition is a set. This is obtained in a different way by the two models of nets that we introduce: ENSs require a stronger enabling condition w.r.t. P/T nets, while CPR nets, intuitively, change the transition relation by allowing to merge tokens of the marking with those produced by the transition.

Definition 13 (inhibitor and reset Elementary Net Systems). *An* inhibitor ENS *is an enriched elementary net* $\langle S, Tr, {}^\bullet(.), (.)^\bullet, \underline{(.)}, {}^{\circledcirc}(.), m\rangle$ *where the distinguished arcs are interpreted as inhibitor arcs. Given a marking* $M \subseteq S$ *and a transition* $t \in Tr$, t *is* ie-enabled *if* ${}^\bullet t \cup \underline{t} \subseteq M$, $M \cap {}^{\circledcirc}t = \emptyset$, *and* $(M \setminus {}^\bullet t) \cap t^\bullet = \emptyset$. *The* ie-transition relation *between markings is defined as*

$$M\,[t\rangle_{ie}\,M' \qquad \textit{if} \qquad t \textit{ is ie-enabled at } M \textit{ and } M' = (M \setminus {}^\bullet t) \cup t^\bullet.$$

A reset ENS *is an enriched elementary net where the distinguished arcs are interpreted as reset arcs. Given* $M \subseteq S$ *and* $t \in Tr$, t *is* re-enabled *if* ${}^\bullet t \cup \underline{t} \subseteq M$ *and* $(M \setminus {}^\bullet t) \cap t^\bullet = \emptyset$. *The* re-transition relation *is defined as*

$$M\,[t\rangle_{re}\,M' \qquad \textit{if} \qquad t \textit{ is re-enabled at } M \textit{ and } M' = ((M \setminus {}^\bullet t) \cup t^\bullet) \setminus {}^{\circledcirc}t.$$

The condition $(M \setminus {}^\bullet t) \cap t^\bullet = \emptyset$ ensures that there is "no contact", i.e., t can produce a token only if it is not in M, or if it is deleted by t itself. As a consequence the $\cup$ operator in the definition of M' is actually a disjoint union.

This is the main difference with respect to CPR nets, where the "no contact" condition is omitted, and the arguments of $\cup$ in the definition of the follower marking might not be disjoint.

Definition 14 (inhibitor and reset CPR nets). *An* inhibitor CPR net *is an enriched elementary net where for a marking $M \subseteq S$ and a transition $t \in Tr$, t is* ic-enabled *if ${}^{\bullet}t \cup \underline{t} \subseteq M$ and $M \cap {}^{\circledcirc}t = \emptyset$; the* ic-transition relation *is defined as*

$$M\,[t\rangle_{ic}\,M' \qquad \textit{if} \qquad t \textit{ is ic-enabled at } M \textit{ and } M' = (M \setminus {}^{\bullet}t) \cup t^{\bullet}.$$

A reset CPR net *is an enriched elementary net where for $M \subseteq S$ and $t \in Tr$, t is* rc-enabled *if ${}^{\bullet}t \cup \underline{t} \subseteq M$; the* rc-transition relation *is defined as*

$$M\,[t\rangle_{rc}\,M' \qquad \textit{if} \qquad t \textit{ is rc-enabled at } M \textit{ and } M' = ((M \setminus {}^{\bullet}t) \cup t^{\bullet}) \setminus {}^{\circledcirc}t.$$

Example 2. Observe that the net N in Fig. 6 can be seen as an ENS. In this case, starting from the initial marking $\{s_0, s_1, s_2, s_4\}$ the transition t_1 cannot fire due to a contact situation in s_4, hence the only possible firing sequence is t_2.

If we interpret N as a CPR net, then t_1 can fire and the reached marking is $\{s_1, s_2, s_3, s_4\}$, where, intuitively, the token generated in s_4 is "merged" with the pre-existing one. In this state, t_2 can fire producing the marking $\{s_2, s_3, s_4, s\}$. If we start by firing t_2, as in the P/T case, t_1 is blocked or can fire (emptying place s), depending on whether we interpret N as an inhibitor or a reset CPR net.

4 From enriched nets to graph transformation systems

This section shows how enriched Petri nets can be encoded as graph grammars. Interestingly, the encoding is essentially the same for all kinds of nets: The different token game flavours are obtained by changing the approach to rewriting.

4.1 Encoding Petri nets as graph grammars

It is part of the folklore (see e.g. the discussion in [2] and the references therein) that (ordinary) Petri nets can be seen as a special kind of graph grammars. The simplest idea is that the marking of a net is represented as a graph with no edges typed over the places: A token in place s is a node typed over s. Then transitions are seen as rules which consume and produce nodes, as prescribed by their pre- and post-set. In this way, Petri nets exactly correspond to graph grammars acting over graphs containing only nodes, where rules preserve no item.

To make the encoding parametric with respect to the chosen class of Petri nets, here we consider a slightly different encoding, where edges, rather than nodes, play the role of tokens. Roughly, the idea of the encoding is the following:

– a place is represented as a node;

– tokens in a place are represented as unary edges connected to the corresponding node;
– a transition becomes a rule, which deletes the tokens in its pre-set, produces the post-set and preserves the tokens in its context; for any place in the distinguished set of t, the corresponding node is deleted and created again.

Note the chosen encoding for the distinguished set of t: In the DPO approach this will prevent the application of the rule if there is at least one token (edge) in the place, thus causing an inhibitor effect. In the SPO approach, the application of the rule will delete as a side-effect any edge possibly attached to the node, thus giving raise to a reset effect.

As a first step, we show how the set of places underlying an enriched net (either P/T or elementary) gives raise to a type graph. In all cases there will be a node s in the type graph for each place s in the net, and the number of edges incident on the node typed over s will represent the number of tokens in that place. Also the way in which markings are encoded as graphs does not depend on the specific kind of nets we are considering.

Definition 15 (type graph, markings). *Let S be a set of places. Then, the associated type graph T_S is (S, S, c), where $c(s) = s$ for all $s \in S$.*

Given a subset of places $S' \subseteq S$ and a marking $M \in S'^{\oplus}$, we define the graph $\mathbf{G}_S(S', M)$ as $(S', E(M), c)$, typed in the obvious way over T_S, such that $E(M) = \{\langle s, i\rangle \mid s \in [\![M]\!] \wedge 0 < i \leq M(s)\}$ and $c(\langle s, i\rangle) = s$ for all $\langle s, i\rangle \in E(M)$. We write simply $\mathbf{G}_S(M)$ for $\mathbf{G}_S(S, M)$.

So, each place contributes a node *and* an edge in the type graph T_S, and a marking can be regarded as a multiset of edges of the type graph.

We next introduce the encoding of net transitions into grammar rules. As mentioned above, the encoding is essentially independent of the kind of nets we are considering: The different firing behaviour will be obtained by changing the considered rewriting approach. Indeed, we next define the encoding of a transition as a DPO rule, but changing the rewriting approach (to SPO or STS) will just require a syntactical change in the presentation of the rule.

Definition 16 (net transitions as DPO rules). *Let t be a transition of an enriched P/T net with place set S. Then t is encoded as a T_S-typed DPO transition*

$$\mathbf{G}_S(t) = \mathbf{G}_S(X \cup {}^{\circledcirc}t, \underline{t} \oplus {}^{\bullet}t) \leftarrow \mathbf{G}_S(X, \underline{t}) \rightarrow \mathbf{G}_S(X \cup {}^{\circledcirc}t, \underline{t} \oplus t^{\bullet})$$

where $X = [\![\, {}^{\bullet}t \oplus \underline{t} \oplus t^{\bullet}]\!]$ and the left and right morphisms are inclusions.

The DPO rule $\mathbf{G}_S(t)$ corresponding to a transition t deletes the edges in its pre-set, preserves the edges in its context and produces the edges in its post-set. The nodes attached to edges in the pre-set, context and post-set (i.e., the set X) are preserved. Finally, the nodes corresponding to the places $s \in {}^{\circledcirc}t$ in the distinguished set of t are deleted and produced again.

It is now immediate to provide the encoding for the different kinds of Petri nets into graph grammars of the appropriate approach.

	Inhibitor	Reset
P/T nets	DPO $\mathbf{G}_S(t)$	SPO $\mathcal{S}(\mathbf{G}_S(t))$
ENS	STS $\mathcal{I}(\mathbf{G}_S(t))$	$\text{STS}^{\sqsubseteq}$ $\mathcal{I}(\mathbf{G}_S(t))$
CPR nets	STS_m $\mathcal{I}(\mathbf{G}_S(t))$	$\text{STS}_m^{\sqsubseteq}$ $\mathcal{I}(\mathbf{G}_S(t))$

Table 2. Encoding Petri nets as graph grammars.

Definition 17. *An enriched Petri net $N = \langle S, Tr, F, C, D, m\rangle$ of one of the six types of nets presented in Definitions 12, 13 and 14 is encoded as a* KND *graph grammar $\mathcal{G}(N) = \langle T, G_s, P, \pi\rangle$ where*

- $T = T_S$
- $P = Tr$
- $G_s = \mathbf{G}_S(m)$

Moreover KND *and the* KND *rule $\pi(t)$ associated to $t \in P$ are defined, according to the type of the net, as shown in Table 2.*

Obviously, the encoding also works for contextual nets (see the first column of Table 1 in the Introduction).

It can be shown that the encoding preserves the firing relation and reachability, in the sense specified by next theorem.

Theorem 1. *Let N be an enriched Petri net of one of the types introduced in Section 3, let* KND *be the type of grammar corresponding to the type of N according to Table 2, and let M be a marking of N. If $M\,[t\rangle\, M'$ in N then $\mathbf{G}_S(M) \Rightarrow_t^{\text{KND}} \mathbf{G}_S(M')$ in the* KND *graph grammar $\mathcal{G}(N)$; vice versa, if $\mathbf{G}_S(M) \Rightarrow_t^{\text{KND}} G'$ in the* KND *graph grammar $\mathcal{G}(N)$ then $M\,[t\rangle\, M''$ in N with $\mathbf{G}_S(M'') = G'$.*

4.2 Examples

In order to provide some more intuition, we next briefly discuss the encoding for the various classes of Petri nets.

P/T Petri nets. As shown in Table 2, the behaviour of P/T Petri nets is faithfully captured by standard DPO or SPO graph grammars.

Inhibitor nets. When N is a P/T inhibitor net, $\mathcal{G}(N)$ is a DPO graph grammar, where the effects of the dangling condition are used to encode inhibitor arcs. As an example, the net in Fig. 6, seen as an inhibitor P/T net, is encoded by the grammar in the same figure, interpreted as a DPO grammar. Observe that since place $s \in {}^{\circledcirc}t_1$, i.e., s inhibits transition t_1, the rule associated with t_1 deletes and produces again the node corresponding to s. In this way the presence of tokens in place s, represented by edges connected to such node, will inhibit the rule because of the dangling condition.

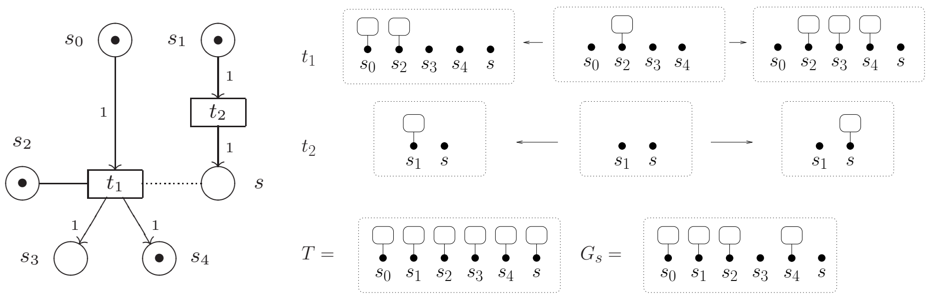


Fig. 6. An enriched Petri net N and the corresponding DPO grammar.

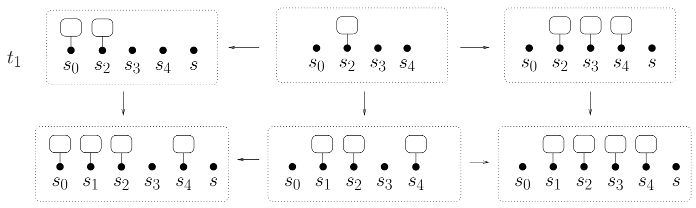
Reset nets. In the case of a P/T reset net N, the encoding $\mathcal{G}(N)$ is an SPO grammar and the side-effects related to node deletion turn out to capture precisely the behaviour of reset arcs. As an example the net in Fig. 6, seen as a reset P/T net, is encoded by the grammar in the same figure, seen as an SPO grammar (by transforming the rules using the function $\mathcal{S}(.)$). The fact that rule t_1 deletes and produces again the node s determines, as side effect, the deletion of all edges connected to such node, representing tokens in place s.

Contextual nets. For contextual P/T nets, i.e., P/T nets where ${}^{\circledcirc}t = \emptyset$ for all t, the rules of the corresponding grammar never delete nodes. Hence, the SPO and the DPO approaches are interchangeable. In particular, ordinary P/T net transitions t, such that $\underline{t} = {}^{\circledcirc}t = \emptyset$, are represented by rules with an interface containing only nodes (see the rule corresponding to t_2 in Fig. 6).

Elementary nets. As shown in Table 2, ENSs are encoded as STSs. As an example, let us consider again the net N in Fig. 6, which can be interpreted as an ENS interpreting, correspondingly, the grammar on the right as an STS.

Observe that, even though there is a match of the rule t_1 in the start graph G_s, i.e., the left-hand side of the rule is a subgraph of G_s, the rule cannot be applied, because there is a contact situation. More precisely, referring to Fig. 7, condition (iv) of Definition 7 (namely, $D \cap R = K$) is not satisfied, as the intersection between the right-hand side of t_1 and the context graph D contains the edge connected to s_4 which is not in K.

If we interpret N as a CPR net and correspondingly the grammar as an STS$_m$, then the diagram in Fig. 7 is a legal derivation: in fact conditions $(i - iii)$ of Definition 8 are satisfied, while condition (iv) is not required anymore.



t_1
s_0 s_2 s_3 s_4 s
s_0 s_2 s_3 s_4
s_0 s_2 s_3 s_4 s
s_0 s_1 s_2 s_3 s_4 s
s_0 s_1 s_2 s_3 s_4
s_0 s_1 s_2 s_3 s_4 s

Fig. 7. A STS$_m$ derivation which is not a legal STS one.

5 Concluding remarks and further works

In this paper we discussed the encoding of different Petri net models into Graph Transformation Systems. Our aim was of a methodological nature, and its accomplishments are summarized by the taxonomy proposed in Tables 1 and 2. Intuitively, the results can be synthesized by the slogan "encode the net once", that is, a Petri net is always encoded essentially in the same way, while different net models correspond to alternative approaches to graph transformation.

A relevant issue, which has been neglected in the present paper, concerns the study of concurrency in Petri nets and in their graph grammar counterparts. Admittedly, there is a shortcoming as far as inhibitor nets are considered (already noted in [16]): If two transitions are inhibited by the same place s, their encodings as DPO rules cannot be executed in parallel, since both rules delete and produce again the node corresponding to s. For instance, in the inhibitor net in Fig. 8,

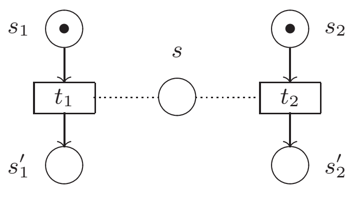


Fig. 8. An inhibitor net N_I: Transitions can fire concurrently. In $\mathcal{G}(N_I)$ they cannot.

the two transitions t_1 and t_2 can fire concurrently. However, in the corresponding DPO grammar the rules associated to t_1 and t_2 delete and generate again the same node s and thus they are forced to be executed sequentially. In general terms, we would like to see how to perform a technology transfer between the less-explored models of nets and GTSs, in order to address the issue of concurrent computations in these yet not fully developed formalisms.

References

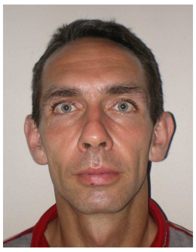
1. Kreowski, H.J.: A comparison between Petri nets and graph grammars. In Noltemeier, H., ed.: Proceedings of the Workshop on Graphtheoretic Concepts in Computer Science. Volume 100 of LNCS, Springer Verlag (1981) 306–317
2. Corradini, A.: Concurrent graph and term graph rewriting. In Montanari, U., Sassone, V., eds.: Proceedings of CONCUR'96. Volume 1119 of LNCS, Springer Verlag (1996) 438–464
3. Bernardinello, L., de Cindio, F.: A survey of basic net models and modular net classes. In Rozenberg, G., ed.: Advances in Petri Nets: The DEMON Project. Volume 609 of LNCS, Springer Verlag (1992) 304–351
4. Rozenberg, G., Engelfriet, J.: Elementary Net Systems. In Reisig, W., Rozenberg, G., eds.: Lectures on Petri Nets I: Basic Models. Volume 1491 of LNCS, Springer Verlag (1996) 12–121
5. Bonchi, F., Brogi, A., Corfini, S., Gadducci, F.: On the use of behavioural equivalences for web services' development. Fundamenta Informaticae **89** (2008) 479–510
6. Christensen, S., Hansen, N.D.: Coloured Petri nets extended with place capacities, test arcs and inhibitor arcs. In Ajmone-Marsan, M., ed.: Proceedings of ICAPTN'93. Volume 691 of LNCS, Springer Verlag (1993) 186–205
7. Montanari, U., Rossi, F.: Contextual nets. Acta Informatica **32** (1995) 545–596
8. Janicki, R., Koutny, M.: Semantics of inhibitor nets. Information and Computation **123** (1995) 1–16
9. Vogler, W.: Efficiency of asynchronous systems and read arcs in Petri nets. In: Proceedings of ICALP'97. Volume 1256 of LNCS, Springer Verlag (1997) 538–548
10. Agerwala, T., Flynn, M.: Comments on capabilities, limitations and "correctness" of Petri nets. Computer Architecture News **4** (1973) 81–86
11. Araki, T., Kasami, T.: Some decision problems related to the reachability problem for Petri nets. Theoretical Computer Science **3** (1977) 85–104
12. Habel, A., Plump, D.: Computational completeness of programming languages based on graph transformation. In Honsell, F., Miculan, M., eds.: Proceedings of FoSSaCS'01. Volume 2030 of LNCS, Springer Verlag (2001) 230–245
13. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109** (1993) 181–224
14. Ehrig, H., Pfender, M., Schneider, H.: Graph-grammars: an algebraic approach. In Book, R., ed.: Switching and Automata Theory, IEEE Computer Society Press (1973) 167–180
15. Corradini, A., Hermann, F., Sobociński, P.: Subobject transformation systems. Applied Categorical Structures **16** (2008) 389–419
16. Baldan, P., Corradini, A., Montanari, U.: Relating SPO and DPO graph rewriting with Petri nets having read, inhibitor and reset arcs. In Ehrig, H., Padberg, J., Rozenberg, G., eds.: Proceedings of PNGT'04. Volume 127(2) of ENTCS, Elsevier (2005) 5–28

17. Ehrig, H., Heckel, R., Korff, M., Löwe, M., Ribeiro, L., Wagner, A., Corradini, A.: Algebraic Approaches to Graph Transformation II: Single Pushout Approach and comparison with Double Pushout Approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
18. Corradini, A., Montanari, U., Rossi, F., Ehrig, H., Heckel, R., Löwe, M.: Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In Rozenberg, G., ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
19. Corradini, A., Montanari, U., Rossi, F.: Graph processes. Fundamenta Informaticae **26** (1996) 241–265
20. Löwe, M., Korff, M., Wagner, A.: An Algebraic Framework for the Transformation of Attributed Graphs. In Sleep, M., Plasmeijer, M., van Eekelen, M., eds.: Term Graph Rewriting: Theory and Practice. Wiley, London (1993) 185–199
21. Lack, S., Sobociński, P.: Adhesive and quasiadhesive categories. Theoretical Informatics and Applications **39** (2005) 511–546
22. Corradini, A., Heindel, T., Hermann, F., König, B.: Sesqui-pushout rewriting. In Corradini, A., Ehrig, H., Montanari, U., Ribeiro, L., Rozenberg, G., eds.: Proceedings of ICGT 2006. Volume 4187 of LNCS, Springer Verlag (2006) 30–45
23. Peterson, J.: Petri Net Theory and the Modelling of Systems. Prentice-Hall (1981)
24. Lakos, C., Christensen, S.: A general systematic approach to arc extensions for Coloured Petri Nets. In Valette, R., ed.: Proceedings of ICAPTN'94. Volume 815 of LNCS, Springer Verlag (1994) 338–357

Prof. Dr. Paolo Baldan

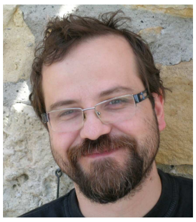
Dipartimento di Matematica Pura e Applicata
Università di Padova
I-35121 Padova (Italy)
baldan@math.unipd.it
http://www.math.unipd.it/~baldan

Paolo Baldan met Hans-Jörg Kreowski for the first time when he was a fresh doctoral student joining the GetGraTS project (General Theory of Graph Transformation Systems) in 1997. Since then, Hans-Jörg's work on concurrency in graph transformation and on the relation between Petri nets and graph rewriting has been a reference for Paolo's research on this subject.



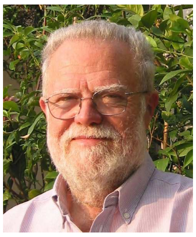
Prof. Dr. Andrea Corradini
Dipartimento di Informatica
Università di Pisa
I-56127 Pisa (Italy)
andrea@di.unipi.it
http://www.di.unipi.it/~andrea

Andrea Corradini met Hans-Jörg Kreowski for the first time in Bremen, at the 4th International Workshop on Graph-Grammars and Their Application to Computer Science (1990). Along the years, they worked as partners in the European projects CompuGraph I and II, GetGraTS, and AppliGraph, but also in the IFIP Working Group 1.3. So they co-chaired the first International Conference on Graph Transformation in Barcelona (2002). Andrea has worked on several research topics on which Hans-Jörg had worked before him, notably on notions of equivalence of graph derivations, and encodings of Petri Nets as graph transformation systems.



Prof. Dr. Fabio Gadducci
Dipartimento di Informatica
Università di Pisa
I-56127 Pisa (Italy)
gadducci@di.unipi.it
http://www.di.unipi.it/~gadducci

Fabio Gadducci met Hans-Jörg Kreowski for the first time in Volterra, at the 1995 Joint CompuGraph/SemaGraph Workshop. The two then met at many further occasions, while Fabio held a grant of the GetGraTS project, dealing with issues related to concurrency for graph transformation. Fabio often benefitted from Hans-Jörg's earlier work, and also from his advice.



Prof. Dr. Ugo Montanari
Dipartimento di Informatica
Università di Pisa
I-56127 Pisa (Italy)
ugo@di.unipi.it
http://www.di.unipi.it/~ugo

Ugo Montanari knows Hans-Jörg Kreowski since his first encounter with the double-pushout graph transformation community, on the 2nd International Workshop on Graph-Grammars and Their Application to Computer Science (1982). Hans-Jörg's work on concurrency has been the basis of lots of contributions by his colleagues in Pisa and himself.

Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework

Enrico Biermann, Claudia Ermel, and Gabriele Taentzer

Abstract. Model transformation is one of the key concepts in model-driven software development. An increasingly popular technology to define modeling languages is provided by the Eclipse Modeling Framework (EMF). Several EMF model transformation approaches have been developed, focusing on different transformation aspects. This paper proposes parallel graph transformation introduced by Ehrig and Kreowski to be a suitable framework for modeling EMF model transformations with multi-object structures. Multi-object structures at transformation rule level provide a flexible way to describe the transformation of structures with a flexible number of recurring structures, dependent on concrete model instances. Parallel graph transformation means massively parallelizing the application of model transformation rules synchronized at a kernel rule. We apply our extended EMF model transformation technique to model the simulation of statecharts with AND-states.

1 Introduction

Model-driven software development is considered as a promising paradigm in software engineering. Models are ideal means for abstraction and enable developers to master the increasing complexity of software systems. Since models are central artifacts in model-driven software development, the quality of generated software is directly dependent on the quality of models. Modifying models, i.e. for behavior simulation or for performing model refactoring [1]) is an important part of model development.

The Eclipse Modelling Framework (EMF) [2] has evolved to one of the standard technologies to define modeling languages. EMF provides a modelling and code generation framework for Eclipse applications based on structured data models. The modelling approach is similar to that of MOF, actually EMF supports Essential MOF (EMOF) as part of the OMG MOF 2.0 specification [3].

EMF models can be manipulated by several approaches to rule-based model transformations. A transformation framework for EMF models which follows the concepts of algebraic graph transformation [4] as far as possible, is presented in [5, 6]. Although graph transformation is an expressive, graphical and formal means to describe computations on graphs, it has limitations. For example, when describing the operational semantics of behavioral models, one often has the problem of modeling an arbitrary number of parallel actions at different places in the same model. A simple example are transformations of object structures of the same class occurring multiple times which all have the same properties (e.g. being contained in the same container, or referencing the same objects). We call such object structures *multi-object structures* in this paper. One way to transform multi-object structures is the sequential application of rules such that we have to explicitly encode an iteration over all the actions to be performed. Usually, this is not the most natural nor efficient way to express the semantics. Thus, it is necessary to have a more powerful means to express parallel actions.

As main contribution of this paper, we propose the use of *amalgamated graph transformation* concepts, based on parallel graph transformation, originally proposed by Ehrig and Kreowski in [7] and extended to synchronized, overlapping rules in [8], to define EMF model transformations with multi-object structures. The essence of amalgamated graph transformation is that (possibly infinite) sets of rules which have a certain regularity, so-called *rule schemes*, can be described by a finite set of *multi-rules* modeling the elementary actions. For the description of such rule schemes the concept of amalgamating rules at *kernel rules* [9] is used in this paper to describe the application of multi-rules in an unknown context. The synchronization of rules along kernel rules forces a transformation step to be maximally parallel in the following sense: an amalgamated rule, induced by a scheme, is constructed by a number of multi-rules being synchronized at the kernel rule. The number of multi-rules is determined by the number of different matches found such that they overlap in the match of the kernel rule. Hence, transforming multi-object structures can be described in a general way though the number of actually occurring objects in the instance model is variable.

In order to respect the special restrictions of EMF models (imposed by the containment hierarchy), we lift the concept of amalgamated graph transformation to amalgamated EMF transformation by showing that the conditions from our previous paper [5] are sufficient to guarantee the consistency of amalgamated EMF model transformations.

We show the usefulness of amalgamated EMF model transformation by simulating the behavior of statecharts with AND-states which may have an arbitrary number of orthogonal components (called *regions* in UML state machines). For example, when the system enters an AND-state, it actually goes to the initial simple state in each region in parallel.

The paper is organized as follows. In Section 2, we introduce EMF models as typed, attributed graphs and present our running example, an EMF model for a simplified variant of statecharts with AND-states. Section 3 reviews the concepts of parallel graph transformation and lifts them to EMF transformations with multi-object structures. This section contains our main result on consistency of amalgamated EMF model transformations. Using EMF transformations with multi-object structures, we model a general simulator for statecharts with AND-states. Section 4 presents related research, and Section 5 ends with the conclusions and future work.

2 EMF Models as Typed, Attributed Graphs with Containment

The Eclipse Modeling Framework (EMF) [2] has evolved to one of the standard technologies to define modeling languages. EMF provides a modeling and code generation framework for Eclipse applications based on structured data models. The modeling approach is similar to that of MOF, actually EMF supports Essential MOF (EMOF) as part of the OMG MOF 2.0 specification. Containment relations, i.e. aggregations, define an ownership relation between objects. Thereby, they induce a tree structure in model instantiations.

In [5], we consider EMF instance models[1] as typed graphs with special containment edges. Typing is expressed by a type graph. It has some similarities to a meta-model, but does not contain multiplicities and other constraints. For simplicity, we consider type graphs without inheritance in this paper. For a complete definition of EMF model transformation based on type graphs with inheritance, see [5].

Since the containment concept plays a special role in EMF models, we distinguish a special kind of edge types defining containments in the type graph.

Example 1 (EMF Model for statecharts with AND-States). Fig. 1 shows the EMF model for statecharts with AND-states, where an arbitrary number of states may be grouped in orthogonal regions of the same AND-state.

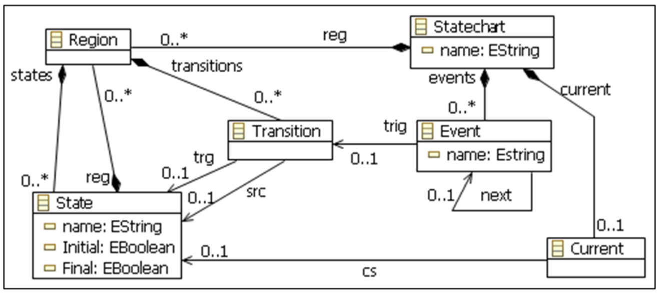


Fig. 1. EMF model for statecharts with AND-states

A *State* may contains *Regions*, each of them containing *States* again. We attribute *States* by Boolean flags denoting whether they are initial or final states. *States* are connected by *Transitions* which are triggered by *Events*. For simulation, a *Current* object is needed which is linked to the currently active *States*. The *Current* object receives an *Event*, the first element of a queue (*Events* linked by *next* links). The type graph with containment corresponding to the EMF model

[1] Note that the EMF community uses the terms "EMF model" for meta-model and "EMF instance model" for a model conforming to a meta-model.

in Fig. 1 looks like the EMF model but has no multiplicities. We have six containment edge types (three of them have type *Statechart* as source, type *states* starts from type *Region* and type *reg* starts from type *State*). Types *states* and *reg* could lead to cycles in EMF instance models (corresponding to graphs typed over the type graph), because there could be theoretically a *Region r* which contains a *State* which transitively contains *Region r* again. Hence, we call such containment edge types *cycle-capable.*

In *consistent* EMF instance graphs, each object node has at most one container and no containment cycles do occur. Graphs fulfilling these requirements are called *graphs with containment.* Although EMF instance models do not need to be rooted in general, this property is important for storing them, or more general, to define the model's extent.

Definition 2 (Graph with containment (C-graph)). *A graph with containment, short C-graph, is a graph $G = (G_N, G_E, s_G, t_G)$ with a distinguished set of containment edges $G_C \subseteq G_E$. The containment edges induce the following binary relation contains$_G$ (the transitive closure of G_C):*

- $contains_G = \{(x, y) \in G_N \times G_N \mid \exists e \in G_C : (s_G(e) = x \wedge t_G(e) = y) \} \cup \{(x, y) \in G_N \times G_N \mid \exists z \in G_N : (x \; contains_G \; z \wedge z \; contains_G \; y)\}$

All containment edges must fulfill the following properties (containment constraints):

- $e_1, e_2 \in G_C : t_G(e_1) = t_G(e_2) \;\Rightarrow\; e_1 = e_2$ *(at most one container).*
- $(x, x) \notin contains_G$ *for all* $x \in G_N$ *(no containment cycles).*

If G is typed over a type graph TG, there is a typing morphism $type\colon G \to TG$ which is consistent with containment, i.e. $\forall e \in G_C : type_{G_E}(e) \in TG_C$.

Please note that a type graph *TG* is no C-graph in general (see e.g. our type graph for statecharts in Fig. 1, which has a containment cycle).

Definition 3 (Rooted graph). *A C-graph G is called* rooted*, if there is a node $r \in G_N$, called* root node*, such that $\forall x \in G_N$ with $x \neq r : r \; contains_G \; x$.*

Example 4 (Consistent EMF instance graph). Fig. 2 shows a statechart with an AND-state. We model an ATM (automated teller machine) where the user can insert a bank card and, after the input of the correct pin, can draw a specified amount of cash from her or his bank account. The *display* region of the AND-state shows what is being displayed on screen, and, simultaneously, the *card-slot* component models whether the card slot is holding a bank card or not. The *enter* event triggers the transition before the AND-state to enter the AND-state. The *card-sensed* event happens if the sensor has sensed a user's bank card being put into the card slot. This event triggers two transitions in parallel. The next events (*pin-input, pin-ok* and *amount-input*) are local to the *display* region. The *end* event again triggers two transitions if the current state is any but the *welcome*

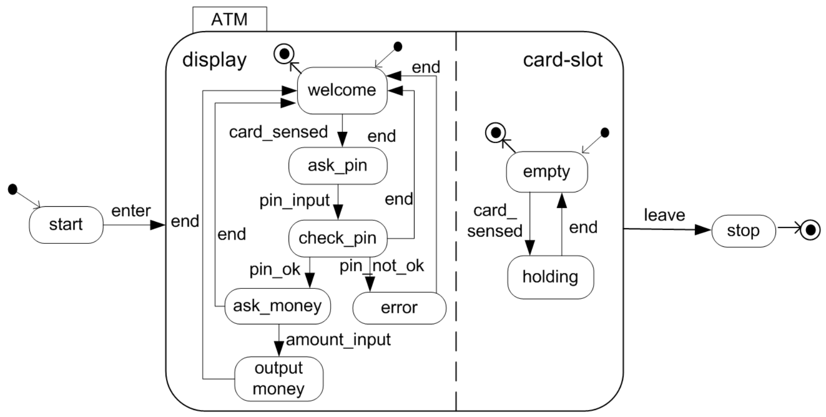


Fig. 2. EMF instance graph: a statechart modelling an ATM

state for the display region and *holding* for the card-slot region. Then, the final states are reached and the AND-state can be left if the *leave* event happens.

Fig. 3 shows the abstract syntax of the EMF instance graph corresponding to the ATM statechart in Fig. 2. This instance graph is typed over the type graph in Fig. 1. The initial state where we want to start the simulation is the *start* state before the AND-state *ATM* is entered. The *Current* object points to the *start* state and is linked to an initial event queue consisting so far of the single event *enter* (the event needed to enter the AND-state) followed by the special event denoting the queue end. During the simulation, events may be added to the event queue such that the queue holds the events that should be processed during the simulation. For better readability, we write names which are not empty in quotation marks and put the name of a boolean attribute type (*Initial* or *Final*) if its value is true. Furthermore, we omitted the containment edges in Fig. 3 from the *Statechart* object named *ATM-SC* to all *Current* and *Event* objects, and from the *Region* objects to the corresponding *Transition* objects.

The EMF instance graph in Fig. 3 is a C-graph since each object is contained in at most one container and there are no containment cycles. The C-graph is rooted, as the root node is the *Statechart* object named *ATM-SC* which contains all objects transitively.

3 EMF Model Transformations with Multi-Object Structures

EMF models can be manipulated by several approaches to rule-based model transformations. A transformation framework for EMF models which follows the concepts of algebraic graph transformation [4] as far as possible, is presented in [6]. But EMF model transformations do not always behave like algebraic graph transformation. The main reason is the difficulty to always satisfy the containment constraints of EMF. Hence, in our previous paper [5], we identify a kind

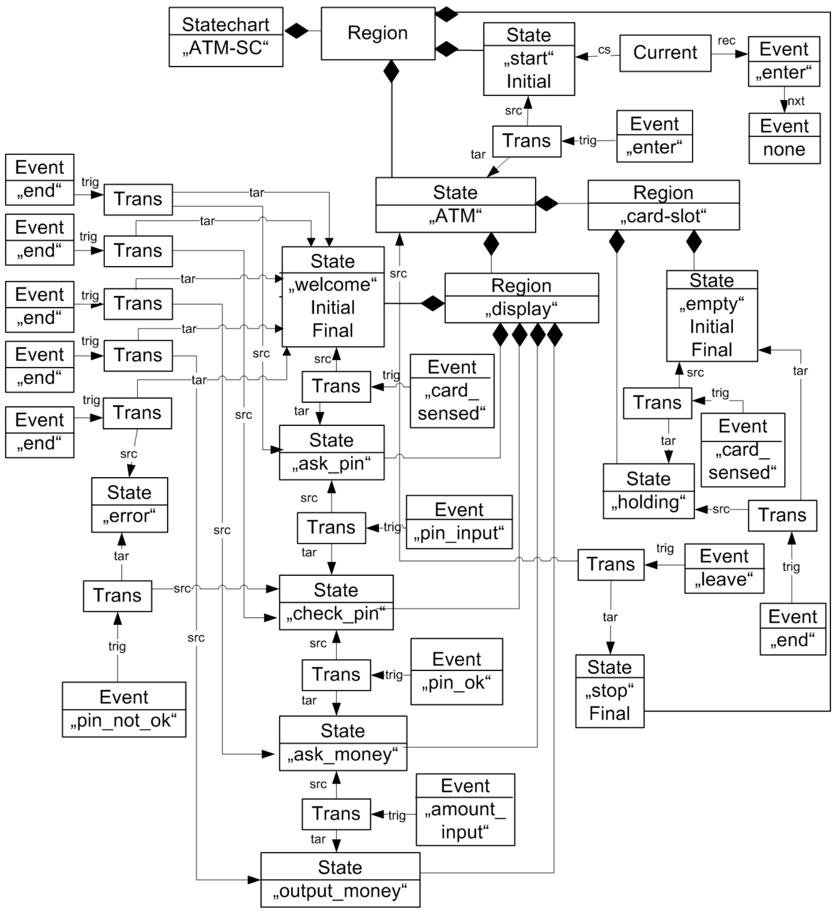


Fig. 3. Abstract Syntax of the ATM statechart in Fig. 2 with *Current* pointer

of model transformation rules which lead to consistent EMF model graphs (i.e. fulfilling the containment constraints), if applied as normal graph transformation rules to consistent EMF model graphs. Thus, we identify a kind of EMF model transformations which behave like algebraic graph transformations. The advantage of this approach is that we provide a basis to apply the rich theory of algebraic graph transformation [4, 11–13] to EMF model transformations.

In Section 3.1, we shortly review the basic notions from [5]. We then introduce amalgamated EMF model transformation, i.e. EMF model transformation with multi-object structures, based on parallel graph transformation concepts in Section 3.2 and expand the capability of consistent EMF transformations by showing that the application of an amalgamated EMF model transformation rule to a consistent EMF model graph results in a consistent transformed EMF model graph again.

3.1 Consistent EMF Model Transformation Based on Graph Transformation Concepts

In order to precisely define consistent graph rules, we have to define relations between typed C-graphs, so-called C-graph morphisms. They define mappings of nodes and edges respectively, such that they are compatible with typing, source and target functions (like typed graph morphisms) and especially preserve containment edge types.

Definition 5 (C-graph morphism). *Given two C-graphs G, H, a pair of functions (f_N, f_E) with $f_N : G_N \to H_N$ and $f_E : G_E \to H_E$ forms a valid C-graph morphism $f : G \to H$, if it has the following properties:*

- *$f_N \circ s_G(e) = s_H \circ f_E(e)$, $f_N \circ t_G(e) = t_H \circ f_E(e)$, and*
- *$\forall e \in G_C \;\Rightarrow\; f_E(e) \in H_C$ (containment edges are preserved).*

If G and H are typed over TG, f must be type compatible, i.e. $type_G = type_H \circ f$. If f_N and f_E are inclusions, then G is called a subgraph of H, denoted by $G \subseteq H$.

Definition 6 (Graph rule). *A graph rule typed over a type graph TG is given by $r = (L \supseteq K \subseteq R,\ type,\ NAC)$, where L, K and R are C-graphs, type is a triple of typing morphisms $type = (type_L \colon L \to TG,\ type_K \colon K \to TG,\ type_R \colon R \to TG)$, and NAC is a set of pairs $NAC_i = (N_i, type_{N_i}), i \in \mathbb{N}$ with $L \subseteq N_i$, and $type_{N_i} \colon N_i \to TG$ a typing morphism, such that $type_L \supseteq type_K \subseteq type_R$.*

As a drawing convention, we omit K. All objects with equal numbers in L and R are also in K and are preserved when the rule is applied. A rule p can contain one or more negative application conditions (NACs) denoting situations which must not exist for the rule to be applicable. Formally this is expressed by attributed graphs NAC_i and morphisms $n_i : NAC_i \leftarrow L$. A rule is *applicable* to a graph G at a match $m : L \to G$ if there is no injective C-graph morphism $n'_i : NAC_i \to G$ such that $m = n'_i \circ n_i$ for all $i \in I$. The application of rule r to graph G leads to the derivation of a graph H. Formally, a *derivation* $G \overset{r}{\Longrightarrow} H$ is a DPO construction in the category of typed attributed graphs and graph morphisms.

Example 7 (Graph rule). Rule *addEvent(e)*, shown in Fig. 4, allows to add a new event of name e into the event queue. In this way, the events that should be processed during a simulation run, can be defined in the beginning of the simulation. Moreover, events also can be inserted while a simulation is running.

Now we define a special kind of graph transformation which formalizes a form of EMF model transformation leading always to EMF models consistent with typing and containment constraints. For that purpose, the form of allowed transformation rules has to be restricted. Consistent transformation rules allow the following kinds of actions which change containments:

1. Delete an object node with its containment relation.

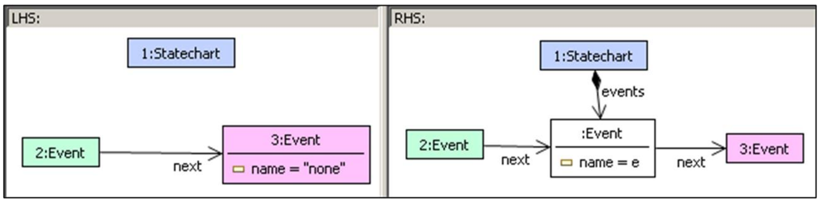


Fig. 4. Rule *addEvent(e)* to insert Event e into the Event Queue

2. Create a new object node and connect it immediately to its container.
3. Delete a containment edge together with its contained object node or change the container of a preserved object node.
4. Create a containment edge with the contained object node or change the container of an existing object node.
5. For an object node contained via a cycle-capable containment edge, change its container only, if the old and the new container of the object node were already transitively related by containment.

In the following definition, we formalize all actions that preserve consistent containment relations which have been described above.

Definition 8 (Consistent graph rule). *Let* $L'_C := L_C - K_C$, $R'_C := R_C - K_C$, $L'_N := L_N - K_N$ *and* $R'_N := R_N - K_N$. *A graph rule* $p = (L \supseteq K \subseteq R, \mathit{type}, \mathit{NAC})$ *is* consistent *wrt. containment if for each rule all of the following constraints are satisfied:*

1. *(node deletion)* $\forall n \in L'_N : \exists e \in L'_C$ *with* $t_{L'_C}(e) = n$,
2. *(node creation)* $\forall n \in R'_N : \exists e \in R'_C$ *with* $t_R(e) = n$,
3. *(containment edge deletion)* $\forall e \in L'_C$ *with* $t_L(e) = n$:
   $n \in L'_N \quad \vee \quad (n \in K_N \wedge \exists e' \in R'_C \text{ with } t_R(e') = n)$
4. *(containment edge creation)* $\forall e \in R'_C$ *with* $t_R(e) = n$:
   $n \in R'_N \quad \vee \quad (n \in K_N \wedge \exists e' \in L'_C \text{ with } t_L(e') = n)$
5. *(creation of cycle-capable containment edges)*
   $\forall e \in R'_C \text{ with } s_R(e) = n \wedge t_R(e) = m : \exists e' \in L'_C \text{ with } s_L(e') = o \wedge t_L(e') = m :$
   $((o, n) \in contains_L \wedge (m, n) \notin contains_L) \vee (n, o) \in contains_L$

Note that for item 5 (creation of cycle-capable containment edges), it is sufficient to inspect the containment in the rule's left-hand side. There cannot be a containment edge from the matched node m to n in G, because then n would have two containers m and o, and hence G would not be a C-graph.

Example 9 (Consistent graph rules). Rule *addEvent(e)* from Example 7 is a consistent graph rule since for each created object its containment edge is created as well. Two further rules are depicted in Fig. 5 which process sequential transitions outside of AND-states. Rule *sequentialTransition* processes a transition in the current state which is triggered by the current event. This rule is consistent since the removed event node is deleted together with its containment edge. Rule *skipEvent* models the situation that no transition is triggered by the current event. In this case, the event is removed from the event queue. Again, the event is deleted together with its containment edge.

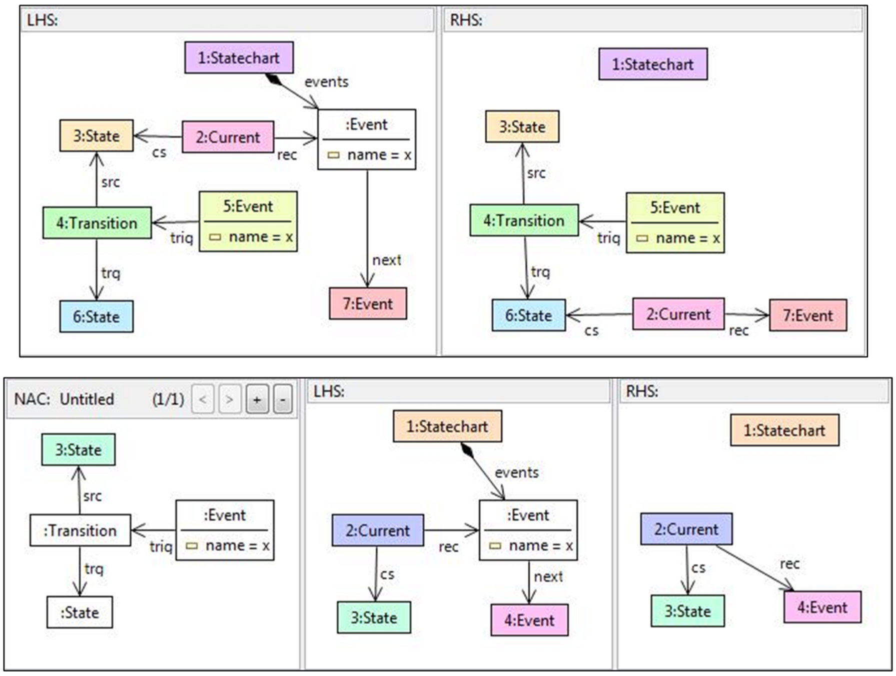


Fig. 5. Rules *sequentialTransition* and *skipEvent*

In our main theorems in [5], we show that the application of a consistent graph rule to a consistent (rooted) EMF instance graph always results again in a consistent (rooted) EMF instance graph.

Theorem 10 (Consistent graph transformation step). *Given a consistent graph rule* $p = (L \supseteq K \subseteq R, type, NAC)$ *and a match* $L \xrightarrow{m} G$ *to a C-graph* G *which is typed by* $type_G \colon G \to TG$ *and satisfies* NAC*. Then, the result graph* $(H, type_H)$ *of direct transformation* $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ *is a C-graph.*

Proof. See [5].

Theorem 11 (Rooted graph transformation step). *A consistent graph transformation step* $(G, type_G) \overset{p,m}{\Longrightarrow} (H, type_H)$ *leads to a* rooted *result graph* H *if graph* G *is rooted.*

Proof. See [5].

3.2 Consistent EMF Model Transformations with Multi-Object Structures

In this section, we lift the essential concepts of parallel graph transformation [8] to EMF model transformation and also lift the consistency result for EMF model transformations from Section 3.1 to transformations with multi-object structures which we also call amalgamated EMF transformations.

Using parallel graph transformation, a system state modeled by a graph can be changed by several actions executed in parallel. Since graph transformation is rule-based without restrictive execution prescription, parallel graph transformation offers the possibility for massively parallel execution. The synchronization of parallel rule applications is described by common subrules, called *kernel rules*.

The simplest type of parallel actions is that of *independent actions*. If they operate on different objects they can clearly be executed in parallel. If they overlap just in reading actions on common objects, the situation does not change essentially. In graph transformation, this is reflected by a *parallel rule* which is a disjoint union of rules. The overlapping part, i.e. the objects which occur in the match of more than one rule, is handled implicitly by the match of the parallel rule. As the application of a parallel rule can model the parallel execution of independent actions only, it is equivalent to the application of the original rules in either order [7].

If actions are not independent of each other, they can still be applied in parallel if they can be synchronized by subactions. If two actions contain the deletion or the creation of the same node, this operation can be encapsulated in a separate action which is a common subaction of the original ones. A common subaction is modelled by the application of a *kernel rule* of all additional actions (modelled by *multi-rules*). The application of rules synchronized by kernel rules is then performed by gluing multi-rule instances at their kernel rules which leads to the corresponding *amalgamated rule*. The application of an amalgamated rule is called *amalgamated graph transformation.*

Formally, the synchronization possibilities of actions (multi-rule applications) are defined by an interaction scheme. For consistent amalgamated EMF transformations (also called EMF model transformations with multi-object structures), we need consistent interaction schemes where all rules are consistent.

Definition 12 (Consistent Interaction Scheme).
An interaction scheme $I = (r_K, M, ke)$ *consists of a kernel rule* r_K*, a set* $M = \{r_i | 1 \leq i \leq n\}$ *of rules called multi-rules, and a set* ke *of kernel rule embeddings* $ke_i : r_K \rightarrow r_i$ *into the multi-rules (i.e.* r_K *is subrule of all multi-rules). I is consistent, if all rules are consistent.*

Example 13. An example of the construction of an amalgamated EMF transformation rule from an interaction scheme is given in Fig. 6.

The common sub-action (adding a loop to a object 1) is modeled by kernel rule r_K. We have only one multi-rule r_1 modeling that at each possible match (the blue) object 2 shall be deleted together with its containment edge, and a new (red) object shall be inserted such that it is contained in object 1. Both the

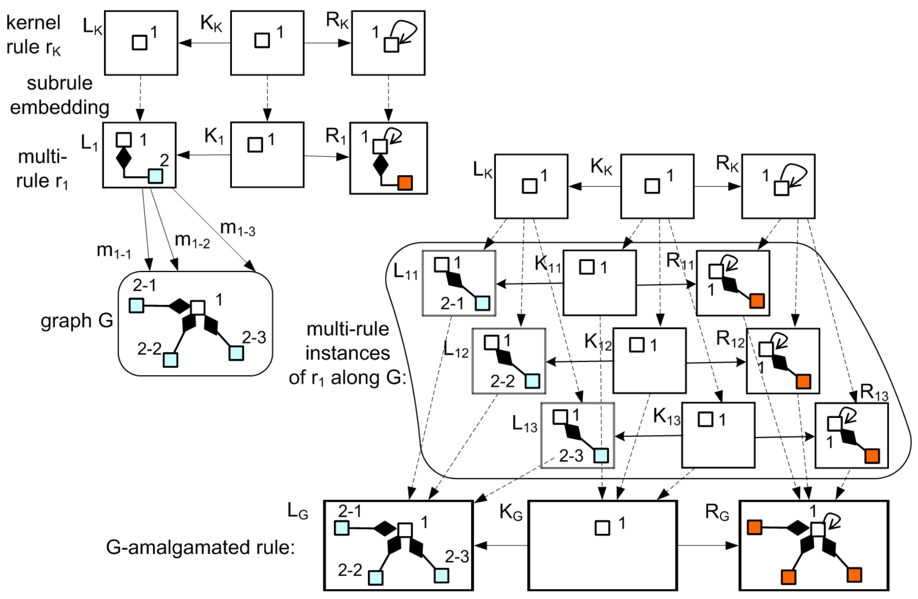


Fig. 6. Construction of an amalgamated graph rule

kernel-rule and the multi-rule are consistent, and we have a subrule embedding from the kernel rule to the multi-rule given by the three C-graph morphisms $L_K \to L_1, K_K \to K_1$ and $R_K \to R_1$. Given graph G, we have obviously three different matches from the multi-rule r_1 to G which overlap in the match from the kernel rule to G. Hence, we have three multi-rule instances, each of them with a different match to G. Gluing the multi-rule instances at their common kernel rule, we get the amalgamated rule with respect to G, shown at the bottom of Fig. 6. The amalgamated rule contains the common action and, additionally, all actions from the multi-rules that do not overlap. Dashed arrows in Fig. 6 indicate rule embedding morphisms, embedding the kernel rule into the corresponding instances of the multi-rules, and the multi-rules into the amalgamated rule.

In addition to specifying how multi-rules should be synchronized, we must decide where and how often a set of multi-rules should be applied. The basic way to synchronize complex parallel operations is to require that a rule should be applied at *all different matches* it has in a given graph (expressing massively parallel execution). In this paper, we restrict the *covering of* G (the image of all different matches from instances of multi-rules in G) to all different matches of multi-rules that overlap in the match of their common kernel rule and do not overlap anywhere else. For more complex covering constructions see [8].

Definition 14. (Amalgamated EMF model transformation rule)
Given a consistent interaction scheme $I = (r_K, \{r_i | 1 \leq i \leq n\}, ke)$ *with* $(L_i - L_K) \cap (L_j - L_K) = \emptyset$ *and a match* m_K *for the kernel rule* r_K*. An* amalgamated

EMF model transformation rule $r_A = (L_A \leftarrow K_A \rightarrow R_A)$ *is an EMF model transformation rule where as many copies* $r_{i_{j_i}}$ *of multi-rules* r_i *are joined as there are different matches* $m_{i_{j_i}} : L_i \rightarrow G$ *such that the copies* $r_{i_{j_i}}$ *of* r_i *in* r_A *all overlap at kernel rule* r_K *and the matches* $m_{i_{j_i}}$ *overlap at match* m_K *only.*

We speak of *amalgamated EMF model transformation* (or, alternatively, of *EMF model transformation with multi-object structures*) if the definition of the EMF model transformation is given by consistent interaction schemes. In this case, the application of an amalgamated EMF model transformation rule defines an EMF model transformation step. Note that a special interaction scheme consists of only one rule, i.e. a kernel rule, such that the interaction scheme is applied like a usual sequential rule.

In order to show that EMF instance graphs resulting from amalgamated EMF model transformation are consistent (Theorem 15), we construct the amalgamated rule from a given consistent interaction scheme and show that this amalgamated rule is a consistent graph rule. Afterwards, we can apply Theorem 10.

Theorem 15. *Given a consistent interaction scheme* $I = (r_K, \{r_i | 1 \leq i \leq n\}, ke)$ *and matches* m_k *and* m_i *to* G *for all* $1 \leq i \leq n$. *Then, if* G *is a C-graph, the resulting amalgamated EMF model transformation rule is consistent.*

Proof.
Case $n = 0$**:** No multi-rule is applied. The amalgamated rule r_A is equal to the kernel rule r_K, which is consistent by assumption (I is consistent).

Case $n = 1$**:** There is one application of a multi-rule. The amalgamated rule r_K is equal to this multi-rule, thus it is consistent by assumption.

Case $n > 1$**:** We have to show that the amalgamated rule r_A satisfies all five consistency constraints for EMF rules according to Def. 8:

1. (node deletion) To show: $\forall n \in L'_{A_N} : \exists e \in L'_{A_C}$ with $t_{L'_{A_C}}(e) = n$.
   W.l.o.g. $n \in L'_{i_N}$: Then, there is $e \in L'_{i_C}$ with $t_{L'_{i_C}}(e) = n$, since r_i is consistent.
2. (node creation) To show: $\forall n \in R'_{A_N} : \exists! e \in R'_{A_C}$ with $t_{R_A}(e) = n$.
   W.l.o.g. $n \in R'_{i_N}$: Then, there is a unique $e \in R'_{i_C}$ with $t_{R'_{i_C}}(e) = n$, since r_i is consistent. There cannot be another $e \in R'_{A_C}$ with $t_{R_A}(e) = n$, since the assumption allows an overlap of multi-rules in the kernel rule only. In this case, they would have to overlap in node n, too, which is not necessarily required here.
3. (containment edge deletion) To show: $\forall e \in L'_{A_C}$ with $t_{L_A}(e) = n$:
   $$n \in L'_{A_N} \quad \vee \quad (n \in K_{A_N} \wedge \exists e' \in R'_{A_C} \text{ with } t_{R_A}(e') = n).$$
   W.l.o.g. $e \in L'_{i_C}$ with $t_{L_i}(e) = n$. Then, $n \in L'_{i_N} \vee (n \in K_{i_N} \wedge \exists e' \in R'_{i_C}$ with $t_{R_i}(e') = n)$, since r_i is consistent.
4. (containment edge creation) To show: $\forall e \in R'_{A_C}$ with $t_{R_A}(e) = n$:
   $$n \in R'_{A_N} \quad \vee \quad (n \in K_{A_N} \wedge \exists e' \in L'_{A_C} \text{ with } t_{L_A}(e') = n)$$

W.l.o.g. $\forall e \in R'_{i_C}$ with $t_{R_i}(e) = n$. Then, $n \in R'_{i_N} \vee (n \in K_{i_N} \wedge \exists e' \in L'_{i_C}$ with $t_{L_i}(e') = n)$, since r_i is consistent.

5. (creation of cycle-capable containment edges)
To show: $\forall e \in R'_{A_{C_{Cycle}}}$ with $s_{R_A}(e) = n \wedge t_{R_A}(e) = m : \exists e' \in L'_{A_C}$ with $s_{L_A}(e') = o \wedge t_{L_A}(e') = m :$
$$((o, n) \in \mathit{contains}_{L_A} \wedge (m, n) \notin \mathit{contains}_{L_A}) \vee (n, o) \in \mathit{contains}_{L_A}.$$
W.l.o.g. $e \in R'_{i_{C_{Cycle}}}$ with $s_{R_i}(e) = n \wedge t_{R_i}(e) = m$. Then, there is $e' \in L'_{i_C}$ with $s_{L_i}(e') = o \wedge t_{L_i}(e') = m :$
$$((o, n) \in \mathit{contains}_{L_i} \wedge (m, n) \notin \mathit{contains}_{L_i}) \vee (n, o) \in \mathit{contains}_{L_i}.$$
In addition, we have to show that there is no $(m, n) \in \mathit{contains}_{L_j}$ for some $j \neq i$. Since r_i and r_j overlap in r_K only, $m, n \in L'_{K_N} \subseteq L'_{i_N}$ and $(m, n) \notin \mathit{contains}_{L_i} \implies (m, n) \notin \mathit{contains}_{L_j}$.

Corollary 16. *Given a consistent interaction scheme $I = (r_K, \{r_i | 1 \leq i \leq n\}, ke)$ and matches m_k and m_i to G for all $1 \leq i \leq n$. Then, if G is a C-graph, the result graph H after applying interaction scheme I to G is a C-graph as well.*

Proof. Due to Theorem 15, the amalgamated rule constructed from I is consistent. By Theorem 10, consistent rules preserve C-graphs. Hence, the result graph H is again a C-graph.

Corollary 17. *Given a consistent interaction scheme I like in Corollary 16. Then, if G is a* rooted *C-graph, the result graph H after applying the interaction scheme I to G is a* rooted *C-graph as well.*

Proof. Due to Theorem 15, the amalgamated rule constructed from I is consistent. By Theorems 10 and 11, we know that consistent rules preserve C-graphs and the rootedness of C-graphs. Hence, the result graph H is a rooted C-graph.

Example 18 (Simulator for statecharts with AND-States).
In our statecharts variant, every region belonging to an AND-state has exactly one initial state and at least one final state. The intended semantics for our statecharts requires that if an AND-state is reached, the active states become the initial ones of each region. A transition is processed if its pre-state is active and its triggering event is the same as the event which is received by the *Current* object (the first event in the queue). Afterwards, the state(s) following the transition become(s) active, the event of the processed transition is removed from the queue, and the previously active state(s) (the pre-state(s) of the transition) is/are not active anymore. More than one transition are processed simultaneously if they belong to different regions of the same AND-state, if their pre-states are all active and if they are all triggered by the same event which is received by the *Current* object. All regions belonging to the same AND-state must have reached a final state before the AND-state can be left and the transition from the AND-state to the next state can be processed. For our simulator we use the *Current* object not only as object which receives the next event (and is linked to the event queue) but also as pointer to the current active states.

Thus, our simulation rules model the relinking of the *Current* object to the next active states and the updating of the event queue.

Note that in the following screenshots of interaction schemes we use an integrated notation, where we define the kernel rule and one multi-rule within one rule picture. This is possible since each of our interaction schemes consists of a kernel rule and one multi-rule only. We distinguish objects belonging to the multi-rule by drawing them as multi-objects (with indicated multiple boxes instead of simple rectangles). The kernel rule consists of all simple objects which are not drawn as multiple boxes. All arcs adjacent to multi-objects belong to the multi-rule only, but not to the kernel rule. All multi-objects together with their adjacent arcs in one multi-rule form a multi-object structure.

The upper part of Fig. 7 shows the interaction scheme *enterRegion* which moves the *Current* pointer along a transition that connects a state to an AND-state. In this case, the *Current* pointer has not only to point to the AND-state afterwards but also to all initial states of all regions of the AND-state. Hence, the amalgamated rule consists of as many copies of the multi-rule as there are regions in the AND-state (provided that each component has exactly one initial state which has to be ensured by a suitable syntax grammar).

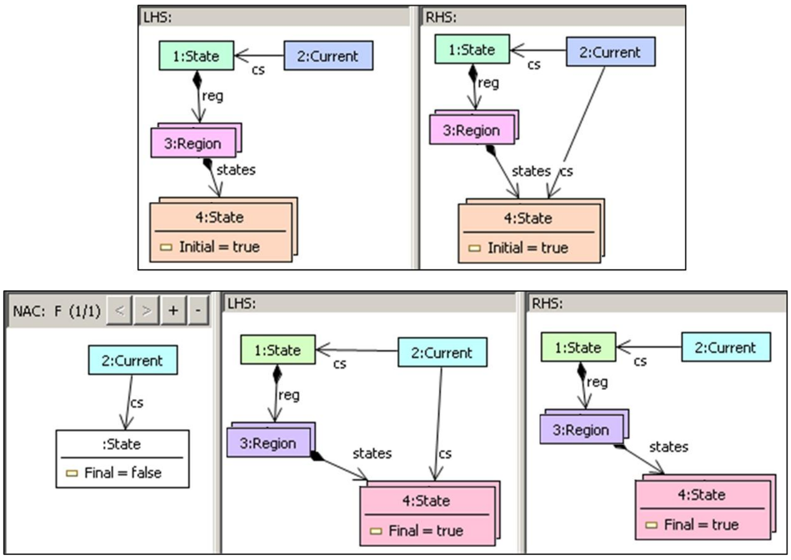


Fig. 7. Interaction Schemes *enterRegion* and *leaveRegion*

Vice versa, when an AND-state is left, the *Current* pointer has to be removed from all of its regions. This step is realized by the interaction scheme *leaveRegion* at the bottom of Fig. 7. The fact that the active states of all regions have to be *Final* is modelled by the NAC. The multi-rule models how all inner links from the *Current* pointer to the regions' final states are removed.

A simultaneous transition is modelled by interaction scheme *simultanTrans* in Fig. 8. Here, an arbitrary number of transitions in different regions of an AND-state are processed if triggered by the same event. In our ATM example this happens at different points of the simulation: When the AND-state is entered and the event *card-sensed* is happening, then the two first transitions of the two regions are processed simultaneously. Similarly, at any state of the display the user can abort the transaction: the *end* event triggers the return of the display region to state *welcome* and the return of the card-slot region to state *empty*.

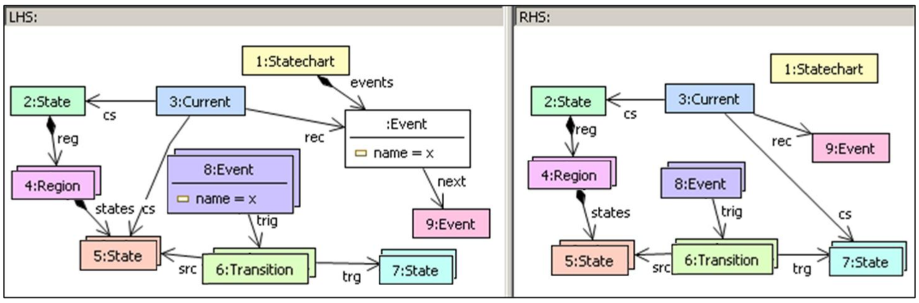


Fig. 8. Interaction Scheme *simultanTrans*

The *simultanTrans* interaction scheme is a good example for a concise way to model simultaneous transitions which are triggered by a single event. This would be quite difficult to model using simple rules. Note that this scheme is applicable also for sequential transition processing within an AND-state. Then there is only one copy of the multi-rule, similar to rule *sequentialTrans*. In the case that no transition leaving an active state is triggered by the current event, we have the situation that there is no copy of the multi-rule of *simultanTrans*, but the kernel rule can be applied anyway. This means that an event which does not trigger any transition inside of an AND-state simply is removed from the event queue. Again, this is similar to applying rule *skipEvent* with the difference that regions are used here.

4 Related Work

There are two tool-based approaches known to us which also realize parallel graph transformation: AToM^3 and GROOVE, where AToM^3 supports the explicit definition of interaction schemes in different rule editors [14] and GROOVE implements rule amalgamation based on nested graph predicates [15]. A related conceptual approach aiming at transforming collections of similar subgraphs is presented in [16]. The main conceptual difference is that we amalgamate rule instances whereas the authors of [16] replace all collection operators (multi-object structures) in a rule by the mapped number of collection match copies. Similarly, Hoffmann et al. define a cloning operator in [17] where cloned nodes correspond to multi-objects, but complete multi-object structures cannot be described. Moreover, the graph transformation tools PROGRES [18] and FuJaBA [19] feature so-called set nodes which are duplicated as often as necessary, but are not based on amalgamated graph transformation. None of the related approaches support the transformation of EMF models.

5 Conclusions and Future Work

This paper presented amalgamated EMF transformation as a valuable means for modelling and simulation. They extend the capabilities of EMF transformation based on simple graph transformation [5] by allowing parallel execution of synchronized EMF transformation rules. This is useful for specifying simulators for formalisms in which parallel actions happen. This is the case for a great number of formalisms, such as statecharts with AND states. It has been shown in the paper that amalgamated EMF transformation always leads to consistent EMF instance models which satisfy the containment constraints of EMF.

In the future, we plan to apply the approach to other kinds of EMF model transformations, such as model refactorings, where multi-object structures are found frequently.

Amalgamated EMF transformation are currently being implemented in the tool EMF TIGER [10, 6] (**T**ransformat**i**on **ge**ne**r**ation), a recently developed Eclipse plug-in supporting modeling and code generation for EMF model transformations, based on structured data models and graph transformation concepts. The goal of EMF Tiger is to provide the means to graphically define rule-based transformations on EMF models. Rule applications change an EMF model instance in-place, i.e. an EMF instance model is modified directly, without copying it before. Moreover, control of rule applications is supported by EMF TIGER, as well as pre-definition of (parts of) the match. EMF TIGER currently consists of a *graphical editor* for visually defining EMF model transformation rules, and a *compiler*, generating Java code from these transformation rules to be included into existing projects performing EMF model transformation. It also contains an *interpreter* which translates EMF transformation rules to AGG. This interpreter is useful for verification purposes.

References

1. Mens, T., Tourwé, T.: A survey of software refactoring. Transactions on Software Engineering **30**(2) (February 2004) 126–139
2. Eclipse Consortium: Eclipse Modeling Framework (EMF) – Version 2.4. (2008) `http://www.eclipse.org/emf`.
3. Object Management Group: Meta Object Facility (MOF) Core Specification Version 2.0. `http://www.omg.org/technology/documents/modeling_spec_catalog.htm\#MOF` (2008)
4. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs in Theor. Comp. Science. Springer (2006)

5. Biermann, E., Ermel, C., Taentzer, G.: Precise Semantics of EMF Model Transformations by Graph Transformation. In Proc. Conf. on Model Driven Engineering Languages and Systems (MoDELS'08). Vol. 5301 of LNCS., Springer (2008) 53–67
6. Biermann, E., Ehrig, K., Köhler, C., Kuhns, G., Taentzer, G., Weiss, E.: Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In Proc. Conf. on Model Driven Engineering Languages and Systems (MoDELS'06). Vol. 4199 of LNCS. Springer (2006) 425–439
7. Ehrig, H., Kreowski, H.J.: Parallel graph grammars. In Lindenmayer, A., Rozenberg, G., eds.: Automata, Languages, Development. North Holland (1976) 425–447
8. Taentzer, G.: Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems. PhD thesis, TU Berlin (1996)
9. Böhm, P., Fonio, H.R., Habel, A.: Amalgamation of graph transformations: a synchronization mechanism. Journal of Computer and System Science **34** (1987) 377–408
10. Tiger Project Team, Technische Universität Berlin: EMF Tiger (2009) `http://tfs.cs.tu-berlin.de/emftrans`.
11. Ehrig, H., Kreowski, H.J., Montanari, U., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Vol 3: Concurrency, Parallelism and Distribution. World Scientific (1999)
12. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools. World Scientific (1999)
13. Rozenberg, G.: Handbook of Graph Grammars and Computing by Graph Transformations, Vol. 1: Foundations. World Scientific (1997)
14. de Lara, J., Ermel, C., Taentzer, G., Ehrig, K.: Parallel Graph Transformation for Model Simulation applied to Timed Transition Petri Nets. In: Proc. Graph Transformation and Visual Modelling Techniques (GTVMT) 2004. (2004)
15. Rensink, A., Kuperus, J.H.: Repotting the geraniums: On nested graph transformation rules. In: Int. Workshop of Graph Transformation and Visual Modelling Techniques (GT-VMT'09). (2009)
16. Grønmo, R., Krogdahl, S., Møller-Pedersen, B.: A collection operator for graph transformation. In: Int. Conf. on Model Transformation (ICMT'09). (2009)
17. Hoffmann, B., Janssens, D., van Eetvelde, N.: Cloning and expanding graph transformation rules for refactoring. In: Int. Workshop on Graph and Model Transformation (GraMoT'05). Vol. 152 of ENTCS, Elsevier (2006) 53–67
18. Schürr, A., Winter, A., Zündorf, A.: The PROGRES-approach: Language and environment. In [12].
19. Fischer, T., Niere, J., Torunski, L., Zündorf, A.: Story diagrams: A new graph rewrite language based on the UML. In Proc. Workshop on Theory and Application of Graph Transformation. Vol. 1764 of LNCS, Springer (2000) 296–309

Enrico Biermann

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
D-10587 Berlin (Germany)
enrico@cs.tu-berlin.de

Dr. Claudia Ermel

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
D-10587 Berlin (Germany)
lieske@cs.tu-berlin.de
http://tfs.cs.tu-berlin.de/˜lieske

Hans-Jörg was the external examiner of Claudia Ermel's doctoral thesis.

Prof. Dr. Gabriele Taentzer

Fachbereich Mathematik und Informatik
Philipps-Universität Marburg
D-35032 Marburg (Germany)
taentzer@mathematik.uni-marburg.de
http://www.informatik.uni-marburg.de/˜taentzer

Hans-Jörg was the external examiner of Gabriele Taentzer's doctoral thesis at the Technical University of Berlin. Due to their common research interests in graph transformation and visual languages, they have several joint publications.

Rechnen, Denken, lebenslange Bildung

Wolfgang Coy

Am Ende von Wittgensteins *Tractatus logico-philosophicus*, nachdem er das menschliche Erkenntnisvermögen vom Reich der Objekte auf sprachliche Aussagen über die Welt und das Räsonnement über diese Aussagen zurückführt hat, steht unter 6.52 die gern überlesene Bemerkung >Wir fühlen, dass selbst, wenn alle *möglichen* wissenschaftlichen Fragen beantwortet sind, unsere Lebensprobleme noch gar nicht berührt sind. Freilich bleibt dann eben keine Frage mehr; und eben dies ist die Antwort.<

*

Alan Mathison Turing steht nicht nur als Cambridge-Absolvent der sprachlichen Wende Wittgensteins sehr nahe. Er nähert sich dem Denken von der Seite der Berechenbarkeit und verschiebt den *linguististic turn* zum Begriff der Berechenbarkeit hin. Rechnen wird bei Turing anders als bei anderen gleichzeitig formulierten formalen Kalkülen der Berechenbarkeit durch die Tätigkeit eines menschlichen Rechners mit Bleistift, Radiergummi und Karopapier beschrieben. Wir können uns ein Schulkind bei seinen Rechenaufgaben vorstellen. Zu jedem Zeitpunkt ist dieser menschliche Rechner in einem von einigen möglichen >Arbeitszuständen<. Betrachtet wird nur ein einziges Karokästchen, in dem ein wohlvertrautes Zeichen aus einem endlichen Vorrat steht. Im Kopf ist eine algorithmische Anweisung, eine von ein paar Anweisungen, was jetzt zu tun sei: Ob das Zeichen durch ein anderes ersetzt wird oder ob das rechts oder das links anschließende Kästchen zu bearbeiten sei, oder ob die Rechnung zu Ende gekommen ist. Eine schlichte Vorstellung, die gleichwohl alles >Rechnen< vollständig beschreiben soll. >It is my contention that these operations include all those which are used in the computation of a number.< Turing nennt dieses wohlerzogene rechnende Schulkind *paper machine*.

Turings Weltbild war vom allgemeinen Stand der Wissenschaften geprägt, soweit sie Mathematikern vertraut war, in den engen Ansätzen der formalen Logik, die sich zu dieser Zeit sehr stark auf das Hilbertsche Grundlagenforschungsprogramm konzentrierten, also die Forderung nach Vollständigkeit, Widerspruchsfreiheit und Entscheidbarkeit axiomatischer Kalküle. Sein Biograf Andrew Hodges hat dies sehr schön auf den Punkt gebracht, als er darauf hinwies, für den ganz jungen Alan sei >Natural Wonders every Child Should Know<, das wichtigste Buch gewesen, eine schlichte Popularisierung und Reduktion aller Naturwissenschaften auf die Mechanik.[1]

Vor diesem Hintergrund wird deutlich, dass Turing zwei Vorstellungen pflegt:

– Alles Denken ist formal und symbolisch als Schlussfolgern beschreibbar (was mit geeignetem Programm von seiner *paper machine* als Rechnung ausführbar wäre).

[1] Vgl. A. Hodges, Alan Turing – Enigma, a.a.O.

– Aus Gründen der endlichen molekularen Struktur der Gehirns lässt sich alles Schlußfolgern in finiten Kalkülen beschreiben. Bei Turing heißt es dazu knapp: >For the present I shall only say that the justification lies in the fact that the human memory is necessarily limited.<[2] Turings Überlegungen bleiben dabei etwas vage. Roger Penrose spitzt dies m.E. angemessen zu: >It seems likely that he [Turing] viewed physical action in general – which would include the action of a human brain – to be always reducible to some kind of Turing-machine action.<[3] Andrew Hodges nennt die die *Turing-These* (im Unterschied zur Church-Turing These).

Er folgt damit dem Wittgensteinschen Programm des Tractatus, ohne sich explizit darauf zu beziehen – und ohne die kritischen Zweifel, die Wittgenstein in den Jahrzehnten nach der Erstveröffentlichung quälten. Turing interessiert sich mehr für die Frage, wie weit seine *paper machines* gehen können. In seinem 1950 für ein philosophisch interessiertes Publikum geschriebenen Aufsatz >Computing machinery and intelligence< schlägt er einen Wettbewerb zwischen Mensch und Maschine vor, um die Leistungsfähigkeit >Intelligenter Programme< zu testen. Der Aufsatz endet mit: >We may hope that machines will eventually compete with men in all purely intellectual fields. But which are the best ones to start with? Even this is a difficult decision. Many people think that a very abstract activity, like the playing of chess would be best. It can also be maintained that it is best to provide the machine with the best sense organs that money can buy, and then teach it to understand and speak English. This process could follow the normal teaching of a child. Things would be pointed out and named, etc. Again I do not know what the right answer is, but I think both approaches should be tried.<

Mit ähnlicher Unbekümmertheit haben John McCarthy und Marvin Minsky diese Vorgehensweise fortgeführt, als sie 1956 auf der berühmten Dartmouth-Conference ein Forschungsprogramm zur künstlichen Intelligenz verkündeten, dessen philosophische Höhenflüge bis heute wenig ergiebig sind, wobei die vorherrschenden Denkfiguren sich von Dekade zu Dekade wandeln – vom Turingtest zum Schachspiel, vom Schachspiel zum Sprachverstehen, vom Sprachverstehen zur Robotik, von Robotern zur Schwarmintelligenz – ohne das sie der Turingschen Frage >Can a machine think?< wirklich näher gekommen sind.

*

Kant wären solch rein computationale Bestimmungen des Denkens wohl fremd geblieben. >So kann sich niemand bloß mit der Logik wagen, über Gegenstände zu urteilen.< heißt es in der Kritik der reinen Vernunft. Kant hat Denken nicht nur als Logik begriffen, sondern im Kontext von Erkenntnis, moralischer und ästhetischer Bewertung und zum Zwecke des Entscheidens und Handelns gesehen. Drei Fragen sind laut Kant zu beantworten:

[2] In “On Computable Numbers with an Application to the Entscheidungsproblem.” a.a.O.

[3] Roger Penrose, Shadows of the Mind a.a.O., S. 21

– Was kann ich wissen?
– Was soll ich tun?
– Was kann ich hoffen?

Zur Beantwortung der Frage, was wir sicher wissen können, wie also menschliche Erkenntnis möglich sei, führte Kant drei Schichten des Denkens ein: die Vernunft, die das logische Denken reguliert, den Verstand, der das analytische Denken konstituiert und die Urteilskraft, die nach Erkenntnis des Sachverhaltes und der Handlungsmöglichkeiten die Abwägung zum Bewerten und Handeln erlaubt. Menschliches Handeln überwindet damit die Zwänge der Natur und erlaubt es, freie Entscheidungen zu treffen.[4] Ein gutes Urteil gründet notgedrungen auf Sinneseindrücken, auf dem, was der Fall zu sein scheint – also auf den Phänomenen, die nach bester Einsicht für wahr gehalten werden. Urteilskraft muss freilich entwickelt werden. Sie ist Ergebnis einer lebenslangen Bildung, Erfahrung und Übung, ein Ergebnis, das freilich nicht immer erreichbar ist: >Der Mangel an Urteilskraft ist eigentlich das, was man Dummheit nennt, und einem solchen Gebrechen ist gar nicht abzuhelfen.< heißt es ohne große Hoffnung in der Kritik der reinen Vernunft.

Was bedeutet dies nun nach dem Eintritt der >Denkmaschinen< in unsere Alltagswelt? Computer sollen die Frage >Was kann ich wissen?< zu beantworten helfen, so dass die Antwort auf die Frage >Was soll ich tun?< leichter fällt. Die Frage >Was kann ich hoffen?< bleibt davon unberührt. Diese wird weder bei Wittgenstein noch bei Turing als wissenschaftliche Frage gesehen – und auch bei Kant bleibt sie zwischen den Zeilen offen. So offen, das der preußische König Friedrich Wilhelm II ihm in einer privat versendeten Kabinettsordre verbot, öffentlich darüber zu räsonnieren (Was der Königsberger Rektor bis zum Tod des Königs auch beachtete). Kants Reflexion Nr. 177 >Was wir denken, können wir nicht immer sagen< aus den >Reflexionen zur Anthropologie< klingt unter diesen Umständen wie Resignation vor der Zensur; es ist aber eine Bemerkung über das Denken, das die Grenze zum Hoffen überschreitet. Damit geht Kant über eine alleinige Bestimmung des Menschen als „denkendem Wesen“ hinaus. Den drei Leitfragen lässt er deshalb eine vierte Frage folgen: >Was ist der Mensch?<[5] Diese kann weder von der Formalen Logik noch von der Informatik beantwortet werden. Jeder Vergleich mit dem Computer führt nur in die Irre.

*

Wittgensteins Neffe Heinz v. Foerster greift die Schlussbemerkung im Tractatus auf seine Weise auf. Wie Turing geht er zum Beispiel der Machine zurück,

[4] Die aktuelle Debatte, ob dieser freie Wille dabei vollständig bewusst verläuft oder auch in tieferen Schichten unseres Gehirns verankert ist, wird davon gar nicht berührt – so wie wir nicht freiwillig für immer aufhören können zu atmen, wohl aber für eine gewisse Dauer.

[5] ... und er fügt in Klammern hinzu >Anthropologie; über die ich schon seit mehr als 20 Jahren jährlich ein Kollegium gelesen habe.< Vgl. Hamburg: Meiner, Briefwechsel, 1989, S. 634

wobei er triviale und nichttriviale Maschinen unterscheidet. Ein funktionierender Automat ist trivial, weil er genau das tut, was er tun soll. Heinz v. Foerster führte als Beispiel gern Rolls-Royce-Motoren an, bei denen er unterstellte, die sie genau so funktionierten, wie die Ingenieure sich das vorstellten. Auch eine deterministische Turingmaschine ist in Bezug auf den nächsten Arbeitsschritt trivial (so wie auch das Programm einer nicht-deterministischen Turingmachine eine nicht genauer diskriminierte Menge möglicher Nachfolgezustände festlegt). Die Frage, ob so eine *paper machine* jemals mit dem Rechnen zu Ende kommt, ist freilich, wie wir dank Alan Turing wissen, nicht entscheidbar. Foersters Dreh ist nun, dass er alle >interessanten< Fragen als nicht entscheidbar einstuft, wobei er von der logischen Unentscheidbarkeit zur Unentscheidbarkeit des richtigen Handelns übergeht. Ob das Weltall unendlich alt oder groß ist oder ob ein Big Bang die Erklärung ist, ob wir jemals Zeitreisen unternehmen können, ob unsere Galaxis in einem schwarzen Loch verschwinden wird, ob Götter oder Dämonen existieren, ob der Mensch seinem Wesen nach gut oder böse ist, all dies sind unentscheidbare Fragen. Das wäre als solches eine billige Beobachtung. Spannend wird sie bei Fragen, die ein Handeln verlangen. Bei der Frage, ob wir diesen Partner oder jenen heiraten sollen oder besser ledig bleiben, ob wir eine Krankheit durch eine Operation bekämpfen lassen wollen, ob wir auswandern sollen oder im Lande bleiben, ob wir diesen Beruf ergreifen oder jenen, ob wir aus dem Haus gehen oder ein Buch lesen, müssen wir uns entscheiden und es gibt keine eindeutige Handlungsanweisung. >Das ist das Amüsante an den prinzipiell unentscheidbaren Fragen; dass es eben keinen Formalismus, keinen Zwang gibt, der mich zwingt, diese Fragen in dieser oder jener Form zu beantworten. Mit dieser prinzipiellen Unentscheidbarkeit ist ein Raum der Freiheit geöffnet, in dem du jetzt entscheiden kannst. Das heisst, prinzipiell unentscheidbare Fragen können nur wir entscheiden, indem wir sagen: *Ich möchte diese Entscheidung wählen, denn ich habe die Freiheit, hier zu wählen, was ich will.* Die Idee, die Freiheit mit der prinzipiellen Unentscheidbarkeit zu kombinieren, bringt jetzt die Idee der Verantwortung mit sich, denn wenn ich eine prinzipiell unentscheidbare Frage entscheide, habe ich mit dieser Entscheidung die Verantwortung für diese Entscheidung übernommen.<[6] An der logischen Kategorie der >nicht entscheidbaren Fragen< spiegelt sich so die ethische Kategorie der >Zu entscheidenden Fragen.< Turings These und mit ihr manche Hoffnungen der Künstlichen Intelligenz lösen sich im Ethischen Imperativ Foersters auf. Noch einmal Heinz v. Foerster: >Unentscheidbarkeit ist die Einladung, sich zu entscheiden. Für diese Entscheidung trägt man dann die Verantwortung.< Das heißt also, selbst wenn alle logisch entscheidbaren Fragen geklärt wären, alles Berechenbare berechnet wäre, bliebe das Ethische unbearbeitet. Dafür brauchen wir Urteilskraft. Das ist das lebenslange Ziel von Erziehung und Bildung.

[6] M. Bröcker & H. V. Foerster, Teil der Welt, a.a.O. S. 178

Literatur

1. Lena Bonsiepen, Folgen des Marginalen, in G. Cyranek und W. Coy (Hrsg): Die maschinelle Kunst des Denkens – Perspektiven und Grenzen der KI. Vieweg, Braunschweig/Wiesbaden, 1994.
2. Monika Bröcker und Heinz v. Foerster, Teil der Welt – Fraktale einer Ethik. Ein Drama in drei Akten. Carl Auer Systeme Verlag, Heidelberg, 2002.
3. Wolfgang Coy, Reduziertes Denken. Informatik in der Tradition des formalistischen Forschungsprogramms, in P. Schefe, H. Hastedt, Y. Dittrich und G. Keil (Hrsg.): Informatik und Philosophie, 31–52. BI Wissenschaftsverlag, Mannheim-Leipzig-Wien-Zürich, 1993.
4. Bernhard Dotzler und Friedrich Kittler (Hrsg.), Alan Turing – Intelligence Service. Brinkmann & Bose, Berlin, 1987.
5. Heinz v. Foerster, Wissen und Gewissen. Suhrkamp, Frankfurt/Main, 1993.
6. Andrew Hodges, Alan Turing – Enigma. Kammerer und Unverzagt, Berlin, 1990.
7. Roger Penrose, Shadows of the Mind: A Search for the Missing Science of Consciousness. Oxford University Press, Oxford, 1994.
8. Alan M. Turing, On Computable Numbers, with an Application to the Entscheidungsproblem, Proc. London Math. Soc. 42(2) (1937), 230-265.
9. Alan M. Turing, Computing machinery and intelligence, Mind, Vol. LIX. No. 236. Oktober 1950, S. 433–460.
10. Ludwig Wittgenstein, Logisch-philosophische Abhandlung. Tractatus logico-philosophicus. Suhrkamp, Frankfurt/Main, 1969.

Prof. Dr. Wolfgang Coy
Institut für Informatik
Humboldt-Universität zu Berlin
D-12489 Berlin (Germany)
coy@informatik.hu-berlin.de
http://www.informatik.hu-berlin.de/~coy

Wolfgang Coy and Hans-Jörg Kreowski were university colleagues from 1982 to 1995 in Bremen and are brothers in arms in *FIfF – Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung* (Forum Computer Professionals for Peace and Social Responsibility). Together with Frieder Nake, Hans-Jörg Kreowski and Wolfgang Coy founded the BIGLab (Labor für Bilder und Grafik – Lab for Images and Graphics) in 1989 to integrate the research of their groups.

Towards the Tree Automata Workbench MARBLES[*]

Frank Drewes

Abstract. We sketch the conceptual ideas that are intended to become the basis for the Tree Automata Workbench MARBLES[1], an extensible system that will facilitate the experimentation with virtually any kind of algorithms on tree automata. Moreover, the system will come with a library and an application programmer's interface that can be used by anyone wanting to apply such algorithms in research and development.

1 Introduction

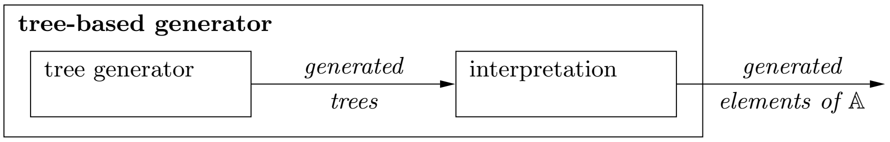
Already in the 1960s, researchers in finite-automata theory realized that large parts of this theory can be generalized by replacing strings with trees, without loosing many of the positive algorithmic results, closure properties, and the like. This observation gave rise to a flourishing theory, including a large number of techniques and algorithms for analysis, modification, and synthesis of various kinds of tree recognizers, tree grammars, and tree transducers [GS84, NP92, GS97, FV98, CDG$^+$07]. Throughout the rest of this paper, all devices that fall into one of these categories will be called tree automata. Today, probably more theoretical research than ever before is done in this area, motivated by a constantly growing number of applications of tree automata in fields such as verification and model checking [GK00, AJMd02, Löd02, FGVTT04], natural language processing [KG05, GKM08], XML processing [Sch07], code selection in compilers [FSW94], graph and picture generation [Eng94, Dre06], and others.

The system TREEBAG[2] uses tree generators to generate sets of objects over arbitrary domains. The central data type of TREEBAG is the ranked and ordered tree, with nodes labelled by symbols taken from a ranked alphabet Σ. In other words, every symbol $f \in \Sigma$ comes with a rank $k \geq 0$, such that a node labelled with f is required to have exactly k children (which are totally ordered). This means that a tree in the sense of TREEBAG is a term, i.e., a well-formed expression composed of abstract (i.e., "meaningless") operation symbols, each having a specified rank that determines the number of subexpressions. TREEBAG deals with two types of tree automata on this type of trees, namely tree grammars and tree transducers. A tree grammar is a device that generates trees out of itself, whereas a tree transducer is one that turns input trees into output trees.

[*] Dedicated to Hans-Jörg Kreowski on the occasion of this 60th birthday.
[1] Tree Automata Workbench = taw = a large marble, a game of marbles (Oxford New Amer. Dict.).
[2] Tree-Based Generator

A tree generator is a tree grammar composed with a (possibly empty) sequence of tree transducers.

In a well-known way, trees of the type described above can be assigned a semantics by choosing a domain $\mathbb{A}$ and associating an operation on $\mathbb{A}$ (of the appropriate arity) with each symbol in the ranked alphabet Σ considered. In other words, a Σ-algebra is specified, so that every tree evaluates to an element of $\mathbb{A}$. This means that a device generating trees provides the syntactic basis for a *tree-based generator* – a system consisting of a tree generator and an interpretation, thus generating elements of $\mathbb{A}$:



Treebag makes it possible to assemble tree-based generators. This is explained in slightly more detail in Section 2, because Marbles is to a certain extent inspired by Treebag. However, in Treebag, all that can be done after having assembled a tree generator is to execute it. In contrast, the usefulness of tree automata in most application areas does not primarily lie in the fact that they can be executed. Their real advantage is that they are simple enough to be effectively analyzed and manipulated. For instance, in a model checking application, tree automata may be generated that model safety and liveness properties of a protocol to be verified. Analyzing these automata then corresponds to checking correctness criteria, thus solving a model checking problem. The idea behind Marbles is, therefore, to make it possible to assemble *algorithms on tree automata*, thus perceiving tree automata mainly as the objects to be analyzed and manipulated, rather than as executable algorithms. The major aim is to provide researchers with a software environment and infrastructure that enables them to create, use, and experiment with algorithms on tree automata.

In addition to Treebag, there are several other systems that implement certain types of tree automata or algorithms on them.

AutoWrite (`http://dept-info.labri.fr/~idurand/autowrite`) is a system that allows the user to check properties of term rewrite systems by means of tree automata constructions. In particular, it allows to load, save, and combine bottom-up tree recognizers. Using the graphical user interface, one can build and manipulate bottom-up tree recognizers related to the term rewrite systems whose properties one wants to check.

Forest FIRE (`http://www.loekcleophas.com`) is a toolkit focusing on recognition, pattern matching, and parsing algorithms in connection with regular tree languages. The system has been developed on the basis of detailed taxonomies, with the major purpose of gaining a deeper conceptual understanding of how the ideas and techniques used in various tree automata constructions are related to each other.

MONA (`http://www.brics.dk/mona`) is a tool for checking the validity of formulas in the weak second-order theory of one successor (WS1S) or of two successors (WS2S). For deciding WS2S, the decision procedures convert a given formula into a so-called guided tree automaton, a variant of a bottom-up tree recognizers, and analyse this automaton.

Tiburon (`http://www.isi.edu/licensed-sw/tiburon`) is a command-line based package of algorithms on weighted regular tree grammars, context-free string grammars, and tree transducers, including various analyzers, modifiers, and synthesizers. The devices and algorithms implemented in Tiburon are typical even for MARBLES, but Tiburon has mainly been developed for applications in Natural Language Processing, without MARBLES' emphasis on a flexible environment that can be adapted and extended to suit the needs of researchers who study tree automata from different points of view.

Timbuk (`http://www.irisa.fr/lande/genet/timbuk`) is a collection of tools for carrying out reachability proofs of term rewrite systems, among other techniques by manipulating nondeterministic bottom-up tree recognizers. It is intended to be used for the verification of programs and cryptographic protocols.

The proposed system MARBLES differs from each of these systems in several respects. Most notably, the systems above have all been developed with a particular application or problem area in mind. They are great for their particular purpose, but they are also restricted to it. In contrast, the intention behind MARBLES is to support tree automata research in general, by providing researchers with a suitable platform and infrastructure for their own extensions, making it possible to experiment with and apply tree automata algorithms of any kind.

The remainder of this paper is structured as follows. The next section presents some aspects of TREEBAG that have, in one way or the other, inspired the intended characteristics of MARBLES. In Section 3, some of the different types of trees, tree automata, and tree automata algorithms that should, in principle, be covered by MARBLES, are discussed. Section 4 presents initial ideas regarding some of the concepts needed for making this possible. Finally, Section 5 concludes the paper.

2 TREEBAG

Let us now have a slightly closer look at the concepts and design principles of TREEBAG. The following description is intentionally kept at a rather abstract level, although concrete classes of, e.g., tree grammars and algebras available in TREEBAG are sometimes mentioned as examples, maninly for readers who happen to be familiar with tree automata theory. Readers who want to inform themselves in more detail should consult the TREEBAG user manual available at `http://www.cs.umu.se/~drewes/treebag` or, for the theory behind, [Dre06].

The work on TREEBAG was started during the second half of the 1990s, when the author was a member of Hans-Jörg Kreowski's research group at the University of Bremen. Around this time, a significant part of our research was dedicated to context-free graph and collage grammars; see, e.g., [HKV91, HKL93, HKT93, DHKT95, DK96, DHK97, DK99]. Both of these can be characterized by combinations of a certain type of tree grammars (namely regular tree grammars) with suitable algebras in the style of Mezei and Wright [MW67], i.e., the grammars can be viewed as tree-based generators. For graphs, this has been made explicit by Engelfriet in [Eng94], and for collages by the author in [Dre96, Dre00]. See also [DEKK03], where this characterization was used to establish certain decidability results for collage languages. In this context, one should not forget to mention Engelfriet's paper [Eng80], where he discusses symbolic computation by tree transductions, which is essentially the same idea, but now for transformation rather than generation: a tree transduction, together with algebras interpreting the input and output trees, is considered as a symbolic algorithm that performs a computation on abstract trees rather than on the concrete objects of the two domains in question.

Whereas the results mentioned above use only regular tree grammars, it is obvious that one may in fact combine arbitrary kinds of tree generators with any sort of algebra, yielding a large number of different grammatical formalisms with comparatively little effort. Being a rather straightforward implementation of this idea (in Java), TREEBAG allows its user to assemble tree-based generators of various kinds. There are four major abstract classes, namely *tree grammars*, *tree transducers*, *algebras*, and *displays*. The first three represent the corresponding formal concepts, whereas displays are required for actually being able to see the results of the generating process. Concrete subclasses of the four abstract classes implement particular types of tree grammars, tree transducers, algebras, and displays. For example, the classes `generators.ET0LTreeGrammar` and `generators.mtTransducer` implement ET0L tree grammars and macro tree transducers, resp. If a class such as these is available, this means that the user can define specific instances (usually in ordinary ASCII text files) and use them in assembling tree-based generators. Such instances are called components in the following.

Figure 1 shows a typical situation when working with TREEBAG. Window 1 is the main window of the system, the so-called worksheet. When the user loads a component, it is represented on the worksheet as a blob. These blobs represent the nodes of a directed acyclic graph whose edges determine the data flow between components. The data-flow edges are interactively established by the user, subject to a few rather obvious rules: The output of a tree grammar or tree transducer can become the input of tree transducers or algebras, and the output of an algebra can become the input of a display. The configuration in Figure 1 consists of a regular tree grammar, a free term algebra with a corresponding tree display, a top-down tree transducer, and two copies of a collage algebra, each with its corresponding collage display. With each display component, a window is associated, namely the windows numbered 3–5. Thus, these windows show the tree generated by the regular tree grammar, its interpretation by the collage grammar, and the interpretation of the transformed tree by (another instance of) the same collage algebra.

An additional window (numbered 2 in the figure) contains buttons that provide access to the user commands of the regular tree grammar. Double clicks on

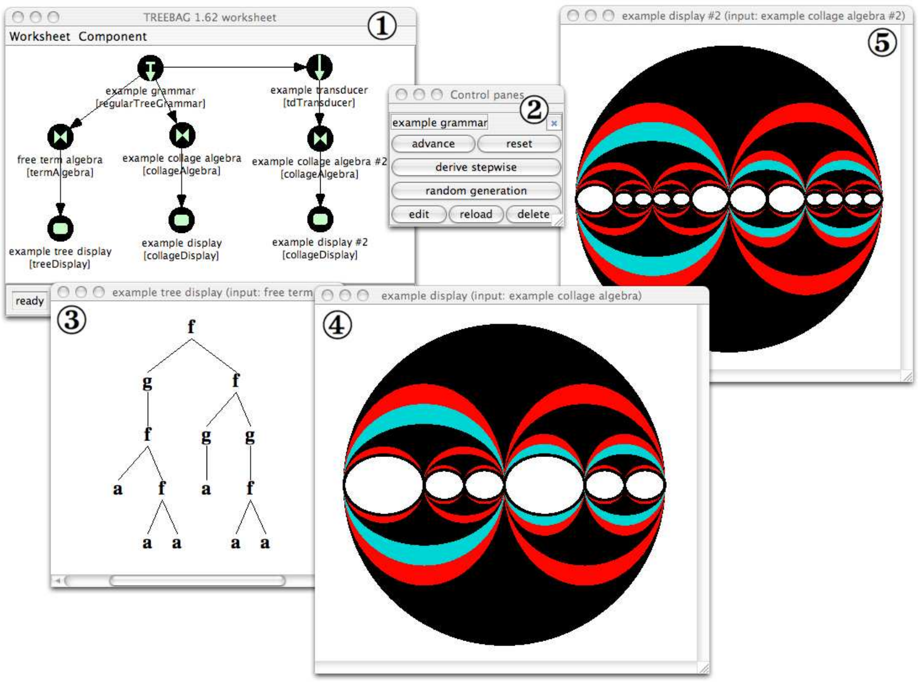

TREEBAG 1.62 worksheet
Worksheet Component
example grammar [regularTreeGrammar]
example transducer [tdTransducer]
free term algebra [termAlgebra]
example collage algebra [collageAlgebra]
example collage algebra #2 [collageAlgebra]
example tree display [treeDisplay]
example display [collageDisplay]
example display #2 [collageDisplay]
ready
Control panes
example grammar
advance
reset
derive stepwise
random generation
edit
reload
delete
example display #2 (input: example collage algebra #2)
example tree display (input: free term
example display (input: example collage algebra)
f g f f g g a f a f a a a a
①
②
③
④
⑤

Fig. 1. A typical configuration of TREEBAG

the other components on the worksheet would open similar sections in this window for them, each one being populated by the individual commands understood by the respective component.

Let us now discuss two aspects of the design of TREEBAG which are expected to have some influence on MARBLES. In fact, these two aspects are quite closely related and can be seen as the two sides of the same coin.

From the point of view of the user, the way in which components can be interconnected depends only on their types, i.e., whether they are tree grammars, tree transducers, algebras or displays. In other words, if the user wants to connect a tree grammar and a tree transducer, this can be done regardless of whether the tree grammar at hand is a regular tree grammar, ET0L tree grammar, context-free tree grammar or whatever type of tree grammar might at some point in time be implemented in TREEBAG. Of course, users must interconnect the "right" components to achieve a particular effect desired, but the system gives users as much freedom as possible. Every concrete component class provides the user with a set of commands that can be used to interact with components of this class (recall Window 2 in Figure 1, containing buttons for the commands provided by the implementation of regular tree grammars). While the commands would be different for, e.g., ET0L tree grammars, this has no influence on the way in which regular tree grammars or ET0L tree grammars can be connected to other components.

The person who implements new classes of tree grammars, tree transducers, algebras or displays will find out that the properties mentioned in the previous paragraph simply reflect properties of the implementation. The core of TREEBAG does not make any distinction between, e.g., different classes of tree grammars. In fact, consider the file defining the regular tree grammar used in Figure 1:

```
generators.regularTreeGrammar("example grammar"):
( { S, A },
  { f:2, g:1, a:0 },
  { S -> f[S,S],
    S -> g[A],
    A -> f[A,A],
    A -> a },
  S )
```

When the user instructs TREEBAG to load this component, it parses only the first line, to discover that the user wishes to load an instance of a component class named `generators.regularTreeGrammar`. The rest of the file uses a syntax which is unknown to (the core of) TREEBAG, as it is specific to the implementation of this class. To handle this, TREEBAG dynamically tries to load the class `generators.regularTreeGrammar` and, upon success, creates an (uninitialized) object of this class. Now, it lets this very object, which is required to contain a method called `parse`, initialize itself by parsing the remainder of the file. Each of the four abstract component types of TREEBAG requires its concrete subclasses to implement such a parsing method. To handle component-specific user commands, each concrete subclass provides two further methods. The first returns, at any point in time, the names of the user commands available at that moment (which means that the list of commands may change), while the second executes a given command.

This structure makes it possible to extend TREEBAG by new classes of tree grammars, tree transducers, algebras, and displays in an easy way, without having to change existing parts of the system. One only has to implement it as a subclass of the appropriate abstract component class and place it in the appropriate directory. Immediately afterwards (provided that everything has been done correctly), it is possible to load instances of this class onto the worksheet, interconnect them with other components, and work with them.

It may be interesting to note that the implementations of some of the classes currently available in TREEBAG make use of decomposition results from the literature. For example, a so-called branching synchronization tree grammar of nesting depth n can be decomposed into a regular tree grammar and a sequence of n top-down tree transducers (see [DE04]). During the parsing step, the implementation of this class in TREEBAG performs this decomposition and writes the $n+1$ components onto the hard disk (in the syntax required by the respective classes). Afterwards, it uses TREEBAG's loading mechanism to load them as internal variables hidden from the user (i.e., so that they do not appear on the worksheet). Every user command is basically forwarded to these internal components, and whichever output tree they produce is returned. In this way, the implementation of the class becomes considerably easier and less error prone than a direct one.

3 Trees, Tree Automata, and Tree Automata Algorithms

As mentioned in the introduction, the major intended purpose of MARBLES is to make it possible to apply and experiment with algorithms on tree automata. The aim is to design MARBLES in such a way that it accommodates virtually all kinds of tree automata algorithms. While this does not mean that all such algorithms should readily be implemented in the system, the design of MARBLES should enable researchers (and application programmers) interested in a particular type of tree automata algorithms to make the necessary extensions. As in the case of TREEBAG, this should be possible without requiring changes of already existing parts. However, compared to TREEBAG, the design challenge is is considerably bigger for MARBLES, because its intended coverage is much wider. It seems to be reasonable to distinguish between (at least) three central categories of objects: trees, tree automata, and tree automata algorithms. Each of them may, in principle, have any number of subcategories one may wish to implement in MARBLES. In the following, some of the possible subcategories of each will be discussed to illustrate this point.

3.1 Trees

In the traditional setting (and in TREEBAG), tree automata work on trees over ranked alphabets, as explained above. This is appropriate, because trees are supposed to be evaluated by algebras by associating with every symbol of rank k a k-ary function on some domain. However, tree automata on unranked trees have received a lot of attention during recent years. Here, symbols are unranked, and a node in a tree can have any finite number of children, regardless of the symbol it is labeled with. It turns out that this variant is well suited for applications in connection with XML, because XML documents can appropriately be viewed as unranked trees. (For example, a node corresponding to a list structure in HTML may have any number of children of type *list item.*) Thus, an XML document type corresponds to a tree language of unranked trees, and a tree transducer on unranked trees corresponds to a transformation between XML document types.

While the two types of trees mentioned are the only ones that play a major role in contemporary research on tree automata, this situation may change in the future. Thus, MARBLES should allow programmers to implement other classes of trees than just these.

3.2 Tree Automata

Tree automata can be classified according to various criteria. An important observation one can make is that the resulting classifications are, to a rather large extent, orthogonal.

Perhaps the most obvious classification is the one that gave rise to the structure of Treebag, distinguishing between tree grammars, tree recognizers (not directly available in Treebag), and tree transducers. From an abstract point of view, a tree grammar is a formal device that generates output trees without requiring input. As usual, the tree recognizer is the dual concept. It takes a tree as input and computes an output value, usually in the range $\{0, 1\}$, indicating whether the tree is accepted or not. Finally, a tree transducer is a formal device transforming input trees into output trees.

The second classification distinguishes between tree automata according to the type of trees they act upon, i.e., tree automata on ranked or unranked trees. Each of the types of tree automata in the first classification can be ranked or unranked. In this sense, these two classifications are orthogonal. In fact, one may even wish to consider tree transducers that turn unranked trees into ranked ones, or vice versa.

Finally, in addition to the traditional case of tree automata, one may consider weighted ones [FV09]. Weighted tree automata deal with tree series instead of tree languages, a tree series being a mapping $\psi\colon \mathrm{T}_\Sigma \to \mathbb{S}$, where T_Σ denotes the set of all trees over a given alphabet, and $\mathbb{S}$ is a semiring. In other words, weighted tree automata generalize the traditional case, which is obtained by choosing as $\mathbb{S}$ the Boolean semiring. Even this third classification is orthogonal to the two previous ones, provided that we define the tree automata according to the first classification in a way general enough to accommodate the weighted case.

It is interesting to note that, from an abstract point of view, but even more from the point of view of system design, weighted tree recognizers are very similar to algebras. Both take a tree as input and compute a value in some other domain.

3.3 Algorithms on Tree Automata

Many useful algorithms on tree automata have been described in the literature. For classification purposes, it is useful to distinguish between analyzers, synthesizers, and decomposition algorithms.

An analyzer for tree automata takes a tree automaton as input and analyses it with respect to certain properties. Well-known examples are algorithms that decide whether the language represented by a tree recognizer or tree grammar is empty or whether it is finite (cf., e.g., [DE98]).

A synthesizer is an algorithm that takes zero or more tree automata (and maybe some additional data) as input and yields a tree automaton as output. There are various important types of synthesizers:

– A generator is an algorithm that outputs tree automata without requiring other tree automata as input. A prominent example is given by grammatical

inference algorithms for tree automata. These are algorithms whose purpose it is to “learn” tree languages. For this, the algorithm is provided with some source of information regarding the tree language (or tree series) to be learned, such as positive and negative examples. It is then expected to construct a tree automaton representing the tree language in question. See, e.g., the references in [Dre09] for a variety of approaches.
Conceptually, a tree automaton A may be considered as a generator that outputs the constant value A.

– Conversion algorithms take a tree automaton as input and yield another tree automaton as output, usually with the same semantics as the input automaton. Well-known examples are conversions between regular tree grammars and finite-state tree recognizers and algorithms that minimize tree automata, make them deterministic, remove useless states or nonterminals, etc (see, e.g., [CDG+07]). There are also conversion algorithms that do not retain the semantics of the tree automaton they are applied to. For example, a macro tree transducer *mtt* [EV85] may be turned into a finite-state tree recognizer that accepts the pre-image of the tree transformation computed by *mtt*. A conversion algorithm that inverts suitable types of top-down tree transducers would be another example.
– Composition algorithms turn n tree automata ($n > 1$) into one. A wealth of such algorithms can be found in the literature. One type of example is, of course, given by composition in the strict sense. For instance, certain types of tree transductions are known to be closed under composition. Another example is the main result of [DE04], which provides an algorithm for converting a regular tree grammar g and n top-down tree transducers $td_1, \ldots, td_n$ into a branching synchronization tree grammar generating the image of $L(g)$ under $td_n \circ \cdots \circ td_1$. Composition algorithms in a more general sense may not perform mathematical composition, but combine tree automata in a different way. For example, two finite-state tree recognizers can be turned into one that recognizes the intersection of the tree languages recognized by the two individual automata.

Finally, decomposition algorithms are the conceptual inverse of composition algorithms, turning one tree automaton into several others. For example, for $\{x, y\} = \{\text{top-down}, \text{bottom-up}\}$, every x tree transducer may be decomposed into two y tree transducers [Eng75]. A similar example is given by the result that every deterministic total macro tree transducer may be decomposed into a top-down tree transducer followed by a YIELD mapping [EV85].

Of course, algorithms on tree automata may additionally be classified according to the types of tree automata they work on, similarly to the fact that tree automata may be classified according to the types of trees they work on.

4 A Proposed Attribute Type System for MARBLES

As mentioned earlier, the goal behind the development of MARBLES is that it should allow its user to assemble configurations of tree automata algorithms in a similar way as Treebag allows its user to assemble various sorts of tree-based generators. In particular, there should be a way to load components representing (tree automata and) tree automata algorithms, establish a data-flow relation between, and execute them. However, while Treebag comes with a fixed set of component types, something like this is neither possible nor desirable for Marbles. In contrast, users should be given the possibility to define and implement their own classes of tree automata algorithms and experiment with them. The following two fictitious scenarios try to illustrate this.[3]

Scenario 1: Test Environment for Minimization Algorithms. Doctoral student X works in a research group using bottom-up tree recognizers for model checking purposes. A typical example is the verification of a process communication protocol P by generating a tree recognizer A_P that models P's behavior and then analyzing A_P to establish P's correctness. The problem is that A_P tends to be huge, and often unnecessarily huge, so that its analysis takes too much time. Unfortunately, A_P is also nondeterministic, which means that it cannot efficiently be minimized.

Therefore, in her thesis, X proposes and studies a number of efficient heuristics for reducing nondeterministic tree recognizers A in size (called minimization, for simplicity). The general technique used is to compute a suitable equivalence $\equiv$ on the state set of A, such that the quotient automaton $A/\equiv$ accepts the same language as A. The various heuristics studied differ only in the concrete definition (and computation) of $\equiv$. Besides studying the minimization algorithms theoretically to establish their correctness and worst case complexity, X wants to study empirically how they behave on real examples arising in the model checking context, in terms of size reduction and efficiency. However, X does not have the time to implement a test environment for her algorithms from scratch, in addition to her theoretical studies. Therefore, she decides to use Marbles.

First, she notices that there is a type of tree automata algorithms called generator, a special type of synthesizer. She defines and implements a simple generator which lets the user choose the name of a protocol (from a fixed set of possible choices) and possibly some other parameters. The generator will then output nondeterministic bottom-up tree recognizers of increasing size, whenever the user presses a certain button.

Next, X discovers that there are so-called converters, and decides to implement a new type of converter as an abstract class. A concrete implementation is obtained by providing a method that, for a given bottom-up tree recognizer A, computes an equivalence relation $\equiv$ on the states. The converter will then return $A/\equiv$.

Fortunately, X finds out that someone else has already implemented two useful auxiliary components. One of them is a wrapper for arbitrary converters that simply executes them, but also reports how much time the execution takes. The other one takes bottom-up tree recognizers as input and saves some statistics

[3] While being fictitious, the scenarios have a real background, as they are inspired by [Kaa08] and ongoing work in our own group, resp.

about them to a file, such as the number of states and transitions. Now, X has everything needed to make the desired tests. All she has to do is to implement the different algorithms yielding the equivalence relations ≡, load and interconnect the required components, and execute them.

Scenario 2: Simulation of Minimal Adequate Teachers Using Corpora. The research group in which researcher Y is working has previously studied grammatical inference algorithms that, within Angluin's learning model of a *minimal adequate teacher* (MAT), construct bottom-up tree recognizers for recognizable tree languages L. Now, they want to find out whether such an algorithm can be used to learn the syntax of natural languages reasonably well, where the necessary data is taken from a corpus.[4]

The major obstacle is the MAT, an oracle capable of answering two types of queries, namely membership queries (*Is the tree t in L?*) and equivalence queries (*Does the bottom-up tree recognizer A satisfy $L(A) = L$? If not, return a counterexample.*) Clearly, a MAT is not available in the situation sketched above. The research question is whether it can (imperfectly) be simulated on the basis of a corpus, so that the inference algorithm as a whole runs with reasonable efficiency and yields acceptable results.

Y decides to try out some approaches and to use MARBLES for that purpose. Thus, she defines two new types of algorithms, namely MATs and learners. A learner is a generator that must be connected to a MAT to create a tree automaton. During the first phase, she only wants to test different realizations of the MAT, to see whether the results are promising enough to continue. Therefore, she implements a single learner (e.g., any of those in [Dre09]). In contrast, a variety of different MATs are implemented, using different approaches for answering membership and equivalence queries based on a corpus.

To find out how good the various approaches are, Y implements a component that has access to a sufficiently large sample of positive and negative examples. It takes a tree recognizer as input, runs it on the samples, and returns statistics regarding its sensitivity and specifitivity. In a second phase of her research work, Y even wants to study other variants of the learner, which can be done in the same setting by replacing the one learner with another.

In scenarios such as those above, the researcher who wants to use MARBLES must implement certain extensions, new types of tree automata and algorithms that become components of MARBLES. For both the system and the user, it is necessary to know in which way instances of these components can be combined. Thus, there must be a possibility to talk about the types of components in an easy, but flexible way. A prerequisite for this is to be able to specify which basic data types exist. Here, we only focus on the perspective of the user and the GUI, which means that the only thing we need is a way to *give name* to different sorts of data.

[4] A corpus is a manually analyzed and annotated database of sentences in a natural language.

While the tree is *the* basic data type in MARBLES, it may not be the only one. Moreover, there may be different sorts of trees. We now define basic types which make it possible to name those structures.

Definition 1 (basic type). *Let $ATTR$ be a finite set of* data attributes *(briefly called attributes). The set $TEXP$ of all* basic types *is the smallest set of pairs such that, for all finite subsets T of $TEXP$ and all attributes a, we have $(a, T) \in TEXP$. If $T = \{t_1, \ldots, t_n\}$, then this basic type is also denoted by $a\langle t_1, \ldots, t_n\rangle$, or by a if $n = 0$.*

As an example, consider trees. We may, e.g., have ranked, unranked, ordered, and unordered trees. Ranked trees may or may not be binary or monadic. Our set of data attributes could then be $ATTR = \{\mathit{tree}, \mathit{ranked}, \mathit{unranked}, \mathit{ordered}, \mathit{unordered}, \mathit{bin}, \mathit{mon}\}$. The basic type for ranked unordered binary trees would then be $\mathit{tree}\langle \mathit{ranked}\langle \mathit{bin}\rangle, \mathit{unordered}\rangle$. The attributes in such a basic type should be seen as assertions stating that the data in question has certain properties. In other words, the presence of an attribute restricts the data type. For example, *tree* is a basic type meaning just any tree, and $\mathit{tree}\langle \mathit{unordered}\rangle$ means "any type of unordered trees". Note that we, intentionally, do not associate a specific semantics with the attributes. It should, however, be *possible* to do this in MARBLES by, e.g., associating an attribute with an abstract class in the implementation. A similar remark applies to the types at the higher levels discussed next.

Next, we define what the type of an automaton looks like. We take a very general approach, where an automaton is a device that turns a finite number of input values of specified basic types and into a finite number of output values, also of specified basic types.

Definition 2 (automaton type). *An* automaton type *is a pair $(\mathit{in}, \mathit{out})$ with $\mathit{in} \in TEXP^k$ and $\mathit{out} \in TEXP^l$ for some $k, l \geq 0$. Such a type will normally be written as $\mathit{in} \to \mathit{out}$. The set of all automaton types is denoted by AUT.*

As an example, a tree grammar of the most general form could be described as being an automaton of type $() \to (\mathit{tree})$, as it takes no input and yields any type of tree as output. Slightly more specific would be a tree grammar generating ranked trees, its type being $() \to (\mathit{tree}\langle \mathit{ranked}\rangle)$. For weighted tree automata over a semifield that work on ranked trees, the type $(\mathit{tree}\langle \mathit{ranked}\rangle) \to (\mathit{semiring}\langle \mathit{semifield}\rangle)$ could be an appropriate description, and for tree transducers on unranked trees one could use $(\mathit{tree}\langle \mathit{unranked}\rangle) \to (\mathit{tree}\langle \mathit{unranked}\rangle)$. Though uncommon in the literature, one may also wish to consider, e.g., tree transducers that take two trees as input and produce one output tree, the corresponding type being $(\mathit{tree}, \mathit{tree}) \to (\mathit{tree})$.

Note that the concept is very general. For example, an algebra can be seen as an automaton of type $(\mathit{tree}\langle \mathit{ranked}\rangle) \to (\mathit{any})$, if we let *any* be the most general basic type, standing for arbitrary data. Also weighted tree automata over multioperator monoids [Kui00] have this type. In fact, the concept covers even devices that do not work on trees at all.

We finally define how to distinguish between different types of algorithms on tree automata.

Definition 3 (algorithm type). *The set ALG of* algorithm types *is inductively defined to be the smallest set containing all triples (in, use, out) such that, for some $k, l, m \geq 0$, $in \in AUT^k$, $out \in AUT^m$, and $use \in ALG^l$. Such a triple is denoted by $in \xrightarrow{use} out$.*

The intuitive interpretation of $in \xrightarrow{use} out$ is that of an algorithm which turns inputs according to *in* into outputs according to *out*, thereby possibly making use of other algorithms given by *use*. A typical example is the MAT learner in Scenario 2, which could be of the type $() \xrightarrow{MAT} (TA)$, where TA is the automaton type $tree\langle ranked\rangle \to bool$.

As mentioned earlier, one of the ideas behind MARBLES is that its GUI, similar to the one of TREEBAG, should allow the user to assemble configurations of tree automata in order to experiment with them. The basic (and still somewhat tentative) plan is that every implementation of a class of tree automata or tree automata algorithms comes with a specified type according to the definitions above. When the user loads an instance of such a component, this information is used in order to determine which connections between these components are possible. For example, an algorithm of the type in Definition 3 will, from the point of view of the user, have $k + l + m$ slots representing the inputs, the used algorithms, and the outputs. For instance, if a component has an output slot s of type $tree\langle ranked\rangle \to bool$ (a recognizer for ranked trees) and another one has an input slot s' of type $tree \to bool$ (a recognizer for any sort of trees), then the data flow can be directed from s to s'.

5 Concluding Remarks

In this paper, ideas and plans regarding a successor of the system TREEBAG have been presented. While this work is still in a very preliminary phase, the overall goal is clear. MARBLES should make it possible to experiment with configurations of tree automata algorithms in a similar way as TREEBAG makes it possible to experiment with tree-based generators. Moreover, MARBLES should be extensible by researchers who are not directly involved in the development of the system itself, but want to use it for their own purposes. For this, concepts such as those presented in Section 4 seem to be a necessity, because the GUI must be able to handle extensions without explicitly being adapted.

An aspect that has not been discussed in the present paper, but which is a necessity as well, is to provide programmers with a well-documented library and a clearly structured application programmer's interface (API). Without this, it would be too difficult, error prone, and time consuming for other researchers to make their own extensions. In fact, it should also be possible to make use of the API without adopting the rest of MARBLES, and especially its GUI. This would programmers give the possibility to apply tree automata algorithms in their own applications. Another aspect that has not yet been decided upon is whether and to what extent MARBLES shall be compatible and able to interoperate with other systems dealing with tree automata, such as those mentioned in Section 1.

Acknowledgment I thank the anonymous referees for their thorough reading of the manuscript and for giving numerous useful comments.

However, most of all, I want to thank you, Hans-Jörg, for the support and inspiration during all those years, for teaching me so much about our profession, and for being a good example in all respects. Happy Birthday to you!

References

[AJMd02] Parosh Aziz Abdulla, Bengt Jonsson, Pritha Mahata, and Julien d'Orso. Regular tree model checking. In E. Brinksma and K. Guldstrand Larsen, editors, *Proc. 14th Intl. Conf. on Computer Aided Verification (CAV'02)*, volume 2404 of *Lecture Notes in Computer Science*, pages 555–568, 2002.

[CDG$^+$07] Hubert Comon, Max Dauchet, Rémi Gilleron, Florent Jacquemard, Christof Löding, Denis Lugiez, Sophie Tison, and Marc Tommasi. *Tree Automata Techniques and Applications*. Internet publication available at `http://tata.gforge.inria.fr`, 2007. Release October 2007.

[DE98] Frank Drewes and Joost Engelfriet. Decidability of the finiteness of ranges of tree transductions. *Information and Computation*, 145:1–50, 1998.

[DE04] Frank Drewes and Joost Engelfriet. Branching synchronization grammars with nested tables. *Journal of Computer and System Sciences*, 68:611–656, 2004.

[DEKK03] Frank Drewes, Sigrid Ewert, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Computing raster images from grid picture grammars. *Journal of Automata, Languages and Combinatorics*, 8:499–519, 2003.

[DHK97] Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*, chapter 2, pages 95–162. World Scientific, Singapore, 1997.

[DHKT95] Frank Drewes, Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Generating self-affine fractals by collage grammars. *Theoretical Computer Science*, 145:159–187, 1995.

[DK96] Frank Drewes and Hans-Jörg Kreowski. (Un-)decidability of geometric properties of pictures generated by collage grammars. *Fundamenta Informaticae*, 25:295–325, 1996.

[DK99] Frank Drewes and Hans-Jörg Kreowski. Picture generation by collage grammars. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation,* Vol. 2: *Applications, Languages, and Tools*, chapter 11, pages 397–457. World Scientific, Singapore, 1999.

[Dre96] Frank Drewes. Language theoretic and algorithmic properties of d-dimensional collages and patterns in a grid. *Journal of Computer and System Sciences*, 53:33–60, 1996.

[Dre00] Frank Drewes. Tree-based picture generation. *Theoretical Computer Science*, 246:1–51, 2000.

[Dre06] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[Dre09] Frank Drewes. Mat learners for recognizable tree languages and tree series. *Acta Cybernetica*, 2009. To appear.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations – a comparison. *Mathematical Systems Theory*, 9:198–231, 1975.

[Eng80] Joost Engelfriet. Some open questions and recent results on tree transducers and tree languages. In R. V. Book, editor, *Formal Language Theory: Perspectives and Open Problems*, pages 241–286. Academic Press, New York, 1980.

[Eng94] Joost Engelfriet. Graph grammars and tree transducers. In S. Tison, editor, *Proceedings of the CAAP 94*, volume 787 of *Lecture Notes in Computer Science*, pages 15–37. Springer, 1994.

[EV85] Joost Engelfriet and Heiko Vogler. Macro tree transducers. *Journal of Computer and System Sciences*, 31:71–146, 1985.

[FGVTT04] Guillaume Feuillade, Thomas Genet, and Valérie Viet Triem Tong. Reachability analysis over term rewriting systems. *Journal of Automated Reasoning*, 33:341–383, 2004.

[FSW94] Christian Ferdinand, Helmut Seidl, and Reinhard Wilhelm. Tree automata for code selection. *Acta Informatica*, 31(8):741–760, 1994.

[FV98] Zoltán Fülöp and Heiko Vogler. *Syntax-Directed Semantics: Formal Models Based on Tree Transducers.* Springer, 1998.

[FV09] Zoltán Fülöp and Heiko Vogler. Weighted tree automata and tree transducers. In Werner Kuich, Manfred Droste, and Heiko Vogler, editors, *Handbook of Weighted Automata.* Springer, 2009.

[GK00] Thomas Genet and Francis Klay. Rewriting for cryptographic protocol verification. In D.A. McAllester, editor, *Proc. 17th International Conference on Automated Deduction (CADE'00)*, volume 1831 of *Lecture Notes in Computer Science*, pages 271–290, 2000.

[GKM08] Jonathan Graehl, Kevin Knight, and Jonathan May. Training tree transducers. *Computational Linguistics*, 34(3):391–427, 2008.

[GS84] Ferenc Gécseg and Magnus Steinby. *Tree Automata.* Akadémiai Kiadó, Budapest, 1984.

[GS97] Ferenc Gécseg and Magnus Steinby. Tree languages. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages.* Vol. 3: *Beyond Words*, chapter 1, pages 1–68. Springer, 1997.

[HKL93] Annegret Habel, Hans-Jörg Kreowski, and Clemens Lautemann. A comparison of compatible, finite, and inductive graph properties. *Theoretical Computer Science*, 110:145–168, 1993.

[HKT93] Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Collages and patterns generated by hyperedge replacement. *Languages of Design*, 1:125–145, 1993.

[HKV91] Annegret Habel, Hans-Jörg Kreowski, and Walter Vogler. Decidable boundedness problems for sets of graphs generated by hyperedge-replacement. *Theoretical Computer Science*, 89:33–62, 1991.

[Kaa08] Lisa Kaati. Reduction Techniques for Finite (Tree) Automata. Doctoral dissertation, Uppsala University, Sweden, 2008.

[KG05] Kevin Knight and Jonathan Graehl. An overview of probabilistic tree transducers for natural language processing. In Alexander F. Gelbukh, editor, *Proc. 6th Intl. Conf. on Computational Linguistics and Intelligent Text Processing (CICLing 2005)*, volume 3406 of Lecture Notes in Computer Science, pages 1–24. Springer, 2005.

[Kui00] Werner Kuich. Linear systems of equations and automata on distributive multioperator monoids. In D. Dorninger, G. Eigenthaler, M. Goldstern, H.K. Kaiser, W. More, and W.B. Müller, editors, *Proc. 58th Workshop on General Algebra (1999)*, volume 12 of *Contributions to General Algebra*, pages 247–256, Klagenfurt, 2000. Johannes Heyn.

[Löd02] Christof Löding. Model-checking infinite systems generated by ground tree rewriting. In M. Nielsen and U. Engberg, editors, *Proc. 5th Intl. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'02)*, volume 2303 of *Lecture Notes in Computer Science*, pages 280–294, 2002.

[MW67] Jorge Mezei and Jesse B. Wright. Algebraic automata and context-free sets. *Information and Control*, 11:3–29, 1967.

[NP92] Maurice Nivat and Andreas Podelski, editors. *Tree Automata and Languages*. Elsevier, Amsterdam, 1992.

[Sch07] Thomas Schwentick. Automata for XML - a survey. *Journal of Computer and System Sciences*, 73(3):289–315, 2007.

Dr. Frank Drewes

Institutionen för datavetenskap
Umeå universitet
S-90187 Umeå (Sweden)
drewes@cs.umu.se
http://www.cs.umu.se/~drewes

Frank Drewes studied Computer Science at the University of Bremen. He was introduced to Theoretical Computer Science by Hans-Jörg Kreowski and Dr. Clemens Lautemann who, during those years, was a member of Hans-Jörg's team. After his graduation in 1990, Frank became a doctoral student supervised by Hans-Jörg. After the defense in 1996, he worked as an assistant professor in Hans-Jörg's team until he accepted an offer from the University of Umeå in 2000.

Processes Based on Biochemical Interactions: Natural Computing Point of View*

Andrzej Ehrenfeucht and Grzegorz Rozenberg

Introduction

In this paper we investigate the interactions between biochemical reactions from the natural computing point of view. Natural computing (see, e.g., [6, 7]) is concerned with human-designed computing inspired by nature and with computation taking place in nature (i.e., it investigates processes taking place in nature in terms of information processing). The former strand of research is quite well-established: some of the well-known examples are evolutionary computing, neural computing, cellular automata, swarm intelligence, molecular computing, quantum computing, artificial immune systems, and membrane computing. Examples of research themes from the latter strand of research are: computational nature of self-assembly, computational nature of developmental processes, computational nature of bacterial communication, computational nature of brain processes, computational nature of biochemical reactions, and system biology approach to bionetworks. A lot of research from this research strand underscores the fact that computer science is also the fundamental science of information processing, and as such a basic science for other scientific disciplines such as, e.g., biology.

This paper is concerned with the computational nature of processes driven by interactions between biochemical reactions in living cells. It presents a formal framework for investigating such processes, called the framework of *reaction systems* (see, e.g., [1–4]). In particular, it provides basic definitions together with the intuition/motivation behind them. The paper is of a tutorial and rather informal style – the reader is advised to consult the references provided in the paper for a precise formal treatment of reaction systems.

1 Reactions

The functioning of a biochemical reaction is based on facilitation and inhibition: a reaction can take place if all of its reactants are present and none of its inhibitors is present. If a reaction takes place, then it creates its product. Therefore to specify a reaction one needs to specify its set of reactants, its set of inhibitors, and its set of products – this leads to the following definition.

* This paper is dedicated to Hans-Joeg Kreowski on the occasion of his 60th birthday.

Definition 1. *A* reaction *is a triplet $a = (R, I, P)$, where R, I, P are finite sets. If S is a set such that $R, I, P \subseteq S$, then a is a* reaction in S.

The sets R, I, P are also denoted by R_a, I_a, P_a, and called the *reactant set of* a, the *inhibitor set of* a, and the *product set of* a, respectively. Also, $rac(S)$ denotes the set of all reactions in S.

For a finite set of reactions A, $R_A = \bigcup_{a \in A} R_a$, $I_A = \bigcup_{a \in A} I_a$, and $P_A = \bigcup_{a \in A} P_a$ are called the *reactant set of* A, the *inhibitor set of* A, and the *product set of* A, respectively.

The effect of a reaction a is conditional: if R_a is present and no element of I_a is present then P_a is produced, otherwise reaction does not take place and "nothing" is produced. This is formalised as follows.

Definition 2. *Let a be a reaction, A a finite set of reactions, and T a finite set.*

(1) a is enabled by T, *denoted by a en T, if $R_a \subseteq T$ and $I_a \cap T = \emptyset$.*

(2) The result of a on T, *denoted by $res_a(T)$, is defined by: $res_a(T) = P_a$ if a en T, and $res_a(T) = \emptyset$ otherwise.*

(3) The result of A on T, *denoted by $res_A(T)$, is defined by:*
$res_A(T) = \bigcup_{a \in A} res_a(T)$.

Clearly, if $R_a \cap I_a \neq \emptyset$, then $res_a(T) = \emptyset$ for every T. Therefore we assume that, for each reaction a, $R_a \cap I_a = \emptyset$; in this paper we will also assume that $R_a \neq \emptyset, I_a \neq \emptyset$, and $P_a \neq \emptyset$.

As an example consider the reaction a with $R_a = \{c, x_1, x_2\}$, $I_a = \{y_1, y_2\}$, and $P_a = \{c, z\}$. We can interpret c as the catalyzer of a (it is needed for a to take place, but is not "consumed" by a), x_1, x_2 as "real" reactants, y_1, y_2 as inhibitors (e.g., acids inhibiting the functioning of c as the catalyzer), and z as the compound that is produced by this reaction. Then a *en* T for $T = \{c, x_1, x_2, z\}$, and a is not enabled on neither $\{c, x_1, x_2, z, y_1\}$ nor on $\{x_1, x_2, z\}$.

An important notion is the *activity* of a set of reactions A on a finite set (state) T – it is denoted by $en_A(T)$, and defined by: $en_A(T) = \{a \in A : a \text{ } en \text{ } T\}$. Hence $en_A(T)$ is the set of all reactions from A that are enabled by (active on) T. Note that $res_A(T) = res_{en_A(T)}(T)$: only the reactions from A which are enabled on T contribute to the result of A on T.

2 Basic Assumptions and Intuition

We will discuss now in more detail the basic notions of enabling and application (result) of reactions and sets of reactions, as they reflect our assumptions about biochemical reactions (motivated by organic chemistry of living organisms), which are very different from the underlying assumptions of majority of models (of human-designed systems) in theoretical computer science.

A reaction a is enabled on a set T if T *separates* R_a from I_a (i.e., $R_a \subseteq T$ and $I_a \cap T = \emptyset$). We make no assumption about the relationship of P_a to either R_a

or I_a. When a is enabled by a finite set T, then $res_a(T) = P_a$. Thus the result of a on T is *"locally determined"* in the sense that it uses only a subset of T, viz., the set of reactants R_a. However the result of the transformation is global: in comparing T with P_a we note that all elements from $T - P_a$ "vanished". This is in great contrast to classical models in theoretical computer science; e.g., in Petri nets (see, e.g., [5]) the firing of a single transition has only a local influence on the global marking which may be changed only on places that are neighbouring the given transition. Our way of defining the result of a reaction on a state T reflects our assumption that there is *no permanency* of elements: an element (molecule) of a global state vanishes unless it is sustained by a reaction.

The result of applying a set of reactions A to a state T is cumulative: it is the union of results of individual reactions from A. We do not set any conditions on the relationship between reactions in A. In particular, we do not have the (standard) notion of conflict here: if $a, b \in A$ with $a\ en\ T$ and $b\ en\ T$, then, even if $R_a \cap R_b \neq \emptyset$, still both a and b contribute to $res_A(T)$, i.e., $(res_a(T) \cup res_b(T)) \subseteq res_A(T)$. Such a conflict of resources (standard in classical models such as, e.g., Petri nets) does not exist here. There is no counting in reaction systems, and so we deal with a qualitative rather than a quantitative model. This reflects our assumption about the "threshold supply" of elements (molecules): either an element is present, and then there is "enough" of it, or an element is not present.

There is a notion in reaction systems that reflects an intuition of conflict, viz., the notion of consistency. A set of reactions A is called *consistent* if $R_A \cap I_A = \emptyset$, i.e., $R_a \cap I_b = \emptyset$ for any two reactions $a, b \in A$; clearly if $R_a \cap I_b \neq \emptyset$, then a and b can never be *together* enabled.

3 Reaction Systems and Interactive Processes

We are ready now to define reaction systems.

Definition 3. *A* reaction system*, abbreviated rs, is an ordered pair $\mathcal{A} = (S, A)$ such that S is a finite set, and $A \subseteq rac(S)$.*

The set S is called the *background set of* $\mathcal{A}$, and A is called the *set of reactions of* $\mathcal{A}$. All the notions and notations introduced for sets of reactions carry over to reaction systems through their underlying sets of reactions. For example, for $T \subseteq S, en_{\mathcal{A}}(T) = en_A(T)$ and $res_{\mathcal{A}}(T) = res_A(T)$ – also, we say that T is *active in* $\mathcal{A}$, if $en_{\mathcal{A}}(T) \neq \emptyset$.

It is important to note here that, in the setup of reaction systems, reactions are primary while structures are secondary. Since we do not have permanency of elements – elements vanish unless they are sustained by reactions, (sets of) reactions *create* states rather than *transform* states. Thus reaction systems do not work in an environment, but rather they create an environment.

The interactions of reaction systems is given by unions. For reaction systems $\mathcal{A}_1 = (S_1, A_1)$ and $\mathcal{A}_2 = (S_2, A_2)$ their union, denoted $\mathcal{A}_1 + \mathcal{A}_2$, is defined by $\mathcal{A}_1 + \mathcal{A}_2 = (S_1 \cup S_2, A_1 \cup A_2)$. This way of combining reaction systems reflects the bottom-up modularity: local descriptions (reaction systems $\mathcal{A}_1, \mathcal{A}_2$) are combined into global picture $(\mathcal{A}_1 + \mathcal{A}_2)$ in such a way that the interactions of local descriptions is provided automatically. Thus a major difference with standard models in theoretical computer science is that *no interface* is given/needed for combining reaction systems: the sheer fact that the sets of reactions A_1, A_2 operate in the same molecular soup (tube) causes A_1, A_2 to interact (again through facilitation and inhibition). Thus union is the basic mechanism for composing/decomposing reaction systems.

The dynamic behaviour of reaction systems is captured through the notion of an interactive process which is formally defined as follows.

Definition 4. *Let $\mathcal{A} = (S, A)$ be a rs. An* interactive proces *in $\mathcal{A}$ is a pair $\pi = (\gamma, \delta)$ of finite sequences such that $\gamma = C_0, C_1, ...C_n$, $\delta = D_1, ..., D_n, n \geq 1$, where $C_0, ..., C_n$, $D_1, ..., D_n \subseteq S$, $D_1 = res_{\mathcal{A}}(C_0)$, and $D_i = res_{\mathcal{A}}(D_{i-1} \cup C_{i-1})$ for each $2 \leq i \leq n$.*

The sequence $C_0, ..., C_n$ is the *context sequence* of π, and the sequence $D_1, ..., D_n$ is the *result sequence* of π. Let $W_0 = C_0$, and $W_i = D_i \cup C_i$ for all $1 \leq i \leq n$. Then the sequence $W_0, ..., W_n$ is the *state sequence of* π, denoted $sts(\pi)$, and W_0 is the initial state of π. For each $0 \leq j \leq n$, C_j is the *context of* W_j. The sequence $E_0, ..., E_{n-1}$ of subsets of A such that $E_i = en_A(W_i)$, for all $0 \leq i \leq n-1$, is the *activity sequence of* π, denoted $act(\pi)$. If $act(\pi)$ consists of nonempty sets only, then $sts(\pi)$ is *active* – in this case all states $W_1, ..., W_{n-1}$ are active. The set of all state sequences of (all interactive processes in) $\mathcal{A}$ is denoted by $STS(\mathcal{A})$.

The basic intuition behind the notion of an interactive process is rather straightforward. Context C_0 represents the initial state of π, i.e., the state in which π begins (is initiated), and the contexts $C_1, ..., C_n$ represent the influence of (the interaction with) the "rest of the world". Then D_1 is the result of $\mathcal{A}$ on C_0, i.e., the result of applying to C_0 all the reactions from $\mathcal{A}$ enabled on C_0. Together with context C_1, D_1 forms the successor state W_1 of the initial state. Then, iteratively, the result of applying $\mathcal{A}$ to state $W_{i-1} = D_{i-1} \cup C_{i-1}$ yields the result D_i which together with the context C_i forms the successor state W_i. Note that even if $D_i = \emptyset$, W_i can still be an active state (if $en_{\mathcal{A}}(C_i) \neq \emptyset$). The definition of an interactive process is illustrated in Figure 1.

4 Extended Reaction Systems

Reaction systems form the basic construct of the broad "framework of reaction systems". However, within this framework we use an "onion approach" meaning that additional levels/components can be incrementally added (or removed) so that the resulting model is well fitted for the research issue at hand. An example of such an (incremental) approach are extended reaction systems which are suitable for investigating the issue of emergence of modules in biochemical systems,

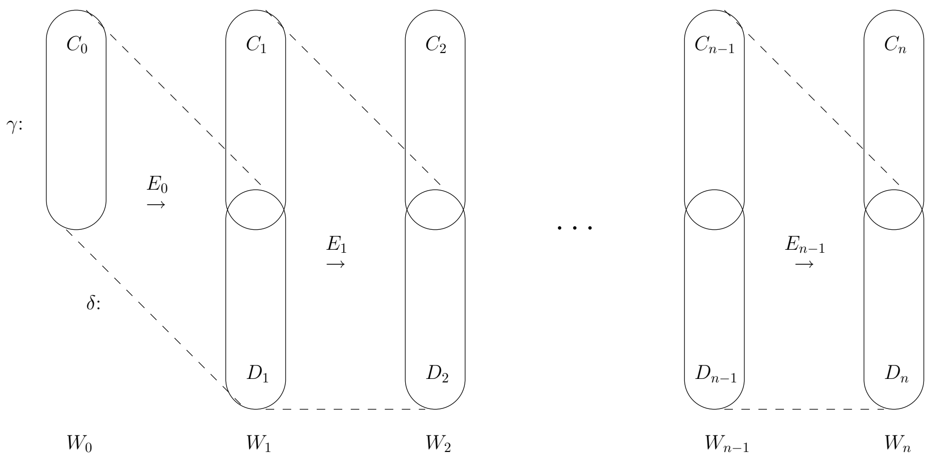


Fig. 1. An interactive process.

investigated in [3] and presented in the next section. We use the notation 2^S to denote the set of subsets of a set S.

Definition 5. *An* extended reaction system*, abbreviated ers, is a triplet* $\mathcal{A} = (S, A, R)$ *such that* (S, A) *is a reaction system, and* R *is a binary relation,* $R \subseteq 2^S \times 2^S$*.*

We refer to (S, A) as the *underlying reaction system of* $\mathcal{A}$ denoted by $und(\mathcal{A})$.

The role of the restriction relation is to restrict the set of interactive processes as follows. An interactive process of $\mathcal{A}$ is an interactive process $\pi = (\gamma, \delta)$ of $und(\mathcal{A})$ such that if $sts(\pi) = W_0, W_1, \ldots, W_n$, then, for each $0 \leq i \leq n-1$, $(W_i, W_{i+1}) \in R$. Thus interactive processes of $\mathcal{A}$ are these interactive processes of $und(\mathcal{A})$, where each two consecutive states in the state sequence are related (allowed) by R. We also require that the restriction relation is not too restrictive, i.e., that for each state sequence $W_0, W_1, ..., W_n$ of $\mathcal{A}$ there exists $W_{n+1} \subseteq S$ such that $W_0, W_1, ..., W_n, W_{n+1}$ is also a state sequence of $\mathcal{A}$. In other words, each interactive process of $\mathcal{A}$ can be extended, as is the case in reaction systems.

A distinct feature of extended reaction systems is the existence of periodic elements – such elements cannot exist in reaction systems. An element t of an *ers* $\mathcal{A}$ is *periodic* (*in* $\mathcal{A}$) if there exists a positive integer n such that for each $W_0, W_1, ..., W_n \in STS(\mathcal{A}), t \in W_0$ if and only if $t \in W_n$; the smallest such n is called the *period of* t. Hence, if t is a periodic element with period $n, W_0, W_1, ..., W_q \in STS(\mathcal{A})$, and $0 \leq i \leq q$, then if $t \in W_i$ then also $t \in W_{i-n}$ (providing that $i - n \geq 0$) and $t \in W_{i+n}$ (providing that $i + n \leq q$). The set of all periodic elements of $\mathcal{A}$ is denoted by $per(\mathcal{A})$, and for each $T \subseteq S$, $per_{\mathcal{A}}(T) = T \cap per(\mathcal{A})$ is the set of periodic elements of T.

The existence of periodic elements motivates the following definition of computing the images of subsets of a given state (of an interactive process) in the successor state. Let $\mathcal{A} = (S, A, R)$ be an *ers*, let $\tau = W_0, W_1, ..., W_n \in STS(\mathcal{A})$, let $i \in \{0, ..., n-1\}$, and let $Q \subseteq W_i$. Then the *image of* Q *in* W_{i+1} (*within* τ), denoted by $im_{\mathcal{A},\tau,i}(Q)$, is defined by $im_{\mathcal{A},\tau,i}(Q) = res_{E_i}(Q \cup per_{\mathcal{A}}(W_i)) - per_{\mathcal{A}}(res_{E_i}(W_i))$. The intuition behind this definition of an image is as follows: since periodic elements are included in fixed states of state sequences "independently of the applied reactions" (i.e., we can predict/compute the states of a state sequence where a periodic element belongs without knowing reactions that are actually applied to states), they are added to a "real argument" (i.e., Q) of the res_{E_i} when computing $im_{\mathcal{A},\tau,i}$. For the same reason we substract the periodic elements of $res_{E_i}(W_i)$, because we want in the image of Q only the "real results" (which excludes elements from $per_{\mathcal{A}}(res_{E_i}(W_i))$ which will be in W_{i+1} anyhow because of their periodicity).

5 Events and Modules

Among all the subsets of a state of an interactive process we will distinguish "material subsets" – these are subsets that are the result of applying reactions of a system to subsets of the predecessor state. More formally, let $\tau = W_0, W_1, \ldots, W_n$ be a state sequence of an *ers* $\mathcal{A}$, and let us consider state W_i for some $1 \leq i \leq n$. A subset $X \subseteq W_i$ is a "material subset" of W_i if there exists a subset $Y \subseteq W_{i-1}$ such that X is the product of the set of reactions enabled on W_{i-1} applied to Y. Such products included in W_i are "modules" of W_i. If we now consider the sequence of modules in consecutive states of τ initiated by $Y \subseteq W_{i-1}$, beginning in W_i (with X) and ending in some W_j for $j \geq i$, then we are tracing the fate of Y (as a sequence of products) through $(j-i+1)$ steps of (an interactive process π behind) τ. Such sequences of modules are called events which are formally defined below. If we are interested in a module Q in some W_k, for $1 \leq k \leq n$, and follow backwards an event that produced Q in W_k, then we get a possible history of Q, hence an explanation of why and how Q was created in W_k.

Definition 6. *Let $\mathcal{A}$ be an ers, let $\tau = W_0, W_1, \ldots, W_n \in STS(\mathcal{A})$, let $i, j \in \{1, \ldots, n\}$ be such that $i \leq j$, and let $\omega = Q_i, \ldots, Q_j$ be such that $Q_i \subseteq W_i, \ldots, Q_j \subseteq W_j$, and all $Q_i, \ldots, Q_{j-1}$ are nonempty. Then ω is* an event *in τ if there is a $Q_{i-1} \subseteq W_{i-1}$ such that, for each $k \in \{i, \ldots, j\}$, $Q_k = im_{\mathcal{A},\tau,k-1}(Q_{k-1})$.*

We say that ω is *passing through* each of $W_i, \ldots, W_j$; if $Q_j = \emptyset$, then ω *dies in* W_j. The sets $Q_i, \ldots, Q_j$ are called the *modules of* ω *in* $W_i, \ldots, W_j$, respectively. More specifically, each module Q_l, $i \leq l \leq j$, is called a *l-module.*

Thus, intuitively, an event (ω) is tracing the fate of a subset (Q_{i-1}) of a state (W_{i-1}) of a state sequence τ within a segment $(W_i, \ldots, W_j)$ of τ. More specifically, suppose that we are interested in a state sequence τ (or in an interactive process π with $sts(\pi) = \tau$), and in particular we are interested in the dynamic development of some $Q_{i-1} \subseteq W_{i-1}$ as τ evolves from W_i on until W_j is reached. This dynamic development of Q_{i-1} in the segment $W_i, \ldots, W_j$ is the sequence $Q_i, \ldots, Q_j$ of material subsets (modules) of $W_i, \ldots, W_j$, respectively. Note that both the notion of the result of transforming Q_l into Q_{l+1}, $l \in \{i, \ldots, j-1\}$, and the notion of a material subset are modified (w.r.t. reaction systems) to take into account the existence of periodic elements in extended reaction systems.

When an event ω is passing through a state W_l then it leaves a "trace" there, viz., its module Q_l. The set of all such traces in W_l left there by all events passing through W_l is called the *snapshot of* W_l *in* τ, denoted by $snp_\tau(k)$. Thus for the given state sequence $\tau = W_0, \ldots, W_n$ we get the corresponding sequence of snapshots $snp(\tau) = \mathcal{S}_1, \ldots, \mathcal{S}_n$, where $\mathcal{S}_i = snp_\tau(i)$ for each $1 \leq i \leq n$, called the *snapshot sequence of* τ, and also called the *snapshot sequence of* $\mathcal{A}$.

Given a snapshot sequence $\rho = \mathcal{S}_1, \ldots, \mathcal{S}_n$ of a state sequence $\tau = W_0, \ldots, W_n$ there exists a natural sequence of partial functions $next_{\tau,1}, next_{\tau,2}, \ldots, next_{\tau,n-1}$ transforming consecutive snapshots of ρ into their successor snapshots, where, for each $1 \leq k \leq n-1$, $next_{\tau,k} : \mathcal{S}_k \to \mathcal{S}_{k+1}$ is defined as follows. For $Q \in \mathcal{S}_k$ and $Q' \in \mathcal{S}_{k+1}$, $next_{\tau,k}(Q) = Q'$ if and only if Q, Q' are nonempty and there exists an event ω in τ such that Q is the module of ω in W_k and Q' is the module of ω in W_{k+1}. If we extend the $next_{\tau,k}$ function also to pairs (Q, Q') with Q' possibly empty, then the resulting function is denoted by $suc_{\tau,k}$. Thus, intuitively, the function $next_{\tau,k}$ connects nonempty modules that are consecutive in an event passing through W_k and W_{k+1}. In this way the sequence of functions $next_{\tau,1}, \ldots, next_{\tau,n-1}$ delineate all the events of τ as they are passing through the states of τ, but it does not explicitly indicate the "moment of death" (if an event dies). The sequence of functions $suc_{\tau,1}, \ldots, suc_{\tau,n-1}$ does indicate also the death moments. As a matter of fact the empty module has really no physical interpretation – it is clearly no material subset, but rather its role is to signal the termination (the death) of an event. It is therefore convenient to consider snapshots with the empty set removed. In this way, for a given snapshot sequence $\rho = \mathcal{S}_1, \ldots, \mathcal{S}_n$ we obtain its $\emptyset$-free version $\bar{\rho} = \bar{\mathcal{S}}_1, \ldots, \bar{\mathcal{S}}_n$, where for each $1 \leq i \leq n$, $\bar{\mathcal{S}}_i = \mathcal{S}_i - \{\emptyset\}$. Accordingly, each $next_{\tau,k}$ function is modified to the $rnext_{\tau,k}$ function which is $next_{\tau,k}$ restricted to $\bar{\mathcal{S}}_k$.

We move now to present the structure of snapshots. First we need a couple of set-theoretical notions.

Definition 7. *Let $\mathcal{L}$ be a family of sets and let $\mathcal{F}_1, \mathcal{F}_2 \subseteq \mathcal{L}$ be nonempty.*

(1) We say that $\mathcal{F}_1$ is embedded *in $\mathcal{F}_2$ if $\bigcup \mathcal{F}_1 \subseteq \bigcap \mathcal{F}_2$.*

(2) We say that $\mathcal{F}_1$ is separated *from $\mathcal{F}_2$* in *$\mathcal{L}$ if there exists $U \in \mathcal{L}$ such that $\bigcup \mathcal{F}_1 \subseteq U \subseteq \bigcap \mathcal{F}_2$.*

Theorem 1. *Let $\mathcal{A}$ be an ers, let $\tau = W_0, W_1, \ldots, W_n \in STS(\mathcal{A})$ where $n \geq 2$, let $snp(\tau) = \mathcal{S}_1, \ldots, \mathcal{S}_n$, and let $1 \leq k \leq n-1$. If $\mathcal{F}_1, \mathcal{F}_2 \subseteq \bar{\mathcal{S}}_k$ are nonempty families of sets such that $\mathcal{F}_1$ is embedded in $\mathcal{F}_2$ and $next_{\tau,k}$ is defined on all modules in $\mathcal{F}_1 \cup \mathcal{F}_2$, then $next_{\tau,k}(\mathcal{F}_1)$ is separated from $next_{\tau,k}(\mathcal{F}_2)$ in $\bar{\mathcal{S}}_{k+1}$.*

This is a remarkable result as it allows us to view (extended) reaction systems as self-organizing systems, where a possible goal of interactive processes is to ensure (improve on) separability!

An interactive process (hence a run) of an *ers* $\mathcal{A}$ produces a sequence ρ of snapshots $\mathcal{S}_1, \ldots, \mathcal{S}_k, \ldots, \mathcal{S}_n$. In general such a sequence may be very "unstable" because there may be no "mathematical similarity" between $\mathcal{S}_k$ and $\mathcal{S}_{k+1}$: remember that the context of the state W_{k+1} (in the state sequence $\tau = W_0, \ldots, W_n$ for which $\rho = snp(\tau)$) can "throw anything" into W_{k+1}. So we can talk about local stability (at W_k) only if there is a strong mathematical similarity between $\mathcal{S}_k$ and $\mathcal{S}_{k+1}$. Perhaps the most natural choice for such a strong similarity is to require that $rnext_{\tau,k}$ is an isomorphism between partial orders $(\bar{\mathcal{S}}_k, \subseteq)$ and $(\bar{\mathcal{S}}_{k+1}, \subseteq)$. When this happens, we get a local stability – it is local because, again, "anything can happen" to $\mathcal{S}_{k+2}$ (through the context of W_{k+2}). Hence we say that $(\mathcal{S}_k, \mathcal{S}_{k+1})$ is a *locally stable situation* if $rnext_k$ is an isomorphism between $(\bar{\mathcal{S}}_k, \subseteq)$ and $(\bar{\mathcal{S}}_{k+1}, \subseteq)$. We want to point out that the situation is quite subtle here, e.g., the fact that $\bar{\mathcal{S}}_k = \bar{\mathcal{S}}_{k+1}$ does not necessarily imply that $rnext_{\tau,k}$ is an isomorphism of $\bar{\mathcal{S}}_k$ onto $\bar{\mathcal{S}}_{k+1}$.

It turns out that local stability is reflected in the structure of the corresponding snapshots.

Theorem 2. *Let $\mathcal{A}$ be an ers, let $\tau \in STS(\mathcal{A})$, and let $\mathcal{S}, \mathcal{S}'$ be two consecutive elements of $snp(\tau)$. If $(\mathcal{S}, \mathcal{S}')$ is a locally stable situation, then both $(\mathcal{S}, \subseteq)$ and $(\mathcal{S}', \subseteq)$ are complete lattices.*

6 Discussion

We have presented in this paper an informal introduction to the framework of reaction systems. It is motivated by organic chemistry of living cells, and more specifically by interactions between biochemical reactions. The basic notions here are reactions and their results, i.e., the way they process states – this way of processing the states of a system is very different from the manner that state processing happens in common models in theoretical computer science. The differences (and motivation between them) are discussed in detail in this paper. The basic model of our framework are reaction systems and the basic notion/tool to investigate their dynamics is an interactive process. Although reaction systems form the core of our framework, the framework is constructed in an "incremental" way: depending on a research issue the notion of reaction system can be modified so that the resulting model is well-suited for the investigation of the given research issue. For example, reaction systems form a qualitative model where we do not have counting (of elements), as is the case for models based on multisets rather than on sets. However there are many situations where one needs to assign quantitative parameters to states (e.g., when dealing with time issues). Our point of view is that a numerical value can be assigned to a state T if there is a measurement of T yielding this value. This leads to the notion of *reaction systems with measurements*, where a finite set of measurement functions is added as a third component to reaction systems (see [4]).

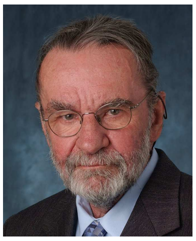
Another example of research leading to an incremental modification of the notion of a reaction system, is the investigation of the way that the products are formed and evolve within the runs of biochemical systems. The resulting extended reaction systems and the formation of products (the topics of [3]) are discussed in detail in this paper. The basic dynamic notion here is the notion of an event which traces the formation of modules (products) within interactive processes of a system. The rather surprising results that (extended) reaction systems can be seen as self-organising systems which in stable situations produce well-structured sets of molecules are also presented.

Altogether this paper presents both the basic setup, its motivation, and some typical research themes and results of the framework of reaction systems.

Acknowledgements The authors are indebted to Robert Brijder and Hendrik Jan Hoogeboom for comments on this paper, and to Robert Brijder and Marloes van der Nat for their help in producing this paper.

References

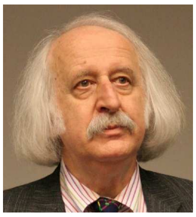
1. Ehrenfeucht, A., Rozenberg, G., Basic notions of reaction systems, Lecture Notes in Computer Science, v. 3340, 27-29, Springer, 2004.
2. Ehrenfeucht, A., Rozenberg, G., Reaction systems, Fundamenta Informaticae, v. 75, 263-280, 2007.
3. Ehrenfeucht, A., Rozenberg, G., Events and modules in reaction systems, Theoretical Computer Science, v. 376, 3-16, 2007.
4. Ehrenfeucht, A., Rozenberg, G., Introducing time in reaction systems, Theoretical Computer Science, v. 410, 310-322, 2009.
5. Engelfriet, J., Rozenberg, G., Elementary net systems, Lecture Notes in Computer Science, v. 1491, 12-121, Springer, 1998.
6. Kari, L., Rozenberg, G., The many facets of natural computing, Communications of the ACM, v. 51, 72-83, 2008.
7. Rozenberg, G., Computer science, informatics, and natural computing – personal reflections, in S.B. Cooper, B. Löwe, A. Sorbi, eds., *New Computational Paradigms – Changing Conceptions of What is Computable*, Springer, Berlin, Heidelberg, 2007.



Prof. Dr. Andrzej Ehrenfeucht

Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430 (U.S.A.)
andrzej@cs.colorado.edu
http://www.cs.colorado.edu/~andrzej

Prof. Dr. Grzegorz Rozenberg



Department of Computer Science
University of Colorado at Boulder
Boulder, CO 80309-0430 (U.S.A.)
and
Leiden Institute of Advanced Computer Science,
Leiden Center for Natural Computing,
Leiden University, NL-2333 CA Leiden (The Netherlands)
rozenber@liacs.nl
http://www.liacs.nl/~rozenber

Grzegorz Rozenberg has been knowing Hans-Jörg Kreowski for about 30 years now. For the first time, they met during one of Grzegorz' visits to Hartmut Ehrig in Berlin. Since then they wrote a number of joint papers, the first one already in 1981. They also participated in a number of joint European research projects.
Hans-Jörg visited Grzegorz many times in Bilthoven, and Grzegorz visited him in Berlin and Bremen. They have become good family friends, with active mutual interest in the artistic careers of their sons Daniel and Kai, and look forward to many more meetings to come.

Parallelism and Concurrency Theorems for Rules with Nested Application Conditions

Hartmut Ehrig, Annegret Habel, and Leen Lambers

Abstract. We present Local Church-Rosser, Parallelism, and Concurrency Theorems for rules with nested application conditions in the framework of weak adhesive HLR categories including different kinds of graphs. The proofs of the statements are based on the corresponding statements for rules without application conditions and two Shift-Lemmas, saying that nested application conditions can be shifted over morphisms and rules.

Keywords: High-level transformation systems, weak adhesive HLR categories, parallelism, concurrency, nested application conditions, negative application conditions.

1 Introduction

Graph replacement systems have been studied extensively and applied to several areas of computer science [1,2,3] and were generalized to high-level replacement (HLR) systems [4] and weak adhesive HLR systems [5,6]. Application conditions restrict the applicability of a rule. Originally, they were defined in [7], specialized to negative application conditions (NACs) [8], and generalized to nested application conditions (ACs) [9].

The Local Church-Rosser, Parallelism, and Concurrency Theorems are well-known theorems for graph replacement systems on rules without application conditions [10,11,12,13,14,15] and are generalized to high-level replacement (HLR) systems [4] and rules with negative application conditions [16]. Nested application conditions (ACs) were introduced in [9] and intensively studied in [17]. They generalize the well-known negative application conditions (NACs) in the sense of [8,16] and are expressively equivalent to first order formulas on graphs. In this paper, we generalize the theorems to weak adhesive HLR systems on rules with nested application conditions.

Theorem	without ACs	with NACs	with ACs
Local Church-Rosser	[10,13,4,6]	[8,16]	this paper
Parallelism	[11,12,4,6]	[8,16]	this paper
Concurrency	[14,15,4,6]	[16]	this paper

The proofs of the theorems are based on the corresponding theorems for weak adhesive HLR systems on rules without application conditions in [6] and facts on nested application conditions in [17], saying that application conditions can be shifted over morphisms and rules.

$$\boxed{\text{Theorem + Shift-Lemmas for ACs} \Rightarrow \text{Theorem for rules with ACs}}$$

The paper is organized as follows: In Sections 2 and 3, we review the definitions of a weak adhesive HLR category, nested conditions, and rules. In Section 4, we state and prove the Local Church-Rosser, Parallelism, and Concurrency Theorems for rules with nested application conditions. The concepts are illustrated by examples in the category of graphs with the class $\mathcal{M}$ of all injective graph morphisms. A conclusion including further work is given in Section 5.

2 Graphs and high-level structures

We recall the basic notions of directed, labeled graphs [13,18] and generalize them to high-level structures [4]. The idea behind the consideration of high-level structures is to avoid similar investigations for similar structures such as Petri-nets and hypergraphs.

Directed, labeled graphs and graph morphisms are defined as follows.

Definition 1 (graphs and graph morphisms). Let $\mathrm{C} = \langle \mathrm{C_V}, \mathrm{C_E} \rangle$ be a fixed, finite label alphabet. A *graph* over C is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_G, \mathrm{m}_G)$ consisting of two finite sets V_G and E_G of *nodes* (or *vertices*) and *edges*, *source* and *target functions* $\mathrm{s}_G, \mathrm{t}_G \colon \mathrm{E}_G \to \mathrm{V}_G$, and two *labeling functions* $\mathrm{l}_G \colon \mathrm{V}_G \to \mathrm{C_V}$ and $\mathrm{m}_G \colon \mathrm{E}_G \to \mathrm{C_E}$. A graph with an empty set of nodes is *empty* and denoted by $\emptyset$. A *graph morphism* $g \colon G \to H$ consists of two functions $g_\mathrm{V} \colon \mathrm{V}_G \to \mathrm{V}_H$ and $g_\mathrm{E} \colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve sources, targets, and labels, that is, $\mathrm{s}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{t}_G$, $\mathrm{l}_H \circ g_\mathrm{V} = \mathrm{l}_G$, and $\mathrm{m}_H \circ g_\mathrm{E} = \mathrm{m}_G$. A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is both injective and surjective. The *composition* $h \circ g$ of g with a morphism $h \colon H \to M$ consists of the composed functions $h_\mathrm{V} \circ g_\mathrm{V}$ and $h_\mathrm{E} \circ g_\mathrm{E}$.

Our considerations are based on weak adhesive HLR categories, i.e. categories based on objects of many kinds of structures which are of interest in computer science and mathematics, e.g. Petri-nets, (hyper)graphs, and algebraic specifications, together with their corresponding morphisms and with specific properties. Readers interested in the category-theoretic background of these concepts may consult e.g. [6].

Definition 2 (weak adhesive HLR category). A category $\mathcal{C}$ with a morphism class $\mathcal{M}$ is a *weak adhesive HLR category*, if the following properties hold:

1. $\mathcal{M}$ is a class of monomorphisms closed under isomorphisms, composition, and decomposition. I.e. for morphisms $g \circ f$: $f \in \mathcal{M}$, g isomorphism (or vice versa) implies $g \circ f \in \mathcal{M}$; $f, g \in \mathcal{M}$ implies $g \circ f \in \mathcal{M}$; and $g \circ f \in \mathcal{M}$, $g \in \mathcal{M}$ implies $f \in \mathcal{M}$.
2. $\mathcal{C}$ has pushouts and pullbacks along $\mathcal{M}$-morphisms, i.e. pushouts and pullbacks, where at least one of the given morphisms is in $\mathcal{M}$, and $\mathcal{M}$-morphisms are closed under pushouts and pullbacks, i.e. given a pushout (1) as in the figure below, $m \in \mathcal{M}$ implies $n \in \mathcal{M}$ and, given a pullback (1), $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.

3. Pushouts in $\mathcal{C}$ along $\mathcal{M}$-morphisms are weak VK-squares, i.e. for any commutative cube in $\mathcal{C}$ where we have the pushout with $m \in \mathcal{M}$ and ($f \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$) in the bottom and the back faces are pullbacks, it holds: the top is pushout iff the front faces are pullbacks.

$$\begin{tikzcd}
A \arrow[r] \arrow[d, hook, "m"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, hook, "n"] \\
B \arrow[r] & D
\end{tikzcd}
\qquad
\begin{tikzcd}
& A' \arrow[rr] \arrow[dl] \arrow[dd] & & C' \arrow[dl] \arrow[dd, "c"] \\
B' \arrow[rr, crossing over] \arrow[dd, "b"'] & & D' & \\
& A \arrow[rr, "f"] \arrow[dl, "m"'] & & C \arrow[dl] \\
B \arrow[rr] & & D \arrow[from=uu, crossing over, "d"'] &
\end{tikzcd}$$

Fact 1. The category $\langle \text{Graphs}, \text{Inj} \rangle$ of graphs with class Inj of all injective graph morphisms is a weak adhesive HLR category [6].

Further examples of weak adhesive HLR categories are the categories of hypergraphs with all injective hypergraph morphisms, place-transition nets with all injective net morphisms, and algebraic specifications with all strict injective specification morphisms [6]. Weak adhesive HLR-categories have a number of nice properties, called HLR properties [4].

Fact 2 (properties of weak adhesive HLR categories [19,6]). For a weak adhesive HLR-category $\langle \mathcal{C}, \mathcal{M} \rangle$, the following properties hold:

1. Pushouts along $\mathcal{M}$-morphisms are pullbacks.
2. $\mathcal{M}$ pushout-pullback decomposition. If the diagram (1)+(2) in the figure below is a pushout, (2) a pullback, $w \in \mathcal{M}$ and ($l \in \mathcal{M}$ or $c \in \mathcal{M}$), then (1) and (2) are pushouts and also pullbacks.
3. Cube pushout-pullback decomposition. Given the commutative cube (3) in the figure below, where all morphisms in the top and the bottom are in $\mathcal{M}$, the top is pullback, and the front faces are pushouts, then the bottom is a pullback iff the back faces of the cube are pushouts.

$$\begin{tikzcd}
A \arrow[r, "c"] \arrow[d, "l"'] \arrow[dr, phantom, "(1)"] & C \arrow[r, "r"] \arrow[d, "s"'] \arrow[dr, phantom, "(2)"] & E \arrow[d, "v"] \\
B \arrow[r, "u"'] & D \arrow[r, "w"'] & F
\end{tikzcd}
\qquad
\begin{tikzcd}
C' \arrow[dr] \arrow[dd] & & A' \arrow[ll] \arrow[dr] \arrow[dd] & \\
& D' & & B' \arrow[ll, crossing over] \arrow[dd] \\
C \arrow[dr] & & A \arrow[ll] \arrow[dr] & \\
(3) & D \arrow[from=uu, crossing over] & & B \arrow[ll]
\end{tikzcd}$$

4. Uniqueness of pushout complements. Given morphisms $c: A \to C$ in $\mathcal{M}$ and $s: C \to D$, then there is, up to isomorphism, at most one B with $l: A \to B$ and $u: B \to D$ such that diagram (1) is a pushout.

In the following, we consider weak adhesive HLR categories with an epi-$\mathcal{M}$ factorization and binary coproducts.

Definition 3 (epi-$\mathcal{M}$ factorization). A weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$ has an *epi-$\mathcal{M}$ factorization* if, for every morphism, there is an epi-mono factorization with monomorphism in $\mathcal{M}$ and this decomposition is unique up to isomorphism.

Remark 1 (binary coproducts). In a weak adhesive HLR category $\langle \mathcal{C}, \mathcal{M} \rangle$ with binary coproducts, the binary coproducts are compatible with $\mathcal{M}$ in the sense that $f, g \in \mathcal{M}$ implies $f{+}g \in \mathcal{M}$. In fact, PO (1) in the figure below with $f \in \mathcal{M}$ implies $(f{+}\mathrm{id}) \in \mathcal{M}$ and PO (2) with $g \in \mathcal{M}$ implies $(\mathrm{id}{+}g) \in \mathcal{M}$, but now $(f{+}g) = (\mathrm{id}{+}g) \circ (f{+}\mathrm{id}) \in \mathcal{M}$ by closure under composition.[1em]

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[d] \arrow[dr, phantom, "(1)"] & B \arrow[dr] & & C \arrow[r, "g"] \arrow[dl] \arrow[dr, phantom, "(2)"] & D \arrow[d] \\
A{+}C \arrow[rr, "f{+}\mathrm{id}"'] & & B{+}C \arrow[rr, "\mathrm{id}{+}g"'] & & B{+}D
\end{tikzcd}$$

[1em] For the category $\langle \mathrm{Graphs}, \mathrm{Inj} \rangle$ of graphs with class Inj of all injective graph morphisms, these specific properties are satisfied.

Fact 3. $\langle \mathrm{Graphs}, \mathrm{Inj} \rangle$ has an epi-Inj factorization and binary coproducts [6].

3 Conditions and rules

We use the framework of weak adhesive HLR categories 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 an epi-$\mathcal{M}$ factorization and binary coproducts.

Conditions are defined as in [9,17]. Syntactically, the conditions may be seen as a tree of morphisms equipped with certain logical symbols such as quantifiers and connectives.

Definition 4 (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 are conditions over P: for conditions c, c_i over P with $i \in I$ (for all index sets I), $\neg\, c$ and $\wedge_{i \in I} c_i$ are conditions over P. $\exists a$ abbreviates $\exists(a, \mathrm{true})$, $\forall(a, c)$ abbreviates $\neg\exists(a, \neg c)$. Every morphism *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 $q \models c$.

$$\begin{tikzcd}
\exists(P \arrow[rr, "a"] \arrow[dr, "p"'] & {=} & C, \arrow[dl, hook, "q"] \arrow[r, dash] & c \;) \\
& G \arrow[ur, phantom, "\models"] &
\end{tikzcd}$$

The satisfaction of conditions over P by morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We write $p \models c$ to denote that the morphism p satisfies c. Two conditions c and c' over P are *equivalent*, denoted by $c \equiv c'$, if for all morphisms p with domain P, $p \models c$ iff $p \models c'$.

Remark 2. The definition of conditions generalizes those in [8,20,21,5]. In the context of rules, conditions are also called *application conditions*. Negative application conditions [8,16] correspond to nested application conditions of the form $\nexists a$. Examples of nested application conditions are given in Figure 1.

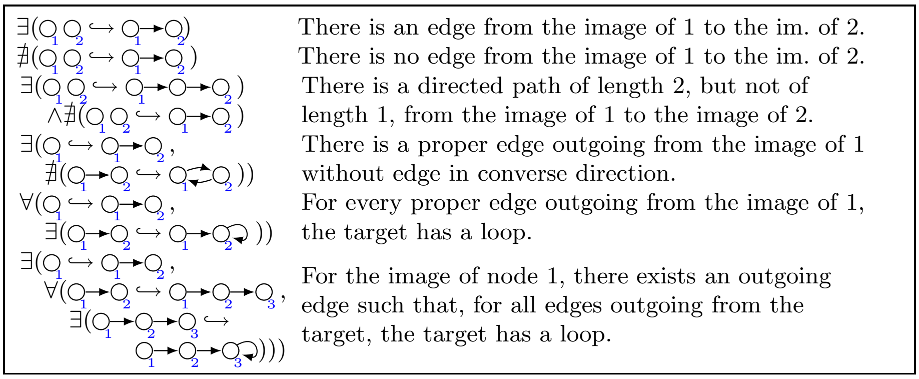

There is an edge from the image of 1 to the im. of 2.
There is no edge from the image of 1 to the im. of 2.
There is a directed path of length 2, but not of length 1, from the image of 1 to the image of 2.
There is a proper edge outgoing from the image of 1 without edge in converse direction.
For every proper edge outgoing from the image of 1, the target has a loop.
For the image of node 1, there exists an outgoing edge such that, for all edges outgoing from the target, the target has a loop.

Fig. 1. Nested application conditions

In the presence of an $\mathcal{M}$-initial object I [17], conditions $\exists(a, c)$ with morphism $a: I \to C$ can be used to define *constraints* for objects G, namely G satisfies $\exists(a, c)$ if the initial morphism i_G satisfies $\exists(a, c)$.

Remark 3. In general, one could choose a satisfiability notion, i.e. a class of morphisms $\mathcal{M}'$, and require that the morphism q in Definition 4 is in $\mathcal{M}'$. Examples are $\mathcal{A}$- and $\mathcal{M}$-satisfiability [22] where $\mathcal{A}$ and $\mathcal{M}$ are the classes of all morphisms and all monomorphisms, respectively.

Conditions can be shifted over morphisms into corresponding conditions over the codomain of the morphism. We present a Shift-construction based on jointly epimorphic pairs of morphisms. A morphism pair (e_1, e_2) with $e_i: A_i \to B$ $(i = 1, 2)$ is *jointly epimorphic* if, for all morphisms $g, h: B \to C$ with $g \circ e_i = h \circ e_i$ for $i = 1, 2$, we have $g = h$. In the case of graphs, "jointly epimorphic" means "jointly surjective": a morphism pair (e_1, e_2) is jointly surjective, if for each $b \in B$ there is a preimage $a_1 \in A_1$ with $e_1(a_1) = b$ or $a_2 \in A_2$ with $e_2(a_2) = b$.

Definition 5 (shift of conditions over morphisms). Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with epi-$\mathcal{M}$-factorization. The transformation Shift is inductively defined as follows:

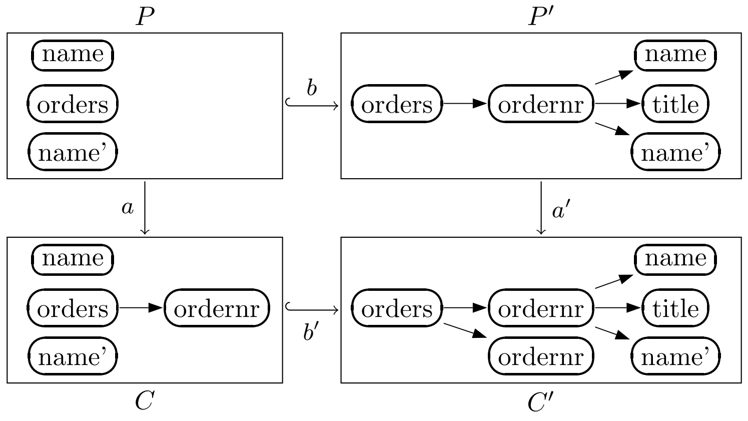
$$\begin{tikzcd}
P \arrow[r, "b"] \arrow[d, "a"'] \arrow[dr, phantom, "(1)"] & P' \arrow[d, "a'"] \\
C \arrow[r, hook, "b'"'] & C' \\
\underset{c}{\blacktriangle} &
\end{tikzcd}$$

$\mathrm{Shift}(b, \mathrm{true}) = \mathrm{true}$.
$\mathrm{Shift}(b, \exists(a, c)) = \bigvee_{(a',b') \in \mathcal{F}} \exists(a', \mathrm{Shift}(b', c))$
with $\mathcal{F} = \{(a', b') \mid (a', b')$ jointly epimorphic, $b' \in \mathcal{M}$, and (1) commutes$\}$.

For Boolean formulas over conditions, Shift is extended in the usual way: For conditions c, c_i with $i \in I$ (for all index sets I), $\mathrm{Shift}(b, \neg c) = \neg\mathrm{Shift}(b, c)$ and $\mathrm{Shift}(b, \wedge_{i \in I} c_i) = \wedge_{i \in I}\mathrm{Shift}(b, c_i)$.

Remark 4. In the special case that $\mathcal{F}$ is empty, the result of the transformation is false. For previous versions of the Shift-construction see [16,17].

Example 1. Given the morphism $b: P \to P'$ below, the condition $\exists\, a$ is shifted into the condition $\mathrm{Shift}(b, \exists\, a) = \exists\, a' \vee \exists\, a'' \vee \exists\, \mathrm{id}_{P'}$ where a' is the morphism depicted in the figure below and a'' obtained from a' by identifying the nodes with label ordernr in C'. The condition can be simplified to true because $\exists\, \mathrm{id}_{P'}$ is equivalent to true. The condition $\nexists\, a$ is shifted into the condition $\mathrm{Shift}(b, \nexists\, a) = \neg\mathrm{Shift}(b, \exists\, a) \equiv \neg\, \mathrm{true} \equiv \mathrm{false}$.



Lemma 1 (shift of conditions over morphisms). Let $\langle \mathcal{C}, \mathcal{M} \rangle$ be a weak adhesive HLR category with epi-$\mathcal{M}$-factorization. Then, for all conditions c over P and all morphisms $b: P \to P'$, $n: P' \to H$, $n \circ b \models c \Leftrightarrow n \models \mathrm{Shift}(b, c)$.

$$\begin{tikzcd}
c \triangleright P \arrow[rr, "b"] \arrow[dr, "n \circ b"'] & & P' \triangleleft \mathrm{Shift}(b, c) \arrow[dl, "n"] \\
& H &
\end{tikzcd}$$

Proof. The statement is proved by structural induction.
Basis. For the condition true, the equivalence holds trivially.
Inductive step. For a condition of the form $\exists(a, c)$, we have to show

$$n \circ b \models \exists(a, c) \Leftrightarrow n \models \mathrm{Shift}(b, \exists(a, c)).$$

"$\Rightarrow$": Let $n \circ b \models \exists(a, c)$. By definition of satisfiability, there is some $q \in \mathcal{M}$ with $q \circ a = n \circ b$ and $q \models c$. Let $(\bar{a}, \bar{b})$ be the pushout in (1) in the left diagram below. By the universal property of pushouts, there is an induced morphism $\bar{q}\colon \bar{C} \to H$ such that $q = \bar{q} \circ \bar{b}$ and $n = \bar{q} \circ \bar{a}$. By epi-$\mathcal{M}$ factorization of $\bar{q}$, $\bar{q} = m \circ e$ with epimorphism e and monomorphism $m \in \mathcal{M}$. Define now $a' = e \circ \bar{a}$ and $b' = e \circ \bar{b}$. Then the diagram $PP'CC'$ commutes. Since $\mathcal{M}$ is closed under decomposition, $q = m \circ b' \in \mathcal{M}$, $m \in \mathcal{M}$ implies $b' \in \mathcal{M}$. Since $\langle \bar{a}, \bar{b} \rangle$ is jointly epimorphic and e is an epimorphism, (a', b') is jointly epimorphic. Thus, $(a', b') \in \mathcal{F}$. By the inductive hypothesis, $q = m \circ b' \models c \Leftrightarrow m \models \mathrm{Shift}(b', c)$. Now $n \models \exists(a', \mathrm{Shift}(b', c))$ and, by definition of Shift, $n \models \exists(b, \mathrm{Shift}(a, c))$.

$$\begin{tikzcd}
P \arrow[r, "a"] \arrow[d, "b"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, hook, "\bar{b}"'] \arrow[dd, dashed, "b'"] \arrow[ddr, bend left, "q"] & \triangleleft\, c \\
P' \arrow[r, "\bar{a}"] \arrow[dr, dashed, "a'"'] \arrow[drr, bend right, "n"'] & \bar{C} \arrow[d, dashed, two heads, "e"] \arrow[dr, bend left, "\bar{q}"] & \\
 & C' \arrow[r, hook, "m"] & H
\end{tikzcd}
\qquad
\begin{tikzcd}
P \arrow[r, "a"] \arrow[d, "b"'] & C \arrow[d, hook, "b'"] & \triangleleft\, c \\
P' \arrow[r, "a'"] \arrow[d, "n"'] & C' \arrow[dl, dashed, hook', "m"] & \\
H & &
\end{tikzcd}$$

"$\Leftarrow$": Let $n \models \mathrm{Shift}(b, \exists(a, c))$. By definition of Shift, there is some $(a', b') \in \mathcal{F}$ with $b' \in \mathcal{M}$ such that $n \models \exists(a', \mathrm{Shift}(b', c))$. By definition of satisfiability, there is some $m \in \mathcal{M}$ such that $m \circ a' = n$ and $m \models \mathrm{Shift}(b', c)$. By the inductive hypothesis, $m \models \mathrm{Shift}(b', c) \Leftrightarrow m \circ b' \models c$. Now $m \circ b' \in \mathcal{M}$, $m \circ b' \circ a = n \circ b$ (see the right diagram above), and $m \circ b' \models c$, i.e., $n \circ b \models \exists(a, c)$. $\square$

Rules are defined as in [5,17]. 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 [13,18].

Definition 6 (rules). A *rule* $\rho = \langle p, ac_L, ac_R \rangle$ 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 and R are called the left- and the right-hand side of p and K the interface; ac_L and ac_R are the *left* and *right* application condition of p.

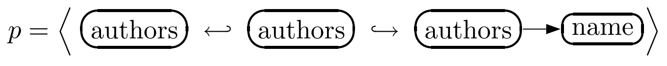
$$\begin{tikzcd}
ac_L \triangleright & L \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(2)"] & R \arrow[d, "m^*"] & \triangleleft\, ac_R \\
 & G & D \arrow[l, hook'] \arrow[r, hook] & H &
\end{tikzcd}$$

A *direct derivation* consists of two pushouts (1) and (2) such that $m \models ac_L$ and $m^* \models ac_R$. We write $G \Rightarrow_{\rho,m,m^*} H$ and say that $m\colon L \to G$ is the match of ρ in G and $m^*\colon R \to H$ is the comatch of ρ in H. We also write $G \Rightarrow_{\rho,m} H$ or $G \Rightarrow_{\rho} H$ to express that there is an m^* or there are m and m^*, respectively, such that $G \Rightarrow_{\rho,m,m^*} H$.

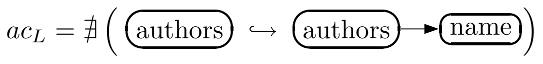
The concept of rules is completely symmetric.

Fact 4. For $\rho = \langle p, ac_L, ac_R\rangle$ with $p = \left\langle L \hookleftarrow K \hookrightarrow R \right\rangle$, $\rho^{-1} = \langle p^{-1}, ac_R, ac_L\rangle$ with $p^{-1} = \left\langle R \hookleftarrow K \hookrightarrow L \right\rangle$, is the *inverse rule* of ρ. For every direct derivation $G \Rightarrow_{\rho,m,m^*} H$, there is a direct derivation $H \Rightarrow_{\rho^{-1},m^*,m} G$ via the inverse rule.

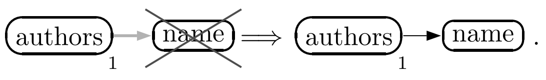
Notation. In the case of graphs, a rule $\left\langle L \hookleftarrow K \hookrightarrow R \right\rangle$ with discrete interface K is shortly depicted by $L \Rightarrow R$, where the nodes of K are indexed in the left- and the right-hand side of the rule. A negative application condition of the form $\nexists(L \hookrightarrow L')$ is integrated in the left-hand side of a rule by crossing the part $L'{-}L$ out. E.g. the rule



with

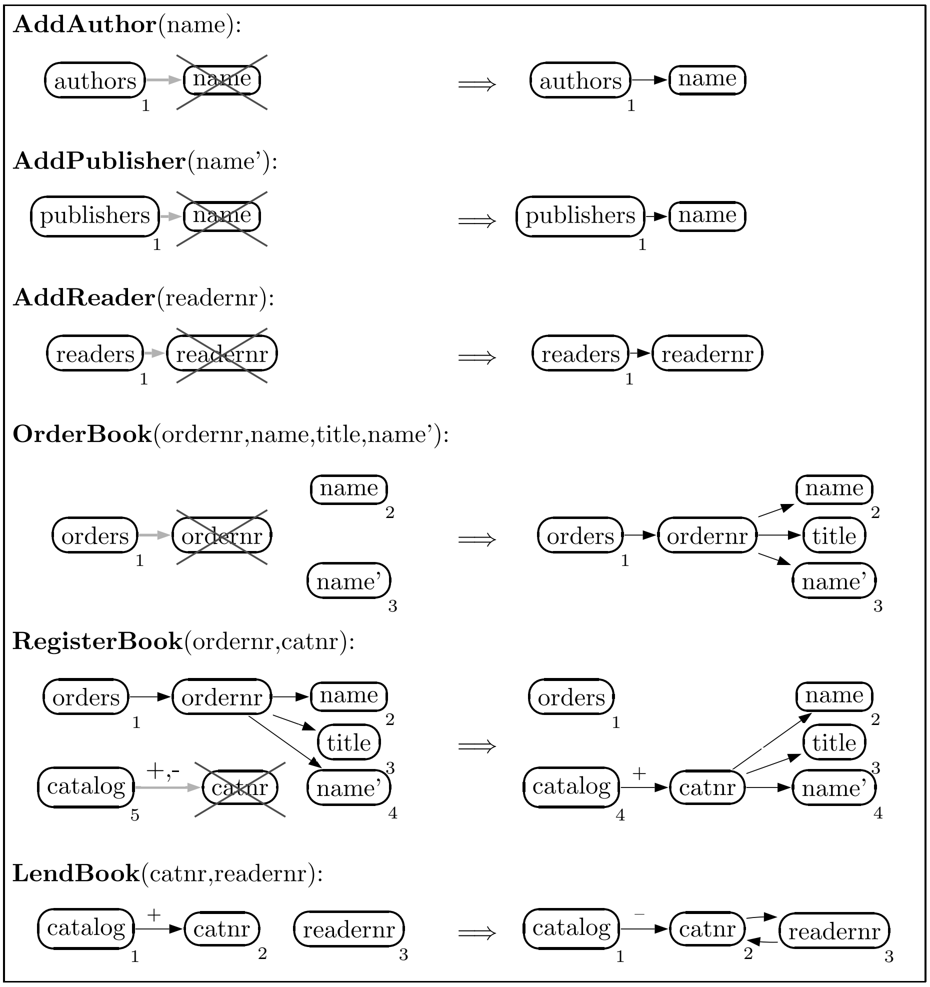


is depicted by



A conjunction $\bigwedge_i \nexists(L_i \hookrightarrow L'_i)$ of negative application conditions is represented by coloring the parts $L'_i{-}L_i$ in grey and crossing them out. A grey edge with labels $l_1, \ldots, l_n$ represents the conjunction of the negative application conditions “There does not exist an l_i-labelled edge” for $i = 1, \ldots, n$.

Example 2. In the figure below, rules with left application conditions are given, corresponding more or less to the operations of the small library system originally investigated in [23].



By Theorem 6 in [17], right application conditions of rules can be shifted into corresponding left application conditions and vice versa.

Lemma 2 (shift of conditions over rules). There are transformations L and R of application conditions such that, for every right application condition ac_R and every left application condition ac_L of a rule ρ and every direct derivation $G \Rightarrow_{\rho,m,m^*} H$, $m \models \mathrm{L}(\rho, ac_R) \Leftrightarrow m^* \models ac_R$ and $m \models ac_L \Leftrightarrow m^* \models \mathrm{R}(\rho, ac_L)$.

$$\begin{tikzcd}
\mathrm{L}(\rho, ac_R) \arrow[r] & L \arrow[d, "m"'] \arrow[rd, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[rd, phantom, "(2)"] & R \arrow[d, "m^*"] & ac_R \arrow[l] \\
 & G & D \arrow[l, hook'] \arrow[r, hook] & H &
\end{tikzcd}$$

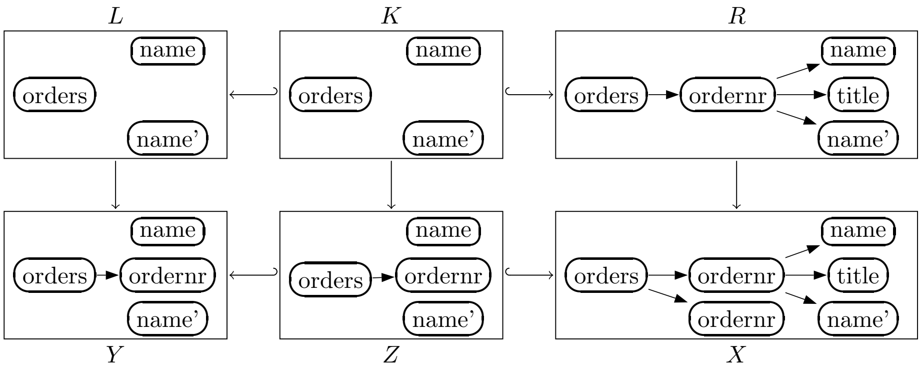
Construction. The transformation L is inductively defined as follows:

$$\begin{tikzcd} L \arrow[d, "b"'] \arrow[dr, phantom, "(2)"] & K \arrow[l, hook', "l"'] \arrow[r, hook, "r"] \arrow[d] \arrow[dr, phantom, "(1)"] & R \arrow[d, "a"] \\ Y \arrow[d, phantom, "\triangle"] & Z \arrow[l, hook', "l^*"] \arrow[r, hook, "r^*"'] & X \arrow[d, phantom, "\triangle"] \\ \mathrm{L}(\rho^*, ac) & & ac \end{tikzcd}$$

$\mathrm{L}(\rho, \mathrm{true}) = \mathrm{true}$
$\mathrm{L}(\rho, \exists(a, ac)) = \exists(b, \mathrm{L}(\rho^*, ac))$ if $\langle r, a\rangle$ has a pushout complement (1) and $\rho^* = \langle Y \leftarrow Z \rightarrow X\rangle$ is the derived rule by constructing the pushout (2).
$\mathrm{L}(\rho, \exists(a, ac)) = \mathrm{false}$, otherwise.

For Boolean formulas over application conditions, L is extended in the usual way: For conditions c, c_i with $i \in I$, $\mathrm{L}(b, \neg c) = \neg\mathrm{L}(b, c)$ and $\mathrm{L}(b, \wedge_{i\in I} c_i) = \wedge_{i\in I}\mathrm{L}(b, c_i)$. The transformation R is given by $\mathrm{R}(\rho, ac_L) = \mathrm{L}(\rho^{-1}, ac_L)$.

Example 3. Given the library rule $\rho =$ **OrderBook**(ordernr, name, title, name′) in the upper row of the figure below, the right application condition $\nexists(R \rightarrow X)$ is shifted over ρ into the left application condition $\nexists(L \rightarrow Y)$.



In the following, we define the equivalence of rules and the equivalence of application conditions with respect to a rule. The equivalence with respect to a rule is more restrictive than the unrestricted one in Definition 4.

Definition 7 (equivalence). Two rules ρ and ρ' are *equivalent*, denoted by $\rho \equiv \rho'$, if the relations $\Rightarrow_\rho$ and $\Rightarrow_{\rho'}$ are equal. For a rule ρ, two left (right) application conditions ac and ac' are *ρ-equivalent*, denoted by $ac \equiv_\rho ac'$, if the rules obtained from ρ by adding the application condition ac and ac', respectively, are equivalent.

There is a close relationship between the transformations L and R: For every rule ρ, Shift of a condition over the rule to the left and then over the rule to the right is ρ-equivalent to the original condition.

Fact 5 (L **and** R**).** For every rule ρ and every application condition ac over R, the right-hand side of the plain rule of ρ, the application conditions $\mathrm{R}(\rho, \mathrm{L}(\rho, ac))$ and ac are ρ-equivalent: $\mathrm{R}(\rho, \mathrm{L}(\rho, ac)) \equiv_\rho ac$.

Proof. By the Shift-Lemma 2, for every direct derivation $G \Rightarrow_{\rho,m,m^*} H$, $m^* \models \mathrm{R}(\rho, \mathrm{L}(\rho, ac)) \Leftrightarrow m \models \mathrm{L}(\rho, ac) \Leftrightarrow m^* \models ac$, i.e., the application conditions $\mathrm{R}(\rho, \mathrm{L}(\rho, ac))$ and ac are ρ-equivalent. □

Remark 5. In general, the application conditions $\mathrm{R}(\rho, \mathrm{L}(\rho, ac))$ and ac are not equivalent in the sense of Definition 4. E.g., for the rule $\rho = \left\langle \emptyset \hookleftarrow \emptyset \hookrightarrow \bigcirc_1 \right\rangle$ and the application condition $ac = \exists(\bigcirc_1 \to \bigcirc_1 \!\to\! \bigcirc)$, $\mathrm{L}(\rho, \neg ac) = \neg \mathrm{L}(\rho, ac) = \neg\mathrm{false} \equiv \mathrm{true}$ and $\mathrm{R}(\rho, \mathrm{L}(\rho, \neg ac)) = \mathrm{R}(\rho, \mathrm{true}) = \mathrm{true} \not\equiv \neg ac$.

Furthermore, there is a nice interchange result of Shift and L saying that, for a rule ρ, the shift of a right application condition over a rule and a match is ρ-equivalent to the shift of the application condition over the comatch and the rule induced by the match.

Lemma 3 (Shift **and** L**).** For every direct derivation $L^* \Rightarrow_{\rho,k,k^*} R^*$ via a rule ρ and every application condition ac, $\mathrm{Shift}(k, \mathrm{L}(\rho, ac)) \equiv_{\rho^*} \mathrm{L}(\rho^*, \mathrm{Shift}(k^*, ac))$, where ρ^* denotes the rule derived from ρ and k. A corresponding statement holds for Shift and R.

$$\begin{tikzcd}
L \arrow[d, "k"'] \arrow[dr, phantom, "(11)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(21)"] & R \triangleleft \arrow[d, "k^*"] \\
L^* & K^* \arrow[l, hook'] \arrow[r, hook] & R^*
\end{tikzcd}$$

Proof. Let $G \Rightarrow_{\rho^*,l,l^*} H$ be a direct derivation, $m = l \circ k$ and $m^* = l^* \circ k^*$. By Shift-Lemmas 1 and 2, we have $l \models \mathrm{Shift}(k, \mathrm{L}(\rho, ac)) \Leftrightarrow m \models \mathrm{L}(\rho, ac) \Leftrightarrow m^* \models ac_R \Leftrightarrow l^* \models \mathrm{Shift}(k^*, ac) \Leftrightarrow l \models \mathrm{L}(\rho^*, \mathrm{Shift}(k^*, ac))$.

$$\begin{tikzcd}
L \arrow[d, "k"'] \arrow[dr, phantom, "(11)"] \arrow[dd, bend right=60, "m"'] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(21)"] & R \triangleleft \arrow[d, "k^*"] \arrow[dd, bend left=60, "m^*"] \\
L^* \arrow[d, "l"'] \arrow[dr, phantom, "(12)"] & K^* \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(22)"] & R^* \arrow[d, "l^*"] \\
G & D \arrow[l, hook'] \arrow[r, hook] & H
\end{tikzcd}$$

□

As a consequence of Shift-Lemma 2, every rule can be transformed into an equivalent one with true right application condition. A rule of the form $\langle p, ac_L, \mathrm{true}\rangle$ is said to be a rule with left application condition and is abbreviated by $\langle p, ac_L\rangle$.

Corollary 1 (rules with left application condition). There is a transformation Left from rules into rules with left application condition such that, for every rule ρ, ρ, and $\mathrm{Left}(\rho)$ are equivalent.

Proof. For a rule $\rho = \langle p, ac_L, ac_R\rangle$, the transformation Left is defined by $\mathrm{Left}(\rho) = \langle p, ac_L \wedge \mathrm{L}(\rho, ac_R)\rangle$. By Definition 6, Shift-Lemma 2, and the definition of Left,

$$\begin{aligned}
G \Rightarrow_{\rho,m,m^*} H &\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models ac_L \wedge m^* \models ac_R \\
&\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models ac_L \wedge m \models \mathrm{L}(\rho, ac_R) \\
&\Leftrightarrow G \Rightarrow_{p,m,m^*} H \wedge m \models ac_L \wedge \mathrm{L}(\rho, ac_R) \\
&\Leftrightarrow G \Rightarrow_{\mathrm{Left}(\rho),m,m^*} H,
\end{aligned}$$

i.e., the rules ρ and $\mathrm{Left}(\rho)$ are equivalent. □

4 Local Church-Rosser, Parallelism, and Concurrency

In this section, we present Local Church-Rosser, Parallelism, and Concurrency Theorems for rules with application conditions. The proofs of the statements are based on the corresponding statements for rules *without application conditions* [6] and Shift-Lemmas 1 and 2, saying that application conditions can be shifted over morphisms and rules.

First, we study parallel and sequential independence of direct derivations leading to the Local Church-Rosser and Parallelism Theorems for rules with application conditions. By Corollary 1, we may assume that the rules are rules with left application condition.

Assumption 2. In the following, let $\rho_1 = \langle p_1, ac_{L_1} \rangle$ and $\rho_2 = \langle p_2, ac_{L_2} \rangle$ be rules with $p_i = \left\langle L_i \hookleftarrow K_i \hookrightarrow R_i \right\rangle$ for $i = 1, 2$.

Roughly speaking, two direct derivations are parallel (sequentially) independent if the underlying direct derivations without application conditions are parallel (sequentially) independent and the induced matches satisfy the corresponding application conditions. For rules with negative application conditions, the definition corresponds to the one in [24].

Definition 8 (parallel and sequential independence). Two direct derivations $H_1 \Leftarrow_{\rho_1,m_1} G \Rightarrow_{\rho_2,m_2} H_2$ are *parallel independent* if there are morphisms $d_2 \colon L_1 \to D_2$ and $d_1 \colon L_2 \to D_1$ such that the triangles L_1D_2G and L_2D_1G commute, $m_1' = c_2 \circ d_2 \models ac_{L_1}$, and $m_2' = c_1 \circ d_1 \models ac_{L_2}$.

$$\begin{tikzcd}
 & & ac_{L_1} & & ac_{L_2} & & \\[-20pt]
R_1 \arrow[d] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & L_1 \triangleleft \arrow[dr] \arrow[drrr, dotted, "d_2"] & & \triangleright L_2 \arrow[dl] \arrow[dlll, dotted, "d_1"'] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l, hook', "c_1"'] \arrow[rr, hook, "="] & & G & & D_2 \arrow[ll, hook', "="'] \arrow[r, hook, "c_2"] & H_2
\end{tikzcd}$$

Two direct derivations $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m_2'} M$ are *sequentially independent* if there are morphisms $d_2 \colon R_1 \to D_2$ and $d_1 \colon L_2 \to D_1$ such that the triangles $R_1D_2H_1$ and $L_2D_1H_1$ commute, $m_1'^* = c_2 \circ d_2 \models \mathrm{R}(\rho_1, ac_{L_1})$ and $m_2 = c_1 \circ d_1 \models ac_{L_2}$.

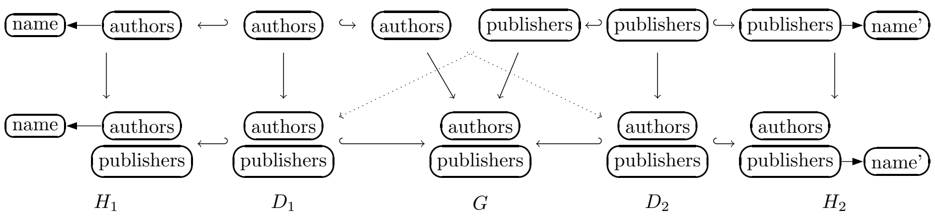
$$\begin{tikzcd}
\overset{ac_{L_1}}{\triangleright} L_1 \arrow[d] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_1 \arrow[dr, "d_1"'] \arrow[rrrd, dotted] & & \overset{ac_{L_2}}{\triangleright} L_2 \arrow[dl, "d_2"] \arrow[llld, dotted] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_2 \arrow[d] \\
G & D_1 \arrow[l, hook', "c_1"'] \arrow[rr, hook, "="] & & H_1 & & D_2 \arrow[ll, hook', "="'] \arrow[r, hook, "c_2"] & M
\end{tikzcd}$$

Two direct derivations that are not parallel (sequentially) independent, are called *parallel (sequentially) dependent.*

By definition, parallel and sequential independence are closely related.

Fact 6 (parallel and sequential independence are closely related). Two direct derivations $H_1 \Leftarrow_{\rho_1,m_1} G \Rightarrow_{\rho_2,m_2} H_2$ are parallel independent iff the two direct derivations $H_1 \Rightarrow_{\rho_1^{-1},m_1^*} G \Rightarrow_{\rho_2,m_2} H_2$ are sequentially independent, where m_1^* is the comatch of ρ_1 in H_1.

Example 4. The two direct derivations $H_1 \Leftarrow_{\rho_1} G \Rightarrow_{\rho_2} H_2$ via the rules $\rho_1 =$ **AddAuthor**(name) and $\rho_2 =$ **AddPublisher**(name′) are parallel independent.



In the proofs of the Local Church-Rosser, Parallelism and Concurrency Theorems, we proceed as follows: (1) We switch from derivations with ACs to the corresponding derivations without ACs, (2) use the results for derivations without ACs, and (3) lift the results without ACs to ACs.

$$\begin{tikzcd}
\text{derivations with ACs} \arrow[r, Rightarrow] \arrow[d] & \text{result with ACs} \\
\text{derivations without ACs} \arrow[r, Rightarrow] & \text{result without ACs} \arrow[u]
\end{tikzcd}$$

Fact 7 (Every derivation with ACs induces a derivation without ACs). For every direct derivation $G \Rightarrow_{\rho,m} H$ via the rule $\rho = \langle p, ac \rangle$, there is a direct derivation $G \Rightarrow_{p,m} H$ via the plain rule p, called the *underlying direct derivation without ACs.*

Fact 8 (independence with ACs implies independence without ACs). Parallel (sequential) independence of direct derivations implies parallel (sequential) independence of the underlying direct derivations without ACs.

Now we present a Local Church-Rosser Theorem for rules with application conditions. It generalizes the well-known Local Church-Rosser Theorems for rules without application conditions [6] and with negative application conditions [24].

Theorem 1 (Local Church-Rosser Theorem). Given two parallel independent direct derivations $H_1 \Leftarrow_{\rho_1,m_1} G \Rightarrow_{\rho_2,m_2} H_2$, there are an object M and direct derivations $H_1 \Rightarrow_{\rho_2,m'_2} M \Leftarrow_{\rho_1,m'_1} H_2$ such that $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$ and $G \Rightarrow_{\rho_2,m_2} H_2 \Rightarrow_{\rho_1,m'_1} M$ are sequentially independent. Given two sequentially independent direct derivations $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$, there are an object H_2 and direct derivations $G \Rightarrow_{\rho_2,m_2} H_2 \Rightarrow_{\rho_1,m'_1} M$ such that $H_1 \Leftarrow_{\rho_1,m_1} G \Rightarrow_{\rho_2,m_2} H_2$ are parallel independent.

$$\begin{tikzcd}
 & H_1 \arrow[dr, Rightarrow, "\rho_2"] & \\
G \arrow[ur, Rightarrow, "\rho_1"] \arrow[dr, Rightarrow, "\rho_2"'] & & M \\
 & H_2 \arrow[ur, Rightarrow, "\rho_1"'] &
\end{tikzcd}$$

Proof. Let $H_1 \Leftarrow_{\rho_1,m_1} G \Rightarrow_{\rho_2,m_2} H_2$ be parallel independent. Then the underlying direct derivations without ACs are parallel independent. By the Local Church-Rosser Theorem without ACs [6], there are an object M and direct derivations $H_1 \Rightarrow_{p_2,m'_2} M \Leftarrow_{p_1,m'_1} H_2$ such that $G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m'_2} M$ and $G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m'_1} M$ are sequentially independent. By assumption, $m_i, m'_i \models ac_{L_i}$ for $i = 1, 2$. Thus, there are direct derivations $H_1 \Rightarrow_{\rho_2,m'_2} M \Leftarrow_{\rho_1,m'_1} H_2$ with ACs. Let $R_1 \to \bar{D}_2$ and $L_2 \to D_1$ be the morphisms in Figure 2. Then $R_1 \to \bar{D}_2 \to H_1 = m_1^*$ and $L_2 \to D_1 \to H_1 = m'_2$. By Shift-Lemma 2, $R_1 \to \bar{D}_2 \to M = m_1'^* \models \mathrm{R}(\rho_1, ac_{L_1})$ and $L_2 \to D_1 \to G = m_2 \models ac_{L_2}$. Thus, the derivation $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$ is sequentially independent. Analogously, the second derivation is sequentially independent.
Vice versa, let $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$ be sequentially independent. Then the underlying direct derivations without ACs are sequentially independent. By the Local Church-Rosser Theorem without ACs [6], there are an object H_2 and direct derivations $G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m'_1} M$ such that $H_1 \Leftarrow_{p_1,m_1} G \Rightarrow_{p_2,m_2} H_2$ are parallel independent. By assumption, we know that $m_1, m'_1 \models ac_{L_1}$, $m_2 \models ac_{L_2}$ (by Shift-Lemma 2, $m_1'^* \models \mathrm{R}(\rho_1, ac_{L_1})$ implies $m'_1 \models ac_{L_1}$). Thus, $G \Rightarrow_{\rho_2,m_2} H_2 \Rightarrow_{\rho_1,m'_1} M$ is a derivation with ACs. Let $L_2 \to D_1$ and $L_1 \to D_2$ in Figure 2 be the morphisms with $L_1 \to D_2 \to G = L_1 \to G$ and $L_2 \to D_1 \to G = L \to G$. Then $L_1 \to D_2 \to H_2 = m'_1$ and $L_2 \to D_1 \to H_1 = m'_2 \models ac_{L_2}$. Thus, the direct derivations $H_1 \Leftarrow_{p_1,m_1} G \Rightarrow_{p_2,m_2} H_2$ become parallel independent. The statement also can be proved with the help of the first statement and Fact 6. □

For clarifying the notations, a sketch a part of the proof of Local Church-Rosser Theorem for rules without ACs is given oriented at the one in [30].

Sketch of proof. Let $H_1 \Leftarrow_{p_1,m_1} G \Rightarrow_{p_2,m_2} H_2$ be parallel independent. Then there are morphisms $L_1 \to D_2$ and $L_2 \to D_1$ such that the triangles $L_1 D_2 G$

andL_2D_1G in the figure below commute.

$$\begin{tikzcd}
R_1 \arrow[d] \arrow[dr, phantom, "(2)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(1)"] & L_1 \arrow[dr] \arrow[drrr, dotted] & & L_2 \arrow[dl] \arrow[dlll, dotted] \arrow[dr, phantom, "(3)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(4)"] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l, hook'] \arrow[rr, hook] & & G & & D_2 \arrow[ll, hook'] \arrow[r, hook] & H_2
\end{tikzcd}$$

The morphisms are used for the decomposition of the pushouts (i) into pushouts (i1),(i2) for $i = 1, \ldots, 4$ (Figure 2.1). The pushouts can be rearranged as in Figure 2.2 and 2.3. Furthermore, diagram (5) is constructed as pushout. Since the composition of pushouts yields pushouts, we obtain direct derivations $H_1 \Rightarrow_{p_2,m'_2} M \Leftarrow_{p_1,m'_1} H_2$ such that the direct derivations $G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m'_2} M$ and $G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m'_1} M$ are sequentially independent. □

$$\begin{tikzcd}
R_1 \arrow[d] \arrow[dd, bend right, "m_1^*"'] \arrow[dr, phantom, "(21)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(11)"] & L_1 \arrow[d] \arrow[ddr, "m_1"] & & L_2 \arrow[d] \arrow[ddl, "m_2"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[dd, bend left, "m_2^*"] \\
\bar{D}_2 \arrow[d, hook] \arrow[dr, phantom, "(22)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(12)"] & D_2 \arrow[dr, hook] & & D_1 \arrow[dl, hook'] \arrow[dr, phantom, "(32)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(42)"] & \bar{D}_1 \arrow[d, hook] \\
H_1 & D_1 \arrow[l, hook'] \arrow[rr, hook] & & G & & D_2 \arrow[ll, hook'] \arrow[r, hook] & H_2
\end{tikzcd}$$

$$\begin{tikzcd}
L_1 \arrow[d] \arrow[dd, bend right, "m_1"'] \arrow[dr, phantom, "(11)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(21)"] & R_1 \arrow[d] \arrow[ddr, "m_1^*"] & & L_2 \arrow[d] \arrow[ddl, "m'_2"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[dd, bend left, "m'^*_2"] \\
D_2 \arrow[d, hook] \arrow[dr, phantom, "(12)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(22)"] & \bar{D}_2 \arrow[dr, hook] & & D_1 \arrow[dl, hook'] \arrow[dr, phantom, "(22)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(5)"] & \bar{D}_1 \arrow[d, hook] \\
G & D_1 \arrow[l, hook'] \arrow[rr, hook] & & H_1 & & \bar{D}_2 \arrow[ll, hook'] \arrow[r, hook] & M
\end{tikzcd}$$

$$\begin{tikzcd}
L_2 \arrow[d] \arrow[dd, bend right, "m_2"'] \arrow[dr, phantom, "(31)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(41)"] & R_2 \arrow[d] \arrow[ddr, "m_2^*"] & & L_1 \arrow[d] \arrow[ddl, "m'_1"'] \arrow[dr, phantom, "(11)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(21)"] & R_1 \arrow[d] \arrow[dd, bend left, "m'^*_1"] \\
D_1 \arrow[d, hook] \arrow[dr, phantom, "(12)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(42)"] & \bar{D}_1 \arrow[dr, hook] & & D_2 \arrow[dl, hook'] \arrow[dr, phantom, "(42)"] & D_0 \arrow[l, hook'] \arrow[r, hook] \arrow[d, hook] \arrow[dr, phantom, "(5)"] & \bar{D}_2 \arrow[d, hook] \\
G & D_2 \arrow[l, hook'] \arrow[rr, hook] & & H_2 & & \bar{D}_1 \arrow[ll, hook'] \arrow[r, hook] & M
\end{tikzcd}$$

Fig. 2. Decomposition and composition

Next, we present the construction of a parallel rule of rules with application conditions. It generalizes the construction of a parallel rule of rules without application conditions [6] and makes use of the Shift of application conditions over morphisms and rules (see Shift-Lemmas 1 and 2). As in [6], we have to assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ has binary coproducts. The application condition of the parallel rule $\rho_1 + \rho_2$ guarantees that, whenever the parallel rule is applicable, the rules ρ_1 and ρ_2 are applicable and, after the application of ρ_1, the rule ρ_2 is applicable and, after the application of ρ_2, the rule ρ_1 is applicable.

Definition 9 (parallel rule and derivation). The *parallel rule* of ρ_1 and ρ_2 is the rule $\rho_1{+}\rho_2 = \langle p, ac'_L \rangle$ where $p = p_1{+}p_2$ is the parallel rule of p_1 and p_2, and $ac_{L'} = ac_L \wedge \mathrm{L}(\rho_1{+}\rho_2, ac_R)$, where

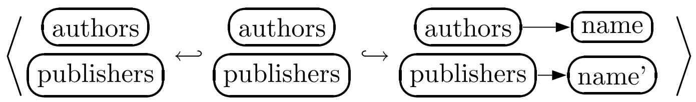
$$\begin{aligned} ac_L &= \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{Shift}(k_2, ac_{L_2}) \\ ac_R &= \mathrm{Shift}(k_1^*, \mathrm{R}(\rho_1, ac_{L_1})) \wedge \mathrm{Shift}(k_2^*, \mathrm{R}(\rho_2, ac_{L_2})). \end{aligned}$$

$$\begin{tikzcd}
\triangleright L_1 \arrow[dd, hook, "k_1"'] & & K_1 \arrow[ll, hook'] \arrow[rr, hook] \arrow[dd, hook] & & R_1 \arrow[dd, hook] & \\
& \triangleright L_2 \arrow[dl, hook', "k_2"] & & K_2 \arrow[ll, hook'] \arrow[rr, hook] \arrow[dl, hook'] & & R_2 \arrow[dl, hook', "k_2^*"] \\
\triangleright L_1{+}L_2 & & K_1{+}K_2 \arrow[ll, hook'] \arrow[rr, hook] & & R_1{+}R_2 \triangleleft &
\end{tikzcd}$$

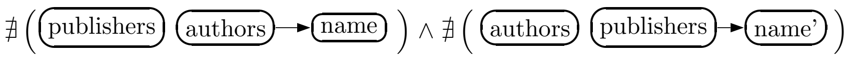
(with k_1^* labelling the arrow $R_1 \to R_1{+}R_2$)

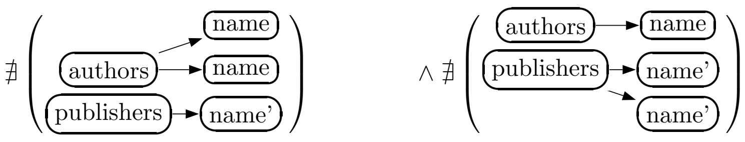
A direct derivation via a parallel rule is called *parallel* direct derivation or parallel derivation, for short.

Example 5. The parallel rule of **AddAuthor**(name) and **AddPublisher**(name′) is the rule with the plain rule

$p =$ 

and the application conditions

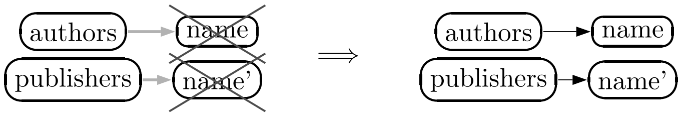
$ac_L = \nexists$ 

$ac_R = \nexists$ 

requiring that "There does not exist an author node with label name", "There does not exist a publisher node with label name′", "Afterwards, there do not exist two author nodes with label name", and "Afterwards, there do not exist two publisher nodes with label name′". Here an author node is a node which is connected with the node with label authors by a directed edge. Shifting the application condition ac_R over the rule ρ yields the application condition ac_L.

Thus, the parallel rule is equivalent to the rule with left application condition depicted below.

AddAuthorPublisher(name, name′):



The connection between sequentially independent direct derivations and parallel direct derivations is expressed by the Parallelism Theorem. We present the Parallelism Theorem for rules with application conditions. It generalizes the well-known Parallelism Theorems for rules without application conditions [6] and with negative application conditions [16].

Theorem 2 (Parallelism). Given sequentially independent direct derivations $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$, there is a parallel derivation $G \Rightarrow_{\rho_1+\rho_2,m} M$. Given a parallel derivation $G \Rightarrow_{\rho_1+\rho_2,m} M$, there are two sequentially independent direct derivations $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$ and $G \Rightarrow_{\rho_2,m_2} H_2 \Rightarrow_{\rho_1,m'_1} M$.

$$\begin{tikzcd}
& H_1 \arrow[dr, Rightarrow, "\rho_2"] & \\
G \arrow[ur, Rightarrow, "\rho_1"] \arrow[rr, Rightarrow, "\rho_1+\rho_2"'] \arrow[dr, Rightarrow, "\rho_2"'] & & M \\
& H_2 \arrow[ur, Rightarrow, "\rho_1"'] &
\end{tikzcd}$$

Proof. Let $G \Rightarrow_{\rho_1,m_1} H_1 \Rightarrow_{\rho_2,m'_2} M$ be sequentially independent. Then the underlying derivation without ACs is sequentially independent and, by the Parallelism Theorem without ACs [6], there is a parallel derivation $G \Rightarrow_{p_1+p_2,m} M$. By Shift-Lemmas 1 and 2, (*) $m \models ac_L$ and $m^* \models ac_R$ if and only if $m_i, m'_i \models ac_{L_i}$ for $i = 1, 2$. This may be seen as follows:

$$\begin{aligned}
m \models ac_L &\Leftrightarrow m \models \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{Shift}(k_2, ac_{L_2})\\
&\Leftrightarrow m_1 \models ac_{L_1} \text{ and } m_2 \models ac_{L_2}\\
m^* \models ac_R &\Leftrightarrow m^* \models \mathrm{Shift}(k_1^*, \mathrm{R}(\rho_1, ac_{L_1})) \wedge \mathrm{Shift}(k_2^*, \mathrm{R}(\rho_2, ac_{L_2}))\\
&\Leftrightarrow m_1'^* \models \mathrm{R}(\rho_1, ac_{L_1}) \text{ and } m_2'^* \models \mathrm{R}(\rho_2, ac_{L_2})\\
&\Leftrightarrow m'_1 \models ac_{L_1} \text{ and } m'_2 \models ac_{L_2}
\end{aligned}$$

$$\begin{tikzcd}
& \triangledown & \\
\rhd L_1 \arrow[r, "k_1"] \arrow[dr, "m_1"'] & L_1+L_2 \arrow[d, dotted, "m"] & L_2 \lhd \arrow[l, "k_2"'] \arrow[dl, "m_2"] \\
& G &
\end{tikzcd}
\qquad
\begin{tikzcd}
& \triangledown & \\
\rhd R_1 \arrow[r, "k_1^*"] \arrow[dr, "m_1'^*"'] & R_1+R_2 \arrow[d, dotted, "m^*"] & R_2 \lhd \arrow[l, "k_2^*"'] \arrow[dl, "m_2'^*"] \\
& M &
\end{tikzcd}$$

By assumption, $m_i, m'_i \models ac_{L_i}$ for $i = 1, 2$. By $(*)$, $m \models ac_L$ and $m^* \models ac_R$, i.e., $G \Rightarrow_{p_1+p_2,m} M$ satisfies ACs. Vice versa, let $G \Rightarrow_{\rho_1+\rho_2,m} M$ be a parallel derivation. Then there is an underlying parallel derivation without ACs, and, by the Parallelism Theorem without ACs [6], there are sequentially independent direct derivations $G \Rightarrow_{p_1,m_1} H_1 \Rightarrow_{p_2,m'_2} M$ and $G \Rightarrow_{p_2,m_2} H_2 \Rightarrow_{p_1,m'_1} M$. By assumption, $m \models ac_L$ and $m^* \models ac_R$. By $(*)$, $m_i, m'_i \models ac_{L_i}$ for $i = 1, 2$, i.e., the sequentially independent direct derivations satisfy ACs. □

Shift operations over parallel rules can be sequentialized into a sequence of shifts over induced rules.

Fact 9 (shift over parallel rules). For every parallel rule $\rho = \rho_1 + \rho_2$, every right application condition ac for ρ, and $i, j \in \{1,2\}$ with $i \neq j$, we have $\mathrm{L}(\rho, ac) \equiv_\rho \mathrm{L}(\rho^*_i, \mathrm{L}(\rho^*_j, ac))$ where ρ^*_i is induced by ρ_i and k_i and ρ^*_j is induced by ρ_j and k'_j.

Proof. By the Parallelism Theorem, for every direct derivation $G \Rightarrow_{\rho,m,m^*} M$ there are direct derivations $G \Rightarrow_{\rho_i,m_i} H_i \Rightarrow_{\rho_j,m_j} M$. By analysis arguments as in the proof of the Parallelism Theorem [6], there are direct derivations $G \Rightarrow_{\rho^*_i,m} H_i \Rightarrow_{\rho^*_j,m'} M$ depicted in Figure 3. By the Shift-Lemma 2, $m \models \mathrm{L}(\rho, ac) \Leftrightarrow m^* \models ac \Leftrightarrow m' \models \mathrm{L}(\rho^*_j, ac) \Leftrightarrow m \models \mathrm{L}(\rho^*_i, \mathrm{L}(\rho^*_j, ac))$, i.e, the application conditions $\mathrm{L}(\rho, ac)$ and $\mathrm{L}(\rho^*_i, \mathrm{L}(\rho^*_j, ac))$ are ρ-equivalent. □

$$\begin{tikzcd}
L_i \arrow[d, "k_i"'] & K_i \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_i \arrow[dr] & & L_j \arrow[dl, "k'_j"'] & K_j \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_j \arrow[d, "k^*_j"] \\
L_i+L_j \arrow[dd, "m"'] & K_i+L_j \arrow[l, hook'] \arrow[rr, hook] \arrow[dd] & & R_i+L_j \arrow[dd, "m'"] & & R_i+K_j \arrow[ll, hook'] \arrow[r, hook] \arrow[dd] & R_i+R_j \arrow[dd, "m^*"] \\
& & K_i+K_j \arrow[ul, dotted] \arrow[urrr, dotted] \arrow[dd, dotted] & & & & \\
G & E_1 \arrow[l, hook'] \arrow[rr, hook] & & H_i & & E_2 \arrow[ll, hook'] \arrow[r, hook] & M \\
& & E \arrow[ul, dotted] \arrow[urrr, dotted] & & & &
\arrow[from=1-1, to=2-2, phantom, "(PO)"]
\arrow[from=1-2, to=2-3, phantom, "(PO)"]
\arrow[from=1-5, to=2-6, phantom, "(PO)"]
\arrow[from=1-6, to=2-7, phantom, "(PO)"]
\arrow[from=2-1, to=4-2, phantom, "(PO)"]
\arrow[from=2-2, to=4-3, phantom, "(PO)"]
\arrow[from=2-4, to=4-6, phantom, "(PO)"]
\arrow[from=2-6, to=4-7, phantom, "(PO)"]
\end{tikzcd}$$

Fig. 3. Sequentialization of a parallel derivation

Finally, we present the construction of a concurrent rule for rules with application conditions. It generalizes the construction of concurrent rules for rules without application conditions [6] and makes use of shifting of application conditions over morphisms and rules (see Shift-Lemmas 1 and 2).

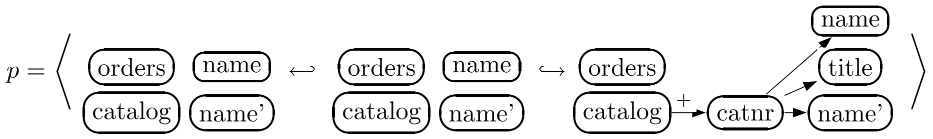
Definition 10 (E-concurrent rule). Let $\mathcal{E}'$ be a class of morphism pairs with the same codomain. Given two rules ρ_1 and ρ_2, an object E with morphisms $e_1 \colon R_1 \to E$ and $e_2 \colon L_2 \to E$ is an *E-dependency relation* for ρ_1 and ρ_2 if $(e_1, e_2) \in$

$\mathcal{E}'$ and the pushout complements (1) and (2) over $K_1 \hookrightarrow R_1 \to E$ and $K_2 \hookrightarrow L_2 \to E$ in the figure below exist. Given such an E-dependency relation for ρ_1 and ρ_2, the *E-concurrent rule* of ρ_1 and ρ_2 is the rule $\rho_1 *_E \rho_2 = \langle p, ac_L\rangle$ where $p = p_1 *_E p_2$ is E-concurrent rule of p_1 and p_2 with pushouts (3), (4) and pullback (5), $\rho_1^* = \left\langle L \hookleftarrow D_1 \hookrightarrow E \right\rangle$ is the rule derived by ρ_1 and k_1, and

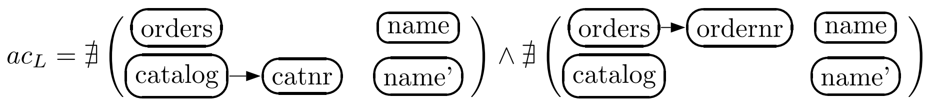
$$ac_L = \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{L}(\rho_1^*, \mathrm{Shift}(k_2, ac_{L_2})).$$

$$\begin{tikzcd}
\rhd L_1 \arrow[d, "k_1"'] \arrow[dr, phantom, "(3)"] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(1)"] & R_1 \arrow[dr] & & \rhd L_2 \arrow[dl, "k_2"'] \arrow[d, phantom, "(2)"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(4)"] & R_2 \arrow[d] \\
\rhd L & D_1 \arrow[l, hook'] \arrow[rr, hook] & & E \arrow[d, phantom, "(5)"] & & D_2 \arrow[ll, hook'] \arrow[r, hook] & R \\
& & & K \arrow[ulll, hook] \arrow[ull, hook] \arrow[urr, hook] \arrow[urrr, hook] & & &
\end{tikzcd}$$

Example 6. The E-concurrent rule of ρ_1=**OrderBook**(ordernr, name, title, name′) and ρ_2 = **RegisterBook**(ordernr, catnr) according to the dependency relation E, being the right-hand side E of ρ_1 and the left-hand side of ρ_2, is the rule

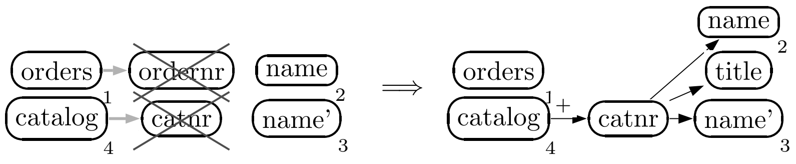


with the left application condition



requiring that "There does not exist a catalog node with label catnr" and "There does not exist an order node with label ordernr". The E-concurrent rule may be depicted as follows.

Order; **RegisterBook**(ordernr, catnr, name, title, name′):



The non-existence of a node with label catnr guarantees that, whenever the E-concurrent rule of ρ_1 and ρ_2 is applicable, then the rule ρ_1 with ordernr is applicable and, afterwards, the rule ρ_2 with catnr is applicable.

For rules without ACs, the parallel rule is a special case of the concurrent rule [6]. For rules with ACs, in general, this is not the case: While the application conditions for the parallel rule must guarantee the applicability of the rules in each order, the application condition for the concurrent rule only must guarantee the applicability of the rules in the given order. Nevertheless, the parallel rule of two rules can be constructed from two concurrent rules of the rules, one for each order.

Fact 10. The parallel rule $\rho_1{+}\rho_2 = \langle p_1{+}p_2, ac_L, ac_R\rangle$ and the rule $\langle p_1{+}p_2, ac_{L_{12}} \wedge ac_{L_{21}}\rangle$ obtained from the $R_1{+}L_2$-concurrent rule $\langle p_1 + p_2, ac_{L_{12}}\rangle$ of ρ_1 and ρ_2 and the $R_2{+}L_1$-concurrent rule $\langle p_2 + p_1, ac_{L_{21}}\rangle$ of ρ_2 and ρ_1 are equivalent.

$$\begin{tikzcd}
\rhd L_1 \arrow[d, "k_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_1 \arrow[dr] & & \rhd L_2 \arrow[dl, "k'_2"'] \\
\rhd\, L_1{+}L_2 & K_1{+}L_2 \arrow[l, hook'] \arrow[rr, hook] & & R_1{+}L_2 &
\end{tikzcd}
\qquad
\begin{tikzcd}
\rhd L_2 \arrow[d, "k_2"'] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] & R_2 \arrow[dr] & & \rhd L_1 \arrow[dl, "k'_1"'] \\
\rhd\, L_2{+}L_1 & K_2{+}L_1 \arrow[l, hook'] \arrow[rr, hook] & & R_2{+}L_1 &
\end{tikzcd}$$

Proof. For every parallel derivation $G \Rightarrow_{\rho_1+\rho_2,m,m^*} M$ (see Figure 3) and $i, j \in \{1, 2\}$ with $i \neq j$, we have

$$\begin{array}{lll}
(***)\ m^* \models \mathrm{Shift}(k_j^*, \mathrm{R}(\rho_j, ac_{L_j})) & \\
\Leftrightarrow \quad m^* \models \mathrm{R}(\rho_j^*, \mathrm{Shift}(k_j, ac_{L_j})) & \text{(Lemma 3)} \\
\Leftrightarrow \quad m \models \mathrm{L}(\rho_j^*, \mathrm{R}(\rho_j^*, \mathrm{Shift}(k_j, ac_{L_j})) & \text{(Shift-Lemma 2)} \\
\Leftrightarrow \quad m \models \mathrm{Shift}(k_j, ac_{L_j})) & \text{(Fact 5)}
\end{array}$$

By the definitions and statement (***),

$$\begin{array}{rlr}
m & \models ac_L \text{ and } m^* \models ac_R & \\
\Leftrightarrow m & \models \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{Shift}(k_2, ac_{L_2}) \text{ and} & \\
m^* & \models \mathrm{Shift}(k_1^*, \mathrm{R}(\rho_1, ac_{L_1})) \wedge \mathrm{Shift}(k_2^*, \mathrm{R}(\rho_2, ac_{L_2})) & \text{(Definition 9)} \\
\Leftrightarrow m & \models \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{L}(\rho_1^*, \mathrm{Shift}(k'_2, ac_{L_2})) \text{ and} & \\
m & \models \mathrm{Shift}(k_2, ac_{L_2}) \wedge \mathrm{L}(\rho_2^*, \mathrm{Shift}(k'_1, ac_{L_1})) & (***) \\
\Leftrightarrow m & \models ac_{L_{12}} \wedge ac_{L_{21}} & \text{(Definition 10)}
\end{array}$$

i.e., the parallel rule and the rule constructed from the concurrent rules are equivalent. □

We consider E-concurrent derivations via E-concurrent rules and E-related derivations via pairs of rules.

Definition 11 (E-concurrent and E-related derivation). A direct derivation via an E-concurrent rule is called *E-concurrent* direct derivation or E-concurrent derivation, for short. A derivation $G \Rightarrow_{\rho_1} H \Rightarrow_{\rho_2} M$ is *E-related* if there are morphisms $E \to H$, $D_1 \to E_1$, and $D_2 \to E_2$ as shown below such that the triangles R_1EH, L_2EH, $K_1D_1E_1$, and $K_2D_2E_2$ in the figure below commute and the diagrams (6) and (7) are pushouts.

$$\begin{tikzcd}
L_1 \arrow[d] & K_1 \arrow[l, hook'] \arrow[rr, hook] \arrow[dr] \arrow[d] & & R_1 \arrow[dr] \arrow[ddr, bend right] & & L_2 \arrow[dl] \arrow[ddl, bend left] & & K_2 \arrow[ll, phantom] \arrow[r, hook] \arrow[dl] \arrow[dd] & R_2 \arrow[dd] \\
& & D_1 \arrow[rr, hook] \arrow[dl, dotted] \arrow[l, phantom, "="] & (6) & E \arrow[d, dotted] & (7) & D_2 \arrow[ll, hook'] \arrow[dr, dotted] \arrow[r, phantom, "="] & & \\
G & E_1 \arrow[l, hook'] \arrow[rrr, hook] & & & H & & & E_2 \arrow[lll, hook'] \arrow[r, hook] & M
\end{tikzcd}$$

Now we present a Concurrency Theorem for rules with application conditions. It generalizes the well-known Concurrency Theorems for rules without application conditions [6] and with negative application conditions [16].

Theorem 3 (Concurrency). Let E be a dependency relation for ρ_1 and ρ_2. For every E-related derivation $G \Rightarrow_{\rho_1,m_1} H \Rightarrow_{\rho_2,m_2} M$, there is an E-concurrent derivation $G \Rightarrow_{\rho_1 *_E \rho_2,m} M$. Vice versa, for every E-concurrent derivation $G \Rightarrow_{\rho_1 *_E \rho_2,m} M$, there is an E-related derivation $G \Rightarrow_{\rho_1,m_1} H \Rightarrow_{\rho_2,m_2} M$.

$$\begin{tikzcd}
& H \arrow[dr, Rightarrow, "\rho_2"] & \\
G \arrow[ur, Rightarrow, "\rho_1"] \arrow[rr, Rightarrow, "\rho_1 *_E \rho_2"'] & & M
\end{tikzcd}$$

Proof. Let $G \Rightarrow_{\rho_1,m_1} H \Rightarrow_{\rho_2,m_2} M$ be E-related. Then the underlying derivation without ACs is E-related and, by the Concurrency Theorem without ACs [6], there is an E-concurrent derivation $G \Rightarrow_{p_1 * p_2,m} M$. By Shift-Lemmas 1 and 2, (**) $m_1 \models ac_{L_1}$ and $m_2 \models ac_{L_2}$ iff $m \models ac_L$. This may be seen as follows:

$$\begin{aligned}
& m_1 \models ac_{L_1} \text{ and } m_2 \models ac_{L_2} \\
\Leftrightarrow\ & m \models \mathrm{Shift}(k_1, ac_{L_1}) \text{ and } m' \models \mathrm{Shift}(k_2, ac_{L_2}) \\
\Leftrightarrow\ & m \models \mathrm{Shift}(k_1, ac_{L_1}) \text{ and } m \models \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, ac_{L_2})) \\
\Leftrightarrow\ & m \models \mathrm{Shift}(k_1, ac_{L_1}) \wedge \mathrm{L}(p_1^*, \mathrm{Shift}(k_2, ac_{L_2})) = ac_L.
\end{aligned}$$

By assumption, $m_i \models ac_{L_i}$ for $i = 1, 2$. By (**), $m \models ac_L$, i.e. the E-concurrent derivation satisfies ACs.

$$\begin{tikzcd}
\rhd L_1 \arrow[d, "k_1"] \arrow[dd, bend right=60, "m_1"'] & K_1 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[l, phantom, shift right=6, "(3)"] & R_1 \arrow[dr] \arrow[ddr, bend right] \arrow[l, phantom, shift right=6, "(1)"] & & \rhd L_2 \arrow[dl, "k_2"'] \arrow[ddl, bend left, "m_2"] & K_2 \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[l, phantom, shift right=6, "(2)"] & R_2 \arrow[d] \arrow[l, phantom, shift right=6, "(4)"] \\
\rhd L \arrow[d, "m"] & D_1 \arrow[l, hook'] \arrow[rr, hook] \arrow[d] \arrow[l, phantom, shift right=6, "(3')"] & (1') & E \arrow[d, "m'"] & (2') & D_2 \arrow[ll, hook'] \arrow[r, hook] \arrow[d] & R \arrow[d] \arrow[l, phantom, shift right=6, "(4')"] \\
G & E_1 \arrow[l, hook'] \arrow[rr, hook] & & H & & E_2 \arrow[ll, hook'] \arrow[r, hook] & M
\end{tikzcd}$$

Vice versa, let $G \Rightarrow_{\rho,m} M$ be an E-concurrent derivation, then the underlying direct derivation without ACs is E-concurrent, and, by the Concurrency Theorem without ACs [6], there is an E-related derivation $G \Rightarrow_{p_1,m_1} H \Rightarrow_{p_2,m_2} M$. By assumption, $m \models ac_L$. By (**), $m_1 \models ac_{L_1}$ and $m_2 \models ac_{L_2}$, i.e., the E-related derivation satisfies ACs. □

5 Conclusion

In this paper we present the well-known Local Church-Rosser, Parallelism, and Concurrency Theorems, known already for rules with negative application conditions [16], for rules with nested application conditions. The proofs are based on the corresponding theorems for rules without application conditions [6] and two Shift-Lemmas [17], saying that application conditions can be shifted over morphisms and rules and assume that $\langle \mathcal{C}, \mathcal{M} \rangle$ is a weak adhesive HLR category with an epi-$\mathcal{M}$-factorization and binary coproducts.

statement	requirements
Local Church-Rosser	Shift 1 & 2
Parallelism	Shift 1 & 2, binary coproducts
Concurrency	Shift 1 & 2
Shift 1	epi-$\mathcal{M}$-factorization
Shift 2	–

Further topics might be the following:

– **Amalgamation Theorem for rules with ACs.** It would be important to generalize the AmalgamationTheorem [25,18] to weak adhesive HLR systems and rules with nested application conditions.
– **Embedding and Local Confluence Theorems for rules with ACs.** It would be important to generalize the Embedding and Local Confluence Theorems [26,13,27,28,6,29] to rules with nested application conditions.
– **Theory to rules with merging.** It would be important to generalize the theory to the case of merging as indicated in [30].

References

1. Rozenberg, G. ed.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations. World Scientific (1997)
2. Ehrig, H. Engels, G. Kreowski, H.J. Rozenberg, G. eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 2: Applications, Languages and Tools. World Scientific (1999)
3. Ehrig, H. Kreowski, H.J. Montanari, U. Rozenberg, G. eds.: Handbook of Graph Grammars and Computing by Graph Transformation. Volume 3: Concurrency, Parallelism, and Distribution. World Scientific (1999)
4. Ehrig, H. Habel, A. Kreowski, H.J. Parisi-Presicce, F.: Parallelism and concurrency in high level replacement systems. Mathematical Structures in Computer Science **1** (1991) 361–404
5. 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)** (2006) 135–166
6. Ehrig, H. Ehrig, K. Prange, U. Taentzer, G.: Fundamentals of Algebraic Graph Transformation. EATCS Monographs of Theoretical Computer Science. Springer, Berlin (2006)

7. Ehrig, H. Habel, A.: Graph grammars with application conditions. In Rozenberg, G. Salomaa, A. eds.: The Book of L. Springer, Berlin (1986) 87–100
8. Habel, A. Heckel, R. Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
9. Habel, A. Pennemann, K.H.: Nested constraints and application conditions for high-level structures. In: Formal Methods in Software and System Modeling. Volume 3393 of LNCS. Springer (2005) 293–308
10. Ehrig, H. Kreowski, H.J.: Parallelism of manipulations in multidimensional information structures. In: Mathematical Foundations of Computer Science. Volume 45 of LNCS. Springer (1976) 284–293
11. Kreowski, H.J.: Manipulationen von Graphmanipulationen. PhD thesis, Technical University of Berlin (1977)
12. Kreowski, H.J.: Transformations of derivation sequences in graph grammars. In: Fundamentals of Computation Theory. Volume 56 of LNCS. Springer (1977) 275–286
13. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In: Graph-Grammars and Their Application to Computer Science and Biology. Volume 73 of LNCS. Springer (1979) 1–69
14. Ehrig, H. Rosen, B.K.: Parallelism and concurrency of graph manipulations. Theoretical Computer Science **11** (1980) 247–275
15. Habel, A.: Concurrency in Graph-Grammatiken. Technical Report 80-11, Technical University of Berlin (1980)
16. 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). Volume 2003 of ENTCS. Elsevier (2008) 43–66
17. Habel, A. Pennemann, K.H.: Correctness of high-level transformation systems relative to nested conditions. Mathematical Structures in Computer Science **19** (2009) 245–296
18. 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. Volume 1. World Scientific (1997) 163–245
19. Lack, S. Sobociński, P.: Adhesive categories. In: Foundations of Software Science and Computation Structures (FOSSACS'04). Volume 2987 of LNCS. Springer (2004) 273–288
20. Heckel, R. Wagner, A.: Ensuring consistency of conditional graph grammars — a constructive approach. In: SEGRAGRA '95. Volume 2 of ENTCS. (1995) 95–104
21. Koch, M. Mancini, L.V. Parisi-Presicce, F.: Graph-based specification of access control policies. Journal of Computer and System Sciences **71** (2005) 1–33
22. Habel, A. Pennemann, K.H.: Satisfiability of high-level conditions. In: Graph Transformations (ICGT 2006). Volume 4178 of LNCS. Springer (2006) 430–444
23. Ehrig, H. Kreowski, H.J.: Applications of graph grammar theory to consistency, synchronization and scheduling in data base systems. Information Systems **5** (1980) 225–238
24. Lambers, L. Ehrig, H. Orejas, F.: Conflict detection for graph transformation with negative application conditions. In: Graph Transformations (ICGT 2006). Volume 4178 of LNCS. Springer (2006) 61–76
25. Boehm, P. Fonio, H.R. Habel, A.: Amalgamation of graph transformations: A synchronization mechanism. Journal of Computer and System Sciences **34** (1987) 377–408

26. Ehrig, H.: Embedding theorems in the algebraic theory of graph grammars. In: Fundamentals of Computation Theory. Volume 56 of LNCS. Springer (1977) 245–255
27. Plump, D.: Hypergraph rewriting: Critical pairs and undecidability of confluence. In: Term Graph Rewriting: Theory and Practice. John Wiley, New York (1993) 201–213
28. Plump, D.: Confluence of graph transformation revisited. In: Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday. Volume 3838 of LNCS. Springer (2005) 280–308
29. Lambers, L. Ehrig, H. Prange, U. Orejas, F.: Embedding and confluence of graph transformations with negative application conditions. In: Graph Transformations (ICGT 2008). Volume 5214 of LNCS. Springer (2008) 162–177
30. Habel, A. Müller, J. Plump, D.: Double-pushout graph transformation revisited. Mathematical Structures in Computer Science **11** (2001) 637–688

Prof. Dr. Hartmut Ehrig

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
D-10587 Berlin (Germany)
ehrig@cs.tu-berlin.de
http://tfs.cs.tu-berlin.de/~ehrig

Hartmut Ehrig knows Hans-Jörg Kreowski since 1970 when he was one of the most engaged students in Hartmut's seminar on *Kategorien und Automaten* at the Mathematical Department of TU Berlin. This seminar was a great success, leading to a textbook with the same title, published 1971 by Walter de Gruyter. In 1974 followed a joint international book *Universal Theory of Automata*, published by Teubner, which was mainly based on Hans-Jörg's Diploma thesis. Meanwhile, Hartmut had become assistant professor at the new Department of Computer Science at TU Berlin, and hired Hans-Jörg as an assistant. The main focus of their joint work switched from Categorical Automata Theory to Graph Transformation, based on the DPO-approach, and Algebraic Specification, following the initial algebra approach of the ADJ-group at IBM Yorktown Heights. A very important contribution in the first area was Hans-Jörg's doctoral thesis *Manipulationen von Graphmanipulationen*, on the concurrent semantics of graph transformation systems. In the area of algebraic specifications, their joint research focussed on parametrized specifications and parameter passing, which led to the well-known ACT-approach and the algebraic specification language ACT ONE. With respect to both areas, this period was most successful for them, with interesting contributions to important conferences and publications in the Springer LNCS series and several well-known journals. Meanwhile, Hans-Jörg finished his habilitation thesis in Berlin. In 1982, he accepted a call for a professorship in Bremen, where he built up a strong research group in the areas of Algebraic Specification and Graph Transformation. Since that time the research groups in Berlin and Bremen have been working together with great success, especially in the European

Research Projects CompuGraph, Compass, GetGraTS, AppliGraph, and SeGraVis.

Prof. Dr. Annegret Habel

Carl v. Ossietzky Universität Oldenburg
Fachbereich Informatik
D-26111 Oldenburg (Germany)
Annegret.Habel@informatik.uni-oldenburg.de
http://theoretica.informatik.uni-oldenburg.de/~habel

Annegret Habel was the first doctoral student of Hans-Jörg Kreowski. She joined his team as a research associate in 1986. Having received her doctoral degree in 1989, she continued to work in his team as an assistant professor until 1995, when she was offered a professorship in Hildesheim, and later moved to Oldenburg.

Leen Lambers

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
D-10587 Berlin (Germany)
leen@cs.tu-berlin.de
http://tfs.cs.tu-berlin.de/~leen

Random Context Picture Grammars: The State of the Art

Sigrid Ewert

Abstract. We present a summary of results on random context picture grammars (rcpgs), which are a method of syntactic picture generation. The productions of such a grammar are context-free, but their application is regulated—permitted or forbidden—by context randomly distributed in the developing picture. Thus far we have investigated three important subclasses of rcpgs, namely random permitting context picture grammars, random forbidding context picture grammars and table-driven context-free picture grammars. For each subclass we have proven characterization theorems and shown that it is properly contained in the class of rcpgs. We have also developed a characterization theorem for all picture sets generated by rcpgs, and used it to find a set that cannot be generated by any rcpg.

Key words: formal languages, picture grammars, syntactic picture generation, image analysis, random context grammars, scene understanding

1 Introduction

Picture generation is a challenging task in Computer Science and applied areas, such as document processing (character recognition), industrial automation (inspection) and medicine (radiology).

Syntactic methods of picture generation have become established during the last decade or two. A variety of methods is discussed and extensive lists of references are given in [9, 11, 12]. Random context picture grammars (rcpgs) [7] generate pictures through successive refinement. They are context-free grammars with regulated rewriting; the motivation for their development was the fact that context-free grammars are often too weak to describe a given picture set, eg. the approximations of the Sierpiński carpet, while context-sensitive grammars are too complex to use.

Random context picture grammars have at least three interesting subclasses, namely random permitting context picture grammars (rPcpgs), random forbidding context picture grammars (rFcpgs) and table-driven context-free picture grammars (Tcfpgs). For each of these classes we have developed characterization theorems. In particular, for rPcpgs we proved a pumping lemma and used it to show that these grammars are strictly weaker than rcpgs [5]. For rFcpgs we proved a shrinking lemma [4], and showed that they too are strictly weaker than rcpgs [6]. In the case of Tcfpgs, we developed two characterization theorems and showed that these grammars are strictly weaker than rFcpgs [1].

Finally, we have developed a characterization theorem for all galleries generated by rcpgs, and used it to find a picture set, more commonly known as a gallery, that cannot be generated by any rcpg [13].

In this paper we present a summary of the above results. We formally define rcpgs in Section 2. In Section 3 we present the pumping lemma for rPcpgs, and use it to show that no rPcpg can generate the approximations of the Sierpiński carpet. In Section 4 we state the shrinking lemma for rFcpgs, and present a gallery that cannot be generated by any rFcpg. Then, in Section 5, we define Tcfpgs, present two characterization theorems for these grammars and show that they are strictly weaker than rFcpgs. In Section 6 we present a property of all galleries generated with rcpgs, and then construct a gallery that does not belong to this class. We briefly touch on a generalization of rcpgs in Section 7. Future work is recommended in Section 8.

2 Random context picture grammars

In this section we introduce random context picture grammars. For picture grammars we need a geometric context; we choose the situation of squares divided into equal squares.

In the following, let $\mathbb{N}_+ = \{1, 2, 3, \ldots\}$. For $k \in \mathbb{N}_+$, let $[k] = \{1, 2, \ldots, k\}$.

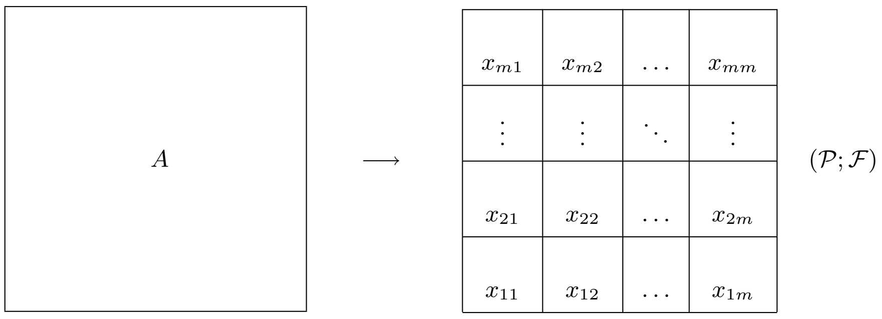


Fig. 1. Production.

Random context picture grammars generate pictures using productions of the form in Figure 1, where A is a variable, $m \in \mathbb{N}_+$, $x_{12}, \ldots, x_{mm}$ are variables or terminals, and $\mathcal{P}$ and $\mathcal{F}$ are sets of variables. The interpretation is as follows: if a developing picture contains a square labelled A and if all variables of $\mathcal{P}$ and none of $\mathcal{F}$ appear as labels of squares in the picture, then the square labelled A may be divided into equal squares with labels $x_{11}, x_{12}, \ldots, x_{mm}$.

We denote a square by a lowercase Greek letter, eg., (A, α) denotes a square α labelled A. If α is a square, $\alpha_{11}, \alpha_{12}, \ldots, \alpha_{mm}$ denote the equal subsquares into which α can be divided, with, eg., α_{11} denoting the left bottom one.

A *random context picture grammar* $G = (V_\mathrm{N}, V_\mathrm{T}, P, (S, \sigma))$ has a finite alphabet V of *labels*, consisting of disjoint subsets V_N of *variables* and V_T of *terminals*. P is a finite set of *productions* of the form $A \to [x_{11}, x_{12}, \ldots, x_{mm}](\mathcal{P}; \mathcal{F})$,

$m \in \mathbb{N}_+$, where $A \in V_{\mathrm{N}}$, $x_{11}, x_{12}, \ldots, x_{mm} \in V$ and $\mathcal{P}, \mathcal{F} \subseteq V_{\mathrm{N}}$. Finally, there is an *initial labelled square* (S, σ) with $S \in V_{\mathrm{N}}$.

A *pictorial form* is any finite set of nonoverlapping labelled squares in the plane. If Π is a pictorial form, we denote by $l(\Pi)$ the set of labels used in Π.

Thirdly, the *size* of a pictorial form Π is the number of squares contained in it, denoted $|\Pi|$.

For an rcpg G and pictorial forms Π and Γ we write $\Pi \Longrightarrow_{\mathrm{G}} \Gamma$ if there is a production $A \to [x_{11}, x_{12}, \ldots, x_{mm}](\mathcal{P}; \mathcal{F})$ in G, Π contains a labelled square (A, α), $l(\Pi \backslash \{(A, \alpha)\}) \supseteq \mathcal{P}$ and $l(\Pi \backslash \{(A, \alpha)\}) \cap \mathcal{F} = \emptyset$, and $\Gamma = (\Pi \backslash \{(A, \alpha)\}) \cup \{(x_{11}, \alpha_{11}), (x_{12}, \alpha_{12}), \ldots, (x_{mm}, \alpha_{mm})\}$. As usual, $\Longrightarrow_{\mathrm{G}}^{*}$ denotes the reflexive transitive closure of $\Longrightarrow_{\mathrm{G}}$.

If every production in G has $\mathcal{P} = \mathcal{F} = \emptyset$, we call G a *context-free picture grammar (cfpg)*; if $\mathcal{F} = \emptyset$ for every production, G is a *random permitting context picture grammar*, and when $\mathcal{P} = \emptyset$, G is a *random forbidding context picture grammar*. The *gallery* $\mathcal{G}(G)$ *generated by a grammar* $G = (V_{\mathrm{N}}, V_{\mathrm{T}}, P, (S, \sigma))$ is $\{\Phi \mid \{(S, \sigma)\} \Longrightarrow_{\mathrm{G}}^{*} \Phi$ and $l(\Phi) \subseteq V_{\mathrm{T}}\}$. An element of $\mathcal{G}(G)$ is called a *picture*.

Let Φ be a picture in the square σ. For any $m \in \mathbb{N}_+$, let σ be divided into equal subsquares, say $\sigma_{11}, \sigma_{12}, \ldots, \sigma_{mm}$. A *subpicture* Γ of Φ is any subset of Φ that fills a square $\sigma_{ij}, i, j \in [m]$, i.e., the union of all the squares in Γ is the square σ_{ij}.

Finally, please note that we write a production $A \to [x_{11}](\mathcal{P}; \mathcal{F})$ as $A \to x_{11}(\mathcal{P}; \mathcal{F})$.

3 Permitting context only

In this section we concentrate on grammars that use permitting context only. We present a pumping lemma for the corresponding galleries, and show that rPcpgs cannot generate $\mathcal{G}_{\mathrm{carpet}}$, the gallery of approximations of the Sierpiński carpet.

We first introduce some notation. Let Π be a pictorial form that occupies a square α, i.e., the union of all the squares in Π is the square α; this we denote by (Π, α). Let β be any square in the plane. Then $(\Pi \to \beta)$ denotes the pictorial form obtained from Π by uniformly scaling (up or down) and translating all the labeled squares in Π to fill the square β, retaining all the labels.

The pumping lemma for rPcpgs and some corollaries are proven in [5]. It states:

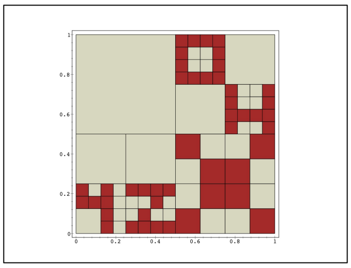
Theorem 1. *For any rPcpg G there is an $m \in \mathbb{N}_+$ such that for any picture $\Phi \in \mathcal{G}(G)$ with $|\Phi| \geq m$ there is a number l, $l \in [m]$, such that:*

1. *Φ contains l mutually disjoint nonempty subpictures $(\Omega_1, \alpha_1), \ldots, (\Omega_l, \alpha_l)$ and l mutually disjoint nonempty subpictures $(\Psi_1, \beta_1), \ldots, (\Psi_l, \beta_l)$, these being related by a function $\vartheta : \{1, \ldots, l\} \to \{1, \ldots, l\}$ such that for each i, $i \in [l]$, $\beta_i \subseteq \alpha_{\vartheta(i)}$ and for at least one i, $i \in [l]$, $\beta_i \subsetneq \alpha_{\vartheta(i)}$;*
2. *the picture obtained from Φ by substituting $(\Omega_i \to \beta_i)$ for (Ψ_i, β_i) for all i, $i \in [l]$, is in $\mathcal{G}(G)$;*

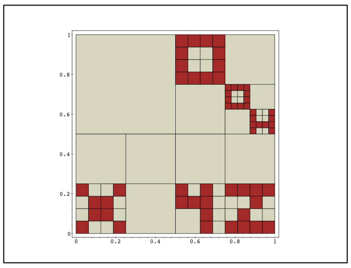
3. *recursively carrying out the operation described in (2) always results in a picture in $\mathcal{G}(G)$.*

Example 1. Consider Φ^1 in Figure 2(a). Let (Ω_1, α_1) be the lower left hand quarter, (Ω_2, α_2) the lower right hand quarter, (Ω_3, α_3) the upper left hand quarter and (Ω_4, α_4) the upper right hand quarter of Φ^1. Furthermore, let (Ψ_1, β_1) be equal to (Ω_2, α_2), (Ψ_2, β_2) the letter Y, (Ψ_3, β_3) the letter Z and (Ψ_4, β_4) the letter H. Then $\vartheta(1) = 2$, $\vartheta(2) = \vartheta(3) = 1$ and $\vartheta(4) = 4$.

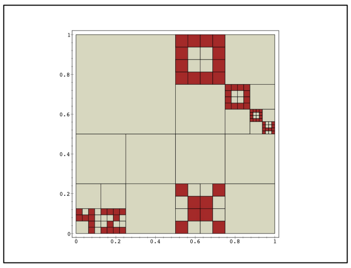
We obtain Φ^2 in Figure 2(b) by substituting $(\Omega_i \to \beta_i)$ for (Ψ_i, β_i), $i \in [4]$, in Φ^1. Then we obtain Φ^3 in Figure 2(c) by carrying out this operation on Φ^2.



(a) Φ^1



(b) Φ^2



(c) Φ^3

Fig. 2. Pumping Φ^1.

An immediate consequence of the pumping property is that the set of sizes of the pictures in an infinite gallery generated by an rPcpg contains an infinite arithmetic progression. From this it follows that $\mathcal{G}_{\text{carpet}}$, two pictures of which are shown in Figure 3, cannot be generated using permitting context only. This gallery can be created by an rFcpg, as is shown in [1].

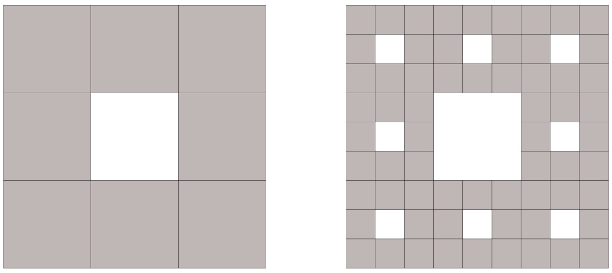


Fig. 3. Two pictures from $\mathcal{G}_{\text{carpet}}$.

4 Forbidding context only

In this section we concentrate on grammars that use forbidding context only and present a shrinking lemma for the corresponding galleries. The lemma is proven in [4] and states:

Theorem 2. *Let G be an rFcpg. For any integer $t \geq 2$ there exists an integer $k = k(t)$ such that for any picture $\Phi \in \mathcal{G}(G)$ with $|\Phi| \geq k$ there are t pictures $\Phi^1, \ldots, \Phi^t = \Phi$ in $\mathcal{G}(G)$ and $t-1$ numbers $l_2, \ldots, l_t$ such that for each j, $2 \leq j \leq t$,*

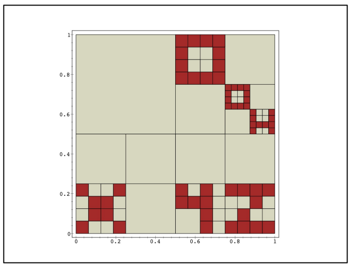
1. *Φ^j contains l_j mutually disjoint nonempty subpictures (Φ_{j1}, α_{j1}), …, $(\Phi_{jl_j}, \alpha_{jl_j})$ and l_j mutually disjoint nonempty subpictures (ϕ_{j1}, β_{j1}), …, $(\phi_{jl_j}, \beta_{jl_j})$, these being related by a function $\vartheta_j : \{1, \ldots, l_j\} \to \{1, \ldots, l_j\}$ such that for each i, $i \in [l_j]$, $\beta_{ji} \subseteq \alpha_{j\vartheta_j(i)}$ and for at least one i, $i \in [l_j]$, $\beta_{ji} \subsetneq \alpha_{j\vartheta_j(i)}$;*
2. *the picture Φ^{j-1} is obtained by substituting $(\phi_{ji} \to \alpha_{ji})$ for (Φ_{ji}, α_{ji}) for all i, $i \in [l_j]$, in Φ^j.*

Example 2. Consider Φ^3 in Figure 4(a). We can choose (Φ_{31}, α_{31}) as the lower left hand quarter, (Φ_{32}, α_{32}) as the lower right hand quarter and (Φ_{33}, α_{33}) as the upper right hand quarter of Φ^3, furthermore (ϕ_{31}, β_{31}) equal to (Φ_{32}, α_{32}), (ϕ_{32}, β_{32}) as the letter X and (ϕ_{33}, β_{33}) as the lower right hand quarter of (Φ_{33}, α_{33}). Here $l_3 = 3$ and $\vartheta_3(1) = 2$, $\vartheta_3(2) = 1$ and $\vartheta_3(3) = 3$.

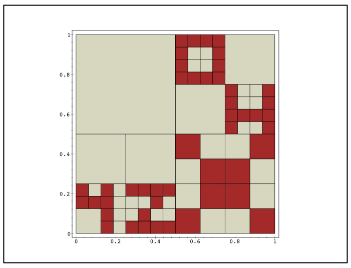
Φ^2 in Figure 4(b) is obtained by substituting $(\phi_{3i} \to \alpha_{3i})$ for (Φ_{3i}, α_{3i}), $1 \leq i \leq 3$, in Φ^3.

Now consider Φ^2 in Figure 4(b). We can choose (Φ_{21}, α_{21}) as the lower left hand quarter, (Φ_{22}, α_{22}) as the lower right hand quarter, (Φ_{23}, α_{23}) as the upper left hand quarter and (Φ_{24}, α_{24}) as the upper right hand quarter of Φ^2, furthermore (ϕ_{21}, β_{21}) equal to (Φ_{22}, α_{22}), (ϕ_{22}, β_{22}) as the letter Y, (ϕ_{23}, β_{23}) as the letter Z and (ϕ_{24}, β_{24}) as the letter H. Here $l_2 = 4$ and $\vartheta_2(1) = 2$, $\vartheta_2(2) = \vartheta_2(3) = 1$ and $\vartheta_2(4) = 4$.

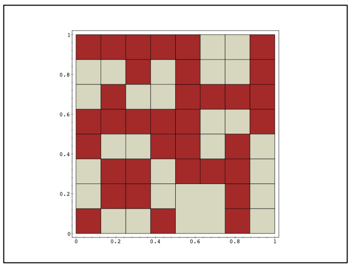
Φ^1 in Figure 4(c) is obtained by substituting $(\phi_{2i} \to \alpha_{2i})$ for (Φ_{2i}, α_{2i}), $1 \leq i \leq 4$, in Φ^2.



(a) Φ^3



(b) Φ^2



(c) Φ^1

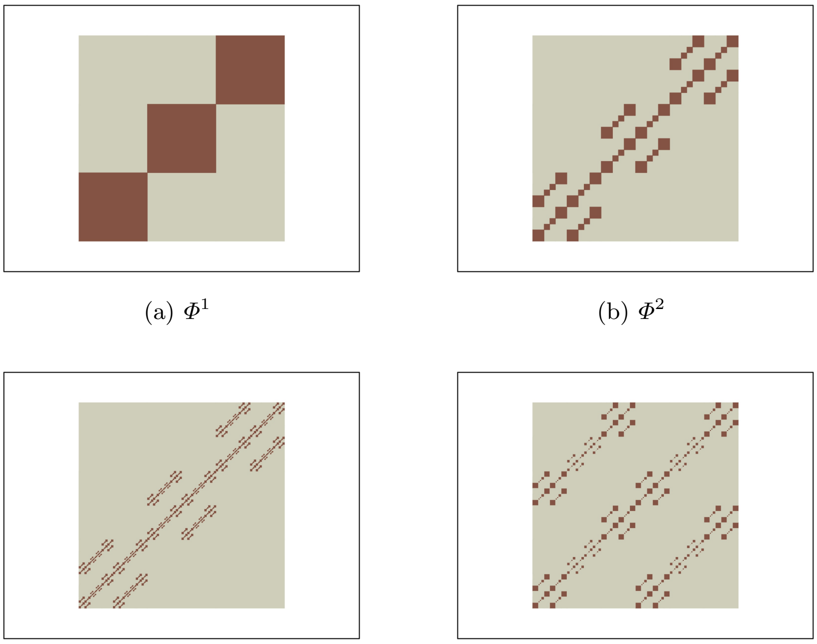
Fig. 4. Shrinking Φ^3.

In [6] we use the technique developed for the proof of the shrinking lemma to show that a certain gallery, $\mathcal{G}_{\text{trail}}$, cannot be generated by any rFcpg, but can be generated by an rcpg. Therefore rFcpgs are strictly weaker than rcpgs.

Consider $\mathcal{G}_{\text{trail}} = \{\Phi^1, \Phi^2, \ldots\}$, where Φ^1, Φ^2 and Φ^3 are shown in Figures 5(a), 5(b) and 5(c), respectively. For the sake of clarity, an enlargement of the bottom left hand ninth of Φ^3 is given in Figure 5(d).

For $i = 2, 3, \ldots$, Φ^i is obtained by dividing each dark square in Φ^{i-1} into four and placing a copy of Φ^1, modified so that it has exactly $i + 2$ dark squares, all on the bottom left to top right diagonal, into each quarter.

The modification of Φ^1 is effected in its middle dark square only and proceeds in detail as follows: The square is divided into four and the newly-created bottom left hand quarter coloured dark. The newly-created top right hand quarter is again divided into four and its bottom left hand quarter coloured dark. This successive quartering of the top right hand square is repeated until a total of $i - 1$ dark squares have been created, then the top right hand square is also coloured dark. The new dark squares thus get successively smaller, except for the last two, which are of equal size.



(a) Φ^1

(b) Φ^2

(c) Φ^3

(d) Bottom left hand ninth of Φ^3 enlarged

Fig. 5. The pictures Φ^1, Φ^2 and Φ^3 from $\mathcal{G}_{\text{trail}}$.

5 Table-driven context-free picture grammars

In [1] we introduce table-driven context-free picture grammars, and compare them to cfpgs, rPcpgs and rFcpgs. We also give two necessary conditions for a gallery to be generated by a Tcfpg, and use them to find galleries that cannot be made by any Tcfpg.

A *table-driven context-free picture grammar* is a system $G = (V_N, V_T, \mathcal{T}, (S, \sigma))$, where V_N, V_T, $V = V_N \cup V_T$ and (S, σ) are as defined in Section 2. $\mathcal{T}$ is a finite set of *tables*, each table $R \in \mathcal{T}$ satisfying the following two conditions:

1. R is a finite set of productions of the form $A \to [x_{11}, x_{12}, \ldots, x_{mm}]$, $m \in \mathbb{N}_+$, where $A \in V_N$, and $x_{11}, x_{12}, \ldots, x_{mm} \in V$.
2. R is complete, i.e., for each $A \in V_N$, there exist an $m \in \mathbb{N}_+$ and $x_{11}, x_{12}, \ldots, x_{mm} \in V$ such that $A \to [x_{11}, x_{12}, \ldots, x_{mm}]$ is in R.

As in the case of rcpgs, the squares containing variables are replaced, but the terminals are never rewritten. Every direct derivation must replace all variables in the pictorial form; the completeness condition ensures that this is possible.

For any production p, say $A \to [x_{11}, x_{12}, \ldots, x_{mm}]$, A is called the left hand side of p, and $[x_{11}, x_{12}, \ldots, x_{mm}]$ the right hand side of p, denoted by $\mathrm{lhs}\,(p)$ and $\mathrm{rhs}\,(p)$, respectively.

For a labelled square (A, α) and a production p with $A = \mathrm{lhs}\,(p)$, say $A \to [x_{11}, x_{12}, \ldots, x_{mm}]$, $m \in \mathbb{N}_+$, we denote $\{(x_{11}, \alpha_{11}), (x_{12}, \alpha_{12}), \ldots, (x_{mm}, \alpha_{mm})\}$ by $\mathrm{repl}\,((A, \alpha))\, p$ [1].

For pictorial form Π, we define $\mathrm{var}\,(\Pi) = \{(A, \alpha) \in \Pi \mid A \in V_\mathrm{N}\}$. For pictorial form Π and table R, we call $\mathrm{b} : \mathrm{var}\,(\Pi) \to R$ a *base* [2] on Π if for each $(A, \alpha) \in \mathrm{var}\,(\Pi)$, $\mathrm{lhs}\,(\mathrm{b}\,((A, \alpha))) = A$.

Let Π and Γ be pictorial forms. We say that Π directly derives Γ ($\Pi \Longrightarrow \Gamma$) if there exists a base b on Π such that

$$\Gamma = \Pi \setminus \mathrm{var}\,(\Pi) \cup \bigcup_{(A,\alpha) \in \mathrm{var}(\Pi)} \mathrm{repl}\,((A, \alpha))\, \mathrm{b}\,((A, \alpha)).$$

For Tcfpgs, the terms $\Longrightarrow_G^*$, *gallery*, and *picture* are defined as for rcpgs in Section 2.

Finally, please note that we write a production $A \to [x_{11}]$ as $A \to x_{11}$.

In [1] we present a Tcfpg that generates $\mathcal{G}_\mathrm{carpet}$. From this it follows that Tcfpgs can generate a gallery that no rPcpg can and that Tcfpgs are strictly more powerful than context-free picture grammars.

In [1] we state two necessary conditions for a gallery to be generated by a Tcfpg.

Before we can state the first such condition, we need a definition. Let Π be a pictorial form and B a set. Then $\#_B\,(\Pi)$ denotes the number of occurrences of elements of B in Π.

Theorem 3. *Let $\mathcal{G}$ be a gallery generated by a Tcfpg with terminal alphabet V_T. Then for every $B \subseteq V_\mathrm{T}$, $B \neq \emptyset$, there exists a positive integer k such that, for every picture $\Phi \in \mathcal{G}$ either*

1. *$\#_B\,(\Phi) \leq 1$, or*
2. *Φ contains a subpicture Ψ such that $|\Psi| \leq k$ and $\#_B\,(\Psi) \geq 2$, or*
3. *there exist infinitely many $\Upsilon \in \mathcal{G}$ such that $\#_B\,(\Upsilon) = \#_B\,(\Phi)$.*

In [1] we use Theorem 3 to show that a certain gallery, $\mathcal{G}_{\mathrm{not-Tcfpg}}$, cannot be generated by any Tcfpg. Consider $\mathcal{G}_{\mathrm{not-Tcfpg}} = \{\Phi^1, \Phi^2, \ldots\}$, where Φ^1, Φ^2 and Φ^3 are given in Figure 6 from left to right. Let the terminals b, g and w represent squares with the colours black, grey and white respectively. Then Φ^n, $n \in \mathbb{N}_+$, is such that the terminals on its diagonal, read from bottom left to top right, form the string $b\,(bg^n)^n$, while the rest of the picture is white.

Before we can state the second necessary condition for a gallery to be generated by a Tcfpg, we introduce the properties *nonfrequent* and *rare*, which are based on properties presented in [3]. Let $\mathcal{G}$ be a set of pictures with labels from the alphabet V_T, and B a nonempty subset of V_T. Then

[1] The use of "repl" was inspired by the concept *repl* defined in [2].

[2] The use of "base" was inspired by the concept *base* defined in [2].

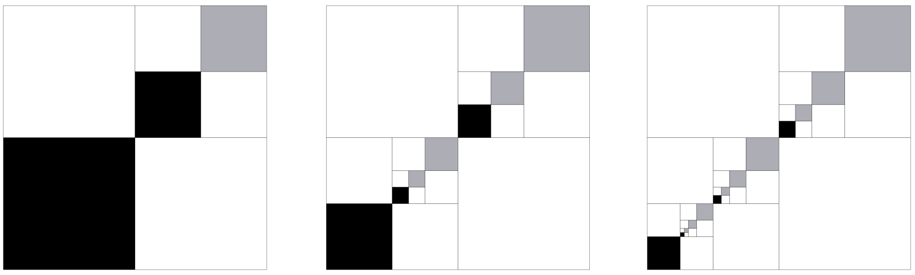


Fig. 6. The pictures Φ^1, Φ^2 and Φ^3 from the gallery $\mathcal{G}_{\text{not}-\text{Tcfpg}}$.

- B is called *nonfrequent* in $\mathcal{G}$ if there exists a constant k such that for every $\Phi \in \mathcal{G}$, $\#_B(\Phi) < k$.
- B is *rare* in $\mathcal{G}$ if for every $k \in \mathbb{N}_+$ there exists an $n_k > 0$ such that for every $n \in \mathbb{N}$ with $n > n_k$, if a picture $\Phi \in \mathcal{G}$ contains n occurrences of letters from B then for each two such occurrences, the smallest subpicture containing those occurrences has size at least k.

Theorem 4. *Let $G = (V_{\mathrm{N}}, V_{\mathrm{T}}, \mathcal{T}, (S, \sigma))$ be a Tcfpg and $B \subseteq V_{\mathrm{T}}, B \neq \emptyset$. If B is rare in $\mathcal{G}(G)$, then B is nonfrequent in $\mathcal{G}(G)$.*

In [1] we use Theorem 4 to show that a certain gallery, $\mathcal{G}_{\text{rFcpg}-\text{Tcfpg}}$, cannot be generated by any Tcfpg. Consider $\mathcal{G}_{\text{rFcpg}-\text{Tcfpg}} = \{\Phi^{m,n} \mid n \geq 1, m \geq n\}$, where $\Phi^{2,2}$ and $\Phi^{4,3}$ are given in Figure 7 from left to right. Let the terminals b, g and w represent squares with the colours black, grey and white respectively. Then $\Phi^{m,n}$ is such that the terminals on its diagonal, read from bottom left to top right, form the string $(bg^m)^n$, while the rest of the picture is white.

In [1] we show that every gallery generated by a Tcfpg can be generated by an rFcpg. Then we present an rFcpg that generates the gallery $\mathcal{G}_{\text{rFcpg}-\text{Tcfpg}}$. From that it follows that Tcfpgs are strictly weaker than rFcpgs.

6 The Limitations of Random Context

In [13] we investigate the limitations of random context picture grammars. First we study those grammars that generate only pictures that are composed of squares of equal size and show that the corresponding galleries enjoy a certain commutativity. This enables us to construct a set of pictures that cannot be generated by any rcpg. Then we generalize the commutativity theorem to the class of all rcpgs.

For the sake of simplicity, we consider only rcpgs of which every production that effects a subdivision produces exactly four subsquares. Also, we let σ be the unit square $((0,0),(1,1))$. The result we state below can be formulated for the case of rcpgs with productions that effect other subdivisions [13].

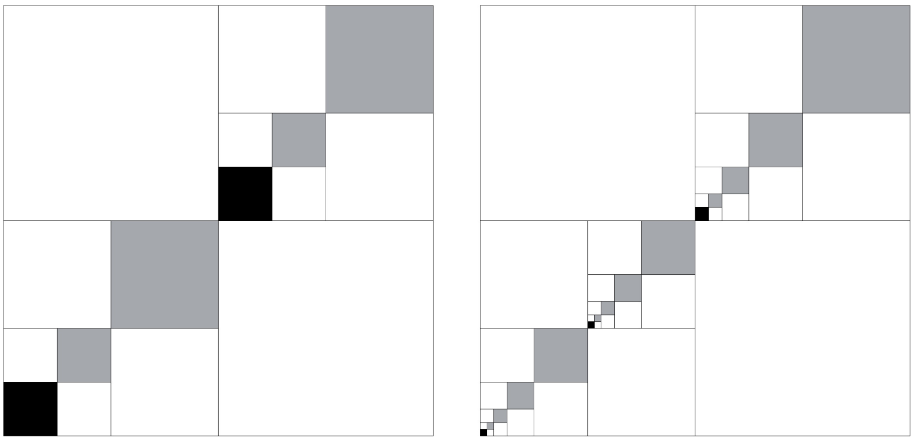


Fig. 7. The pictures $\Phi^{2,2}$ and $\Phi^{4,3}$ from the gallery $\mathcal{G}_{\mathrm{rFcpg-Tcfpg}}$.

Before we can state the theorem, we need some definitions. A picture is called *n-divided*, for $n \in \mathbb{N}_+$, if it consists of 4^n equal subsquares, each labeled with a terminal. For example, the picture on the left hand side of Figure 8 is 4-divided. A *level-m subsquare* of an n-divided picture, with $1 \le m \le n$, is a square $((x2^{-m}, y2^{-m}), ((x+1)2^{-m}, (y+1)2^{-m}))$, where x and y are integers and $0 \le x, y < 2^m$. Note that, for $m < n$, a level-m subsquare consists of all 4^{n-m} labeled subsquares contained in it. For example, the upper left hand quarter of the above mentioned picture is a level-1 subsquare of the picture and consists of 4^3 labeled subsquares.

Two n-divided pictures Φ^1 and Φ^2 are said to *commute at level m* if Φ^1 contains two different level-m subsquares α and β such that Φ^2 can be obtained by simply interchanging the labeling of α and β. A picture Φ^1 is called *self-commutative at level m* if Φ^1 and Φ^1 commute at level m.

In [13] we give a proof of the following theorem:

Theorem 5. *Let $G = (V_N, V_T, P, (S, \sigma))$ be an rcpg that generates an infinite gallery of n-divided pictures, where $n \in \mathbb{N}_+$. Then there exist an m and a c such that each picture that is c-divided is either self-commutative at level m or commutes with another picture in the gallery at level m.*

We now use Theorem 5 to construct a gallery that cannot be generated by any rcpg.

Let $m \in \mathbb{N}_+$. Consider the $2m$-divided picture Φ that is constructed as follows: For any level-m subsquare α in Φ, if α is in row i and column j of Φ, then the level-$2m$ subsquare in row i and column j of α is coloured dark. All the other level-$2m$ subsquares are coloured light.

For example, in Figure 8, $m = 2$. The picture on the left hand side is 2×2-divided, i.e., 4-divided. On the right hand side, we show the level-2 subsquares

α_1 in row 2, column 2, and α_2 in row 3, column 4. The level-4 subsquare in row 2, column 2 of α_1 is coloured dark and all other level-4 subsquares of α_1 are coloured light. Similarly, the level-4 subsquare in row 3, column 4 of α_2 is coloured dark and all other level-4 subsquares of α_2 are coloured light.

Then Φ is not self-commutative at level m. Thus we have:

Theorem 6. *There exists a set of pictures, each consisting of the unit square subdivided into equal subsquares and coloured with two colours, that cannot be generated by an rcpg.*

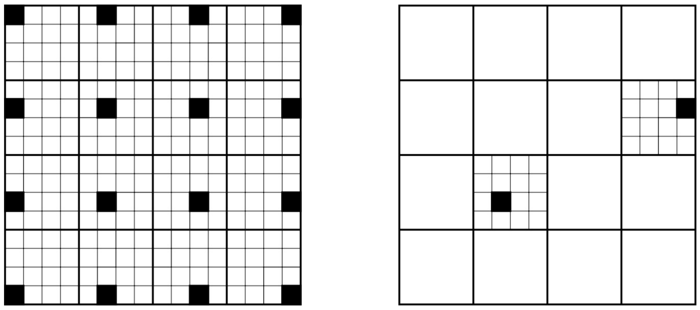


Fig. 8. 4-divided.

In [13] we generalize Theorem 5 to the class of all rcpgs.

7 Generalized Random Context Picture Grammars

As geometric context for random context picture grammars we used squares divided into equal, non-overlapping squares. Clearly we could start with another shape, eg. a triangle, and divide it successively into non-overlapping triangles of equal size. We could also divide a shape into shapes that do not all have the same size or have the same shape as the original. For any given gallery there may be a combination of shapes and arrangement of subshapes that is most effective. This leads us to a generalization of rcpgs, so-called generalized random context picture grammars (grcpgs). In [8] we define grcpgs as grammars where the terminals are subsets of the Euclidean plane and the replacement of variables involves the building of functions that will eventually be applied to terminals. Context is again used to enable or inhibit the application of production rules.

In this form generalized random context picture grammars can be seen as a generalization of (context-free) collage grammars [9].

Iterated Function Systems (IFSs) are among the best-known methods for constructing fractals. In [8] we show that any picture sequence generated by an IFS can also be generated by a grcpg that uses forbidding context only. Moreover, since grcpgs use context to control the sequence in which functions are applied, they can generate a wider range of fractals or, more generally, pictures than IFSs [8].

Mutually Recursive Function Systems (MRFSs) are a generalization of IFSs. In [10] we show that any picture sequence generated by an IFS can also be generated by a grcpg that uses forbidding context only. Moreover, grcpgs can generate sequences of pictures that MRFSs cannot [10].

8 Future work

In this paper we give a summary of results for random context picture grammars and three of their more interesting subclasses, namely random permitting context picture grammars, random forbidding context picture grammars and table-driven context-free picture grammars.

It has been established that Tcfpgs are strictly weaker than rFcpgs. Moreover, it is known that Tcfpgs can generate a gallery that rPcpgs cannot, namely the gallery of approximations of the Sierpiński carpet. However, it is not known whether there exists a gallery that can be generated by an rPcpg, but not by any Tcfpg. Moreover, it is also not known if there is a gallery that can be generated by an rPcpg, but not by any rFcpg.

Acknowledgments The author would like to thank the referees for their helpful comments.

References

1. C. Bhika, S. Ewert, R. Schwartz, and M. Waruhiu. Table-driven context-free picture grammars. *International Journal of Foundations of Computer Science*, 18(6):1151–1160, 2007.
2. F. Drewes, R. Klempien-Hinrichs, and H.-J. Kreowski. Table-driven and context-sensitive collage languages. *Journal of Automata, Languages and Combinatorics*, 8(1):5–24, 2003.
3. A. Ehrenfeucht and G. Rozenberg. On proving that certain languages are not ET0L. *Acta Informatica*, 6:407–415, 1976.
4. S. Ewert and A. van der Walt. Generating pictures using random forbidding context. *International Journal of Pattern Recognition and Artificial Intelligence*, 12(7):939–950, 1998.
5. S. Ewert and A. van der Walt. Generating pictures using random permitting context. *International Journal of Pattern Recognition and Artificial Intelligence*, 13(3):339–355, 1999.
6. S. Ewert and A. van der Walt. A hierarchy result for random forbidding context picture grammars. *International Journal of Pattern Recognition and Artificial Intelligence*, 13(7):997–1007, 1999.
7. S. Ewert and A. van der Walt. Random context picture grammars. *Publicationes Mathematicae (Debrecen)*, 54 (Supp):763–786, 1999.

8. S. Ewert and A. van der Walt. Shrink indecomposable fractals. *Journal of Universal Computer Science*, 5(9):521–531, 1999.
9. A. Habel, H.-J. Kreowski, and S. Taubenberger. Collages and patterns generated by hyperedge replacement. *Languages of Design*, 1:125–145, 1993.
10. H. Kruger and S. Ewert. Translating mutually recursive function systems into generalised random context picture grammars. *South African Computer Journal*, (36):99–109, 2006.
11. A. Nakamura, M. Nivat, A. Saoudi, P. S. P. Wang, and K. Inoue, editors. *Proceedings of the Second International Conference on Parallel Image Analysis, ICPIA '92, Ube, Japan, December 1992*, volume 654 of *Lecture Notes in Computer Science*, Berlin. Springer.
12. A. Rosenfeld and R. Siromoney. Picture languages—a survey. *Languages of Design*, 1:229–245, 1993.
13. A. van der Walt and S. Ewert. A property of random context picture grammars. *Theoretical Computer Science*, (301):313–320, 2003.

Prof. Dr. Sigrid Ewert
School of Computer Science
University of the Witwatersrand, Johannesburg
Private Bag 3
Wits, 2050 (South Africa)
sigrid.ewert@wits.ac.za
http://www.cs.wits.ac.za/~sigrid/

Sigrid Ewert was a guest researcher in Hans-Jörg Kreowski's group from 1999 to 2000.

From Algebraic Specifications to Graph Transformation Rules to UML and OCL Models

Martin Gogolla and Karsten Hölscher

Abstract. Graph transformation and algebraic specification are well-established techniques in theoretical and practical computer science and claim to support software development with fundamental methods in a formal manner. Equational algebraic specifications can be translated into a graph transformation system in a systematic way. A graph transformation system in turn can be analyzed and processed by a number of tools. This paper studies how to step from equational algebraic specifications to graph transformation and from there to an operational representation in various graph transformation tools. We work with USE (UML-based Specification Environment), AGG (Attributed Graph Grammar system), and GrGen (Graph rewrite Generator). In particular, we discuss how to establish a connection between algebraic specifications and UML class diagrams and OCL constraints.

1 Introduction

Equational algebraic specifications [EM85,EGL89,Wir90] as well as graph grammars and graph transformation [Roz97,EEKR99] are two fields which have been studied since about thirty years and which share the use of fundamental categorical and algebraic techniques. Both fields claim to support software development with fundamental methods in a formal manner. In recent years, graph transformation attracted substantial research effort because of its closeness to model-driven and model transformation-oriented approaches. For a graph transformation system, practically applicable tools like AGG [dLT04], FUJABA [BGN$^+$04], GrGen [GK07], GReAT [BNvBK06], MOFLON [AKRS06] or GROOVE [Ren03] have been developed and UML tools like USE have been extended to cope with graph transformation [BG06].

The transformation in our paper basically follows the method for translating algebraic specifications into graph transformation which has been proposed in [Löw90] but which has not been realized in a tool (in contrast to the work presented in this paper). Our work is based on the UML and OCL tool USE developed in our group since about ten years [GBR05,GBR07] and on the tools AGG [dLT04] and GrGen [GK07]. This selection of tools was determined by the fact that the authors have experience and knowhow in the use of these tools. We are sure that the other mentioned tools and further ones can be used for our purpose as well.

One main result of the paper is our observation that it is feasible to build a conceptual bridge between so distant fields like *"hard"* algebraic specifica-

tion (AlgSpec) and *"soft"* popular approaches like the Unified Modeling Language (UML). We regard Graph Transformation (GraTra) as the *missing link*. GraTra tools provide the possibility of analyzing the underlying model. For example, GraTra tools apart from validating the model are able to check the model consistency or can provide a critical pair analysis of the underlying equations in the algebraic specification. Thus a central ingredient in the interplay between AlgSpec and UML is the ability of GraTra to mediate between the other fields and to broadcast results in both directions. There are some scientists which have been working in all three fields. Among them is Hans-Jörg Kreowski. According to DBLP, his earliest contribution in the GraTra field is from 1977 [Kre77], the earliest one on AlgSpec is from 1978 [EKW78], and the earliest one on UML is from 2002 [KGKK02].
The structure of the rest of the paper is as follows. Section 2 introduces our simple running example within the context of equational algebraic specification and graph transformation. Sections 3, 4 and 5 discuss the realization of this example in the tools USE, AGG, and GrGen, respectively. The paper is finished with concluding remarks in Sect. 6.

2 Running Example

We will study the relationship between equational algebraic specification, graph transformation, and tool realizations of graph transformation by a very simple equational algebraic specification for the natural numbers as shown in Fig. 1. The specification includes the two constructors `zero` and `succ` and an operation `plus` realizing the addition on natural numbers. Any term incorporating the operation `plus` can be reduced by means of the equations to a term using only the constructors `zero` and `succ`, and the different terms built over `zero` and `succ` represent all values for the sort `nat`.

```
spec Nat
srts nat
opns zero: -> nat
     succ: nat -> nat
     plus: nat nat -> nat
vars N, M: nat
eqns plus(zero,N) = N
     plus(succ(N),M) = succ(plus(N,M))
```

Fig. 1. Algebraic Example Specification for Addition on Natural Numbers

In Fig. 2 we have represented the two example equations as two graph transformation rules with left and right side: Each operation symbol and each variable becomes a node and the edges connect operation symbols with their arguments. The first argument of the operation `plus` is established with edges labelled `PlusNat1` and the second argument with label `PlusNat2`. Analogously, the argument of the constructor `succ` shows the label `SuccNat`. Although the

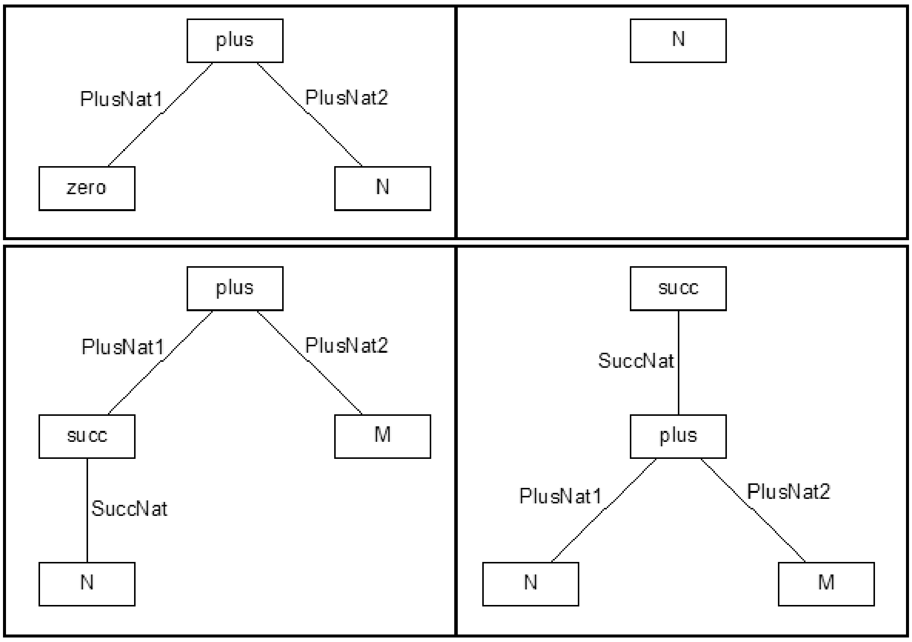


Fig. 2. Naive Representation of Example Equations as a Graph Transformation System

representation seems straight forward, it is too naive for general graph transformation and must be extended as explained below.

The main problem with the above representation in Fig. 2 lies in the fact that the context in which the rules are to be applied is not handled properly. For example, if the first rule is applied in the term `succ(plus(zero,zero))`, the context information that in this case `plus` is a subterm of `succ` is not preserved by the rule. In order to preserve this context information additional nodes are introduced. These nodes embody the context information and explicitly state the type information for each term. This extended representation of the rules is pictured in Fig. 3. The context information is held within the `NatType` nodes. In both rules it is essential, that the topmost `NatType` nodes are preserved by the rules, i.e., the topmost `NatType` nodes appear in the left *and* right side of the rules. Roughly speaking, incoming edges for both rules are handled by the `NatType` nodes `(1)`. Outgoing edges for the first rule are handled by node `(6)` whereas outgoing edges for the second rule are handled by nodes `(6)` and `(8)`.

3 USE

The example graph transformation system corresponding to the algebraic specification is given to USE in textual form. First, the underlying UML class diagram including classes, inheritance relationships, and associations is stated. The overall class diagram is shown in Fig. 4. Currently, our translator from rules to OCL

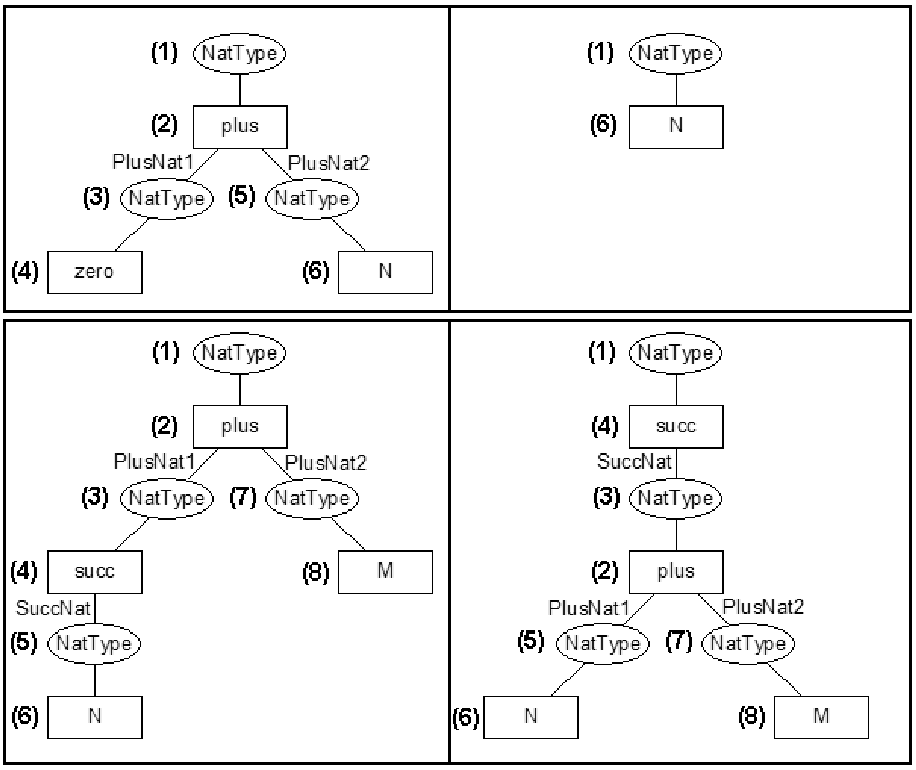


Fig. 3. Representation of Example Equations as a Graph Transformation System

only supports `0..*` multiplicities. Therefore only those multiplicities are stated in the class diagram, although more restricting multiplicities could be chosen.

In addition to the class diagram, the two rules are stated in textual form in Fig. 5 and are called `plusZeroN_2_N` and `plusSuccNM_2_succPlusNM`. These names will be used for generated UML operations. The rules basically make declarations for nodes and edges on the left and right hand side of the rule. In the UML class diagram context, nodes correspond to objects and edges to links. Additionally, OCL conditions could be declared in the left or right hand side, although this feature is not used in this example. OCL conditions in the left side correspond to rule preconditions and OCL conditions in the right side to rule postconditions. Left hand side conditions are called application conditions within the graph transformation area. A representation of the rules in form of UML object diagrams is pictured in Fig. 6.

In Fig. 7 the class diagram generated from the rules resp. the operations generated from the rules are pictured. Each rule induces three operations: The first operation is responsible for applying the rule and for replacing in the so-called working graph a matching left-hand side by the rule's right hand side; the second

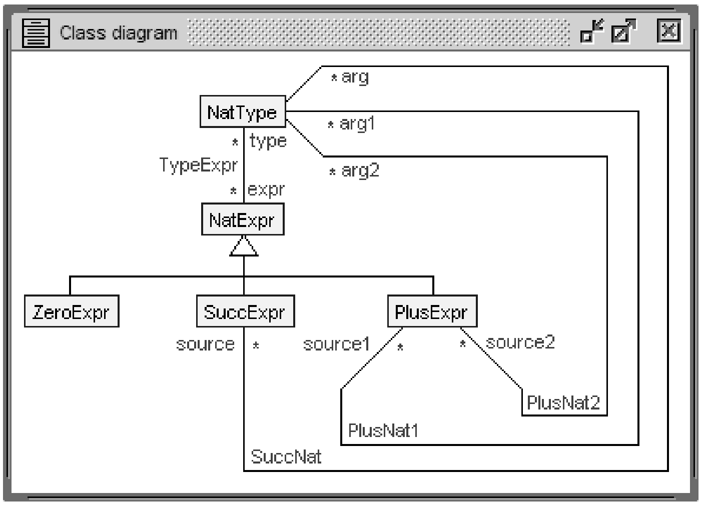


Fig. 4. UML Class Diagram for Terms over Natural Numbers Employed in USE

operation checks the precondition of the rule, and the third operation searches in the working graph for rule redexes, i.e., locations in the working graph where the rule can be applied. The parameters of the operations are determined by the objects (nodes) appearing on the left hand side of the rule.

An example for the reduction of a working graph is given in Fig. 8, basically as a sequence of working graphs in form of UML object diagrams. The reduction corresponds to the calculation `[(0+1)+(0)]+1 = [(0)+(0)]+1+1 = 0+1+1`. The lower part shows the calculation close to a mathematical notation, the middle part uses the naive term representation, and the upper part employs the correct detailed representation with nodes representing the types. The left column represents the term `[(0+1)+(0)]+1`, the middle column the term `[(0)+(0)]+1+1` and the right column the term `0+1+1`. The transition from the left column to the middle column is basically induced by an operation call to `plusSuccNM_2_succPlusNM` corresponding to an application of the second rule from Fig. 3 and the transition from the middle column to the right column by an operation call to `plusZeroN_2_N` corresponding to an application of the first rule from Fig. 3.

The example calculation is also pictured in Fig. 9 in form of a UML sequence diagram. The commands and calls in the sequence diagram can be classified into three parts: The first commands build up the working graph by creating objects representing the start term `[(0+1)+(0)]+1`, the second part is the application of the second rule, and the third part shows the application of the first rule. In the first part, also the links are introduced, however this is not shown in the sequence diagram in order to keep the diagram small. Because object-oriented ideas stand behind USE, every operation call must be directed to an object. Therefore, exactly one object `rc` of class `RuleCollection` is created. The following calls are directed to this object `rc`. The operation call in the third part which corresponds

```
-- plus(zero,N) = N
rule  plusZeroN_2_N
left  N:NatExpr
      NT:NatType
      zero:ZeroExpr
      zeroT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (zeroT,zero):TypeExpr
      (plusT,plus):TypeExpr
      --
      (plus,zeroT):PlusNat1
      (plus,NT):PlusNat2
right N:NatExpr
      --
      plusT:NatType
      --
      (plusT,N):TypeExpr
```

```
-- plus(succ(N),M) = succ(plus(N,M))
rule plusSuccNM_2_succPlusNM
left  N:NatExpr
      NT:NatType
      M:NatExpr
      MT:NatType
      succ:SuccExpr
      succT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (MT,M):TypeExpr
      (succT,succ):TypeExpr
      (plusT,plus):TypeExpr
      --
      (succ,NT):SuccNat
      (plus,succT):PlusNat1
      (plus,MT):PlusNat2
right N:NatExpr
      NT:NatType
      M:NatExpr
      MT:NatType
      succ:SuccExpr
      succT:NatType
      plus:PlusExpr
      plusT:NatType
      --
      (NT,N):TypeExpr
      (MT,M):TypeExpr
      (plusT,succ):TypeExpr
      (succT,plus):TypeExpr
      --
      (succ,succT):SuccNat
      (plus,NT):PlusNat1
      (plus,MT):PlusNat2
```

Fig. 5. Representation of Example Equations in Textual Form

to the application of first rule eliminates certain objects which corresponds to the fact that this rule deletes nodes. The redexes for rule application are determined by calls to the `redexes` operations.

Finally let us comment on the role of OCL within our approach. In USE, OCL plays a central role. In our view, OCL is the formal specification language of the UML and is comparable to other formal specification languages like Z or CASL although the emphasis is much more on practical usability than on theoretical underpinning, for example, on proof theory. The representation of graph transformation in USE is achieved by representing a rule as an operation which is characterized by OCL pre- and postconditions and an operational realization as a command script. Furthermore, OCL can be employed for analyzing the work-

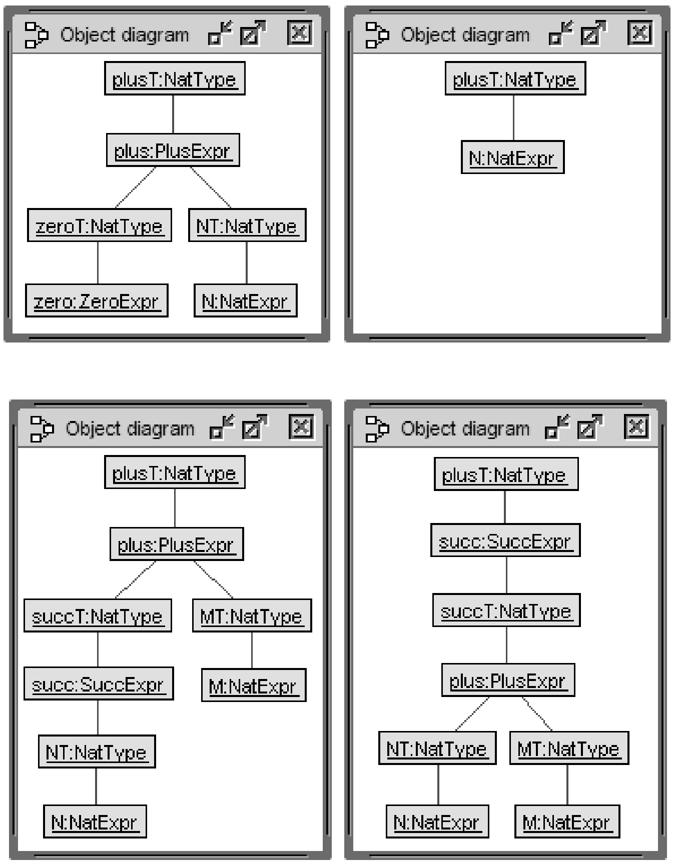
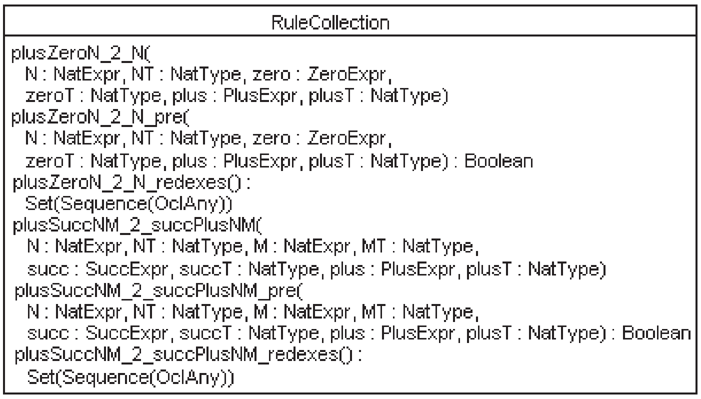


Fig. 6. Representation of Example Equations as Object Diagram Pairs



```
RuleCollection
plusZeroN_2_N(
  N : NatExpr, NT : NatType, zero : ZeroExpr,
  zeroT : NatType, plus : PlusExpr, plusT : NatType)
plusZeroN_2_N_pre(
  N : NatExpr, NT : NatType, zero : ZeroExpr,
  zeroT : NatType, plus : PlusExpr, plusT : NatType) : Boolean
plusZeroN_2_N_redexes() :
  Set(Sequence(OclAny))
plusSuccNM_2_succPlusNM(
  N : NatExpr, NT : NatType, M : NatExpr, MT : NatType,
  succ : SuccExpr, succT : NatType, plus : PlusExpr, plusT : NatType)
plusSuccNM_2_succPlusNM_pre(
  N : NatExpr, NT : NatType, M : NatExpr, MT : NatType,
  succ : SuccExpr, succT : NatType, plus : PlusExpr, plusT : NatType) : Boolean
plusSuccNM_2_succPlusNM_redexes() :
  Set(Sequence(OclAny))
```

Fig. 7. Class Diagram Induced by the Rules

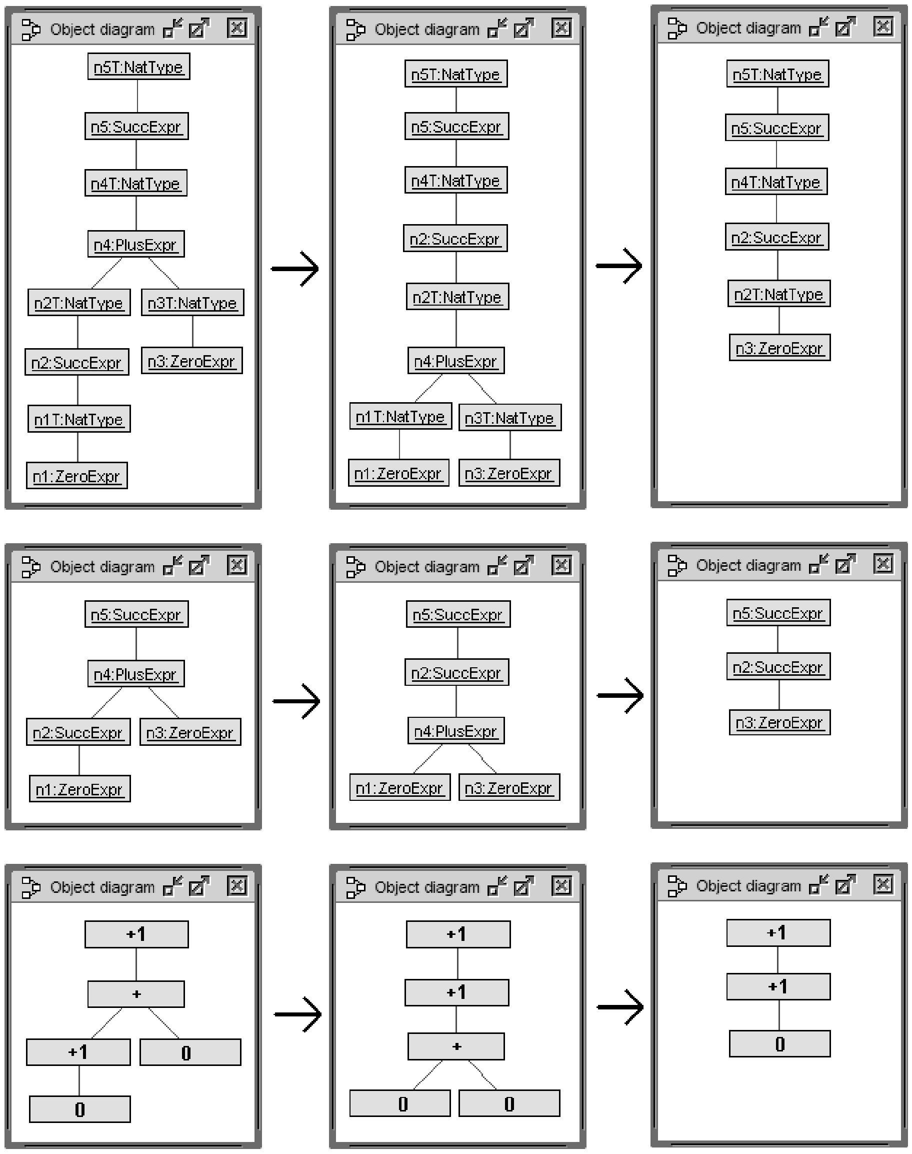


Fig. 8. Example Calculation as an Object Diagram Filmstrip

ing graph at any stage during its development by querying the underlying UML object diagram.

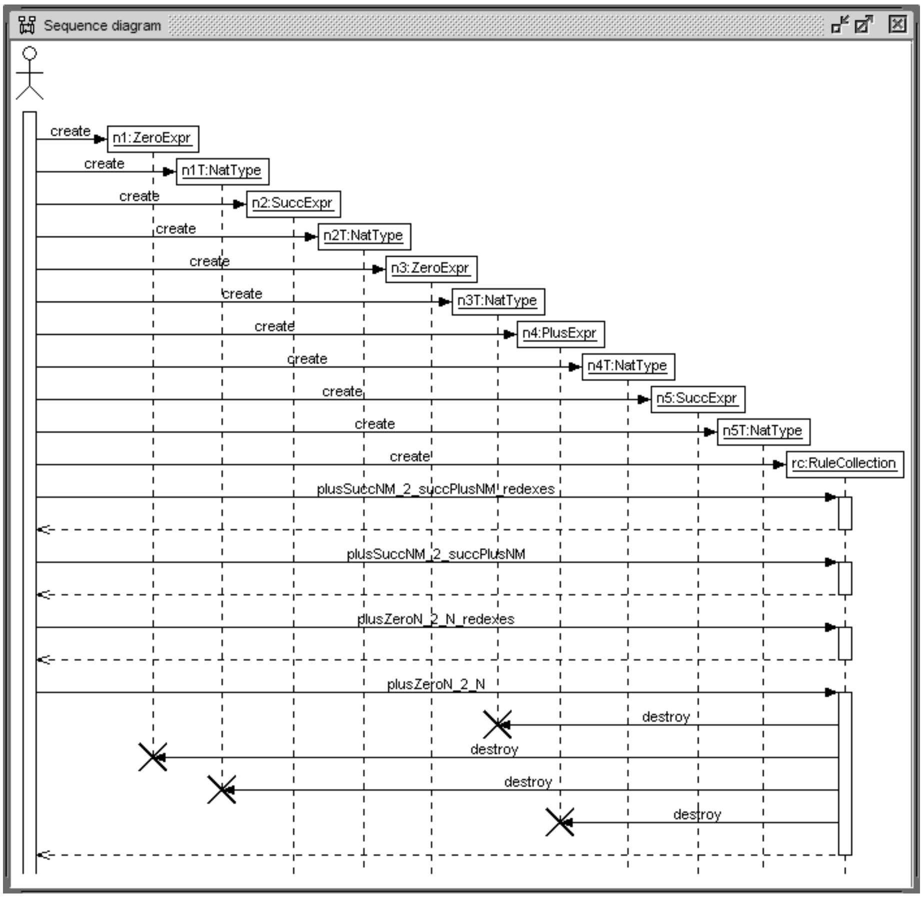


Fig. 9. Example Calculation as a UML Sequence Diagram

4 AGG

In order to animate graph transformation in AGG it is necessary to specify a graph grammar first. Such a grammar makes declarations for the node and edge types of the needed graph items. In the case of the sample algebraic specification from Fig. 1, a straightforward translation requires a node of type `Nat` as a base node for the sort. Additionally, a node of type `NatType` is needed for the same reason as explained in Section 2. Now for every operation of the algebraic specification a corresponding node type being a subtype of `Nat` is needed. An inheritance relation cannot be specified in the AGG grammar, but in a type graph for the corresponding grammar. For this reason the type graph in Fig. 10 is created. In AGG this can be done in a graphical editor. The type graph specifies the nodes `PlusExp`, `SuccExp`, and `ZeroExp` to be subtypes of the node `Nat`.

Additionally edge types are declared for the arguments of the operations defined in the algebraic specification. Since the order of these arguments is usually im-

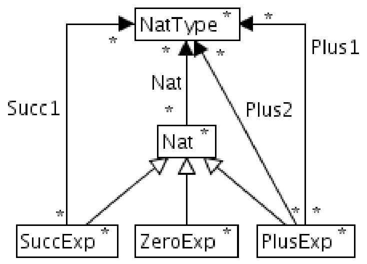


Fig. 10. Type Graph for Terms over Natural Numbers Employed in AGG

portant, a digit is appended to the type names. So the additional edge types are `Succ1`, `Plus1`, and `Plus2`.

Having specified a suitable graph grammar, it is now possible to create a host graph for the transformation, which can also be done in a graphical editor. Fig. 11 shows the host graph for the sample term `succ(plus(succ(zero),zero))`. The recipe for creating a host graph of a given equation is to read the equation from left to right and add the corresponding nodes and edges in that order. For every node that represents a sort it is additionally necessary to add a corresponding node for the context.

The sample term in Fig. 11 starts with `succ`, so a node of type `SuccExp` is added. This node is connected to a newly created context node `NatType` using an edge of type `Nat`. Since a `Nat` node is always attached to a `NatType` context node in this way, it is hence refered to as a `Nat` node pair.

The next operation occurring in the term is `plus`, so a `PlusExp` node pair is inserted into the graph. Since this node pair is an argument to the previous `succ` operation, an edge of type `Succ1` is added which connects the `SuccExp` node with the `NatType` node belonging to the new `PlusExp` node. The remainder of the equation can be treated in an analogous way.

Now that we have specified a host graph, the actual graph transformation rules can be derived. The specification contains two equations. These equations can be translated into a graph transformation rule in an analogous way as the translation above. The left-hand side of the equation is translated to a graph which becomes the left-hand side of the rule. Similarly the right-hand side of the equation becomes the right-hand side of the rule. The rule creation is finished by the specification of those elements that have to be preserved when the rule is actually applied.

Consider the first equation `plus(zero,N)=N` of the specification. The left-hand side of this equation, i.e., `plus(zero,N)` has to be translated into a graph first. As previously explained, this is accomplished by reading the expression from left to right and inserting corresponding elements into the graph. The expression starts with `plus`, so a new `PlusExp` node pair is inserted into the graph. The first argument of the operation `plus` is `zero`, so a new `ZeroExp` node pair is added to

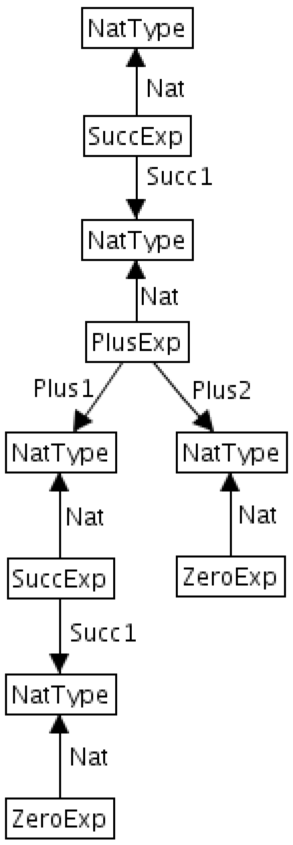


Fig. 11. AGG Graph Representation of `succ(plus(succ(zero),zero))`

the graph. Additionally the `PlusExp` node is connected to the `NatType` node of the `ZeroExp` node pair with an edge of type `Plus1`. The second argument of `plus` in the equation is `N`, so a new `Nat` node pair is created. The `NatType` node of this pair is connected to the `PlusExp` node with an edge of type `Plus2`. Since `N` is not further specified, the concrete subtype is not known. For this reason a node of the supertype `Nat` is used.

The right-hand side of the equation is `N`. So the graph for the right-hand side of the new rule contains only a `Nat` node pair.

In AGG a rule can also be created in a graphical editor. Fig. 12 shows a screenshot of the rule corresponding to the equation `plus(zero,N)=N`.

The graph to the left of the vertical bar is the left-hand side of the rule while the graph to its right is the right-hand side. The items that have to be preserved are represented by identical numbers in graph elements of the left-hand and the right-hand side. In this case, only the `NatType` node indicated by `1:` and the `Nat` node indicated by `2:` are specified as elements that have to be preserved when the rule is applied. Since the equation specifies that `plus` and `zero` do not occur

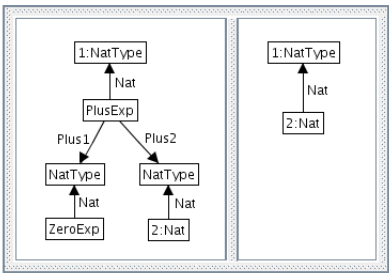


Fig. 12. AGG Rule Representing the Equation `plus(zero,N)=N`

in the right-hand side, the graph transformation rule specifies to delete the corresponding nodes. It may be an intuitive approach to simply keep the `N:Nat` node pair when the rule is applied, but this may yield a wrong result. If the `NatType` node of the `PlusExp` node pair is connected to another node, e.g., to a `SuccExp` node via a `Succ1` edge (representing an expression like `succ(plus(...))`, then this connection would be deleted together with the respective `NatType` node. For this reason, the `NatType` node of the `Nat` node pair representing the first term in the equation expression always has to be preserved. The additional context following `N` is preserved anyway, since the `Nat` node representing `N` is preserved and with it all its connections.

So the rule creation works as stated above bearing in mind that the the topmost (in the sense of no incoming edges) `NatType` of the left-hand side has to be preserved.

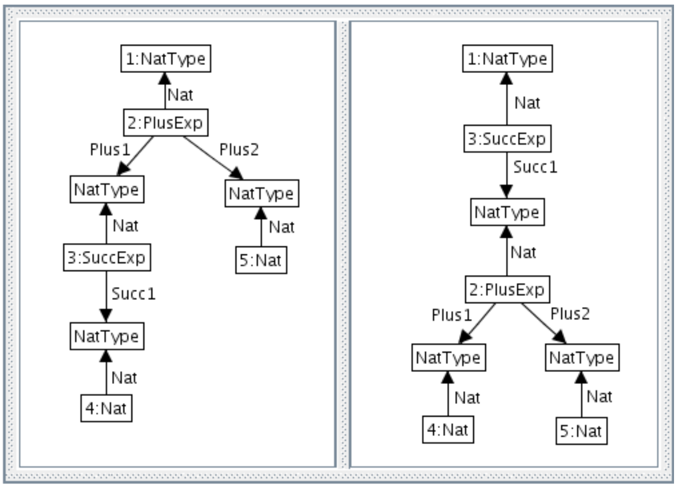


Fig. 13. AGG Rule Representing the Equation `plus(succ(N),M)=succ(plus(N,M))`

Fig. 13 pictures the AGG rule that corresponds to the second equation `plus(succ(N),M) = succ(plus(N,M))` of the specification. It has been created analogously to the first rule.

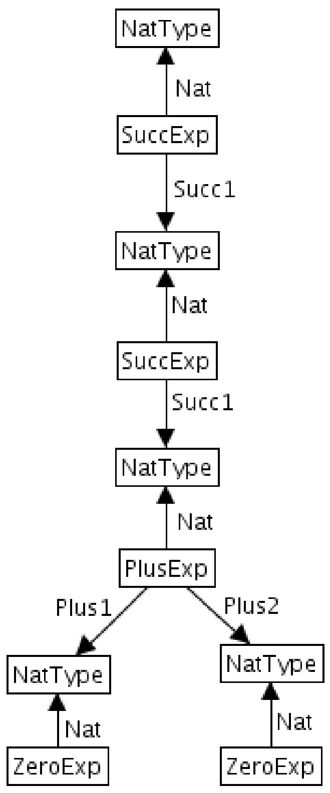


Fig. 14. Transformed Graph After Applying the First Rule in AGG

Considering the host graph from Fig. 11 the first rule cannot be applied. This holds, since the first argument of `plus` would have to be `zero` for the rule to be applicable. The second rule can be applied in exactly one fashion and in the same way as one would apply the second equation. It yields the transformed graph in Fig. 14. The figure shows a screenshot of the actual transformation in AGG which can directly be observed in the GUI version of the tool.

The second rule is not applicable to the transformed graph. This holds, since the rule expects a `succ` as first argument of `plus`. But in the expression represented by the graph, the first argument of the only `plus` is `zero`. For this reason the first graph transformation rule is applicable, yielding the transformed graph depicted in Fig. 15. It represents the term `succ(succ(zero))`, which is the expected result.

5 GrGen

In GrGen the specification of the underlying graph model as well as the graph transformation rules is given in textual form. Since GrGen supports subtypes for nodes and edges as well as subtype matching in rule application, the algebraic

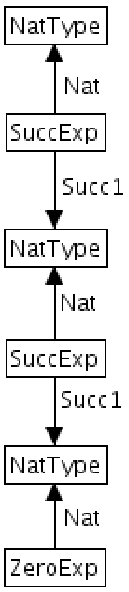


Fig. 15. Transformed Graph After Applying the Second Rule in AGG

specification can be translated in a straight-forward way. The graph model can be derived from a specification in the following way. For every sort there is a node type with the same name and a context node type. Then for every operation that yields a certain sort, there is a node type which extends the node type representing the sort. For every argument of an operation there is an edge type with a type name consisting of the operation name and a successive number to indicate the order of the arguments. Therefore, in the case of the running example specification from Fig. 1, the GrGen graph model description can be stated textually as follows.

```
node class NatType;
node class Nat;

node class Zero extends Nat;
node class Succ extends Nat;
node class Plus extends Nat;

edge class nat;
edge class succ1;
edge class plus1;
edge class plus2;
```

This model can also be pictured as a UML class diagram as shown in Fig. 16. In addition to the simple, but sufficient model we have used here, GrGen would also allow to declare the edges to possess more specific types.

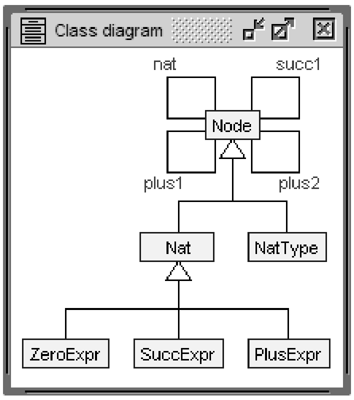


Fig. 16. UML Class Diagram for Terms over Natural Numbers Employed in GrGen

In GrGen a graph transformation rule consists of a `pattern` part and a `replace` part. The `pattern` part represents the left-hand side of the rule, while the `replace` part corresponds to the right-hand side. The graph items that have to be preserved are indicated by using the same identifiers for nodes and edges in both parts.

In order to specify the left-hand side of the rule for `plus(zero,N)` a node pair consisting of the corresponding `Nat` node connected to its context `NatType` node is inserted for every `Nat` expression. Then connecting edges are specified for the operation arguments, which can be directly derived from the specification. The right-hand side is specified analogously. Similarly to the AGG specification the topmost `NatType` node of the left-hand side has to be mapped to the topmost `NatType` node of the right-hand side in order to preserve a possible context. So the first rule looks like this:

```
rule plusZeroN {
  plus:Plus -:nat-> plusType:NatType;
  zero:Zero -:nat-> zeroType:NatType;
  n:Nat -:nat-> nType:NatType;
  plus -:plus1-> zeroType;
  plus -:plus2-> nType;

  replace {
    n -:nat-> plusType;
  }
}
```

Analogously the second rule looks like this:

```
rule plusSuccNM {
  plus:Plus -:nat-> plusType:NatType;
  suc:Succ -:nat-> sucType:NatType;
  n:Nat -:nat-> nType:NatType;
  m:Nat -:nat-> mType:NatType;
  plus -:plus1-> sucType;
  plus -:plus2-> mType;
  suc -:succ1-> nType;

  replace {
    suc -:nat-> plusType;
    suc -:succ1-> newPlusType:NatType;
    plus -:nat-> newPlusType;
    plus -:plus1-> newNType:NatType;
    n -:nat-> newNType;
    plus -:plus2-> newMType:NatType;
    m -:nat-> newMType;
  }
}
```

The actual graph transformation is executed in GrGen's *grshell.* Within grshell, a graph transformation specification can be loaded and a graph can be created manually or by a script. Testing the above specification using an initial graph representing the sample term `succ(plus(succ(zero),zero))` yields the expected result. Initially only the rule `plusSuccNM` and after that only the rule `plusSuccNM` is applicable. For debuging purposes, GrGen is shipped with yComp, a graph visualization tool that draws the current host graph handled in GrGen. Fig. 17 shows a screenshot of the graph after the two rule applications. As expected it is the graph representation of the term `succ(succ(zero))`.

6 Conclusion

This paper has explained how algebraic specification in their basic form as conditional equations can be represented as a graph transformation system and how the result can be validated, animated, and executed in various graph transformation tools. These graph transformation tools offer the possibility of analyzing the underlying model (although we have not demonstrated this feature). We have considered the equational specification as a rewriting system which works in one direction only. The work shows that classical software specification techniques still have a close connection to modern object-oriented techniques like UML. The translation may also be seen as an example for a conceptual model transformation from one computer science field (Algebraic Specification) into another one (UML and OCL). Another aspect of the current work was to show and compare the graph models in the different tools by formally fixed UML class diagrams. These different graph models underpin the flexibility of current graph transformation tools.

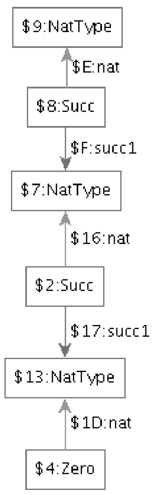


Fig. 17. Final Graph in GrGen Drawn with yComp

Future work might concentrate on the question how to utilize the strengths and analysis features of the different tools in order to give feedback to system developers. Apart from the considered tools, other tools like FUJABA, MOFLON, GReAT, GROOVE, VIATRA, or VMTS might be taken into consideration. The equations might also be treated as rules in both directions. We think that for courses on formal software development the translation which we have proposed gives insight into connections between the different computer science fields, namely algebraic specification, graph transformation, and model-driven development.

References

[AKRS06] C. Amelunxen, A. Königs, T. Rötschke, A. Schürr. MOFLON: A Standard-Compliant Metamodeling Framework with Graph Transformations. In A. Rensink, J. Warmer, editors, *Proc. 2nd Eur. Conf. Model Driven Architecture (ECMDA'2006)*, 361–375. LNCS 4066, Springer, Berlin, 2006.

[BG06] F. Büttner and M. Gogolla. Realizing Graph Transformations by Pre- and Postconditions and Command Sequences. In A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, editors, *Proc. 3rd Int. Conf. Graph Transformations (ICGT'2006)*, 398-412. LNCS 4178, Springer, Berlin, 2006.

[BGN$^+$04] S. Burmester, H. Giese, J. Niere, M. Tichy, J.P. Wadsack, R. Wagner, L. Wendehals, and A. Zündorf. Tool Integration at the Meta-Model Level: The FUJABA Approach. *STTT*, 6(3):203-218, 2004.

[BNvBK06] D. Balasubramanian, A. Narayanan, C.P. van Buskirk, and G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *ECEASST*, 1, 2006.

[dLT04] J. de Lara and G. Taentzer. Automated Model Transformation and Its Validation Using AToM 3 and AGG. In A.F. Blackwell, K. Marriott, and A. Shimojima, editors, *Diagrams*, LNCS 2980, 182-198. Springer, 2004.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools.* World Scientific, Singapore, 1999.

[EGL89] H.-D. Ehrich, M. Gogolla, U.W. Lipeck. *Algebraische Spezifikation abstrakter Datentypen.* Teubner, Stuttgart, 1989.

[EKW78] H. Ehrig, H.-J. Kreowski, H. Weber. *Algebraic Specification Schemes for Data Base Systems.* In S. Bing Yao, editor, Proc. 4th Int. Conf. Very Large Data Bases, IEEE, 427-440, 1978.

[EM85] H. Ehrig and B. Mahr. *Fundamentals of Algebraic Specification.* Springer, Berlin, Germany, 1985.

[GBR05] M. Gogolla, J. Bohling, and M. Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386-398, 2005.

[GBR07] M. Gogolla, F. Büttner, and M. Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27-34, 2007.

[GK07] R. Geiß and M. Kroll. GrGen.Net: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *AGTIVE*, *LNCS 5088*, 568-569. Springer, 2007.

[Kre77] H.-J. Kreowski. *Transformations of Derivation Sequences in Graph Grammars.* In M.Karpinski, editor, Proc. 1st Int. Conf. Fundamentals of Computation Theory, Springer, LNCS 56, 275-286, 1977.

[KGKK02] S. Kuske and M. Gogolla and R. Kollmann and H.-J. Kreowski. *An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation.* In M.J. Butler, L. Petre, K. Sere, editors, Proc. 3rd Int. Conf. Integrated Formal Methods, Springer, LNCS 2335, 11-28, 2002.

[Löw90] M. Löwe. Implementing Algebraic Specifications by Graph Transformation Systems. *Elektronische Informationsverarbeitung und Kybernetik*, 26 (11/12):615-641, 1990.

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In J.L. Pfaltz, M. Nagl, and B. Böhlen, editors, *AGTIVE*, *LNCS 3062*, 479-485. Springer, 2003.

[Roz97] G. Rozenberg, editor. *Handbook on Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

[Wir90] M. Wirsing. Algebraic specification. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science.* North-Holland, 1990.

Appendix: USE Protocol for the Example Calculation

```
----------------------------------------------------------------------
use> open alg2ocl.use
----------------------------------------------------------------------
use> !create n1:ZeroExpr
use> !create n1T:NatType
use> !insert (n1T,n1) into TypeExpr

use> !create n2:SuccExpr
use> !create n2T:NatType
use> !insert (n2T,n2) into TypeExpr

use> !insert (n2,n1T) into SuccNat

use> !create n3:ZeroExpr
use> !create n3T:NatType
use> !insert (n3T,n3) into TypeExpr

use> !create n4:PlusExpr
use> !create n4T:NatType
use> !insert (n4T,n4) into TypeExpr

use> !insert (n4,n2T) into PlusNat1
use> !insert (n4,n3T) into PlusNat2

use> !create n5:SuccExpr
use> !create n5T:NatType
use> !insert (n5T,n5) into TypeExpr

use> !insert (n5,n4T) into SuccNat
----------------------------------------------------------------------
use> !create rc:RuleCollection

use> ?rc.plusZeroN_2_N_redexes()
     Set{}: Set(Sequence(OclAny))
use> ?rc.plusSuccNM_2_succPlusNM_redexes()
     Set{Sequence{@n1,@n1T,@n3,@n3T,@n2,@n2T,@n4,@n4T}}:
       Set(Sequence(OclAny))
----------------------------------------------------------------------
use> read plusSuccNM_2_succPlusNM_find_redex.cmd
     -- the following commands are executed by reading the command file
CMD> !let _redex = rc.plusSuccNM_2_succPlusNM_redexes()->any(true)
CMD> !let _N = _redex->at(1)
CMD> !let _NT = _redex->at(2)
CMD> !let _M = _redex->at(3)
CMD> !let _MT = _redex->at(4)
CMD> !let _succ = _redex->at(5)
CMD> !let _succT = _redex->at(6)
CMD> !let _plus = _redex->at(7)
CMD> !let _plusT = _redex->at(8)
CMD> !openter rc plusSuccNM_2_succPlusNM
       (_N,_NT,_M,_MT,_succ,_succT,_plus,_plusT)
     precondition ‘plusSuccNM_2_succPlusNM_pre’ is true
CMD> !insert(_plusT,_succ) into TypeExpr
CMD> !insert(_succT,_plus) into TypeExpr
CMD> !insert(_succ,_succT) into SuccNat
CMD> !insert(_plus,_NT) into PlusNat1
CMD> !delete(_succT,_succ) from TypeExpr
CMD> !delete(_plusT,_plus) from TypeExpr
CMD> !delete(_succ,_NT) from SuccNat
CMD> !delete(_plus,_succT) from PlusNat1
CMD> !opexit
     postcondition ‘plusSuccNM_2_succPlusNM_post’ is true
------------------------------------------------------------------------
use> ?rc.plusZeroN_2_N_redexes()
     Set{Sequence{@n3,@n3T,@n1,@n1T,@n4,@n2T}}: Set(Sequence(OclAny))
use> ?rc.plusSuccNM_2_succPlusNM_redexes()
     Set{}: Set(Sequence(OclAny))
------------------------------------------------------------------------
use> read plusZeroN_2_N_find_redex.cmd
     -- the following commands are executed by reading the command file
CMD> !let _redex = rc.plusZeroN_2_N_redexes()->any(true)
CMD> !let _N = _redex->at(1)
CMD> !let _NT = _redex->at(2)
CMD> !let _zero = _redex->at(3)
CMD> !let _zeroT = _redex->at(4)
CMD> !let _plus = _redex->at(5)
CMD> !let _plusT = _redex->at(6)
CMD> !openter rc plusZeroN_2_N(_N,_NT,_zero,_zeroT,_plus,_plusT)
     precondition ‘plusZeroN_2_N_pre’ is true
CMD> !insert(_plusT,_N) into TypeExpr
CMD> !delete(_NT,_N) from TypeExpr
CMD> !delete(_zeroT,_zero) from TypeExpr
CMD> !delete(_plusT,_plus) from TypeExpr
CMD> !delete(_plus,_zeroT) from PlusNat1
CMD> !delete(_plus,_NT) from PlusNat2
CMD> !destroy _NT
CMD> !destroy _zero
CMD> !destroy _zeroT
CMD> !destroy _plus
CMD> !opexit
     postcondition ‘plusZeroN_2_N_post’ is true
------------------------------------------------------------------------
```

Prof. Dr. Martin Gogolla
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
gogolla@informatik.uni-bremen.de
http://www.db.informatik.uni-bremen.de

Being a professor for Computer Science at the University of Bremen, Martin Gogolla has been a colleague of Hans-Jörg Kreowski since 1994. They first met in 1982 the 1st Workshop on Abstract Data Types, and have collaborated in both international (e.g., COMPASS) and national projects (e.g., UML-AID).

Dr. Karsten Hölscher
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hoelsch@informatik.uni-bremen.de

Karsten Hölscher studied Computer Science at the University of Bremen. He was introduced to Theoretical Computer Science by Hans-Jörg Kreowski and Frank Drewes, who, during that time, was an assistant professor in Hans-Jörg's team. Karsten graduated in 2003 under Hans-Jörg's supervision and became a doctoral student in his group, working as a research associate from 2003 to 2008. Karsten received his doctoral degree in September 2008 under Hans-Jörg's supervision. After a short intermezzo in software industry, Karsten returned to the University of Bremen, switching sides to a more practical field. He now works as a research associate in the Software Engineering Group headed by Rainer Koschke.

Conditional Adaptive Star Grammars

Berthold Hoffmann

Abstract. The precise specification of software models is a major concern in model-driven design of object-oriented software. In this paper, we investigate how program graphs, a language-independent model of object-oriented programs, can be specified precisely, with a focus on static structure rather than behavior. Graph grammars are a natural candidate for specifying the structure of a class of graphs. However, neither star grammars—which are equivalent to the well-known hyperedge replacement grammars—nor the recently proposed adaptive star grammars allow all relevant properties of program graphs to be specified. So we extend adaptive star rules by positive and negative application conditions, and show that the resulting conditional adaptive star grammars are powerful enough to generate program graphs.

1 Introduction

Model-driven design of object-oriented software aims at describing the static structure, dynamic behavior, and gradual evolution of a system in a comprehensive way. Typically, a software model is a collection of graph-like diagrams that is often specified by a meta-model. For instance, the static structure of a system is often defined by class diagrams of the UML. Since graph grammars are another candidate for specifying graph-like structures, we investigate how they can be used to define software models. As a case study, we consider program graphs, a language-independent model of object-oriented programs that has been devised for specifying refactoring operations on programs [14]. Several kinds of graph grammars have been proposed in the literature. Here we need a formalism that is *powerful* so that all properties of models can be captured, and *simple* in order to be practically useful, in particular for *parsing* models in order to determine their consistency. However, neither star grammars (equivalent to the well-known hyperedge replacement grammars [13, 4]), nor node replacement grammars [12] are powerful enough for our purpose. Even the recently proposed adaptive star grammars [6, 5] fail for certain more delicate properties of program graphs. So we define *conditional adaptive star grammars* in this paper. In these grammars, adaptive star rules are extended by positive and negative application conditions. (Informally, application conditions have already been considered in [9, 7].) Conditional adaptive star grammars capture all relevant properties of program graphs.

The paper is structured as follows. In Section 2, we recall how object-oriented programs can abstractly be represented as *program graphs*. Then we introduce star grammars in Section 3, show that they can define *program trees*, a substructure of program graphs, and discuss why they cannot define program graphs themselves. In Section 4, we therfore recall the *adaptive star grammars* devised in [6, 5]. Close inspection reveals that even this formalism fails to capture properties of program graphs. So we extend adaptive star grammars further, by rules with positive and negative application conditions, in Section 5. These *conditional adaptive star grammars*, finally, allow program graphs to be defined completely. We conclude with some remarks on related and future work in Section 6.

2 Graphs Representing Object-Oriented Software

In model-driven software development, software is represented by diagrams, e.g., of Uml. Formally, such diagrams can be defined as many-sorted graphs.

Definition 2.1 (Graph). Let $\Sigma = \langle \dot{\Sigma}, \bar{\Sigma} \rangle$ be a pair of disjoint finite sets of *sorts*.

A *many-sorted directed graph over* Σ (*graph*, for short) is a tuple $G = \langle \dot{G}, \bar{G}, s, t, \sigma \rangle$ where $\dot{G}$ is a finite set of *nodes*, $\bar{G}$ is a finite set of *edges*, the functions $s, t\colon \bar{G} \to \dot{G}$ define the *source* and *target* nodes of edges, and the pair $\sigma = \langle \dot{\sigma}, \bar{\sigma} \rangle$ of functions $\dot{\sigma}\colon \dot{G} \to \dot{\Sigma}$ and $\bar{\sigma}\colon \bar{G} \to \bar{\Sigma}$ associate nodes and edges with sorts.

Given graphs G and H, a pair $m = \langle \dot{m}, \bar{m} \rangle$ of functions $\dot{m}\colon \dot{G} \to \dot{H}$ and $\bar{m}\colon \bar{G} \to \bar{H}$ is a *morphism* if it preserves sources, targets and sorts. A morphism m is *surjective* or *injective* if both $\dot{m}$ and $\bar{m}$ have the respective property. If the morphism $m\colon G \to H$ is both injective and surjective, it is an *isomorphism*, and G and H are called *isomorphic*, written $G \cong H$.

In figures of graphs, different sorts of edges are represented by arrows drawn with different widths or dashing, whereas nodes are distinguished by their shape, which may be a box or a circle, and by a label inscribed to that shape.

Program graphs have been devised as a language-independent representation of object-oriented code that can be used for studying refactoring operations [14]. They capture concepts that are common to many object-oriented languages, like single inheritance and method overriding, whereas properties particular to a few languages—like multiple inheritance—are left out.

Example 2.1 (A Program Graph). Figure 1 depicts a program graph for a simple object-oriented program from [1], which is shown in Figure 2. The nodes of a program graph, drawn as circles, represent syntactic entities of a program: classes (C), variables (V), method signatures (M) and bodies (B), and expressions (E). Edges establish relations between entities: "↓" is pronounced "*contains*" , and "⇣" is pronounced "*refers to*".

Nodes of sort C are called "class nodes" or just "classes", and so for the other sorts of nodes. The variables contained in a method signature are called its *parameters*, and we say that a class c' is a *super-class* of a class c if either c' equals c, of if some class contained in c' is a super-class of c. In a similar way, we define a *sub-expression* of a body or expression. If a body b refers to a method signature m, we say that b *implements* m. In expressions, only data flow is represented: a reference to a method represents a *call*; a reference to a

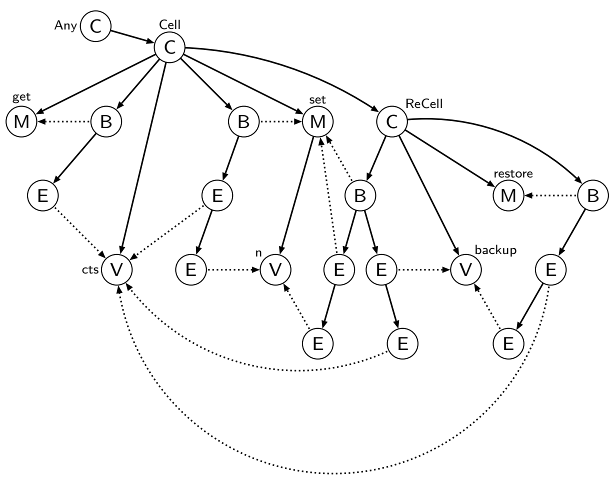


Fig. 1. A program graph

```
class Cell is
   var cts: Any;
   method get() Any is
      return cts;
   method set(var n: Any) is
      cts := n

subclass ReCell of Cell is
   var backup: Any;
   method restore() is
      cts := backup;
   override set(var n: Any) is
      backup := cts;
      super.set(n)
```

Fig. 2. A simple OO program

variable represents an *access* that either *uses* its value, or *assigns* the value of an expression to it.

Definition 2.2 (Program Graph). A graph G is a *program graph* if the following conditions are satisfied:

1. In G, nodes and edges are of the sorts $\{\mathsf{C}, \mathsf{V}, \mathsf{M}, \mathsf{B}, \mathsf{E}\}$ and {↓, ⇣}, respectively.
2. Nodes and edges may be incident as shown in Figure 3. In particular,
   (a) a body contains at least one expression, and implements exactly one method signature, and
   (b) an expression either calls exactly one method, or it accesses exactly one variable; in the latter case, it contain at most one expression.
3. The subgraph $\bar{G}$ of G induced by ↓-edges is a spanning tree of G; the root of $\bar{G}$ is a class.
4. An expression may call any method contained in any class.
5. An expression contained in a class c may access any variable contained in any super-class of c.
6. An expression e may access a parameter of a method m if e is a sub-expression of a body implementing m.

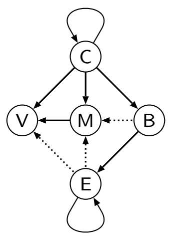


Fig. 3. Incidence of nodes and edges in program graphs

7. A body b contained in some class c may implement any method signature contained in any super-class of c.
8. Every class may contain at most one body implementing a particular method signature m.
9. If an expression e calls a method m, the number of m's parameters must match the number of expressions contained in e.

The class of program graphs is denoted by $\mathcal{P}$.

The graph in in Figure 3 is called a type graph in [10], and a graph schema in Progres [17]; Properties 2.2.1–3 could be described by a class diagram in Uml. Property 2.2.4 defines the visibility of all methods as *public*, and Property 2.2.5 defines the visibility of all variables as *protected*, in the terminology of Java.

The graph-theoretic structure of program graphs is as follows.

Definition 2.3. A rooted, connected, acyclic graph is called a *collapsed tree.*

Fact 2.1. Program graphs are collapsed trees.

Proof Sketch. Acyclicity follows from Property 2.2.2: Cyclic incidences are allowed only for nesting of classes and expressions, but these occur in the underlying the spanning tree so that they do not lead to cycles. Property 2.2.3 implies connectedness; The root class of the spanning tree is the root of the program graph as well, because property 2.2.2 forbids references to classes.

3 Star Grammars

Star grammars are a special case of double pushout (DPO) graph transformation [10], and equivalent to hyperedge replacement grammars [13, 4], a well-understood context-free kind of graph grammars. They are recalled just as a basis for the extensions defined in Sections 4 and 5.

Definition 3.1 (Variable). From now on we assume that the node sorts contain *variable sorts* $\dot{\Sigma}_\mathrm{v} \subseteq \dot{\Sigma}$ that define the *terminal node sorts* as $\dot{\Sigma}_\mathrm{t} = \dot{\Sigma} \setminus \dot{\Sigma}_\mathrm{v}$.

Consider a star-like graph X, with one center node c_X of sort $x \in \dot{\Sigma}_\mathrm{v}$, and with some border nodes (of terminal sorts from $\dot{\Sigma}_\mathrm{t}$) so that the edges connect c_X to every border node; Then X is called a (*syntactic*) *variable named* x. A variable is *straight* if every border node is incident with exactly one edge.

A graph G is a *graph with variables* if all nodes named with variables are not adjacent to each other.[1] Let $\mathcal{X}$ denote the class of (*syntactic*) *variables*, $\mathcal{G}(\mathcal{X})$ the class of graphs with variables, and $\mathcal{G}$ be the class of graphs without variables (with node sorts from $\dot{\Sigma}_\mathrm{t}$).

Definition 3.2 (Star Replacement). A *star rule* is written $L ::= R$, where the *left-hand side* $L \in \mathcal{X}$ is a straight variable and the *replacement* is a graph $R \in \mathcal{G}(\mathcal{X})$ that contains the border nodes of L.

[1] Then all nodes labeled with variable names are centers of stars.

A variable Y in a graph G is a *match* for a star rule $L ::= R$ if there is a surjective morphism $m\colon L \to Y$ where $\bar{m}$ is bijective. Then a *star replacement* yields the graph denoted as $G[Y/_m R]$, which is constructed by adding the nodes $\dot{R} \setminus \dot{L}$ and edges $\bar{R}$ disjointly to G, and by replacing, for every edge in $\bar{R}$, every source or target node $v \in \dot{L}$ by the node $\dot{m}(v)$, and by removing the edges $\bar{Y}$ and center node c_Y.

Let $\mathcal{R}$ be a finite set of star rules. Then we write $G \Rightarrow_{\mathcal{R}} H$ if $H = G[Y/_m R]$ for some $L ::= R \in \mathcal{R}$, some variable Y in G, and some match Y, and denote the reflexive-transitive closure of this relation by $\Rightarrow^*_{\mathcal{R}}$.

Example 3.1 (Star Replacement). Figure 4 shows a star rule $L ::= R$ for an assignment expression. The center nodes of variables are drawn as boxes enclosing the variable name. We shall draw a star rule "blowing up" the center node of L and placing the new nodes and edges of R inside, as can be seen on the right-hand side of Figure 4. A star rule can be represented as it is drawn, as a single *rule graph* wherein a variable is distinguished as the rule's left-hand side. This way, graph operations can be applied to star rules as well. Figure 5 shows a schematic star replacement $G_0 \Rightarrow_{\mathsf{ass}} G_1$ using the rule.

Definition 3.3 (Star Grammar). $\Gamma = \langle \mathcal{G}(\mathcal{X}), \mathcal{X}, \mathcal{R}, Z\rangle$ is a *star grammar* with a *start variable* $Z \in \mathcal{X}$. The *language* of Γ is obtained by exhaustive star replacement with its rules, starting from the start variable:

$$\mathcal{L}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_{\mathcal{R}} G\}$$

Example 3.2 (Star Grammar for Program Trees). Figure 6 shows the star rules generating the program trees that underlie program graphs. The rules define a star grammar PT according to the following convention: The left-hand side of the first rule indicates the start variable, a variable named **Prg** with a class as a border node in this case. The sorts used in the rules define the sorts of the grammar.

Boxes with dashed lines and/or shades indicate *multiple subgraphs*: a shade indicates that the subgraph may have several copies, which are called *clones* as each of them is connected to the remaining graph in the same way; a dashed line indicates that a subgraph may be present, or missing. (In hy, a hierarchy

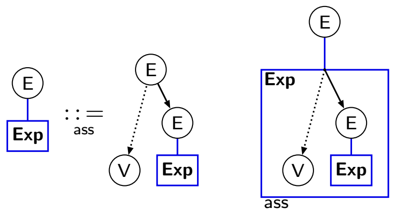


Fig. 4. A star rule and its boxed form

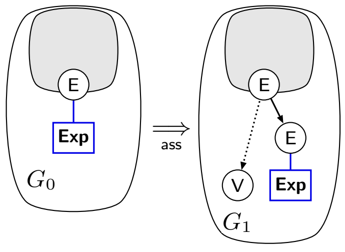


Fig. 5. A star replacement

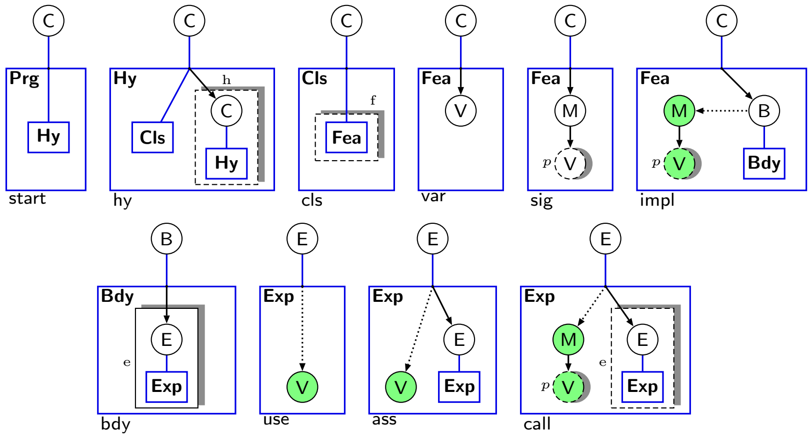


Fig. 6. The rules of the star grammar PT generating program trees

may have $n \geqslant 0$ sub-hierarchies; in bdy, a body contains $n \geqslant 1$ expressions.) Analogously, the dashed and shaded variables in sig, impl, and call indicate multiple nodes that may have $n \geqslant 0$ clones. Note that multiple nodes and subgraphs are just abbreviations, which can be replaced by auxiliary variables that are defined by auxiliary star rules, just as the iteration operators of the extended Backus-Naur Form for context-free word grammars.

Grey nodes designate nodes in the program tree that have to be identified with nodes representing their declarations in order to get a program graph according to Definition 2.2: These are the method and parameters generated in impl and call, and the variables accessed in use and ass.

Inspection of the rules in PT reveals the following.

Fact 3.1. $\mathcal{L}(\mathsf{PT})$ is a language of trees.

The language of PT is closely related to program graphs.

Definition 3.4. The *unraveling* $\hat{G}$ of a collapsed tree G is a tree so that there is a surjective morphism $r\colon \hat{G} \to G$. Let $\hat{\mathcal{P}} = \{\hat{G} \mid G \in \mathcal{P}\}$ denote the unravelings of program graphs.

Fact 3.2. $\hat{\mathcal{P}} \subsetneq \mathcal{L}(\mathsf{PT})$.

Proof Idea. (I): $\hat{\mathcal{P}} \neq \mathcal{L}(\mathsf{PT})$. Inspection of the rules shows that the trees generated with PT satisfy Properties 2.2.1–2.

Also, every graph G in $\mathcal{P}$ satisfies Properties 2.2.1–9. Properties 2.2.1–2 are preserved in the unraveling $\hat{G}$. Properties 2.2.4–8 are irrelevant for $\hat{G}$, as the methods, variables, and parameters referred are copied by the unraveling operation. Property 2.2.9 is unchanged under unraveling

(II): $\hat{\mathcal{P}} \neq \mathcal{L}(\mathsf{PT})$. Rule call may have clones where the formal parameters p have n clones, whereas the actual parameters e have $m \neq n$ clones. However, a

tree generated with such a clone cannot be the unraveling of a program graph, which satisfies Property 2.2.9. (For other rules, one can find similar examples.)

Star grammars are context-free in the sense defined by Courcelle [3]. This suggests that their generative power is limited. Indeed, we have the following

Conjecture 3.1. *There is no star grammar $\varGamma$ with $\mathcal{L}(\varGamma) = \mathcal{P}$.*

Consider Figure 5 to see why we have this conjecture. The rule ass allows to derive trees of assignments. However, for generating a program graph, the rule should insert a reference to a variable that already exists in graph G_0, and is accessible in the expression. Due to the restricted form of star rules, such a node had to be on the border of ass. Since assignments may set every accessible variable, rule ass must have all these variables on its border nodes so that one of them can be selected in the rule. However, the number of accessible variables depends on the size of the program, and is unbounded. Thus a finite set of star rules cannot suffice to define all legal assignments.

4 Adaptive Star Grammars

Proposition 3.1 gives a clue how the limitation of star grammars can be overcome. We make the left-hand sides of star rules *adaptive* wrt. the numbers of border nodes, as proposed in [6, 5]. Formally, this is defined by cloning.

Definition 4.1 (Singular and Multiple Nodes). We assume that the sorts $\varSigma = \langle \dot{\varSigma}, \bar{\varSigma} \rangle$ are given so that the terminal node sorts $\dot{\varSigma}_{\mathrm{t}}$ contain a set $\ddot{\varSigma}_{\mathrm{t}}$ of *multiple* sorts so that every remaining *singular sort* $s \in \dot{\varSigma}_{\mathrm{t}} \setminus \ddot{\varSigma}_{\mathrm{t}}$ has a unique multiple sort $\ddot{s} \in \ddot{\varSigma}_{\mathrm{t}}$, and vice versa.

From now on, $\mathcal{X}$, $\mathcal{G}$ and $\mathcal{G}(\mathcal{X})$ denote classes of graphs with singular sorts only, whereas $\ddot{\mathcal{X}}$, $\ddot{\mathcal{G}}$ and $\ddot{\mathcal{G}}(\ddot{\mathcal{X}})$ denote classes of *adaptive graphs* that may contain multiple sorts as well.

A star rule $L ::= R$ is called *adaptive* if $L \in \ddot{\mathcal{X}}$ and $R \in \ddot{\mathcal{G}}(\ddot{\mathcal{X}})$.

Definition 4.2 (Cloning). Let G be a graph in $\ddot{\mathcal{G}}(\ddot{\mathcal{X}})$ with a multiple node v that is labeled with $\ddot{\ell} \in \ddot{\varSigma}$, and incident with edges $e_1, \dots, e_n$ ($n \geqslant 0$). Then $G\frac{v}{k}$ denotes the *clone* of G in which v is replaced by $k \geqslant 0$ singular nodes $v_1, \dots, v_k$ that are labeled with ℓ, where every clone v_i is incident with copies $e_{i,1}, \dots, e_{i,n}$ of the edges $e_1, \dots, e_n$ incident with v.

If $r = L ::= R$ is an adaptive star rule with a multiple node v, its clone $r\frac{v}{k}$ is obtained by cloning its rule graph.

Example 4.1 (Adaptive Star Cloning, and Label Specialization). The star rule ass on the left of Figure 7 is adaptive: its variable a is a multiple node, and shall match a set of $n \geqslant 0$ variables in the host graph that are accessible in the expression. On the right-hand side, a schematic view of the clone $\mathsf{ass}\frac{a}{n}$ is given, for $n \geqslant 0$.

The *abstract sort* F of nodes a and a_i is a placeholder for the concrete subsorts V and M. (F stands for for *feature.*) Before applying the clone $\mathsf{ass}\frac{a}{n}$, each of the labels F is specialized either to V or M. As with multiple nodes and subgraphs, a star rule with abstract sorts is just an abbreviation for a set of star rules wherein these abstract sorts are replaced with any combination of concrete sub-sorts.

Definition 4.3 (Adaptive Star Grammar). Let $\Gamma = \langle \ddot{\mathcal{G}}(\ddot{\mathcal{X}}), \ddot{\mathcal{X}}, \mathcal{R}, Z \rangle$ be a star grammar over adaptive variables and graphs. Then Γ is called *adaptive* if $Z \in \mathcal{X}$ (i.e., has no multiple nodes).

Let $\ddot{\mathcal{R}}$ denote the set of all possible clones of a set $\mathcal{R}$ of adaptive star rules. Then Γ generates the language

$$\ddot{\mathcal{L}}(\Gamma) = \{G \in \mathcal{G} \mid Z \Rightarrow^*_{\ddot{\mathcal{R}}} G\}$$

The set of star rules $\ddot{\mathcal{R}}$ generated from a set of adaptive star rules is infinite if at least one of the adaptive star rules contains a multiple node or subgraph. It has been shown in [5] that this gives adaptive star grammars greater generative power than grammars based on hyperedge [13] or node replacement [12], but they are still parseable [6].

Example 4.2 (Adaptive Star Grammar for Program Graphs). The adaptive star rules in Figure 9 define an adaptive star grammar PG that systematically extends the program tree grammar PT of Figure 6.

With two exceptions, the rules of PG just extend those of PT. In PG, rule meth defines a method declaration, which combines a signature sig with an (optional) implementation impl, whereas ovrd defines the overriding of a method in the subclass of the original method definition.

In Figure 8 we show the general form of variables in PG and of the program subgraphs they generate. (In derivations, the multiple nodes d, v, and o of X are cloned.) The sorts of edges indicate the following roles of the border nodes. Node r is the *root* of the program subgraph G_X derived from X. Clones of d are the features *declared* in G_X. Clones of v are the features that are *visible* in G_X. Clones of o are the methods that are *overridable* in G_X. Features may have multiple roles in X and G_X: every feature declared by X is also visible in X, and

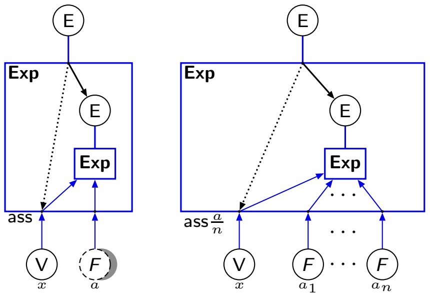


Fig. 7. An adaptive rule and its clones

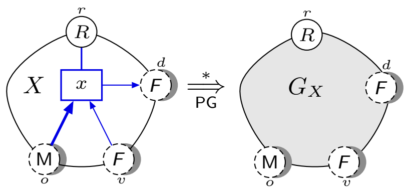


Fig. 8. Variables and derivations in PG

overridable methods are visible as well so that some clones d and v, and some clones of v and o in X may identified. On the left-hand side of rules, the clones of d, v and o in a variable X are be distinct (they are required to be straight) so that they must be identified by matching. The graph G_X is directed and acyclic. Some of its visible border nodes may be isolated. The rest is a collapsed tree with root r.

The rules in Figure 9 extend the rules of Figure 6 by adding border nodes to variables according to the roles explained above. The rules for **Fea** declare a variable or a method (or just override an existing method). The rule cls declares its member variables and methods. A hierarchy declares all methods of its top class and of its sub-hierarchies, makes the variables of the top class visible in the class itself and in the sub-hierarchies, and makes the methods of the top class overridable in the classes of its sub-hierarchies. The rule start makes all methods declared by the program hierarchy visible in it. All rules pass visible features down to the leaves of the program graph. The rules for **Exp** then select visible variables for being used or assigned to, and methods for being called; rule ovrd selects an overridable method signature for overriding it with a body.

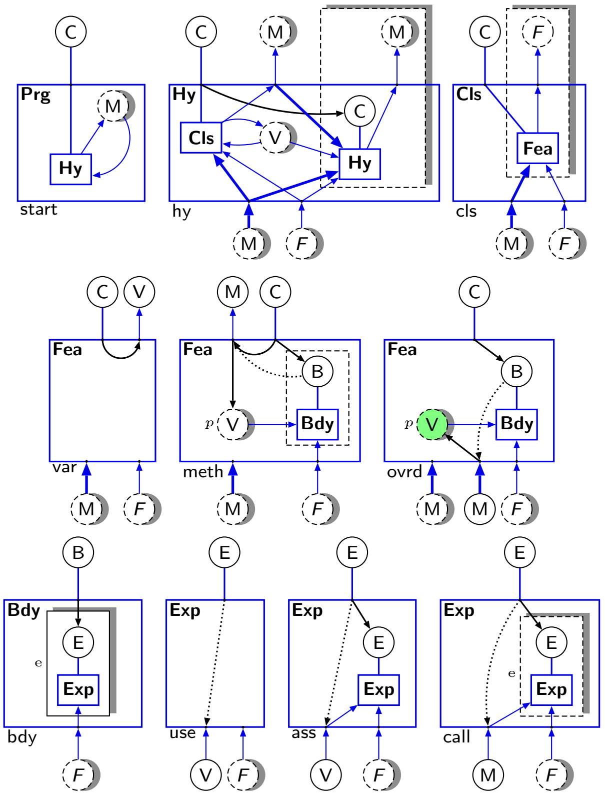


Fig. 9. Rules of the adaptive star grammar PG defining program graphs

Figure 10 shows parts of a derivation of the program graph shown in Figure 1 with PG. We simplify the drawing of edges as follows: A pair of counter-parallel edges “⇄” is drawn as a single line “——”, and a pair of parallel edges of the form “⇉” is drawn as a single arrow “⟶”.

The class hierarchy is derived in the first row. Exponents of the rules (if present) indicate how many clones are made of the multiple subgraphs and nodes on the right-hand side; for border nodes, it is easy to see how many clones have to be made, and how their labels have to be specialized. Classes Cell and Recell will introduce three and two features, resp.; the methods are visible in both classes, but the variables introduced are only visible in the defining class and

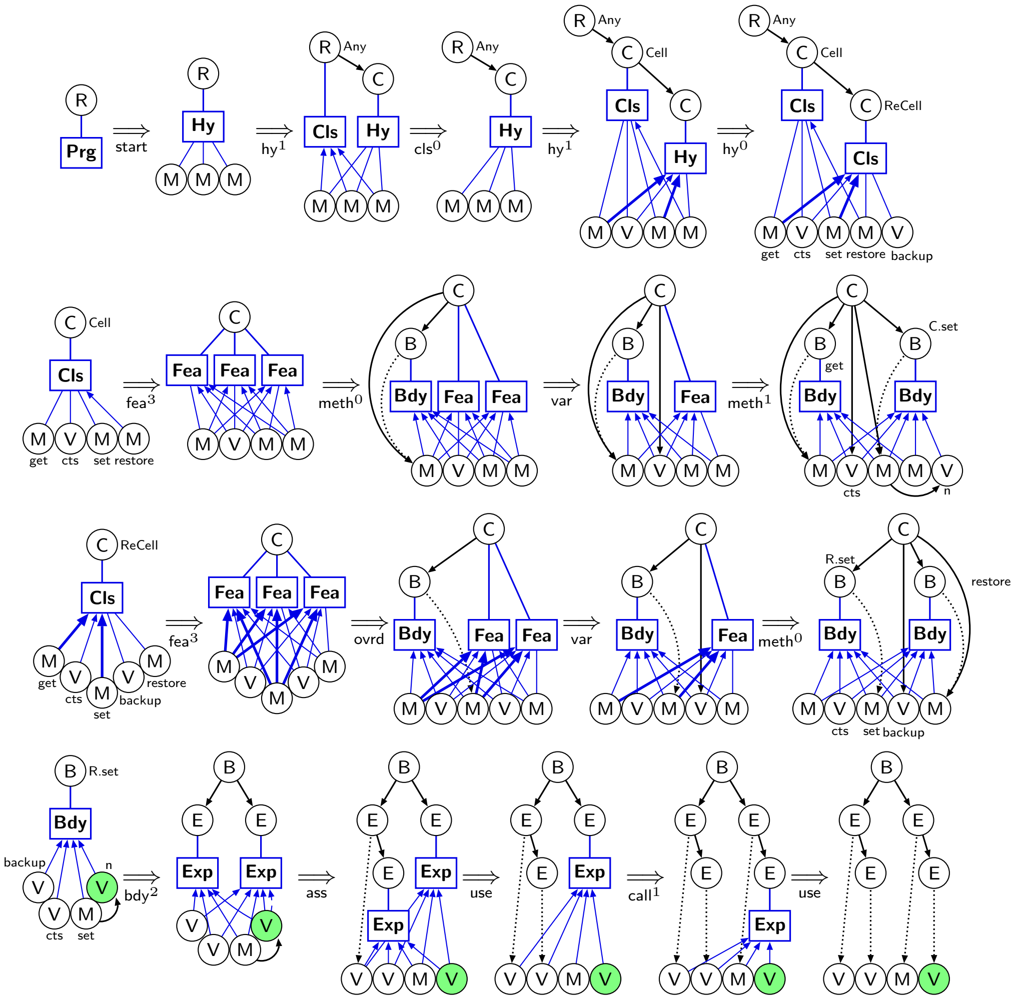


Fig. 10. Deriving the program graph of Figure 1 with PG

in its subclasses so that the variable backup in ReCell will not be visible in Cell. The methods defined in Cell are overridable in ReCell.

The features get, backup, and restore of the class Cell are introduced in the second row, and the features of the class ReCell are derived in the third row: the variable backup and the method restore are introduced, and the method set of Cell is overridden. The last row shows a derivation of the body overriding the method set of class Cell in ReCell.

The derivations in rows one to three can be combined to one big derivation by embedding. However, the start graph of the last row *cannot* be embedded into the final graph of the derivation in the third row. This is because the rule ovrd does not make the parameter n (drawn in grey) of the signature of set visible in the overriding body. The parameter is needed to derive the body, and it should be visible in it. This reveals one of two problems in the grammar, which cannot be overcome with adaptive star grammars.

Theorem 4.1. *A graph G is in $\mathcal{L}(\mathsf{PG})$ iff it satisfies Properties 2.2.(1–5,7).*

Proof Sketch. "⇒": Inspection of the rules (as done in Example 4.2 and Figure 8 above) shows that the border nodes of variables do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathcal{L}(\mathsf{PG})$ satisfies Properties 2.2.(1–5,7).

"⇐": Let G satisfy Properties 2.2.(1–5,7). We construct a derivation

$$Z = H_0 \underset{\mathsf{PG}}{\Longrightarrow} H_1 \underset{\mathsf{PG}}{\Longrightarrow} \cdots \underset{\mathsf{PG}}{\Longrightarrow} H_n$$

so that there are injective morphism $h_i \colon \bar{H}_i \to G$, where $\bar{H}_i$ is the terminal subgraph of H_i with all nodes that are reachable from the root class via edges of types $\{\downarrow, \downarrow\}$, for $0 \leqslant i \leqslant n$.

By Fact 2.1, G is a collapsed tree that has a class, say c, as its root, which is also the root of a spanning tree induced by ↓-edges (Property 2.2.3). The unique class in Z is then mapped onto c. The number, say s, of direct subclasses of c determines the instance $\mathsf{hy}\frac{h}{s}$ that must be applied to $\hat{c}$ so that the clones of the C-nodes in H_1 can be mapped to the subclasses of c in G so that they can be extended to an injective morphism $h_1 \colon \bar{H}_1 \to G$. (The number of clones for the border nodes in Z and $\mathsf{hy}\frac{h}{s}$ are left open for the moment.) The construction of further graphs $\bar{H}_i$ can be continued in the same way. (See the program graph in Figure 1 as an example.) When a graph $\bar{H}_n$ has been constructed so that h_i is injective and surjective, no variables are left in H. Then all the open multiple border nodes in graphs $H_0, \ldots, H_{n-1}$ can be determined by considering the number of their clones as multiplicity variables, the values of which are determined by the rules used in the derivation. In the rule used in the first step, for instance, the multiplicity of the multiple M-node in Z is given as the sum of the multiplicities of all declared multiple M-nodes in $\mathsf{hy}\frac{h}{s}$, which in turn equals the multiplicity of the multiple F-node in $\mathsf{hy}\frac{h}{s}$. The multiplicity for the overridable M-node in $\mathsf{hy}\frac{h}{s}$ equals 0, and the multiplicity of the multiple C-node in $\mathsf{hy}\frac{h}{s}$ is determined by the rule that has been applied to **Cls**. The process of finding equations for the multiplicities of nodes yields a unique solution, because the equations satisfy the invariants on border nodes. The solutions for the multiple nodes define complete instances for all multiple nodes in the graphs H_i and in the rules of the derivation. Thus H_n is in $\mathcal{L}(\mathsf{PG})$, so that h_i is an isomorphism for $\hat{H}_n = H_n$, and $H_n \cong G$.

The proof constructs a derivation for a given graph. The construction is unique up to isomorphism so that we get the following:

Corollary 4.1. *PG is unambiguous.*

The grammar PG falsely derives some graphs that are not program graphs. In a graph $G \in \mathcal{L}(\mathsf{PG}) \setminus \mathcal{P}$, a class may contain several bodies that override the same method, method bodies that access a wrong number of parameters, and method calls with a wrong number of actual parameters. Let us discuss why adaptive star grammars fail to describe two properties of program graphs.

Property 2.2.6. In rule ovrd, the parameters of the method m being overridden cannot be made visible in its body. Parameters are only visible in the body of their first definition, so they are not among the clones of the F-node in that rule. (See the overriding of the method set in class ReCell discussed in Example 4.2.)

We could pass around all parameters of all methods (not in the role "visible", but in their role as "parameters"). Then, we had to select the parameters of m to be passed on to its body. We thus have to distinguish the parameters of m from those of other visible methods. However, the number of visible methods is unbounded, whereas our supply of edge sorts is finite. So this is not possible. Alternatively, we could generate copies of the formal parameters for every overridden body. But then we must know how many formal parameters m has. Again, this information cannot be made available.

Property 2.2.9. In rule call, the number of actual parameters needs not match the formal parameters of the method being called. As in the previous case, we would need access to the parameters of the method being called, or need to know their number in order to assure parameter correspondence.

These considerations lead to the following

Conjecture 4.1. *There is no adaptive star grammar Γ with $\mathcal{L}(\Gamma) = \mathcal{P}$.*

5 Conditional Adaptive Star Grammars

To overcome the deficiencies of adaptive star grammars, we extend adaptive star rules with *application conditions.* This has been proposed for general (DPO) graph transformation rules in [11], and has been discussed informally for adaptive star grammars in [8].

Definition 5.1 (Conditional Adaptive Star Replacement). An *application condition A* for an adaptive star rule $L ::= R$ is one of the following: *(i)* a *positive condition* C or *(ii)* a *negative condition* $\neg C$ where $C \in \ddot{\mathcal{G}}$, or *(iii)* a *clone condition* $\forall x : A'$ where x is a multiple node in L, and A' is an application condition for L wherein the node x occurs as a singular node that carries the same label as in L. The graphs in an application condition may contain further nodes from L, which carry the same label and are singular or multiple in both A and L. If $A_1, \ldots, A_n$ are application conditions for L, $r = A_1 \wedge \cdots \wedge A_n \parallel L ::= R$ is a *conditional adaptive star rule.* If $n = 0$, the rule r is written without the symbol "$\parallel$", like an unconditional rule.

In a *clone* $\ddot{r} = \ddot{A}_1 \wedge \cdots \wedge \ddot{A}_n \parallel \ddot{L} ::= \ddot{R}$ of a conditional rule r, all multiple nodes have as many clones in $\ddot{A}$ as in $\ddot{L}$. Let $m : \ddot{L} \to Y$ be a match of $\ddot{r}$ with a variable Y in some host graph G. We define recursively over the structure of application conditions in which cases m *satisfies* an application condition A, written $m \vDash A$:

– $m \vDash C$ if m can be extended to C;
– $m \vDash \neg C$ if m cannot be extended to C;
– $m \vDash \forall x : A$ if $m \vDash A(x/x')$ for every clone x' of x, where $A(x/x')$ is obtained from A by renaming x to x'.

If $m \vDash \ddot{A}_i$ for $1 \leqslant i \leqslant n$, the star replacement $G[Y/_m\ddot{R}]$ is a *conditional star replacement*, and we write $G \overset{c}{\Longrightarrow}_{\ddot{r}} H$.

When drawing conditional rules, as in Figure 11, we indicate shared nodes of application conditions and left-hand sides of conditional rules by attaching the same letters to them.

Definition 5.2 (Conditional Adaptive Star Grammar). Let $\mathcal{C}$ be a finite set of conditional adaptive star rules. Then $\Gamma = \langle \ddot{\mathcal{G}}(\ddot{\mathcal{X}}), \ddot{\mathcal{X}}, \mathcal{C}, Z \rangle$ is a *conditional adaptive star grammar* over if $Z \in \mathcal{X}$.

Let $\ddot{\mathcal{C}}$ denote the set of all possible clones of a set $\mathcal{C}$ of conditional adaptive star rules. Then Γ generates the language

$$\ddot{\mathcal{L}}(\Gamma) = \{G \in \mathcal{G} \mid Z \overset{c}{\Longrightarrow}{}^{*}_{\ddot{\mathcal{C}}} G\}$$

Example 5.1. [Conditional Adaptive Star Grammar for Program Graphs] Figure 11 shows the rules of the conditional adaptive star grammar PG_c, which refines the adaptive star grammar PG of Example 4.2 as follows. The variables in PG_c are attached to the border nodes used in PG, and may be attached to two additional sets of nodes, see Figure 12: Outgoing dashed edges $\dashrightarrow$ represent the formal parameters *contained* in variables named **Hy**, **Cls**, and **Fea**, and ingoing dashed edges represent the formal parameters *known* in a variable. The rules make that all formal parameters contained in the features, classes and hierarchies of the program are known to every variable.

In rule call, the positive condition on nodes m and p (where p is multiple as it occurs inside a multiple subgraph) requires that the clones of p are formal parameters of m, and the negative clone condition on nodes m and o forbids, for every other clone o' of o, i.e., for every other parameter known in the program,

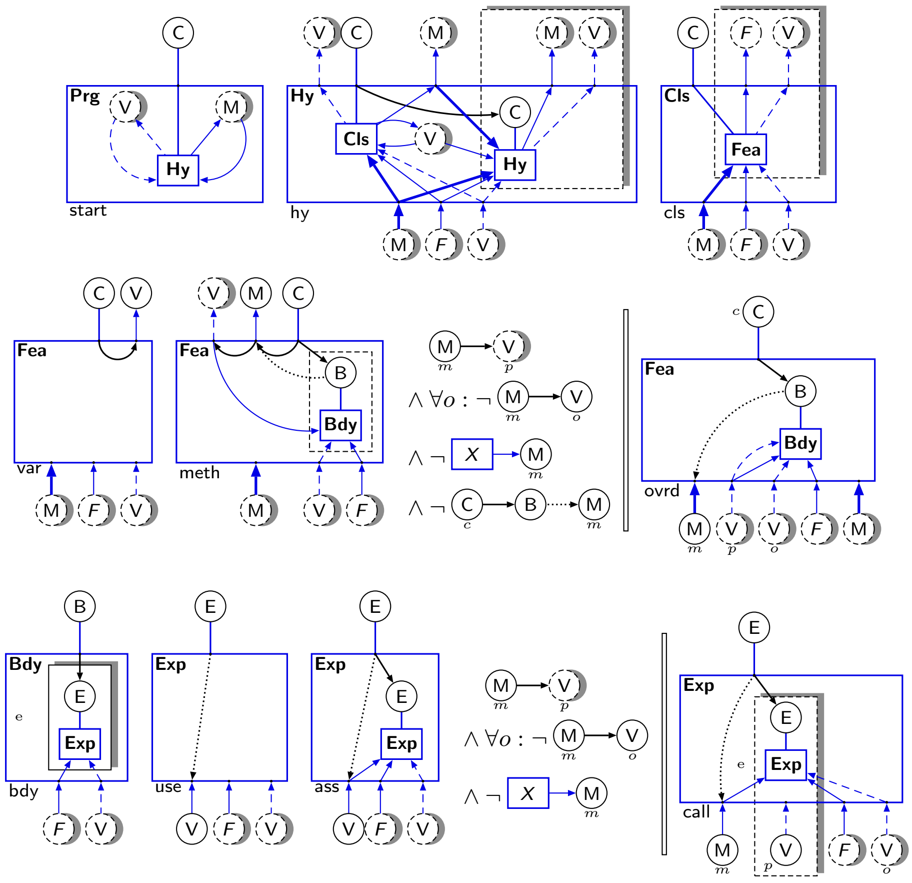


Fig. 11. Rules of the conditional adaptive star grammar $\mathsf{PG_c}$ defining program graphs

that o' is a parameter of m. Thus the nhclones of p are all parameters of m. The remaining condition forbids m to be a declared node of any variable named $X \in \Sigma_v$ to m. This makes sure that the parameters of m have been generated before rule call is applied. Since the multiple subgraph contains the formal parameter p as well as the variable named **Exp** generating the actual parameters, this makes

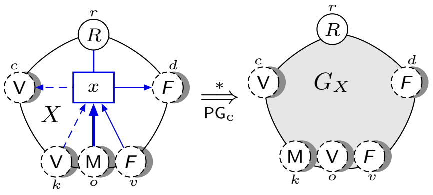


Fig. 12. Variables and derivations in $\mathsf{PG_c}$

sure that call will generate an actual parameter for every formal parameter of m. Thus Property 2.2.9 is respected.

In rule ovrd, the first three application conditions (which equal that of call) make sure that the clones of p are all formal parameters of m. These parameters are not only made known to the overriding body of m, but also made visible to it so that they may be accessed as variables in use and ass. Thus Property 2.2.6 is respected. The fourth application condition makes sure that no other method body contained in the current class c does override the same method m; this guarantees Property 2.2.8.

In Figure 13, we show some steps of a derivation with $\mathsf{PG_c}$ that could eventually derive the program graph in Figure 1. The grey region contains nodes representing the declarations of get, n, backup, and restore. A pair of counter-parallel edges “⇠⇢” is drawn as a single line “----”.

Note that rule meth, which generates the definition of set in class Cell makes the parameter n visible, as a parameter, to the entire program.

When the rule ovrd is applied to the method set, n is made visible as a variable inside its body. The other part of the applicability condition holds as well: Class ReCell does not contain another body overriding set, and no variable has m as a declared border node (but just as a visible border node of **Bdy** and an overridable border node of **Fea**). Note that in class ReCell, the method set cannot be overridden by another body since this would violate the application condition of ovrd. Now the derivation in the last row of Figure 10 can be inserted

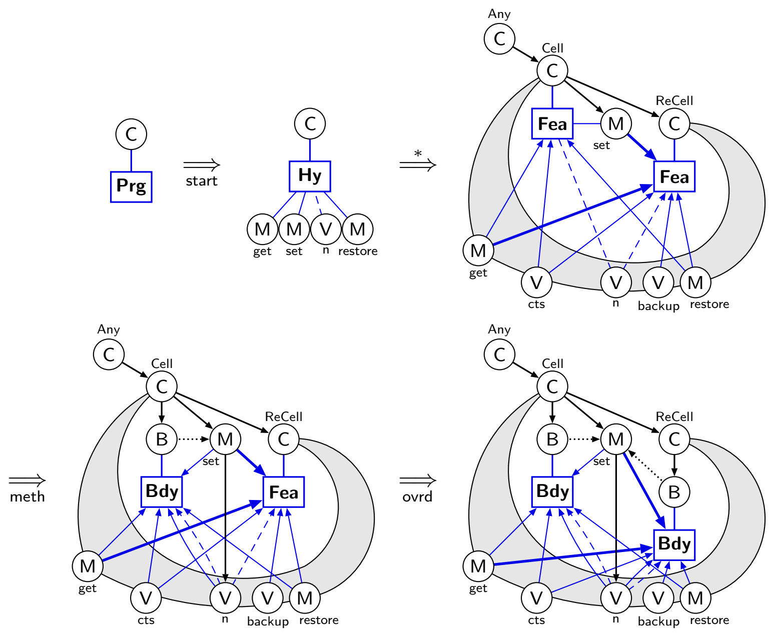


Fig. 13. Deriving the program graph of Figure 1 with $\mathsf{PG_c}$

for the body of set in ReCell because n is present. In that derivation, in the step using rule call, the application condition of $\mathsf{PG_c}$ guarantees that exactly one expression will be generated as an actual parameter since method set has one formal parameter.

Definition 5.3 (Complete Node). Consider a graph $G \in \mathcal{G}(\mathcal{X})$ and a conditional adaptive star grammar Γ.

A node $v \in \dot{G}$ is called *complete* wrt. structural edges if for every derivation $G \overset{c}{\Longrightarrow}{}^{*}_{\Gamma} H$, v is incident to the same terminal edges in H as it was in G.

Fact 5.1. In graphs derived with $\mathsf{PG_c}$, M-nodes are complete wrt. structural edges if they are not declared nodes of any variables.

Proof Sketch. By inspection of the right-hand sides of the rules for these variables in $\mathsf{PG_c}$, it is clear that structural edges are added only to declared nodes of these rules' left-hand sides.

According to this fact, application conditions over structural edges can safely be checked as soon as the relevant nodes are only visible or overridable border nodes of variables. This is the case for the conditions concerning the parameters of methods.

Thus $\mathsf{PG_c}$ generates the program graph in Figure 1, and will not generate calls with mismatching parameters, nor with methods that are overridden twice in a class.

Theorem 5.1. $\mathcal{L}(PG_c) = \mathcal{P}$.

Proof Sketch. The proof is similar to that of Theorem 4.1.

"$\Rightarrow$": Inspection of the rules (as done in Example 5.1 and Figure 12 above) shows that the border nodes of variables do indeed play the roles given to them. Using these invariants, it can be shown by induction over the structure of rules that every $G \in \mathcal{L}(\mathsf{PG})$ satisfies all Properties 2.2.(1–9) of a program graph.

"$\Leftarrow$": Given a program graph $G \in \mathcal{P}$, we can construct a derivation according to the underlying structure (with edges of type $\downarrow$) first, before we determine the clones for border nodes according to the equations on the multiplicity variables. At last, it can be verified that the conditional rules ovrd and call satisfy their application conditions.

Application conditions do not sacrifice parseability of adaptive star grammars. Because, checking a condition, which consists of a positive and negative terminal graph, is always decidable. In contrast to simple adaptive star rules, the matches of conditional adaptive star rules in a graph may have critical overlaps. The application condition of one rule may contradict the application application condition of another rule. Consider, e.g., the node ReCell in the rightmost graph in the top row of Figure 13. The rule ovrd matches every **Fea** node in reCell. However, if the match includes the same method (get or set), then the application of the rule to one feature would disable the other application, due to the negative application concerning unique implementation. The critical pair analysis for graph transformation rules (as implemented in the AGG-system) applies to conditional graph transformation rules; it might be used to analyze conflicts in conditional adaptive star rules if we can extend it to multiple nodes.

6 Conclusions

In this paper, we have attempted to define the well-known class of program graphs [14] by graph grammars. This seems to be impossible with star grammars and even adaptive star grammars [6, 5], whereas it can be done with the conditional adaptive star grammars that have been introduced informally in [9, 8]. A richer class of program graphs, featuring more general visibility rules, contextual rules for abstract methods and classes, control flow in method bodies, and static typing of variables and methods has been specified in [9] by conditional adaptive star grammars as well.

There are too many kinds of graph grammars to relate conditional adaptive star grammars to all of them. So we restrict our discussion to approaches that aim at a similar application. Context-embedding rules [15] extend hyperedge-replacement grammars by rules that add a single edge to an arbitrary graph pattern. They are used to define and parse diagram languages and are not powerful enough to define models like program graphs. Graph reduction grammars [2] have been proposed to define and check the shape of data structures with pointers. The form of their rules is not restricted, but reductions with the inverse rules are required to be terminating and confluent, providing a backtracking-free parsing algorithm. It is an open question whether graph reduction grammars suffice to define program graphs.

A lot of work has to be done until we get a graph grammar mechanism that is useful for defining software models. Yet another problem is to convince software engineers that it is a practical benefit for their daily work!

First of all, graph grammars should be compared with the conventional software models, like UML diagrams. For instance, can such a model be derived from a grammar? Can at least parts of a model be obtained "automatically"? There is some indication that a class diagram specifying Properties 2.2.(1–3) of program graphs can be inferred from the rules of a (conditional) adaptive star grammar.

Even if conditional adaptive star grammars are powerful enough, their rules tend to be rather complicated, both to write and to read. So a more general challenge would be to come up with yet another graph grammar formalism that is easier to use, but enjoys many of the formal properties of (adaptive) star rules.

The proof of conjectures 3.1 and 4.1 poses the theoretical challenge to disprove membership in a class of graph languages. While there are at some concepts for star languages (e.g., the pumping lemma for the equivalent hyperedge replacement languages [13, 4]), nothing is known for (conditional) adaptive star languages.

Acknowledgments. I thank my favorite co-authors for their constructive reviews of this paper.

Special Thanks. *Danke, Hans-Jörg!* For your long-lasting support in form of scientific (and other) discussions, opportunities to visit conferences, never-ending supply of co-authors from your PhD students, and—last but not least—for fruit and cake in meetings!

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects.* Monographs in Computer Science. Springer, New York, 1996.
2. Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. *Mathematical Structures in Computer Science*, 2009. Accepted for publication.
3. Bruno Courcelle. An axiomatic definition of context-free rewriting and its application to NLC rewriting. *Theoretical Computer Science*, 55:141–181, 1987.
4. Frank Drewes, Annegret Habel, and Hans-Jörg Kreowski. Hyperedge replacement graph grammars. In Rozenberg [16], chapter 2, pages 95–162.
5. Frank Drewes, Berthold Hoffmann, Dirk Janssens, and Mark Minas. Adaptive star grammars and their languages. Technical Report 2008-01, Departement Wiskunde-Informatica, Universiteit Antwerpen, 2008.
6. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Adaptive star grammars. In Andrea Corradini, Hartmut Ehrig, Ugo Montanari, Leila Ribeiro, and Grzegorz Rozenberg, editors, *3rd Int'l Conference on Graph Transformation (ICGT'06)*, number 4178 in Lecture Notes in Computer Science, pages 77–91. Springer, 2006.
7. Frank Drewes, Berthold Hoffmann, Dirk Janssens, Mark Minas, and Niels Van Eetvelde. Shaped generic graph transformation. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07)*, number 5088 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
8. Frank Drewes, Berthold Hoffmann, and Mark Minas. Adaptive star grammars for graph models. In Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors, *4th International Conference on Graph Transformation (ICGT'08)*, number 5214 in Lecture Notes in Computer Science, pages 201–216. Springer, 2008.
9. Niels Van Eetvelde. *A Graph Transformation Approach to Refactoring.* Doctoral thesis, Universiteit Antwerpen, May 2007.
10. Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, and Gabriele Taentzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs on Theoretical Computer Science. Springer, 2006.
11. Hartmut Ehrig and Annegret Habel. Graph grammars with application conditions. In Grzegorz Rozenberg and Arto Salomaa, editors, *The Book of L*, pages 87–100. Springer, Berlin, 1986.
12. Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In Rozenberg [16], chapter 1, pages 1–94.

13. Annegret Habel. *Hyperedge Replacement: Grammars and Languages.* Number 643 in Lecture Notes in Computer Science. Springer, 1992.
14. Tom Mens, Niels Van Eetvelde, Serge Demeyer, and Dirk Janssens. Formalizing refactorings with graph transformations. *Journal on Software Maintenance and Evolution: Research and Practice*, 17(4):247–276, 2005.
15. Mark Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
16. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
17. Andy Schürr, Andreas Winter, and Albert Zündorf. The PROGRES approach: Language and environment. In Gregor Engels, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. II: Applications, Languages, and Tools*, chapter 13, pages 487–550. World Scientific, Singapore, 1999.

Dr. Berthold Hoffmann

Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hof@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~hof

Berthold Hoffmann became a colleague of Hans-Jörg Kreowski in 1978 at TU Berlin. After both of them moved to the University of Bremen in 1982, he took part in research activities of Hans-Jörg's group, particularly in the EC Working Groups COMPUGRAPH, APPLIGRAPH, and SEGRAVIS.

Assemblies as Graph Processes

Dirk Janssens

Abstract. This short paper explores the potential of embedding-based graph rewriting as a tool for understanding natural computing, and in particular self-assembly. The basic point of view is that aggregation steps in self-assembly can be adequately described by graph rewriting steps in an embedding-based graph transformation system: the building blocks of an assembly correspond to occurrences of rewriting rules, and hence assemblies correspond to graph processes. However, meaningful algorithms do not consist only of aggregation steps, but also of global steps in which assemblies are modified. A theoretical algorithm is presented in which the two kinds of steps are combined: on the one hand aggregation steps that build assemblies, and on the other hand global steps which act on the assemblies.

1 Introduction

The study of self-assembly has been an interesting and promising part of the fascinating area of natural computing for several years [WLWS98,KR08,Cas06]. The phenomenon is an important aspect of biological systems [ETP$^+$04] and has potential applications in nanotechnology, chemistry and material sciences [GJC91]. The basic idea is that components such as molecules or proteins aggregate to form assemblies that have interesting emerging properties which are not present in the original components. It is obviously important to control this aggregation process, i.e. we want to be able to design the building blocks in such a way that certain a priori known structures emerge as a result of spontaneous aggregation. These structures may in their turn interact in a meaningful way with other components.

The basic step in an assembly process is sketched in Figure 1: two components (left) aggregate to form an assembly (right). It is assumed that this happens because there is a particular relationship between their surface structures: these contain active parts (bold segments) that spontaneously stick together; one may think of atoms or molecules that form bonds between them, like in the case of Watson-Crick complementarity. Components will be called assemblies whenever we want to stress that they are built by self-assembly.

The use of graph rewriting [EKMR97,EEPT] as a tool for studying natural computing and self-assembly has been explored before [HLP08,KGL04]. However there are a lot of possible directions to follow because of the variety of processes that need to be described as well as the variety of graph rewriting mechanisms. The aim of this paper is to explore, in a very preliminary and perhaps naive way, how the work on graph rewriting with embedding and the corresponding theory of graph processes from, e.g., [VJ02] can be used in this context. It turns out that components, surface structure and assemblies correspond to rules, graphs and graph processes, respectively.

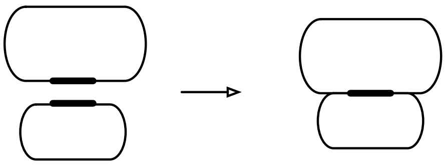


Fig. 1. Aggregation

The fact that both the surface structure and the assemblies are described within the same formal framework enables one to switch between the graph/surface structure and the process/assembly view: the former allows one to describe the aggregation steps (combination of various assemblies based on their surface structure) whereas the latter allows one to describe global steps, caused by external manipulation (heating, extraction, ...). The first kind of step occurs when a large amount of simple building blocks (molecules) is put into a solution and allowed to form assemblies; the latter kind happens in response to other manipulations acting in a uniform way on the assemblies as a whole (e.g. partially breaking them down). We present an algorithm in which the two kinds of steps are combined; it (theoretically, at least) allows one to recognize the difference between two solutions, a "pure" one and one that is contaminated with one or more extra (but unknown) components, by building assemblies that encode the contaminating components (but only those). These assemblies can then be extracted and used to mark the contaminating components, so that they can be removed from the contaminated solution.

In Section 2 the basic assumptions underlying this work are given, and the relationship is discussed between components, surface structure and assemblies on the one hand and rules, graphs and graph processes on the other hand. The example algorithm is presented in Section 3 and the paper ends by a brief discussion section.

2 Components, rules and processes

In this section the basic assumptions of the approach are given, and the necessary elements of graph rewriting, embedding mechanism and processes are briefly sketched. A formal treatment can be found in, e.g., [VJ02].

2.1 Basic assumptions

A graph transformation system consist essentially of a set of *rules* that describe local changes applied to graphs. Traditionally, a rule has a *left-hand side* and a *right-hand side*, which are both also graphs. A rule is applied to a graph g by matching its left-hand side with a subgraph of g. That subgraph is then removed and replaced by the right-hand side. In the approach used in this paper the rules are equipped with additional information, called "embedding mechanism", which is used to determine the edges of the resulting graph.

The way graph rewriting is related to aggregation is the following. Consider an assembly step, like the one depicted in the upper part of Figure 2: an existing assembly (top) gets larger by aggregating with a building block (bottom). The surface structure of the former is represented by a graph g_1 with nodes a, b, c, d, e. The aggregation leads to a larger assembly with a modified surface structure, represented by a graph g_2 with nodes c, d, e, f, g, h. Thus the effect of adding the new building block on the surface structures is that g_1 is changed into g_2, in a way that can be captured by graph rewriting: the building block is viewed as a graph rewriting rule and the aggregation step corresponds to its application, removing the nodes a, b and replacing them by f, g, h. Evidently one also has to deal with the edges, which represent relationships between surface elements. We come back to this in subsection 2.2. The approach entails the following three assumptions concerning aggregation.

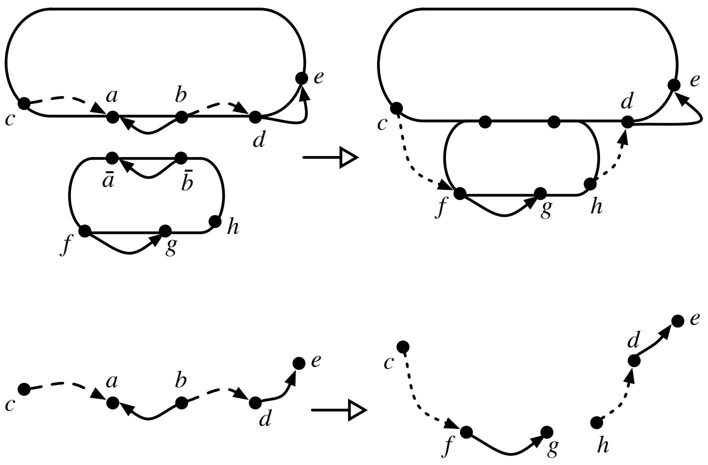


Fig. 2. Aggregation and graph transformation

The first working hypothesis is that the surface structure of components, which governs the way in which they aggregate, can be adequately described as a labeled graph. The nodes represent atoms, molecules, ... that are present at specific locations on the surface, the node labels distinguish between various kinds of such surface elements, and the edges describe relationships between these elements that are important for determining whether a group of nodes is active in the sense that it causes aggregation. One may think of spatial relationships or vicinity, but there may be others. In Figure 2 the symbols $\overline{a}$ and $\overline{b}$ are used to indicate the fact that binding or aggregation between physical components is caused by elements that are "complementary" in some sense (e.g. having opposite polarities, Watson-Crick complements, ...). In our approach this information is implicitly represented by the fact that building blocks are described by graph rewriting rules which have a designated left-hand side. This is why we will not use complementary labels such as a and $\overline{a}$ in the next section; the usual notion of matching suffices.

The second hypothesis is that the relevant relationships between the elements of the new surface (i.e., the new edges) can be determined from (1) the surface of the existing component and (2) the new component. Thus the components, such as the ones depicted in the left part of Figure 1 will not be treated equally: one of them (the upper one in the figures) may be thought of as a large assembly that grows by aggregating with the other one, which is small and simple. Hence the large assembly "grows" by adding a new building block. As a result of this, the building blocks of an assembly are partially ordered, making them similar to graph processes. The surface of the assembly after the aggregation step consists of most of the "old" surface combined with a small, new part that belongs to the building block. It is assumed that in determining the new surface structure, one does not need the entire internal structure of the large assembly. The upper half of Figure 2 depicts an aggregation step where the surface structures are graphs. Technically the letters $a, b, \ldots$ are node labels, but throughout this section we need not to distinguish between nodes and their label. The lower half of Figure 2 depicts the transformation of the surface structure, which is now a graph transformation.

A last assumption is that the effect of an aggregation step is *local*: a surface element that is irrelevant for a location does not suddenly become relevant when an aggregation takes place involving that location: e.g. in Figure 2, e is not relevant to the locations a and b involved in the aggregation – and thus e is not connected to either of them. In terms of graph rewriting, the assumption means that the newly introduced nodes can only be connected to those existing nodes that are neighbors of the nodes removed by the rewriting. Thus the neighbors of the new nodes f, g, h are either also new or chosen among the "old" neighbors c, d of a and b.

2.2 The embedding mechanism

The lower half of Figure 2 depicts the change in surface structure that corresponds to the aggregation step in the upper half of the figure. This change can be described by the application of a graph rewriting rule to the graph on the left: the rule removes nodes a, b and creates f, g, h. The edges of the new surface are either edges of the old surface, such as (d, e), or edges of the surface of the newly added building block, such a (f, g), or edges that connect the new nodes with the old ones, such as (c, f) or (h, d). The mechanism for establishing the latter kind of edges is known as an *embedding mechanism.* Here the embedding mechanism is very simple: each of the new nodes may take over the incoming and/or outgoing edges of one or more of the nodes that have been removed. In

Figure 2, f takes over the incoming edges from a and h takes over the outgoing edges of b. The rule applied is depicted in Figure 3: it consists of two graphs (the left-hand side and the right-hand side) and two binary relations *in* and *out* that express the way edges are established. In Figure 3 both relations contain only one pair; in general $in, out \subseteq N_L \times N_R$ where N_L and N_R are the sets of nodes of the left-hand side and the right-hand side, respectively.

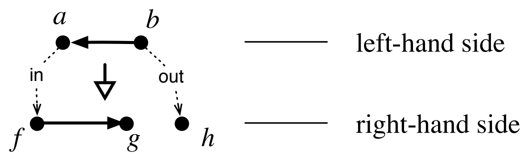


Fig. 3. A rule

2.3 Processes

In [CMR96,VJ02] graph processes are proposed as a way to describe "runs" of a graph rewriting system. Informally, a graph process is a structure obtained by gluing together the rule occurrences of a run, where the gluing is consistent with the way the rules are applied. Thus a graph process is essentially a directed acyclic graph where the nodes are those that occur in the run and where the edges represent the direct causal dependency relation: whenever a rule occurrence removes a node a and introduces a new node b, then b is directly causally dependent on a. This DAG is further decorated with extra information: the initial graph of the run is given as well as the rule occurrences. However there is no information on the order in which the rules are applied other than the causality relation. Figure 4 depicts a process: the initial graph is the linear structure at the top and there are three occurrences of the rule depicted at the left. There is no information about the relative order of rule occurrences 1 and 2, and so the process describes in fact three possible runs: the three rule occurrences may happen either in the order 1,2,3, or 2,1,3, or 1 and 2 may happen simultaneously, followed by 3. The dotted edges are established according to the embedding mechanism.

Using this notion one has three ways to view the components that act as building blocks in aggregation steps such as the one considered in Figure 2: a component with a surface structure, a graph rewriting rule, and a process. Since such components are not composed of smaller ones they are called "atomic". Similarly, the processes that represent a single rule are called atomic processes. Figure 5 depicts the three views; the arrows/lines labeled *in* and *out* represent the embedding mechanism.

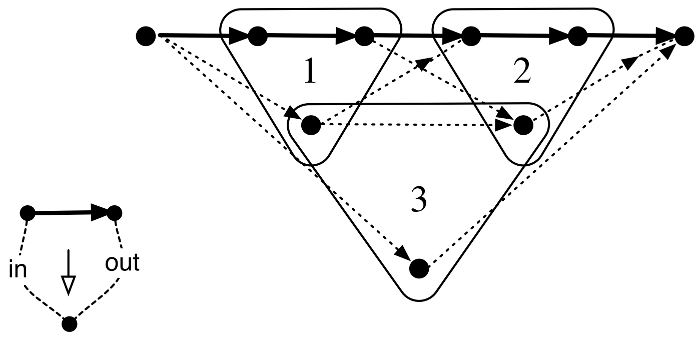


Fig. 4. Process

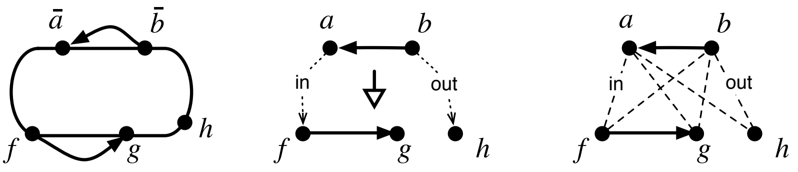


Fig. 5. Atomic component, rule and atomic process

The relationship between processes and assemblies is illustrated in Figure 6 (in the process, on the right, only the direct causality relation is represented). An important property of processes is that, when the embedding satisfies certain conditions, each slice (maximal set of causally unrelated nodes) *uniquely* determines a graph corresponding to that slice: the graph obtained by applying the rule occurrences that precede the slice in the causality relation to the initial graph – in any order consistent with the causality relation. In particular, the graph resulting from the process is uniquely determined; it is the configuration corresponding to the set of maximal nodes of the causality relation. It has been proven earlier [VJ02] that the embedding mechanism used in this paper satisfies the necessary condition. In Figure 6, one may e.g. consider the situation of the assembly after building blocks 1 and 2 are added. The corresponding slice consists of the square nodes. The property then means that the surface structure at this point of the aggregation process does not depend on the order in which blocks 1 and 2 were added; a posteriori inspection of an assembly (which blocks are present and how are they glued together) suffices to determine its surface structure. Since the order in which the aggregation takes place would probably be very hard to control, this property is of crucial importance.

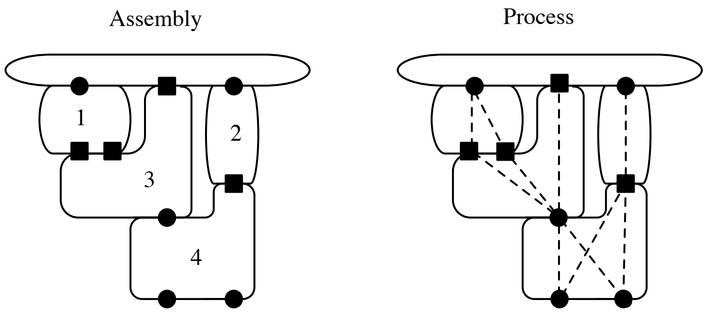


Fig. 6. Assembly and process

3 The algorithm

The aim of this section is to sketch how aggregation steps (building assemblies) and global steps (acting in a uniform way on all components) may be combined into a meaningful algorithm. In the context of this paper an algorithm describes a sequence of steps in which test tubes containing a solution are manipulated in order to obtain a solution with certain desired properties. One important way to manipulate a test tube is to cause a self-assembly process in it: atomic components (encoding graph transformation rules) are added to the test tube and self-assembly is allowed to happen.

The design of aggregation steps is now viewed as the design of a suitable set of graph transformation rules, and thus it is implicitly assumed that for each of those rules a component can be constructed which has the right surface structure, and that this component interacts in the right way with the other components. It has to be noted that the latter assumption is not obvious; however there is evidence that unintended interactions can be made improbable by a clever design of the components. The problem is similar to the DNA code word problem, which is an interesting research topic on its own.

The global steps, where all components in a solution are modified in a uniform way, are not local changes based on the surface structure of components, and thus the rewriting of graphs describing their surface structure is not a natural way to formalize them. However the more complete description of an assembly by a graph process provides information that is sufficient to express the global steps: the atomic components it is built from and the way they are combined. Two kinds of global steps are needed.

1. *extract*(m), where the symbol m represents a marker, i.e. a part of a component that can easily be detected by its physical properties. The operation removes all components in which the marker occurs from the test tube.
2. *disassemble*(R), where R is a set of rules used for aggregation. The operation removes all atomic components corresponding to rules of R from the assem-

blies in the test tube. Hence this operation may cause assemblies to fall apart into smaller pieces. The physical implementation would be a manipulation that breaks down the components corresponding to R, and only those. This could be achieved by designing these components in such a way that they are less stable or less resistant to heat than the other components.

The problem we focus on is the following. Consider two test tubes X and Y; Y contains the same components as X but also some additional ones; these are viewed as a contamination that has to be removed. The algorithm has to recognize the contaminating components, by yielding a test tube containing assemblies that encode the latter, where "encoding" means that their surface structure contains a copy, up to a relabeling, of that of the encoded components. The relabeling is needed to distinguish the encoding from the original.

It is assumed that only some of the components are relevant; in the example these have the surface structure depicted in Figure 7, where the x_i belong to the set $\{a, b\}$. The symbols a, b, l, r represent certain kinds of surface elements and n is even. The symbols a, b, l, r are relabeled $\tilde{a}, \tilde{b}, \tilde{l}, \tilde{r}$ in the encoding.

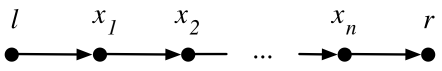


Fig. 7. Initial structure

For our purposes X and Y are sets of concrete structures, And obviously both of them will in general contain many isomorphic copies of each of their elements.

The algorithm consists of the following steps:

1. Recognize the relevant components of Y: prepare their encoding by forming a suitable assembly around each of them.
2. Extract these assemblies from Y.
3. Add the result of this to X and form assemblies that mark the encodings of components that occur in X.
4. Extract the marked encodings; the remaining ones are the desired ones.

To realize step 1, first ignore the relabeling. Then the step can be carried out by adding to Y the rules (i.e. components realizing the rules) of Figure 8: these form an assembly that is essentially a binary tree where the leaves are labeled a or b and each two consecutive leaves have the same parent.

The graph process in the upper part of Figure 9 represents an intermediate stage in the formation of such an assembly: one more step (applied to the two square nodes) will complete the tree. Only if the component is of the right form

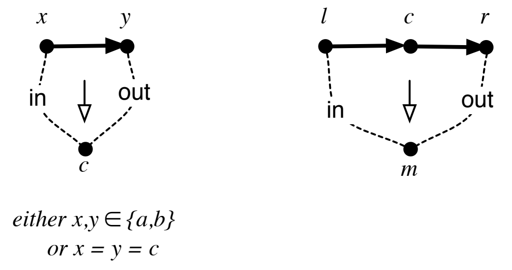


Fig. 8. Rules for assembly in step 1

the process can be completed by an application of the rule at the right of Figure 8 which attaches the marker m to the assembly (lower left part of Figure 9). The relabeling of a, b, l, r into $\tilde{a}, \tilde{b}, \tilde{l}, \tilde{r}$ can be taken into account by modifying the rules in the way depicted in Figure 10: a relabeled copy of the encoded structure is produced.

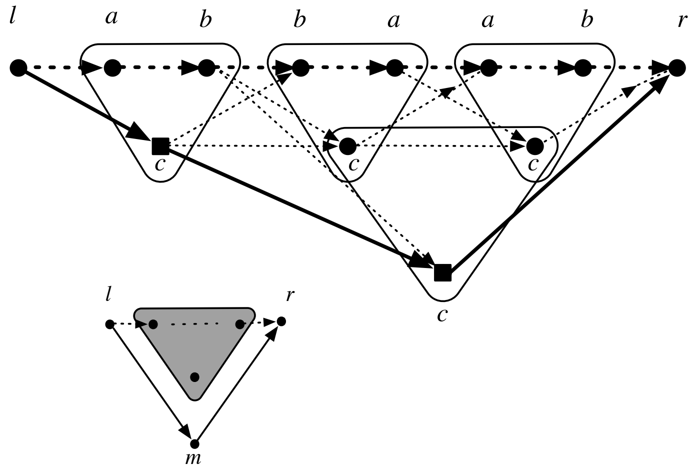


Fig. 9. Assembly in step 1

Step 2 can be realized by an *extract*(m) operation. A potential problem is that the rules of step 1 can be applied to a component of the right form in such a way that the assembly obtained is a forest, but not a tree; then that component is not encoded. However, one may improve the result by using an

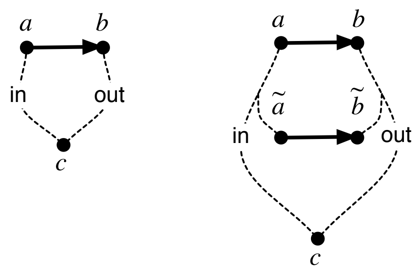


Fig. 10. Modified rule

iterative procedure: repeat the sequence (step 1, *extract*(m), *disassemble*(R_1)), where R_1 is the set of rules of step 1, until nothing is extracted.

Step 3 can be realized by adding the assemblies extracted in step 2 and the rules of Figure 11 to X. The effect is depicted in Figure 12: an encoding (starting with $\tilde{l}$) and a component (starting with l) are traversed, until L becomes adjacent to r and $\tilde{r}$. In that case the encoding is marked, using the rule at the right of Figure 11. The dotted edges in Figure 12 are established by the embedding mechanism. If the encoding and the component do not correspond, then L does not become adjacent to both r and $\tilde{r}$ and the marking does not occur. Step 4 can be done by an *extract* operation. Again, there is a potential problem because the assemblies with the encodings may get neutralized by trying to combine with the wrong component: e.g. when an assembly encoding *aaba* combines with component *aabba* the aggregation process of Figure 12 gets stuck after 4 steps. Again, an iterative procedure is needed: repeat the sequence (step 3, step 4, *disassemble*(R_3)) where R_3 is the set of rules of step 3.

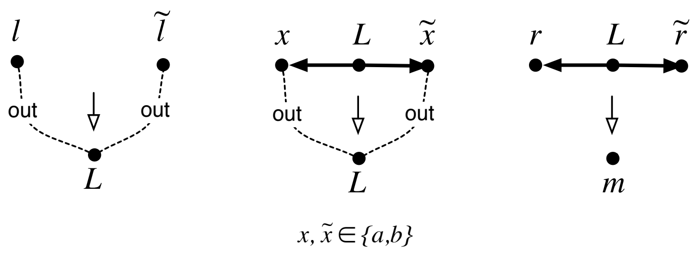


Fig. 11. Rules for traversal

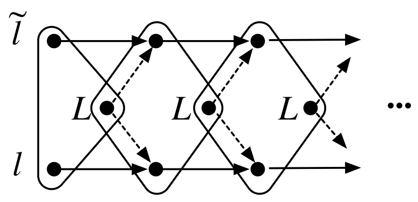


Fig. 12. Traversal - assembly view

The result of the algorithm is a test tube containing assemblies which encode the contaminants, i.e. the components of Y that do not occur in X. Using an iterative procedure similar to the one combining steps 3 and 4 above, it is in principle possible to use this to remove the contaminants from Y.

4 Discussion

The aim of the paper is to explore the use of graph rewriting based on embedding for the understanding of self-assembly and natural computing. The basic idea is that graphs capture the active surface structure that controls the way components in a solution aggregate, and that the way in which such aggregation changes the surface structure can be captured by graph rewriting. However one may expect that most meaningful algorithms in this context do not only require the building of ever larger assemblies, but also operations that break down or modify such assemblies, and in the algorithm sketched in Section 3 a few of these operations are used: extracting certain components according to particular "marker" labels, or removing certain atomic components from the assemblies in a solution. Thus what seems to be needed is an interplay between aggregation operations, which are described by graph transformation rules, and which act on the graphs that describe the active surface of components, and global operations in which all assemblies of a given kind in a solution are modified. Since assemblies correspond to processes of the graph rewriting systems that describe their formation, the theory of graph rewriting and graph processes may provide a way to obtain a formal framework in which both kinds of operations can be combined in an elegant way.

Obviously, the material presented here is of a very speculative nature, since the implicit assumptions concerning the possible realization of the approach in the physical world may turn out to be naive or unrealistic. To mention just a few: when reducing the problem of controlling self-assemby to the problem of writing a suitable graph transformation system, it is assumed that each rule written down can be realized by a component that behaves exactly in the right way: not only does it aggregate with another component when the structure corresponding to its left-hand side matches part of the structure of the other component, but this is also the *only* way it interacts with other components. Another thorny issue is the assumption about the information to be encoded into the edges, information that is handled by the embedding mechanism: what are exactly the relationships between locations on a component that are relevant? How to encode spatial information into those edges? Also for the the global operations many questions remain: on the one hand they may seem rather ad-hoc, but on the other hand they are quite simple. In spite of these reservations, however, the correspondence between graph rewriting and graph processes on the one hand and aggregation and assemblies on the other hand seems simple and natural enough to deserve further attention.

References

[Cas06] Leandro N. De Castro. *Fundamentals of natural computing: basic concepts, algorithms, and applications.* Published by CRC Press, 2006.

[CMR96] A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundam. Inf.*, 26(3-4):241–265, 1996.

[EEPT] H. Ehrig, K. Ehrig, U. Prange, and G. Taenzer. *Fundamentals of Algebraic Graph Transformation.* EATCS Monographs in Theoretical Computer Science.

[EKMR97] H. Ehrig, H-J. Kreowski, U. Montanari, and G. Rozenberg. *Handbook of Graph grammars and Computing by Graph Transformation.* World Scientific, 1997.

[ETP$^+$04] A. Ehrenfeucht, T.Harju, I. Petre, D.M. Prescott, and G. Rozenberg. *Computation in Living Cells - Gene Assembly in Ciliates.* Natural Computing Series. Springer Verlag, 2004.

[GJC91] GM.Whitesides, JP.Mathias, and CT.Seto. Molecular self-assembly and nanochemistry - a chemical strategy for the synthesis of nanotructures. *Science*, 254:1312–1319, 1991.

[HLP08] Tero Harju, Chang Li, and Ion Petre. Graph theoretic approach to parallel gene assembly. *Discrete Applied Mathematics*, 156(18):3416–3429, 2008.

[KGL04] Eric Klavins, Robert Ghrist, and David Lipsky. Graph grammars for self assembling robotic systems. In *Proceedings of the 2004 IEEE International Conference on Robotics and Automation*, pages 5293–5300, 2004.

[KR08] Lila Kari and Grzegorz Rozenberg. The many facets of natural computing. *Commun. ACM*, 51(10):72–83, 2008.

[VJ02] N. Verlinden and D. Janssens. Algebraic properties of processes for local action systems. *Mathematical. Structures in Comp. Sci.*, 12(4):423–448, 2002.

[WLWS98] E. Winfree, F. Liu, L.A. Wenzler, and N.C. Seeman. Design and self-assembly of two-dimensional dna crystals. *Nature*, (394):539–544, 1998.

Prof. Dr. Dirk Janssens

Department of Computer Science
University of Antwerp
2020 Antwerpen
Dirk.Janssens@ua.ac.be

Dirk Janssens and Hans-Jörg Kreowski have co-authored several papers. Moreover, Hans-Jörg Kreowski was in Dirk Janssens' Ph.D. jury in 1983. When being asked for their relation, Dirk points out that it has always been a pleasure to meet Hans-Jörg Kreowski, not only because of his inspiring scientific ideas and his way of concentrating on the right questions, but also because of his kindness and patience.

Generation of Celtic Key Patterns with Tree-based Collage Grammars

Renate Klempien-Hinrichs and Caroline von Totth

Abstract. Tree-based collage grammars are a syntactic device for modelling visual languages. Celtic art provides such languages, which follow precise rules of construction, for instance key patterns and knotwork. In this paper, we study the syntactic generation of Celtic key patterns using tree-based collage grammars. Moreover, we compare the regulation mechanisms employed here in order to ensure that only consistent key patterns are generated with those that were used in a previous study for knotwork.

1 Modelling with collage grammars

Collage grammars are a device to generate sets of pictures in a syntactic manner [HK91,HKT93,DK99]. They were developed by equipping hyperedge replacement grammars with spatial information: nonterminals come with a location and an extension in some Euclidean space, and the role of terminals is played by collages, where a collage is a union of parts and a part is a set of points. Equivalently, collage grammars may be seen as a combination of a tree grammar and an algebra interpreting the generated trees as collages [Dre00,Dre06], so-called tree-based collage grammars.

The initially proposed collage grammars were context-free. More powerful grammar types include the table-driven (or T0L) grammars, where nonterminals are replaced in parallel with rules from one of finitely many tables [KRS97] (for T0L collage grammars see [DKK03]). The behaviour of a table-driven tree grammar may be emulated by a regular tree grammar generating unary trees as input for a top-down tree transducer; such a unary tree basically states the sequence of the applied tables in the T0L grammar [Dre06, Lemma 2.5.7].

For any picture-generating mechanism, it is of particular interest to find out whether a certain class of pictures can be generated. For table-driven collage grammars, the case study in [DK00] (see also [Dre06, Sect. 3.5]) shows a way to produce Celtic knotwork. The idea is to identify a small pattern that is repeated in a tiling-like manner to form the whole design (see [Bai90,Slo95] for distinct ways to divide a design).

In addition to knotwork, there are three other major design types in Celtic art: key patterns, spirals, and interlacings with animal or human forms (see [All93,Bai51] for overviews). Historically, the designs were drawn in medieval manuscripts, carved on standing stones, or forged in precious metalwork.

In this paper, we present a case study modelling Celtic key patterns, sometimes also called maze patterns, by means of collage grammars. Where a Celtic knot consists of continuous cords that interweave, key patterns have continuous

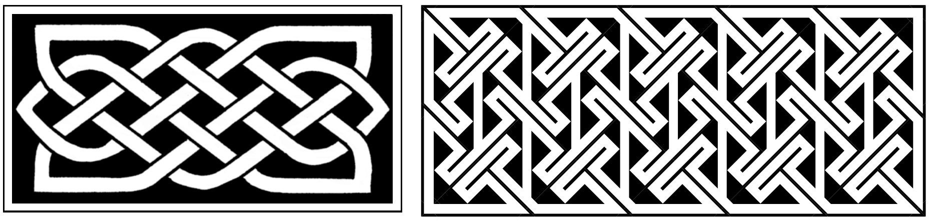


Fig. 1. Celtic designs: a knot (left) and a key pattern (right)

paths that follow straight lines, usually at 45° angles, and meet at crossings (see Figure 1 for examples).

Methods for the construction of key patterns by hand, that may have been used by the original Celtic artists, are described in [Bai90,Bai93,Mee02]; these methods are mainly oriented at which straight long lines may be drawn. On the other hand, small patterns or *tiles* may be identified that are repeated throughout the whole design. For this, Sloss [Slo97] overlays a key pattern with a grid whose axes run parallel to the pattern border, obtaining rectangles of various sizes. In contrast (and more in line with [Bai93]), we propose to use triangles and squares with a 45° slope as tiles. We then use tree-based collage grammars to show how these tiles can be structurally combined to yield well-formed key patterns. Moreover, we discuss the structural differences between knotwork [DK00] and key patterns. All collage grammars for key patterns that we developed were implemented using Frank Drewes' system Treebag [Dre06, Sect. 8].

The paper is organised as follows: In Section 2, basic notions of tree-based collage grammars are recalled. Section 3 contains a structural analysis of basic key patterns and a description of the collage grammar that we used to generate them. In Section 4, variations of the first model are proposed. The paper concludes with a brief summary and ideas for various lines of future research.

2 Tree-based collage grammars

The basic notions for tree-based collage grammars collected in this section follow presentations given in [DKL03,DEKK03,Dre06]. Please consult these, [Dre06] in particular, for more detail, including examples.

Let $\mathbb{N} = \{0, 1, 2, 3, \dots\}$ denote the set of natural numbers, and $[n] = \{1, \dots, n\}$ for $n \in \mathbb{N}$. Moreover, $\mathbb{R}$ denotes the set of real numbers, and $\mathbb{R}^2$ the Euclidean plane, which contains points $(x, y) \in \mathbb{R}^2$.

We will use expressions to represent collages and call such an expression a *term* (or *tree*) over a certain signature. In general, a *signature* is a finite set Σ of symbols, each symbol $f \in \Sigma$ being assigned a unique *rank* $\mathrm{rank}_\Sigma(f) \in \mathbb{N}$. The set T_Σ of *terms over* Σ is the smallest set such that for $n \in \mathbb{N}$, $f[t_1, \dots, t_n] \in T_\Sigma$ for all $f \in \Sigma$ with $\mathrm{rank}_\Sigma(f) = n$ and all $t_1, \dots, t_n \in T_\Sigma$. For $n = 0$, we may omit the parentheses in $f[\,]$, writing just f instead.

It is now possible to define collages and the relationship between collages and terms in T_Σ formally. A *part* $p \subseteq \mathbb{R}^2$ is a bounded subset of the Euclidean plane; intuitively, p is a set of black points. A collage C is a finite set of parts, and the set of all collages is denoted by $\mathcal{C}$. Transformations $f\colon \mathbb{R}^2 \to \mathbb{R}^2$ on $\mathbb{R}^2$ are canonically extended from points to parts and collages, i.e., $f(p) = \{f(x, y) \mid (x, y) \in p\}$ for a part p, and $f(C) = \{f(p) \mid p \in C\}$ for a collage C. For affine transformations $f_1, \dots, f_n$ on $\mathbb{R}^2$, $\langle\!\langle f_1 \cdots f_n \rangle\!\rangle$ denotes the operation $f\colon \mathcal{C}^n \to \mathcal{C}$ given by $f(C_1, \dots, C_n) = \bigcup_{i\in[n]} f_i(C_i)$ for all $C_1, \dots, C_n \in \mathcal{C}$. A *collage operation* is either an operation of the form $\langle\!\langle f_1 \cdots f_n \rangle\!\rangle$ for affine transformations $f_1, \dots, f_n$ ($n \geq 1$) or a constant collage, viewed as an operation of arity 0. A *collage signature* is a finite signature Σ consisting of collage operations, where ranks coincide with arities. For a term $t \in T_\Sigma$, its *value* is val(t), i.e., the collage $\mathrm{val}(t) = f(\mathrm{val}(t_1), \dots, \mathrm{val}(t_n))$ if $t = f[t_1, \dots, t_n]$.

By such an interpretation of ranked symbols as collage operations, appropriate grammatical devices for generating sets of terms allow us to generate sets of collages. Among such devices, we consider the regular tree grammar and the top-down tree transducer, and we call the combination of a tree generator and a collage algebra a *tree-based collage grammar*. Since we deal exclusively with tree-based collage grammars in this paper, in the following we will refer to them simply as collage grammars.

A regular tree grammar works analogously to a regular string grammar, but by replacing nonterminals in terms and having nonterminals occur only without subterms. Formally, a *regular tree grammar* is a system $G = (N, \Sigma, P, S)$ consisting of

- a finite set N of *nonterminals*, which are considered to be symbols of rank 0,
- a signature Σ disjoint with N,
- a finite set $P \subseteq N \times T_{\Sigma\cup N}$ of *productions*, and
- an *initial nonterminal* $S \in N$.

A term $t \in T_{\Sigma\cup N}$ *directly derives* a term $t' \in T_{\Sigma\cup N}$, denoted by $t \longrightarrow_P t'$, if there is a production $A ::= s$ in P such that t' is obtained from t by replacing an occurrence of A in t with s. The *language generated by* G is $L(G) = \{t \in T_\Sigma \mid S \longrightarrow_P^* t\}$, where $\longrightarrow_P^*$ denotes the reflexive and transitive closure of $\longrightarrow_P$; such a language is called a *regular tree language*.

A tree transducer transforms some input tree into an output tree by traversing the input tree symbol by symbol, possibly storing some information in one of finitely many states. Formally, a *top-down tree transducer* is a system $td = (\Sigma, \Sigma', \Gamma, R, \gamma_0)$ consisting of

- finite *input* and *output signatures* Σ and Σ',
- a finite signature Γ of *states* of rank 1, where $\Gamma \cap (\Sigma \cup \Sigma') = \emptyset$,
- a finite set R of *rules*, and
- an *initial state* $\gamma_0 \in \Gamma$.

Each rule in R has the form

$$\gamma[f[x_1,\ldots,x_n]] \to t[\![\gamma_1[x_{i_1}],\ldots,\gamma_m[x_{i_m}]]\!],$$

where $\gamma,\gamma_1,\ldots,\gamma_m \in \Gamma$ are states, n is the rank of symbol $f \in \Sigma$ and $x_1,\ldots,x_n$ are pairwise distinct variables (of rank 0 and not occurring in $\Sigma \cup \Sigma' \cup \Gamma$); and t is a term with symbols from Σ' and $m \in \mathbb{N}$ subterms of the form $\gamma_j[x_{i_j}]$ with $i_j \in [n]$, i.e., $x_{i_j} \in \{x_1,\ldots,x_n\}$.

There is a *computation step* of *td* from a term s to a term s', denoted by $s \Rightarrow_R s'$, if R contains a rule $\gamma[f[x_1,\ldots,x_n]] \to t[\![\gamma_1[x_{i_1}],\ldots,\gamma_m[x_{i_m}]]\!]$, s has a subterm of the form $\gamma[f[t_1,\ldots,t_n]]$, and replacing this subterm in s with $t[\![\gamma_1[t_{i_1}],\ldots,\gamma_m[t_{i_m}]]\!]$ (which is formed as in the rule, but with corresponding subtrees $t_{i_1},\ldots,t_{i_m}$ from s instead of variables $x_{i_1},\ldots,x_{i_m}$) yields s'. The *tree transformation computed by td* is given by $td(s) = \{s' \in T_{\Sigma'} \mid \gamma_0[s] \Rightarrow_R^* s'\}$ for every tree $s \in T_\Sigma$, where $\Rightarrow_R^*$ denotes the reflexive and transitive closure of $\Rightarrow_R$.

3 The structure of Celtic key patterns

In this section, we show a way to structure Celtic key patterns so that they can be easily generated by collage grammars. We first identify a small set of tiles – the constant collages – that allows us to put together a whole pattern, and then describe the syntactic synthesis.

Following [Bai93], a typical key pattern will often be a coherent composition of isosceles right-angled triangles to which a border line needs to be added, see Figure 2 left.

Moreover, note that the smooth borders must be produced by triangle hypotenuses. In particular, the top and bottom border triangles have horizontal hypotenuses, whereas the hypotenuses of all other triangles are vertical. This

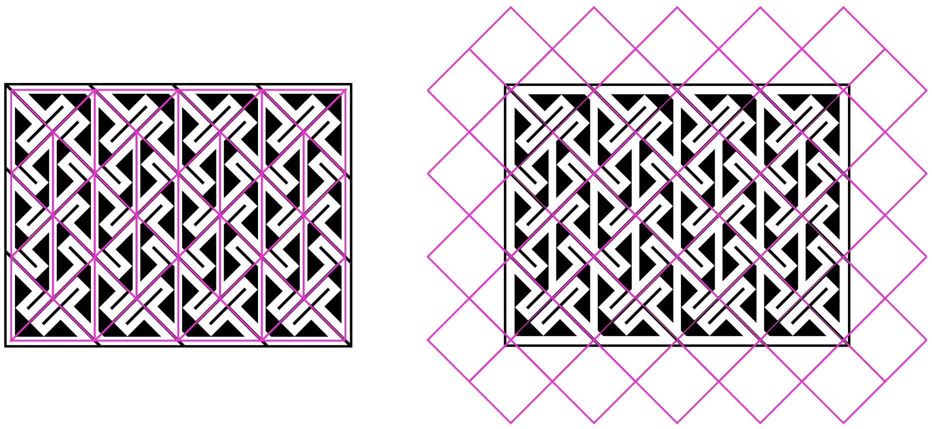


Fig. 2. Division of a key pattern into triangles (left) and squares (right)

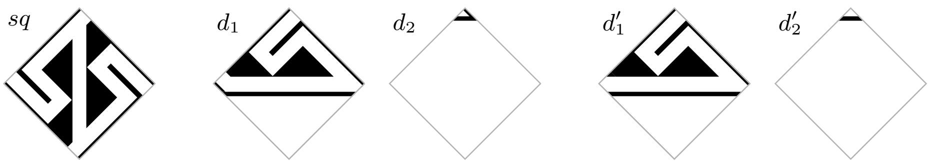


Fig. 3. The set of basic tiles

suggests a division into square tiles as indicated in Figure 2 right, which also takes care of drawing the border lines. The resulting set of basic tiles (modulo reflections and 90° rotations) is shown in Figure 3. At first glance, the last two of the five tiles appear to be identical to the preceding two. There are, however, subtle differences, for two reasons. First, key patterns have two different types of corners since a corner is obtained by so-called mitring, which means reflecting a basic triangle at one of its shorter sides; compare, e.g., the two right corners of a pattern in Figure 2. Secondly, a line always comes with a thickness, and where a hand-drawn line lies on the connecting sides of two tiles, then each of the graphic tiles gets a line of half the thickness. Thus, the two middle tiles d_1 and d_2 in Figure 3 have a short line at a 135° angle that is not present in the last two tiles, and in tile d'_1 the left corner of its left black triangle is pointed.

Now let us come to the syntactic arrangement of these tiles in a rectangular key pattern. The task is to overlay the pattern with a tree whose interpretation through collage operations yields the pattern. In order to find the tree, we reuse the idea from [DK00] for rectangular knotwork, namely to develop the pattern from its centre. One can immediately observe from Figure 2 that the left half of the pattern may be obtained from the right half by a rotation of 180° about the centre. The right half in turn may be seen as having a horizontal backbone through the centre, where each vertical line of square tiles above the backbone may be obtained from the same line below the backbone by a rotation of 180° about the crossing point of the vertical line with the backbone. (The border, however, will need special treatment, which we will discuss at the end of the section.) Thus, we can concentrate on the lower right section as sketched in Figure 4 to get an idea of the required collage signature.

As proposed earlier, the figure displays a portion of the tree for the key pattern in Figure 2 overlaying the pattern structure. The constant symbols sq, d_1 and d_2 refer to the tiles given in Figure 3; their placement results from the operations above them in the tree and reflects the layout of the tiles in the pattern. Constants r_1 and r_2 result from mitring d_1 and d_2, respectively, as described above. Constant sq_m results also from a reflection of sq, but this time in the vertical (or, equivalently, horizontal) axis through the middle of the tile. This reflection is necessary because two adjoining tiles need to agree in the path linking them (this is unlike the knotwork tiles considered in [DK00], where the cord always leaves through the centre of a tile side).

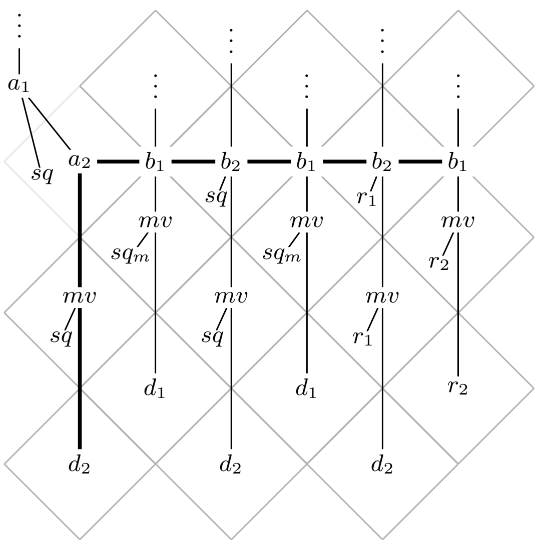


Fig. 4. A sketch of a tree over collage operations that generates the lower right section of the key pattern in Figure 2

The operations work as follows. For all collages C_1, C_2, C_3, C_4:

- $mv(C_1, C_2)$ puts C_1 at the current position (meaning the affine transformation applied to C_1 is the identity), and translates C_2 downward for the diagonal length of a tile;
- $b_1(C_2, C_3, C_4)$ moves C_2 half this length downward, C_3 half the length to the right, and C_4 half the length downward followed by a rotation of 180° about its current position;
- $b_2(C_1, C_2, C_3, C_4)$ behaves as b_1, but has in addition collage C_1 which is put at the current position;
- $a_2(C_1, C_2)$ moves C_1 one diagonal length downward and C_2 half the length to the right; and
- $a_1(C_1, C_2, C_3)$ puts C_1 and C_2 both at the current position and rotates C_3 by 180° about that position.

Now these operations have to be organised in order to yield well-formed key patterns. Let us first take a look at a successful generation process, like the one shown in Figure 5, where pictorial representations of terms are given for easier understanding. Starting from some initial item, the idea is to specify the width of the pattern before the height.

In order to get uniform growth both horizontally and vertically, some regulation mechanism has to be employed. For this, we use a regular tree grammar producing unary terms over the symbols w_1, w_2, h_1 of rank 1 and h_2, wh_2 of rank 0 such that the terms without parentheses belong to the string language of the regular expression $w_1^*(w_2h_1h_1^*h_2|wh_2)$. Symbols w_i specify the width and symbols h_j the height of the pattern, symbols x_1 (i.e., symbols with the index 1) the

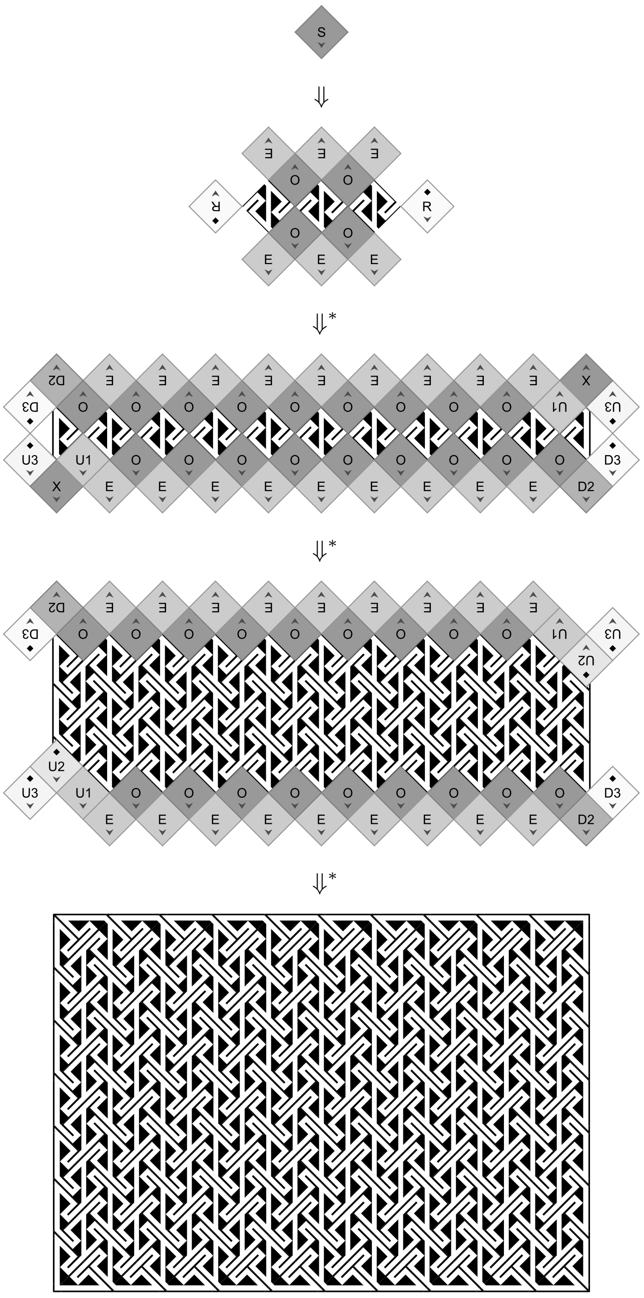


Fig. 5. Typical derivation sequence for a key pattern

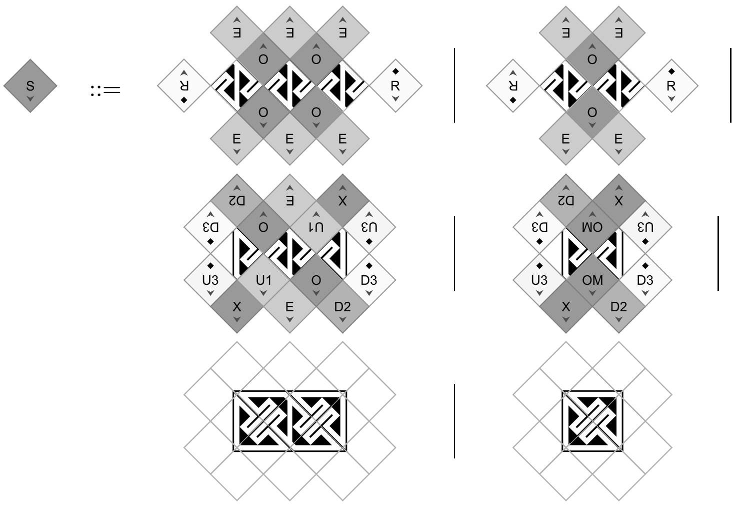


Fig. 6. Transducer rules to start a computation from the initial state

internal growth and symbols y_2 the border construction, with symbol wh_2 for the special case that no vertical growth is to take place. These control terms are then used as input for a top-down tree transducer that consumes in each step the first symbol of the term and copies the rest of the term identically to all newly introduced states. The rules of the transducer are shown in Figures 6–10, in the same pictorial representation as above, and with grey squares representing the named transducer states.

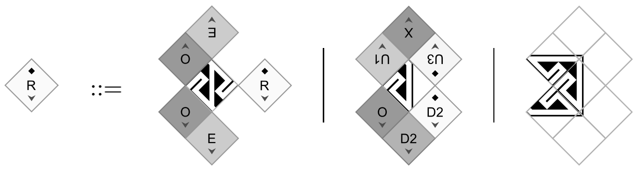


Fig. 7. Transducer rules to grow a design horizontally

– State S is the initial state and may encounter symbols w_1, w_2, wh_2 (Figure 6). As the centre of a key pattern may either lie inside a tile sq or inbetween two such tiles, there are two options for processing each of the three symbols, making the transducer nondeterministic. The right-hand-sides in the first row of Figure 6 are created when w_1 is consumed by S. The next two right-hand-sides result when state S encounters immediately symbol w_2, i.e., when no horizontal growth takes place. In this case, state OM is introduced to work analogously to state O by producing tiles sq_m, but producing tile d'_1 instead

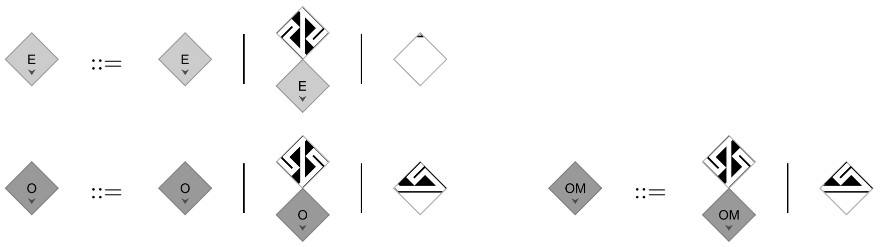


Fig. 8. Transducer rules to grow a design vertically

of d_1 for the top and bottom border (Figure 8 on the right). We leave it to the interested reader to find out why nevertheless the key pattern shown in Figure 1 cannot be generated by the resulting set of transducer rules, and how to remedy the problem.

- State R executes horizontal growth and may encounter symbols w_1, w_2, wh_2 (Figure 7). The first right-hand-side is the result of processing w_1, the second results from consuming w_2, and the last from consuming wh_2, respectively.
- State E ignores symbols w_i (i.e., it consumes them and proceeds to their subtree without any further action) and produces the even-numbered vertical lines using tile sq; analogously, state O produces the odd-numbered vertical lines using tile sq_m. Finally, state OM behaves nearly the same as O, with the difference that it places tile d'_1 instead of d_1 for the border (Figure 8).
- States D2 and D3 encounter only symbols h_i, from which they grow the lower half of the right border (and the upper half of the left border).

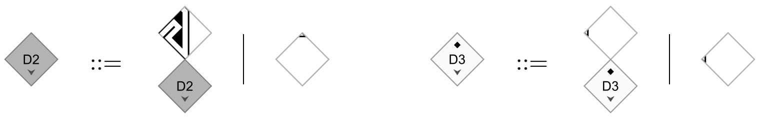


Fig. 9. Transducer rules to grow the vertical borders

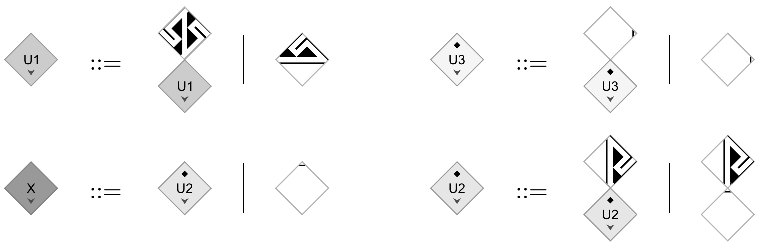


Fig. 10. Transducer rules to produce the second type of corner

- The upper half of the right border (and the lower half of the left border) is grown through states U1, U2 and U3 from symbols h_i (Figure 10). These states produce the alternative corner with tiles d'_1 and d'_2 as follows: As may be seen in Figure 5, state U2 is one step behind the other two states, with the retardation provided by state X on the first occurrence of symbol h_1. This is because U2 has to produce (a rotation of) tile d_1 while reading symbol h_1, but on encountering h_2, i.e., for the corner, it has to be tile d'_1 instead, which is suitably combined with tile d'_2.

Reconsidering the complete model, one may ask why one should start growing key patterns from their centre, and not rather from the centre of one of their sides, or indeed a corner. These are, in fact, viable and sometimes even preferable alternatives. The question will be discussed with the variations of key patterns studied in the next section.

4 Variations of Celtic key patterns

In Celtic key patterns, both the basic tiles and their structural arrangements offer many possiblities for distinctive designs. A small collection of alternatives is presented in this section.

The considerations of the preceding section started with identifying a basic triangle tile in a complete key pattern. In the original Celtic artwork, many variations of this first, very typical, triangle may be found. A small collection of triangles is shown in the first row of Figure 11. Note that in all triangles, the

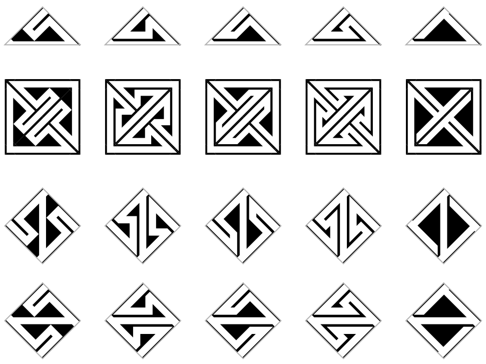


Fig. 11. Various basic triangles, smallest corresponding key patterns, and derived square tiles for larger key patterns

path enters at the same position close to the right angle, continues along the hypotenuse with half the original width, and leaves again at the acute angle opposite of the entrance. Any triangle complying with these geometric constraints can be used instead of the basic triangle of the previous section to produce a well-formed pattern with continuous paths. The smallest such patterns, consisting entirely of border triangles, are shown in the second row of the figure. In order to grow larger patterns with the method of the preceding section, one needs to compose a triangle with its 180° rotation about the centre of the hypotenuse to form a square tile. The derived square tiles are shown in the last two rows of Figure 11: The third row contains the squares with vertical centre path (as used in the preceding section), and the fourth row the squares with horizontal centre path, obtained from the upper squares by a 90° rotation. A small selection of key patterns using these squares is shown in Figure 12.

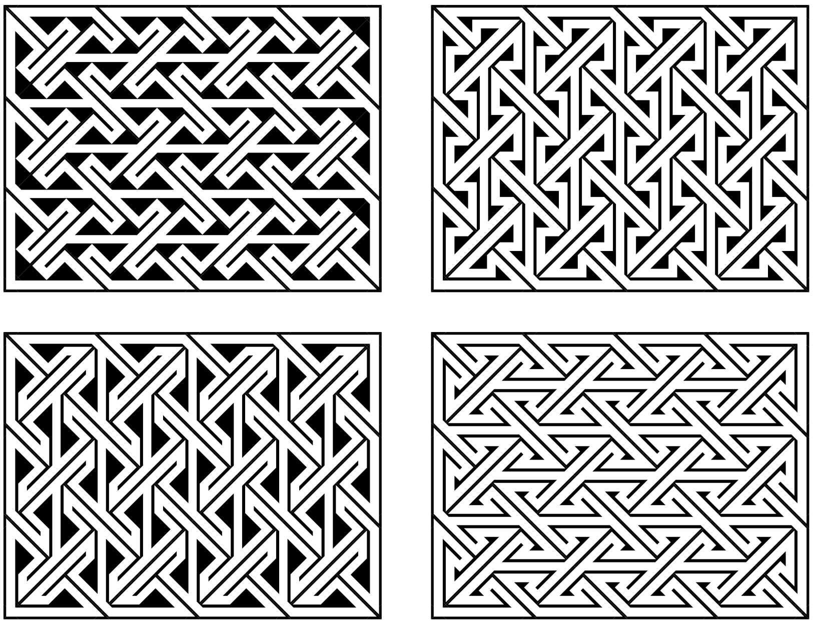


Fig. 12. Key patterns based uniformly on a triangle

The geometric constraints for triangles imply similar constraints for squares: A path enters at a fixed position on each side of a square, and the paths within a square are connected in some way. In Celtic art, many such squares have been used (see Figure 13 for a small sample). Since these squares cannot be divided satisfyingly into triangles, they have to be combined with border triangles so that full key patterns may be generated. Some samples are shown in Figure 14.

One can also combine more than one square type in a key pattern. To avoid disorderly arrangements, suitable syntactic rules may be employed. One such rule is to distinguish between constants *sq* and sq_m (rather than having sq_m as a reflection of *sq*) and thus admit interpretations by different squares; the key patterns in Figure 15 are of this kind. This rule may be refined by requiring

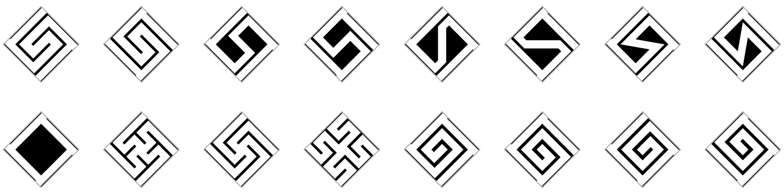


Fig. 13. Sample collection of square tiles

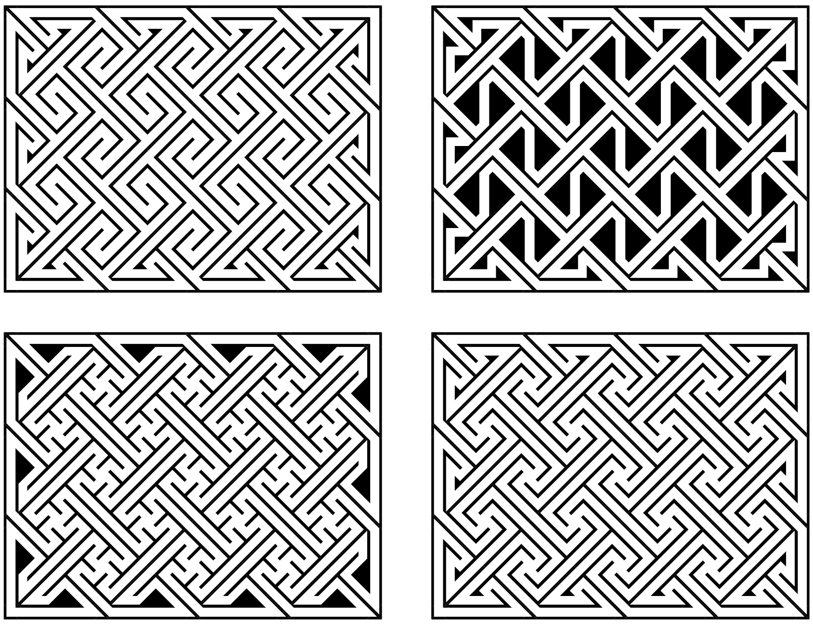


Fig. 14. Key patterns based on a square with a triangle border

that two distinct squares be used alternately to interpret *sq*, which leads to key patterns as shown in Figure 16.

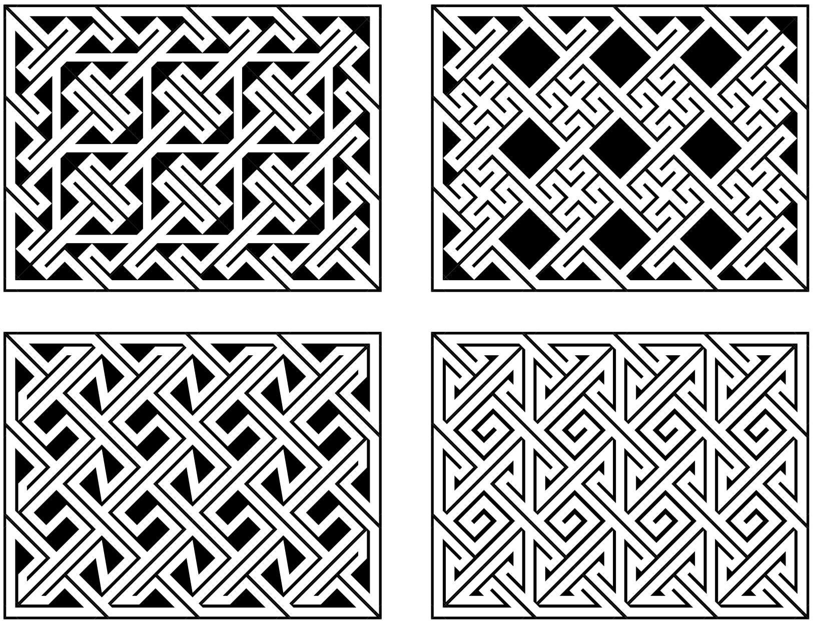


Fig. 15. Key patterns where sq and sq_m are interpreted differently

Further distribution rules for different squares in a key pattern include:

- Using different squares for the lower half and the upper half; this may be implemented in our model by doubling the states of the transducer so that the two halves may be treated independently, but to the same vertical extension as given by the input term.
- Using different squares for each row of squares, but having the choice for the upper half reflected in the lower half; this may be implemented in the input term of the transducer by replacing the symbols h_1 with references to the squares that shall be used.
- Finally, one may use branching synchronization to achieve a more general symmetric distribution of synchronized squares. The method used to generate breaklines within the rectangular Celtic knot designs in [Dre06] is well-suited for this. A nesting depth of 2 for the resulting branching collage grammar is sufficient to ensure that the overall design maintains horizontal and vertical symmetry even though blocks of different squares may alternate in a nondeterministic fashion.

Coming back to the question at the end of Section 3, the two rules above are good reasons to start the growth of a key pattern from the centre at least of the

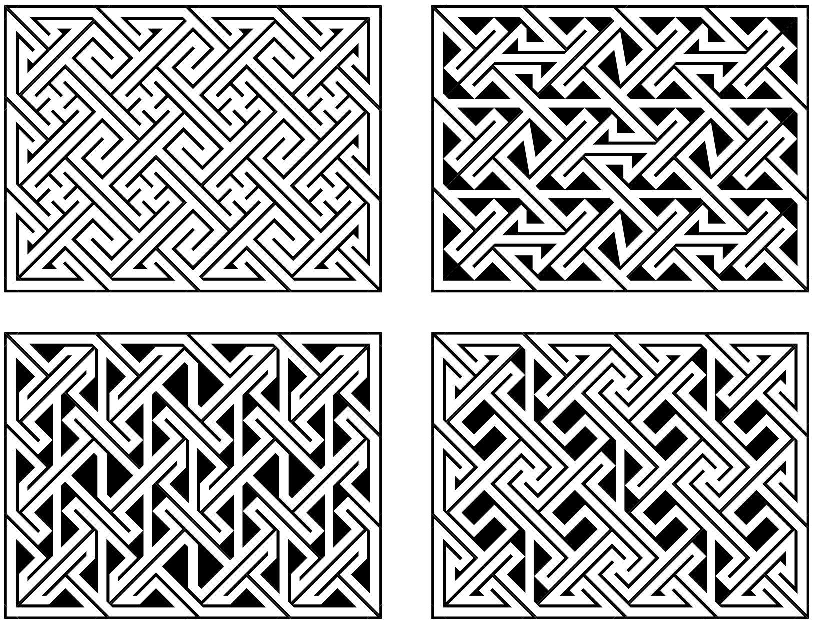


Fig. 16. Key patterns where *sq* is interpreted alternately by two distinct squares

vertical direction. A further reason lies in the last four tiles shown in Figure 13, i.e., the dead-end spiral, the only type of tile given in that figure that does not have (at least) 180° rotational symmetry. Our model admits rotation of this tile between the lower and the upper half of a key pattern; see, e.g., the last pattern in Figure 15. If, however, it is desired that all occurrences of this tile have the same orientation, then it is preferable to start vertical growth from the top (or the bottom) of the pattern.

5 Conclusion

In this paper we have shown how rectangular key patterns with interior and border variations can be modelled using tree-based collage grammars to interpret the generated terms. All considered key pattern variations are covered by a class of tree generators that combine a regular tree grammar for unary terms with a top-down tree transducer.

The model for rectangular knotwork as proposed in [DK00,Dre06] is also based on square tiles, but uses branching tree grammars to encode a horizontally and vertically symmetric distribution of so-called breaklines over a knot. Consequently, the syntactic structure of rectangular knotwork with breaklines may be assumed to be one level more complex than the structure of rectangular key patterns with regular distribution of tiles.

For both knotwork and key patterns, it is interesting to study how evenly placed holes of varying sizes and shapes can be added to the base pattern. These holes can either be left plain, or they can be filled with any type of Celtic tiling. The designs so produced are called carpet-page designs. We note that for an authentic look, the holes will have to be distributed in a symmetric fashion across the expanse of the base design.

While the basic shape of such a hole is rectangular, in Celtic art holes also come in L, cross or crosslet shapes. The boundary of the hole itself needs to be sealed off with specific border tiles. Of course, holes need not be restricted to the interior of the key pattern boundary, but may also lie directly on it, breaking up the rectangular border. If these border cutouts are regularly distributed as well, the resulting key pattern designs display a multitude of interesting shapes, of which the cross is the most basic.

In [DK00,Dre06], a way to include such holes is proposed for square knotwork panels that are grown diagonally from the centre to the corners and thus come with a natural vertical/horizontal synchronisation. It would be nice to have a generalised method that works for *rectangular* patterns (and therefore needs additional synchronisation), in any kind of tiling with defined border tiles.

In Celtic art, it is often customary to fill such holes with some contrasting decorative pattern. This presents a modelling problem, since whatever pattern is created may not grow beyond the boundaries of the hole, and collage grammars do not offer context-sensitive queries. No matter whether the new pattern is created by subdivision or growth, information about the shape and size of the hole is required in order to create a correct pattern with a border that seamlessly joins the boundary of its parent hole. Then, a formalism is required that can deal with the multitude of possible shapes and sizes and create matching patterns. Additionally, some synchronisation between the scale of the tiles in the base panel – which inform the dimensions of possible holes – and the scale of hole-filling tiles must take place.

The creation of round key pattern (or knotwork) panels with a circular tesselation pattern calls for another type of construction method altogether. Collage grammars as they are used here rely on local replacement, which cannot be used to recompute the scaling and placement operations necessary to evenly place tiles along a growing circle. A suitable construction method for circular tesselation patterns might also shed some light on how to generate Celtic spiral patterns. It may be interesting to note that in [Dre06], a method is suggested for generating a tiling of concentric rings of triangles, and from there spiral tilings, including the Frazer spiral. This which might work for circular Celtic patterns, as the basic idea is growing concentric rings of tiles outward from an inner circle which is subdivided in a fan-like arrangement of triangles. This process, however, cannot generate a truly circular outer border by subdivision.

In the illuminated pages of Celtic manuscripts, key patterns come with colour. Often, this just entails giving a different colour to the paths than to the background, which can be achieved just like having white paths on a black background. Just as often, however, a sophisticated colouring scheme is used based on colour blocks in rectangular or lozenge shapes. How to add colour to key patterns in such a way is an open problem, though colour operations as presented in [Dre06] might allow simulating at least a simplified version of the original Celtic colour schemes.

Finally, there are two classes of Celtic key patterns for which we have not yet devised a collage grammar-based generation technique. The first class uses squares tiles that are obtained from basic triangles by reflection at the hypotenuse. Consequently, these tiles do not have rotational symmetry with respect to path entry points at their sides, so that they have to be arranged differently to form entire patterns. The second class uses hook-like squares such as the four last squares in Figure 13, but some of the path entries may be sealed off. The reason for this can be seen in Figure 17: There are many long straight black lines that do not finish off by properly meeting with other black lines at each end. Of course, lengthening these lines at their ends requires the two neighbouring tiles which form the line end to agree. Moreover, the orientation of the tiles in hook patterns is not so uniform as in the key patterns considered in this paper. Next steps for future work may include writing collage grammars for these classes of key patterns, too.

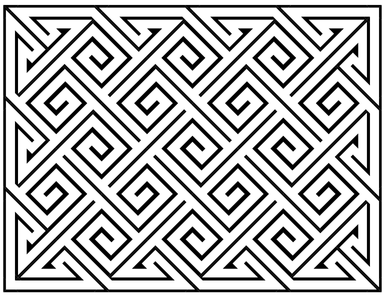


Fig. 17. Pseudo hook pattern

References

[All93] J. Romilly Allen. *Celtic Art in Pagan and Christian Times.* Bracken Books, London, 1993.

[Bai51] George Bain. *Celtic Art. The Methods of Construction.* Constable, London, 1951.

[Bai90] Iain Bain. *Celtic Knotwork.* Constable, London, 1990.

[Bai93] Iain Bain. *Celtic Key Patterns.* Constable, London, 1993.

[DEKK03] Frank Drewes, Sigrid Ewert, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Computing raster images from grid picture grammars. *Journal of Automata, Languages and Combinatorics*, 8(3):499–519, 2003.

[DK99] Frank Drewes and Hans-Jörg Kreowski. Picture generation by collage grammars. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages, and Tools*, chapter 11, pages 397–457. World Scientific, 1999.

[DK00] Frank Drewes and Renate Klempien-Hinrichs. Picking knots from trees – the syntactic structure of celtic knotwork. In *Proc. 1st Intl. Conference on Theory and Application of Diagrams 2000*, volume 1889 of *Lecture Notes in Artificial Intelligence*, pages 89–104. Springer, 2000.

[DKK03] Frank Drewes, Renate Klempien-Hinrichs, and Hans-Jörg Kreowski. Table-driven and context-sensitive collage languages. *Journal of Automata, Languages and Combinatorics*, 8(1):5–24, 2003.

[DKL03] Frank Drewes, Hans-Jörg Kreowski, and Denis Lapoire. Criteria to disprove context freeness of collage languages. *Theoretical Computer Science*, 290:1445–1458, 2003.

[Dre00] Frank Drewes. Tree-based picture generation. *Theoretical Computer Science*, 246:1–51, 2000.

[Dre06] Frank Drewes. *Grammatical Picture Generation – A Tree-Based Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[HK91] Annegret Habel and Hans-Jörg Kreowski. Collage grammars. In H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors, *Proc. 4th Intl. Workshop on Graph Grammars and Their Application to Computer Science*, volume 532 of *Lecture Notes in Computer Science*, pages 411–429, 1991.

[HKT93] Annegret Habel, Hans-Jörg Kreowski, and Stefan Taubenberger. Collages and patterns generated by hyperedge replacement. *Languages of Design*, 1:125–145, 1993.

[KRS97] Lila Kari, Grzegorz Rozenberg, and Arto Salomaa. L systems. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. I: Word, Language, Grammar*, chapter 5, pages 253–328. Springer, 1997.

[Mee02] Aidan Meehan. *Maze Patterns.* Thames & Hudson, London, 2002.

[Slo95] Andy Sloss. *How to Draw Celtic Knotwork: A Practical Handbook.* Blandford Press, 1995.

[Slo97] Andy Sloss. *How to Draw Celtic Key Patterns: A Practical Handbook.* Blandford Press, 1997.

Dr. Renate Klempien-Hinrichs
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
rena@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~rena

Renate Klempien-Hinrichs was a member of Hans-Jörg Kreowski's team from 1993 to 2009. In 2000, she received her doctoral degree from the University of Bremen under Hans-Jörg's supervision.

Caroline von Totth
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
caro@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/˜caro

Hans-Jörg Kreowski supervised Caroline von Totth's diploma thesis. Since 2004, she is a member of his group, working on her doctoral thesis, again under his supervision.

Autonomous Units for Solving the Capacitated Vehicle Routing Problem Based on Ant Colony Optimization [*]

Sabine Kuske, Melanie Luderer, and Hauke Tönnies

Abstract. Communities of autonomous units and ant colony systems have fundamental features in common. Both consists of a set of autonomously acting units that transform and move around a common environment that is usually a graph. In contrast to ant colony systems, the actions of autonomous units are specified by graph transformation rules which have a precisely defined operational semantics and can be visualized in a straighforward way. In this paper, we model an ant colony system solving the capacitated vehicle routing problem as a community of autonomous units. The presented case study shows that the main characteristics such as tour construction and pheromone update can be captured in a natural way by autonomous units. Hence, autonomous units provide a formal and visual framework for ant colony optimization algorithms.

1 Introduction

Communities of autonomous units are rule-based systems, in which the units act and interact autonomously in a common environment while striving for a goal (cf. [KK07,KK08,HKK09]). More concretely, every autonomous unit is composed of a set of graph transformation rules, a control condition, a specification of initial private states, and a goal. Moreover, it can ask auxiliary units for help. Autonomous units transform the common environment and their private states simultaneously, can communicate with each other via the common environment, and may act in parallel. A current state of an autonomous unit consists of a common environment and a private state which are both graphs. An autonomous unit specifies all state transformation processes that (1) start with an initial private state and an arbitrary common environment (2) are allowed by the control condition, and (3) can be obtained via (parallel) applications of the unit's rules, auxiliary units, and other autonomous units in the community. Hence, the semantics of a single autonomous unit includes actions of other autonomous units which are not known by the unit. This means that the semantics of an autonomous unit is loose in the sense that it is defined with respect to a set of (parallel) rules that model the actions of other units. These rules are called metarules. A state transformation process is called successful if it meets the goal.

[*] 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).

A community consists of a set of autonomous units, a specification of initial common environments, a global control condition, and an overall goal. A current state of a community is composed of a current common environment plus a private state for every autonomous unit. The semantics of a community consists of all state transformation processes performed by the autonomous units, allowed by the global control condition, and starting with an initial state. A transformation process is successful if it reaches the overall goal. The basic components of communities are provided by a graph transformation approach consisting of a class of graphs, a class of graph class expressions, a class of rules with a rule application operator, and a class of control conditions. In the literature there exists a variety of graph transformation approaches that differ mainly in the kind of rules and graphs (cf. [Roz97] for an overview on graph transformation approaches). They all can be used as underlying approach for communities.

Ant colony systems consist of a set of autonomously behaving artificial ants that move around a common graph and make their decisions according to the pheromone concentration in their neighbourhood. They are inspired by the way how ants find short routes between food and their formicary and have been shown to be well-suited not only for the solving of shortest path problems, but for a series of more complex problems, typically ocurring in logistics (cf. [DS04]). Basically, in an ant colony system, a set of ants constructs solutions for a given problem (mostly NP-hard) by moving along the edges of an underlying graph. According to the quality of the constructed solutions the ants walk back and put some pheromone on the traversed items, i.e., the better the solution is the more pheromone is placed by an ant. During solution construction the pheromone concentration as well as some further heuristic value help the ants to decide where to go in each step. Every ant has a memory for storing important information such as the length of the traversed path, etc.

In this paper, we show that ant colony systems can be modeled by communities of autonomous units in a natural way. This is illustrated with the example of the Capacitated Vehicle Routing Problem (CVRP) (cf., e.g., [RDH04,DS04]). The advantages of modeling ant colony systems as communities of autonomous units are the following. (1) Autonomous units provide ant colony systems with a well-founded operational semantics so that verification techniques for graph transformation can be applied to ant colony systems. (2) The fact that ant actions can be specified as graph transformation rules allows for a visual modeling of ant algorithms and hence for a visual representation of ant colony behaviour. (3) Existing graph transformation tools such as GrGEN [GK08] or AGG [ERT99] can be used to implement ant algorithms.

This paper is organized as follows. In Section 2, ant colony systems for the heuristic solving of optimization problems are briefly introduced and a particular ant colony optimization algorithm for solving the CVRP is recalled. Section 3 presents a particular graph transformation approach that is used throughout this paper. Section 4 introduces autonomous units and communities of autonomous units. Section 5 shows how fundamental features of ant colony systems can be modeled with autonomous units by translating an ant colony system solving the CVRP into a community. The conclusion is given in Section 6.

2 Ant Colony Opimization

Ant colony optimization (ACO) systems are algorithmic frameworks for the heuristic solving of optimization problems, typically problems belonging to the complexity class NP-hard, since no efficient algorithms for this kind of problems are known that always solve the problem. The idea of ACO originates in the observation of how ants find short ways between food and their formicary. An individual ant can hardly see and has a very narrow perspective of its environment. While searching for food, it leaves a chemical substance on the ground, called pheromone, which can be sensed by other ants and influence their route decision. The higher the concentration of pheromone along a way, the higher the probability that an ant will choose this way as well, thus leaving even more pheromone. The crucial point is that pheromone evaporates with time. An ant following a short route to food will return sooner to the formicary (returning obviously on the same way) so that the pheromone concentration on shorter routes becomes more intense than on longer routes. The higher pheromone concentration makes more ants choose the short route which in turn raises the pheromone concentration further. Finally, almost all ants end up choosing one short route, although not necessarily the shortest one. Since typical optimization problems can be nicely modeled as graphs, it is the prefered data structure for ACO. The graphs used in this paper are edge-labeled and undirected and can be defined as follows.

Definition 1 (Graphs). A *graph* is a tuple (V, E, att, m), where V is a finite set of *nodes*, E is a finite set of *edges* such that V and E are disjoint, $att : E \to \bigcup_{k\in\{1,2\}} \binom{V}{k}$ assigns to every edge a set of one or two *sources* in V, and m is a mapping that assigns a *label* to every edge in E. A graph with no nodes and no edges is called the *empty graph* which is denoted by $\emptyset$. The components of G are also denoted by V_G, E_G, respectively. The set of all graphs is denoted by $\mathcal{G}$.

A solution to an optimization problem consists typically of a tour (e.g. an ordered sequence of nodes) within the given graph. Intuitively, the complexity of most NP-hard optimization problems lies in the exponentially growing number of possible tours when new nodes and edges are added. The lack of an efficient search method for the 'best' way requires an (almost) exhaustive search of all the possible tours. To solve an optimization problem with ACO, some additional information is needed. We define optimization problems as follows.

Definition 2 (Optimization Problem). An optimization problem is a 6-tuple $(CG, d, \tau, \eta, S, g)$ where $CG \in \mathcal{G}$ is a *construction graph*, d is a function that associates every edge with a cost value (e.g. the distance), τ is a function that associates every edge with a pheromone value, η is a function that associates every edge with a number as an heuristic value for the quality of the edge, $S \subseteq V^*$ is the set of *solutions*, and g assigns a *cost* $g(s)$ to every $s \in S$.

Basically, ACO works as follows. At first, a predefined number of ants are placed randomly at some nodes. These ants decide in parallel which edge they follow in the next step according to a transition rule. Let a be an index to choose one of n ants and U_a the set of all edges that can be chosen from ant a residing at some node. The decision, which edge $e \in U_a$ to take, is probability-based. The probabilities are calculated as follows.

$$p_a(e) = \frac{[\tau(e)]^\alpha \cdot [\eta(e)]^\beta}{\sum_{e \in U_a} [\tau(e)]^\alpha \cdot [\eta(e)]^\beta} \quad \forall e \in U_a$$

In words this formula states that ants prefer edges with low cost und a high concentration of pheromone. The experimental parameters α and β control the influence of the pheromone resp. heuristic value in the decision. In every step this formula is applied, until all the ants have constructed a complete tour.

The next step concerns the pheromone values. Simulating the evaporation, the values of τ are reduced: $\tau(e) \leftarrow (1-\rho)\cdot\tau(e) \quad \forall e \in E_{CG}$ where ρ is a pheromone decay parameter in the intervall $(0, 1]$. Furthermore the release of pheromone of the ants is simulated:

$$\tau(e) \leftarrow \tau(e) + \sum_{a=1}^{n} \Delta\tau_a(e), \text{ with } \Delta\tau_a(e) = \begin{cases} \frac{1}{length(tour_a)} & , e \in tour_a \\ 0 & otherwise \end{cases}$$

where $tour_a$ is the solution constructed by ant a. In contrast to nature, the release of pheromone takes place after the ants constructed a complete tour, since the amount of pheromone corresponds to the overall quality of the tour (e.g. the length of the tour). Furthermore, in some ACO systems not every ant leaves pheromone, but just the ones having constructed the best tours.

Now the ants are placed again at some randomly chosen nodes and the algorithm starts with the modified values of pheromone. Some variants of this basic ACO yielding better performance have been proposed in the literature. Details can be found in [DS04].

2.1 Application: Capacitated Vehicle Routing Problem

An important application field of ACO concerns all kinds of tour planning with the Traveling Salesperson Problem (TSP) as the most famous one. Another problem often occurring in distribution logistics is the so called Capacitated Vehicle Routing Problem (CVRP), which can be described as follows. A number of customers must be served with some goods that are stored at a central depot. A number of vehicles with finite and equal capacity is available. The aim is to find a set of tours such that the demands of all customers are met and the total cost (the sum of the distances of the tours) is minimized. Combinatorially, a solution can be formally described as a partition of the cities into m routes $\{R_1, \ldots, R_m\}$. Each route must satisfy the condition $\sum_{j \in R_i} dem_j \leq k$, where dem_j describes the demand of the j-th customer and k is the capacity restriction of the vehicles. Within each partition, an associated permutation function specifies the customer order.

Relaxing the conditions by allowing any partition (respectively setting $k = \infty$), the CVRP is transformed into an instance of the Multiple Traveling Salesperson Problem. Leaving the condition unchanged but with a cost function that counts the number of partitions CVRP becomes the well-known bin packing problem. CVRP contains in this sense two NP-hard problems, which in practice makes it a lot more complicated to solve than TSP for example and it seems a good idea to use ACO. A formulation of CVRP according to Definition 2 is quickly found. Nevertheless, there are different ways to design the function $\eta : E_{CG} \rightarrow \mathbb{R}$. One easy possibility consists of the reciprocal cost-value of the edge.

Nevertheless, sometimes other methods are used to calculate the heuristic values; one elegant way is based on the so-called *Savings algorithm*. Starting from the initial (and unfavoured) solution, where every route consists of exactly one customer, it is calculated, how the quality of the solution changes (how much one would save), putting two customers i and j in one route. Let d_{i0} denote the distance between customer i and the depot and d_{ij} the distance between customer i and j. Then the saving value obtained by merging the routes R_i and R_j together is calculated as follows:

$$\begin{aligned} s_{ij} &= 2 * d_{i0} + 2 * d_{j0} - (d_{i0} + d_{ij} + d_{j0}) && (1) \\ &= d_{i0} + d_{j0} - d_{ij} && (2) \end{aligned}$$

Elaborated experiments concerning the performance of ACO and Saving Algorithm for the CVRP can be found in [RDH04].

3 A Graph Transformation Approach

Graph transformation approaches provide the main ingredients for communities of autonomus units. They consist of a class of graphs, a class of rules, a class of control conditions, and a class of graph class expressions. The graphs are used to represent the common environments and the private states of communities. The rules are needed to transform these graphs. Moreover, control conditions can restrict the non-determinism of rule application, and with graph class expressions one can specify specific graph sets such as initial environments or goals to be reached. In the literature, there exists a series of different graph transformation approaches (cf. [Roz97]).

In the following, we tailor a particular graph transformation approach that can be used for modeling ACO algorithms. Concretely, the rule class and the graph class are based on the double-pushout approach [CEH+97]. Additionally, we introduce a class of control conditions that is suitable for autonomous units running in parallel. These control conditions are proactive meaning that rules must always be applied as soon as possible. The class of graph class expressions allows to specify graph languages in a rule-based way.

3.1 Graphs and Rules

The graphs we use are edge-labeled and undirected as presented in Section 2. Subgraphs and graph morphisms are defined as follows.

Definition 3 (Subgraph, graph morphism). For $G, G' \in \mathcal{G}$, the graph G is a *subgraph* of G', denoted by $G \subseteq G'$, if $V_G \subseteq V_{G'}$, $E_G \subseteq E_{G'}$, $att(e) = att'(e)$, and $m(e) = m'(e)$ for all $e \in E_G$. A *graph morphism* $g\colon G \to G'$ is a pair (g_V, g_E) of mappings with $g_V\colon V_G \to V_{G'}$ and $g_E\colon E_G \to E_{G'}$ such that labels and sources are kept, i.e., for all $e \in E_G$, $g_V(att_G(e)) = att_{G'}(g_E(e))$ and $m_{G'}(g_E(e)) = m_G(e)$.[1] The image of G in G' is the subgraph $g(G)$ of G' such that $V_{g(G)} = g_V(V_G)$ and $E_{g(G)} = g_E(E_G)$.

Remark. In the following, the subscripts V and E of g_V and g_E are often omitted, i.e., $g(x)$ means $g_V(x)$ for $x \in V$ and $g_E(x)$ for $x \in E$.

Graphs are depicted as usual with round or boxed nodes and lines as edges. A loop can be omitted by putting its label inside the node to which the loop is attached. This can be done for at most one loop per node. We assume the existence of a special label *unlabeled* that is omitted in graph drawings.

Graphs can be modified by rules consisting of a negative context, a left-hand side, a gluing graph, and a right-hand side. Roughly speaking, the negative context specifies components that must not occur in the graph to which the rule is applied. The left-hand side, the gluing graph, and the right-hand side are used to determine which components should be deleted, kept and added, respectively. In every computation step of a community, the autonomous units transform the common environment and their private states simultaneously. For this purpose, every unit applies pairs of rules (r_1, r_2), where the first rule r_1 is applied to the common environment and r_2 to the private state.

Definition 4 (Rule, rule pair). A *rule* r is a quadruple (N, L, K, R) of graphs with $N \supseteq L \supseteq K \subseteq R$ where N is the *negative context*, L is the *left-hand side*, K is the *gluing graph*, and R is the *right-hand side*. If all components of r are empty, r is the *empty rule*. The set of all rules is denoted by $\mathcal{R}$. A *rule pair* is a pair of rules $r = (r_1, r_2)$ where r_1 is called the *global rule* and r_2 the *private rule*. The set of all rule pairs is denoted by $\widetilde{\mathcal{R}}$.

Remark. A rule pair $r = (r_1, r_2)$ where r_2 is the empty rule can be regarded as a single rule. Hence, in the following, we often do not distinguish between single rules and rule pairs with an empty private rule.

A rule (N, L, K, R) is depicted as $N \to R$ where the nodes and edges of K have the same forms, labels, and relative positions in N and R. The nodes of N that do not belong to L are coloured grey. The edges of N that do not belong to L are dashed. Fig. 1 shows a rule in which the negative context consists of a round node and two rectangle nodes. Each of the rectangle nodes has exactly

[1] For a mapping $f\colon A \to B$ and $C \subseteq A$ the set $f(C)$ is defined as $\{f(x) \mid x \in C\}$, i.e., $g_V(att_G(e)) = \{g_V(v) \mid v \in att_G(e)\}$.

one loop labeled with a and b, respectively. The round node is connected to both rectangle nodes. The left-hand side contains the round node, the a-node (i.e., the rectangle node with the a-loop) and the edge between both. The gluing graph consists of the round node, and the right-hand side is obtained from the gluing graph by connecting the round node with a new b-node.

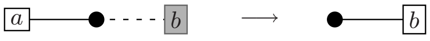


Fig. 1. A rule

A rule pair $r = ((N_1, L_1, K_1, R_1), (N_2, L_2, K_2, R_2))$ (with non-empty private rule) is depicted as $L_1|L_2 \to R_1|R_2$ where the negative contexts and the gluing graphs are represented as in single rules.

A rule (N, L, K, R) is applied to a graph as follows. (1) Choose an image $g(L)$ of L in G. (2) Check if $g(L)$ has no negative context given by N up to L. (3) Delete $g(L)$ up to $g(K)$ from G provided that no dangling edges are produced. (4) Glue R and the remaining graph in K. The construction needed in the fourth step can be defined as follows.

Definition 5 (Gluing of graphs). Let $K \subseteq R$ and $h: K \to Z$. Let $\approx_V$ be the equivalence relation on $V_Z + V_R$ generated by the relation $\{(v, h_V(v)) \mid v \in V_K\}$ and let $\approx_E$ be the equivalence relation on $E_Z + E_R$ generated by $\{(e, h_E(e)) \mid e \in E_K\}$. Let $(V_Z + V_R)/\approx_V$ and $(E_Z + E_R)/\approx_E$ be the quotient sets of the disjoint union $V_Z + V_R$ and $E_Z + E_R$, respectively. Then the *gluing* of Z and R in K with respect to h yields the graph $D = ((V_Z + V_R)/\approx_V, (E_Z + E_R)/\approx_E, att, m)$ where for all $e \in (E_Z + E_R)/\approx_E$

$$att(e) = \begin{cases} [att_Z(\overline{e})] \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_Z{}^{2} \\ [att_R(\overline{e})] \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_R - E_K \end{cases}$$

$$m(e) = \begin{cases} m_Z(\overline{e}) \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_Z \\ m_R(\overline{e}) \text{ if } e = [\overline{e}] \text{ for some } \overline{e} \in E_R - E_K \end{cases}$$

The application of a rule to a graph is formally defined as follows.

Definition 6 (Rule application). Let $r = (N, L, K, R) \in \mathcal{R}$, let $G \in \mathcal{G}$, and let $g: L \to G$ such that g is injective and the following *gluing condition* is satisfied.

- If $L \subset N$, there exists no $g': N \to G$ with $g'(x) = g(x)$ for all $x \in V_L \cup E_L$.
- For all $e \in E_G - E_{g(L)}$, $att_G(e) \subseteq V_G - (V_{g(L)} - V_{g(K)})$.

[2] For a quotient set $A/\approx$, $[]: A \to A/\approx$ denotes its natural associated function.

Then r is applied to G by (1) deleting $V_{g(L)} - V_{g(K)}$ and $E_{g(L)} - E_{g(K)}$, and (2) constructing the gluing of the resulting graph D and R in K with respect to $g|K: K \to D$ where $g|K(x) = g(x)$ for all $x \in V_K \cup E_K$. The *semantic relation of* r is denoted by $SEM(r)$ and consists of all pairs (G, G') such that G' can be derived from G via the application of r. For a set $P \subseteq \mathcal{R}$, we define $SEM(P) = \bigcup_{r \in P} SEM(r)$. For $(r_1, r_2) \in \widetilde{\mathcal{R}}$, the semantic relation is equal to $\{((G_1, G_2), (G_1', G_2')) \mid (G_i, G_i') \in SEM(r_i), i = 1, 2\}$.

Remark. The described kind of applying graph transformation rules corresponds to the double-pushout approach presented in e.g. [CEH+97], where also non-injective matchings of the left-hand side are allowed.

The rule in Fig. 1 can be applied to a graph containing a node v connected to an a-node but not connected to a b-node. Its application removes the a-node plus the edge to v and adds a b-node and an edge from this b-node to v. Because of the gluing condition, the a-node is only connected to v but not to other nodes; otherwise its deletion would produce dangling edges.

In general, the autonomous units of a community apply their rules in parallel. A parallel rule application step involving two rules can be defined as follows.

Definition 7 (Parallel rule application). Let $G \in \mathcal{G}$ and for $i = 1, 2$, let $r_i = (N_i, L_i, K_i, R_i)$ be two rules. Let $g_i: L_i \to G$ be two injective graph morphisms that satisfy the gluing condition of Definition 6 and the *independence condition* $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$.[3] Then r_1 and r_2 can be applied in parallel to G by (1) deleting $V_{g_i(L)} - V_{g_i(K)}$ and $E_{g_i(L)} - E_{g_i(K)}$ (for $i = 1, 2$), and (2) constructing the gluing of the resulting graph D and $R_1 + R_2$ in $K_1 + K_2$ with respect to $g: K_1 + K_2 \to D$, where $g(x) = g_i(x)$ if $x \in V_{K_i} \cup E_{K_i}$, for $i = 1, 2$.[4]

Remarks.

1. Definition 7 can be extended in a straightforward way from two rules to arbitrary non-empty multisets of rules. For a multiset m of rules, $SEM(m)$ denotes the set of all $(G, G') \in \mathcal{G} \times \mathcal{G}$ where G' is derived from G via the parallel application of the rules in m. A multiset m of rules will be called a *parallel rule*, and for a set $P \subseteq \mathcal{R}$, the set of all parallel rules over P is denoted by P_*.
2. For a rule pair $r = (r_1, r_2)$, $SEM(r||m)$ denotes all $((G_1, G_2), (G_1', G_2')) \in (\mathcal{G} \times \mathcal{G}) \times (\mathcal{G} \times \mathcal{G})$ where G_1' is derived from G_1 by applying the multiset obtained from adding r_1 to m, and $(G_2, G_2') \in SEM(r_2)$.

3.2 Control Conditions

It is often desirable to restrict the non-determinism of rule application. This can be achieved with control conditions. Concretely, we use as control conditions

[3] For $G_1, G_2 \in \mathcal{G}$ the intersection $G_1 \cap G_2$ yields the pair (V, E) where $V = V_{G_1} \cap V_{G_2}$ and $E = E_{G_1} \cap E_{G_2}$. Moreover, we have $(V_1, E_1) \subseteq (V_2, E_2)$ if $V_1 \subseteq V_2$ and $E_1 \subseteq E_2$.

[4] The morphism g may be non-injective.

regular expressions equipped with *as long as possible* and the parallel operator $||$.

Definition 8 (Control conditions). Let ID be a set such that $P \subseteq ID$ for some set P of rule pairs. Then the class $\mathcal{C}(ID)$ of *control conditions* over ID is inductively defined as follows.

1. $\{lambda\} \cup ID \cup \{x! \mid x \in P\} \subseteq \mathcal{C}(ID)$.
2. For $c, c_1, c_2 \in \mathcal{C}(ID)$, we have $(c_1 + c_2), (c_1\,; c_2), (c^*), (c_1||c_2) \in \mathcal{C}(ID)$.

Remark. For practical applications, the set ID would consist of names refering to rule pairs (or units) but for technical simplicity we do not distinguish between rule pairs (units) and their names.

If ID consists only of rule pairs, a semantics of control conditions can be defined in an intuitive way. Roughly speaking, the condition *lambda* applies no rule. Every rule pair r is a control condition that prescribes one application of r. The condition $c_1 + c_2$ stands for applying c_1 or c_2, $c_1\,; c_2$ means that c_1 must be applied before c_2, c^* applies c arbitrarily often, $r!$ requires that the pair r be applied as long as possible, and $c_1||c_2$ allows only transformations where c_1 and c_2 are applied in parallel. For example, the expression $r_1; r_2^* + r_3!$ allows all sequences in which r_1 is applied before an arbitrarily often application of r_2 or in which r_3 is applied whenever this is possible.[5]

As stated before, the application of a rule pair by an autonomous unit *aut* is generally done in parallel with transformations of the common environment executed by other autonomous units. Moreover, it may also happen that other units perform actions before or after rule applications of *aut*. When defining the semantics of control conditions the rules of other units are not known. Hence, the semantics is loose, i.e., it is defined with respect to a set $\mathcal{MR}$ of parallel rules called *metarules*. For modeling ant-based systems in a suitable way, we require additionally that rules be applied as early as possible. This leads to proactive transformation processes. In more detail, for a set P of rule pairs, every proactive transformation process s specified by a control condition $c \in \mathcal{C}(P)$ must have the following property: If in s an application of a rule pair $r \in P$ is preceded by a sequence of k mere metarule applications, the application of r cannot be shifted to any of the k preceding steps. For defining a proactive semantics of control conditions, i.e., a semantics that consists only of proactive transformation processes, we also have to define proactive transformation processes in which no metarules are applied after the last application of a rule in P. For example, for $i = 1, 2$, let $SEM_{\mathcal{MR}}(r_i)$ be the proactive transformation processes specified by the rule r_i. In order to get all proactive transformation processes specified by $r_1\,; r_2$, we cannot take the sequential composition of the processes in $SEM_{\mathcal{MR}}(r_1)$ and $SEM_{\mathcal{MR}}(r_2)$, because of the following reason. Let s_1 be a transformation process in $SEM_{\mathcal{MR}}(r_1)$ in which some metarules are applied after r_1. Let s_2 be a transformation process in $SEM_{\mathcal{MR}}(r_2)$. Then the sequential composition

[5] The operator * has a stronger binding than ; which in turn has a stronger binding than $+$.

$s_1 \circ s_2$ is not proactive if r_2 is applicable after the application of r_1 in s_1 but before the end of s_1 because in this case the application of r_2 can be shifted to an earlier step. Hence, before sequentially composing s_1 and s_2 we have to cut all metarule applications from s_1 that take place after the application of r_1.

In the following, a proactive semantics of control conditions is defined for the case where *ID* consists of rules, only. In Section 4 we show how this definition can be employed for the more general case where *ID* contains units, too.

Definition 9 (Proactive semantics of control conditions). Let $\mathcal{MR} \subseteq \mathcal{R}_*$ be a set of parallel rules called *metarules* and let $P \subseteq \widetilde{\mathcal{R}}$. Then for each control condition in $\mathcal{C}(P)$ its *proactive semantics* is defined as follows.

1. $SEM_{\mathcal{MR}}(lambda)$ consists of all sequences $(G_0, \ldots, G_n)$ of graph pairs such that for $i = 1, \ldots, n$, $(G_{i-1}, G_i) \in SEM(m)$ for some $m \in \mathcal{MR}$.[6] Moreover, we define $CUT(SEM_{\mathcal{MR}}(lambda)) = \mathcal{G} \times \mathcal{G}$.
2. $SEM_{\mathcal{MR}}(r)$ consists of all sequences $s = (G_0, \ldots, G_n)$ for which there exist some $j \in \{1, \ldots, n\}$ and $m_1, \ldots, m_n \in \mathcal{MR}$ such that for $i = 1, \ldots, j-1$ and $i = j+1, \ldots, n$, $(G_{i-1}, G_i) \in SEM(m_i)$, $(G_{j-1}, G_j) \in SEM(r||m_j)$ and for $i = 0, \ldots, j-1$, there is no $G \in \mathcal{G} \times \mathcal{G}$ such that $(G_i, G) \in SEM(r||m_i)$. Moreover, we define $cut(s) = \text{TRUE}$ iff $j = n$ and $CUT(SEM_{\mathcal{MR}}(r)) = \{s \in SEM_{\mathcal{MR}}(r) \mid cut(s) = \text{TRUE}\}$.
3. $SEM_{\mathcal{MR}}(c_1 + c_2) = SEM_{\mathcal{MR}}(c_1) \cup SEM_{\mathcal{MR}}(c_2)$ and
$$CUT(SEM_{\mathcal{MR}}(c_1 + c_2)) = CUT(SEM_{\mathcal{MR}}(c_1)) \cup CUT(SEM_{\mathcal{MR}}(c_2)).$$
4. $SEM_{\mathcal{MR}}(c_1 \,; c_2) = CUT(SEM_{\mathcal{MR}}(c_1)) \circ SEM_{\mathcal{MR}}(c_2)$ and
$$CUT(SEM_{\mathcal{MR}}(c_1 \,; c_2)) = CUT(SEM_{\mathcal{MR}}(c_1)) \circ CUT(SEM_{\mathcal{MR}}(c_2)).$$[7]
5. $SEM_{\mathcal{MR}}(c^*) = SEM_{\mathcal{MR}}(lambda) \cup CUT(SEM_{\mathcal{MR}}(c))^* \circ SEM_{\mathcal{MR}}(c)$. Moreover, $CUT(SEM_{\mathcal{MR}}(c^*)) = (CUT(SEM_{\mathcal{MR}}(c)))^*$.
6. $SEM_{\mathcal{MR}}(r!) = CUT(SEM_{\mathcal{MR}}(r^*)) \circ \{(G_0, \ldots, G_k) \in SEM_{\mathcal{MR}}(lambda) \mid G_i \in red(r) \text{ for } i = 1, \ldots, k\}$ where for $(r_1, r_2) \in \widetilde{\mathcal{R}}$, $red(r_1, r_2)$ consists of all $(G_1, G_2) \in \mathcal{G} \times \mathcal{G}$ such that r_1 is not applicable to G_1 or r_2 is not applicable to G_2. Moreover, $CUT(SEM_{\mathcal{MR}}(r!)) = \{(G_0, \ldots, G_k) \in CUT(SEM_{\mathcal{MR}}(r^*)) \mid G_k \in red(r)\}$.
7. $SEM_{\mathcal{MR}}(c_1||c_2) = SEM_{\mathcal{MR} \cup Rules(c_2)_*}(c_1) \cap SEM_{\mathcal{MR} \cup Rules(c_1)_*}(c_2)$ where for $i = 1, 2$, $Rules(c_i)$ is the set of all rule pairs occurring in c_i. Moreover, $CUT(SEM_{\mathcal{MR}}(c_1||c_2)) = SEM_{\mathcal{MR}}(c_1||c_2) \cap (CUT(SEM_{\mathcal{MR} \cup Rules(c_2)_*}(c_1)) \cup CUT(SEM_{\mathcal{MR} \cup Rules(c_1)_*}(c_2)))$.

Remark. In the above definition $SEM_{\mathcal{MR}}(c)$ is the set of proactive transformation processes specified by c whereas $CUT(SEM_{\mathcal{MR}}(c))$ is needed for defining the proactive semantics of control conditions involving sequential composition.

[6] In this transformation, the second component of every graph pair remains unchanged, because m is a multiset of single rules.

[7] For sets of sequences S, S', their sequential composition is denoted by $S \circ S'$, and S^* is defined as $\bigcup_{i \in \mathbb{N}} S^i$ with $S^0 = \mathcal{G} \times \mathcal{G}$ and $S^{i+1} = S^i \circ S$.

3.3 Graph Class Expressions

In order to use graph transformation in a meaningful way, it should be possible to specify initial and terminal graphs of graph transformation processes with graph class expressions. In general, a graph class expression can be any expression that specifies a set of graphs. In particular, the graph class expressions used in this paper are the following.

Definition 10 (Graph class expressions). The class $\mathcal{X}$ of all graph class expressions is defined as follows.

1. $all, empty, red(P), (P, C) \in \mathcal{X}$ with $P \subseteq \mathcal{R}$ and $C \in \mathcal{C}(P)$ where $SEM(all) = \mathcal{G}$, $SEM(empty) = \emptyset$, $SEM(red(P))$ consists of all graphs G to which no rule of P can be applied, and $SEM(P, C)$ consists of all graphs G for which there is a sequence $(G_0, \ldots, G_n)$ such that $G_n = G$, for $i = 1, \ldots, n$ $(G_{i-1}, G_i) \in SEM(P)$ and $(G_0, \ldots, G_n) \in SEM_{\emptyset}(C)$.[8]
2. For $I, T \in \mathcal{X}$, $P \subseteq \mathcal{R}$, and $C \in \mathcal{C}$, $(I, P, C, T) \in \mathcal{X}$ where $SEM(I, P, C, T) = SEM(P, C) \cap (SEM(I) \times SEM(T))$.

One example of graph class expressions of the second type is $complete = (empty, \{nodes, edges\}, nodes^*; edges^*, red(\{edges\}))$, where *nodes* and *edges* are the rules in Fig. 2.

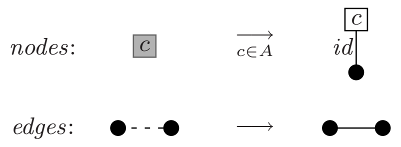


Fig. 2. The rules *nodes* and *edges*

Given some alphabet A, the expression *complete* specifies all complete graphs composed of round nodes in which every round node is additionally connected to exactly one uniquely labeled boxed node via an *id*-edge. The labels of the boxed nodes are taken from A. It is worth noting that the rule *edges* cannot produce loops because we only use injective morphisms to choose a matching of the left-hand side. In addition, we technically distinguish between round and boxed nodes by using particularly labeled loops that indicate the respective node type (*round* or *boxed*).

4 Communities of Autonomous Units

Every community is mainly composed of a set of autonomous units that act and interact in a common environment (see e.g. [HKK09] where a sequential and a parallel semantics of communities is introduced).

[8] Control conditions can be used to define sequences of graphs (instead of sequences of graph pairs) because, as stated before, rules can be regarded as rule pairs with empty private component.

4.1 Autonomous Units

Autonomous units transform a common graph and have an additional private graph where they can store private information. Since the rule set of an autonomous unit can be very large, structuring concepts should be provided to keep it manageable. Autonomous units allow to import auxiliary units and provide control conditions as well as graph class expressions. Auxiliary units differ from autonomous units in the sense that they do not contain graph class expressions. The graph class expressions of every autonomous unit are used to specify the initial private states as well as the goal. The latter consists of a private goal concerning the private state and a goal concerning the common environment that the autonomous unit wants to reach.

Definition 11 (Autonomous units).

1. A *unit* of *import depth* 0 is a system $unit = (I, U, P, C, g)$ where $I \in \mathcal{X}$ is the *initial private graph class expression*, $U = \emptyset$ is the empty set, $P \subseteq \widetilde{\mathcal{R}}$ is a set of rule pairs, $C \in \mathcal{C}(P \cup U)$ is a control condition, and $g \in \mathcal{X} \times \mathcal{X}$ is the *goal*.
2. A *unit* of *import depth* $n+1$ ($n \in \mathbb{N}$) is a system $unit = (I, U, P, C, g)$ where U is a set of units of import depth at most n, and I, P, C, and g are defined as in point 1.
3. (I, U, P, C, g) is an *auxiliary unit* if $I = all$, $g = (all, all)$, and every $u \in U$ is an auxiliary unit.
4. (I, U, P, C, g) is an *autonomous unit* if every $u \in U$ is an auxiliary unit. The set of autonomous units is denoted by AUT
5. The components of *unit* are also denoted by I_{unit}, U_{unit}, P_{unit}, C_{unit}, and g_{unit}, respectively.

Every unit can be converted into a flattened unit with import depth zero. The rule set and the control condition of the flattened unit can be constructed as follows.

Definition 12 (Flattening). For $unit = (I, U, P, C, g)$ its *flattened rule set* $Rules(unit)$ and its *flattened control condition* $flC(unit)$ is defined as follows. If $U = \emptyset$, $Rules(unit) = P$ and $flC(unit) = C$. If $U \neq \emptyset$, $Rules(unit) = P \cup \bigcup_{u \in U} Rules(u)$ and $flC(unit) = C[a]$ where $a{:}\, U \to \mathcal{C}(\widetilde{\mathcal{R}})$ is defined as $a(u) = flC(u)$.[9]

The parallel semantics of autonomous units consists of all transformation sequences that start with a pair consisting of an initial private graph and an arbitrary common environment and that are allowed by the flattened control condition. Like for control conditions, we assume the existence of a set of metarules specifying the common environment transformations that can be performed by other units. If the transformation reaches the goal, it is called successful.

[9] For a control condition c and a mapping $a{:}\, U \in \mathcal{C}$, $C[a]$ is obtained by replacing every occurrence of u with $a(u)$, for all $u \in U$.

Definition 13 (Parallel semantics). Let $aut = (I, U, P, C, (g_1, g_2))$ be an autonomous unit, let $\mathcal{MR} \subseteq \mathcal{R}_*$, and let $s = ((G_0, G_0'), \ldots, (G_n, G_n'))$ be a sequence of graph pairs. Then $s \in PAR_{\mathcal{MR}}(aut)$ if $G_0' \in SEM(I)$ and $s \in SEM_{\mathcal{MR}}(\mathit{flC}(aut))$. Moreover, s is *successful* if $(G_n, G_n') \in SEM(g_1) \times SEM(g_2)$.

Remark. In general, the transformation processes of autonomous units may also be infinite which is appopiate to describe infinite processes and in particular to investigate convergence behavior of ant-based systems. However, in this first approach we consider only the finite case, but an extension to the infinite case is straightforward (cf. [HKK09]).

A community consists of a set of autonomous units, a specification of all possible initial environments, a global control condition, and an overall goal. In the following, global control conditions are regular expressions equipped with the parallel operator $||$.

Definition 14 (Global control conditions). Let $Aut \subseteq AUT$. Then the set of *global control conditions* $\mathcal{GLC}(Aut)$ is recursively defined as follows.

1. $Aut \cup \{aut_1 || \cdots || aut_k \mid aut_i \in Aut, i = 1, \ldots, k\} \subseteq \mathcal{GLC}(Aut)$
2. For $c, c_1, c_2 \in \mathcal{GLC}(Aut)$, we have $(c_1 + c_2), (c_1 \,; c_2)$, $(c^*) \in \mathcal{GLC}(Aut)$.

Global control conditions specify sequences of states where every state consists of a common environment plus a private state for every autonomous unit in a community. Roughly speaking, the global control condition aut specifies all transformation processes of aut where the private states of all other units are not changed. The global control condition $aut_1 || \cdots || aut_k$ prescribes the parallel running of $aut_1, \ldots, aut_k$. The semantics of the remaining control conditions are defined as expected. In the following we define *Aut*-states and the semantics of global control conditions.

Definition 15 (Aut-states and semantics of global control conditions). For $Aut \subseteq AUT$, an *Aut-state* is a pair (G, map) where $G \in \mathcal{G}$ and $map\colon Aut \to \mathcal{G}$ is a mapping. The *semantics* of each global control condition in $\mathcal{GLC}(Aut)$ is defined as follows.

1. $SEM_{Aut}(aut)$ consists of all sequences $((G_0, map_0), \ldots, (G_n, map_n))$ of *Aut*-states such that $((G_0, map_0(aut)), \ldots, (G_n, map_n(aut))) \in SEM_{\emptyset}(\mathit{flC}(aut))$, and for each $aut' \in Aut - \{aut\}$, $map_0(aut') = \cdots = map_n(aut')$.
2. $SEM_{Aut}(aut_1 || \cdots || aut_k)$ consists of all $((G_0, map_0), \ldots, (G_n, map_n))$ such that for $i = 1, \ldots, k$,

$$((G_0, map_0(aut_i)), \ldots, (G_n, map_n(aut_i))) \in SEM_{\mathcal{MR}(aut_i)}(\mathit{flC}(aut_i)),$$

   where $\mathcal{MR}(aut_i) = (\bigcup_{aut \in \{aut_1, \ldots, aut_k\} - \{aut_i\}} Rules(aut))_*$, and for each $aut \in Aut - \{aut_1, \ldots, aut_k\}$, $map_0(aut) = \cdots = map_n(aut)$.
3. $SEM_{Aut}(c_1 + c_2) = SEM_{Aut}(c_1) \cup SEM_{Aut}(c_2)$,
4. $SEM_{Aut}(c_1 \,; c_2) = SEM_{Aut}(c_1) \circ SEM_{Aut}(c_2)$, and
5. $SEM_{Aut}(c^*) = SEM_{Aut}(c)^*$.

The components of communities are given in the following definition.

Definition 16 (Community). A *community* is a tuple $(\mathit{Init}, \mathit{Aut}, \mathit{Cond}, \mathit{Goal})$ where $\mathit{Init}, \mathit{Goal} \in \mathcal{X}$, $\mathit{Aut} \subseteq AUT$, and $\mathit{Cond} \in \mathcal{GLC}(\mathit{Aut})$.

The parallel semantics of a community consists of all state sequences that are allowed by the global control condition and start with an initial state consisting of an initial common environment and an initial private state for each autonomous unit. The state sequences are successful if they reach the overall goal.

Definition 17 (Parallel community semantics). Let

$$\mathit{COM} = (\mathit{Init}, \mathit{Aut}, \mathit{Cond}, \mathit{Goal})$$

be a community. Then the *parallel community semantics* of COM, denoted by $PAR(\mathit{COM})$ consists of all Aut-state sequences $s = ((G_0, map_0), \ldots, (G_n, map_n))$ such that $G_0 \in \mathit{SEM}(\mathit{Init})$, $map_0(aut) \in \mathit{SEM}(I_{aut})$ (for each $aut \in \mathit{Aut}$), and $s \in \mathit{SEM}_{\mathit{Aut}}(\mathit{Cond})$. Moreover, s is *successful* if $G_n \in \mathit{SEM}(\mathit{Goal})$.

5 An ACO Community for Solving the CVRP

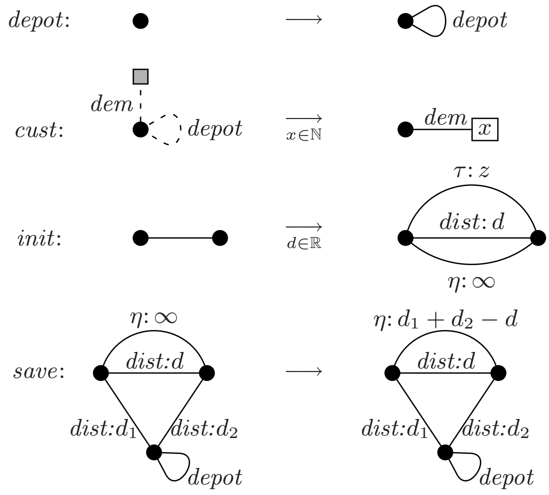
In this section we present the components of the ACO community COM_{CVRP} for modeling the Capacitated Vehicle Routing Problem (CVRP) introduced in Section 2. The initial environment specification of COM_{CVRP} specifies the construction graph of the problem; the set of autonomous units consists of the autonomous units $\mathit{Ant}_1, \ldots, \mathit{Ant}_k$ ($k \in \mathbb{N}$), and *Evap&Select*; and the global control condition *Cond* is equal to $(\mathit{Ant}_1 || \ldots || \mathit{Ant}_k || \mathit{Evap\&Select})^*$. In our first approach the overall goal is equal to *all*.

Roughly speaking, the community COM_{CVRP} works as follows. The ant units $\mathit{Ant}_1 \ldots \mathit{Ant}_k$ model the ants, which in parallel traverse the graph according to the savings heuristics introduced in Section 2 and the current pheromone trails, and search for a solution for the CVRP. When all ants have finished their search, the autonomous unit *Evap&Select* first carries out evaporation of the current pheromone trails. After that it selects the w best solutions. Now each ant which provides one of the best solutions leaves a pheromone trail on its solution path according to the quality of the solution. All the units act in parallel. To ensure the described order we use negative application conditions as well as control conditions.

5.1 The Initial Environment

The underlying structure of the construction graph of the ACO system modeling the CVRP is a complete graph with some additional information such as initial pheromone concentration, distances, etc. Therefore the construction graph for the CVRP can be defined by the graph class expression depicted in Fig. 3. It uses as initial expression the graph class expression *complete* introduced in 3.3.

Construction_graph
initial: *complete*
rules:



conds: *depot* ; (*cust* + *init* + *save*)*
goal: $red(\{init, save, cust\})$

Fig. 3. The graph class expression *Construction_graph*

Its rule *depot* selects the depot and has to be applied exactly once. The rule *cust* adds a number representing the demand to every customer node, i.e., to every node apart from the depot. The rule *init* labels every edge e of the initial graph with a *distance* d and it inserts two edges between each two nodes of the graph, one labeled with the *heuristic value* ∞ the other with an *initial pheromone value* z. The rule *save* computes the heuristic value of every edge based on the savings heuristics. The control condition requires that the depot is selected first. The terminal graph class expression $red(\{init, save, cust\})$ guarantees that the rules *cust*, *init*, and *save* are applied as long as possible.

5.2 The Ant Units

In general, every ant builds a solution tour by traversing the common environment according to the current pheromone trails. It first selects its initial position. Afterwards, it constructs a solution tour t. Then it puts some pheromone on t if it is selected to do so. Every ant unit Ant_j uses the auxiliary units $tour_j$, and put_phero_j. The control condition is equal to $initial\_position_j$; $tour_j$; put_phero_j where $initial\_position_j$ is the rule depicted in Fig. 4. It puts the ant Ant_j to the depot and generates its memory M_j where it stores the current load of the vehicle represented by Ant_j (*load*), the capacity of the vehicle (*cap*), its current location (*sit*) and the total length of the tours (*len*). This information is represented by edges labeled with the respective labels (*load*, *cap*, *sit* and *len*), which are each attached to a node labeled with the corresponding value.

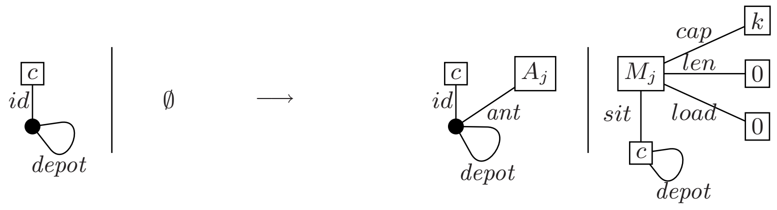


Fig. 4. The rule *initial_position*$_j$

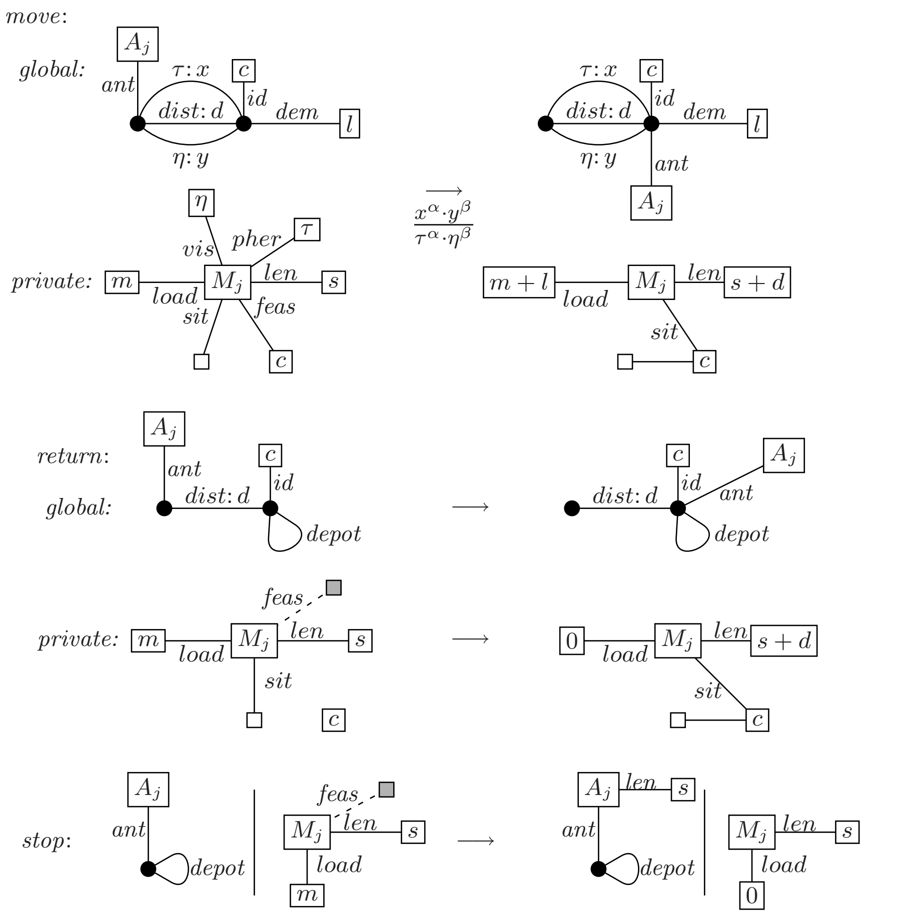
The unit *tour*$_j$ is given in Fig. 5. The global and private parts of the unit's rules are depicted one below the other. With *tour*$_j$ the ant builds a solution tour depending on probabilities for the next move to a feasible neighbour calculated from the savings heuristics and the current pheromone trails. It contains the auxiliary units *feasible_neighbours*$_j$ and *prob*$_j$, and the rules *move*, *return* and *stop*. The control condition requires to apply the unit *feasible_neighbours*$_j$ first. This unit is given is given in Fig. 6. It computes the feasible neighbours for an ant unit *Ant*$_j$ and stores them in the memory of the ant. Feasible neighbours are customer-nodes that are not yet visited and whose demand still fits into the vehicle. Every application of the only rule *feas* adds one feasible neighbour to the memory. Moreover, it uses the auxiliary unit *delete_nonfeasible* that removes all neighbours from the memory that are connected via a *feas*-edge to M_j and whose demand exceeds the remaining capacity of the vehicle.[10] This is necessary because after adding a feasible customer to a tour, the former feasible neighbours may not fit into the vehicle anymore. For reasons of space limitations a drawing of *delete_nonfeasible* is omitted.

The unit *prob*$_j$ is given in Fig. 7. It provides some of the values that are needed by the unit *tour*$_j$ for computing the probability that a feasible neighbour is chosen for a next move. This is done by summing up the pheromone and the heuristic values of all feasible neighbours. The rule *begin* initializes these values with 0. The rule *sum* must be applied as long as possible. For not counting a feasible neighbour several times *sum* changes each label *feas* into *ok*. At the end the unit *relabel_all_private*$_j$ *(ok,feas)* is applied which undoes this relabeling, i.e., it changes all *ok*-edges into *feas*-edges. It is very simple and hence not depicted.

With the rule *move* the ant moves to a feasible neighbour with the probability depicted under the arrow of the rule *move* in Fig. 5. Moreover, in the memory the current load of the vehicle, the path followed so far, and the total length of the tour are updated. With the rule *return* the ant returns to the depot if no feasible neighbour is left and resets its current load to 0. Afterwards it starts to construct a new subtour. Finally, when all nodes are visited, the rule *stop*

[10] We assume that the demand of each customer fits into one vehicle.

tour$_j$
uses: *feasible_neighbours$_j$*, *prob$_j$*
rules:



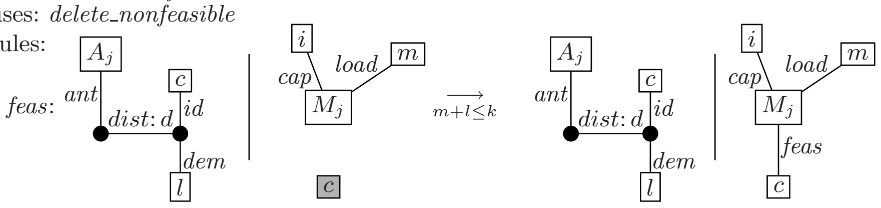
conds: (*feasible_neighbours$_j$* ; (*prob$_j$* ; *move* + *return*))* ; *feasible_neighbours$_j$* ; *stop*

Fig. 5. The auxiliary unit *tour$_j$*

is applied to reset the current load in the memory of the ant to 0. The rule *stop* also it adds the information about the length of the found solution to the common environment by inserting an edge labeled with *len* from the ant-node A_j to a new node labeled with the length of the solution.

The unit *put_phero$_j$* is depicted in Fig. 8. It works a little different for ants, who should leave a pheromone trail and those who should not. Both kinds of ants apply different rules, but the structure of rule applications is the same. In both

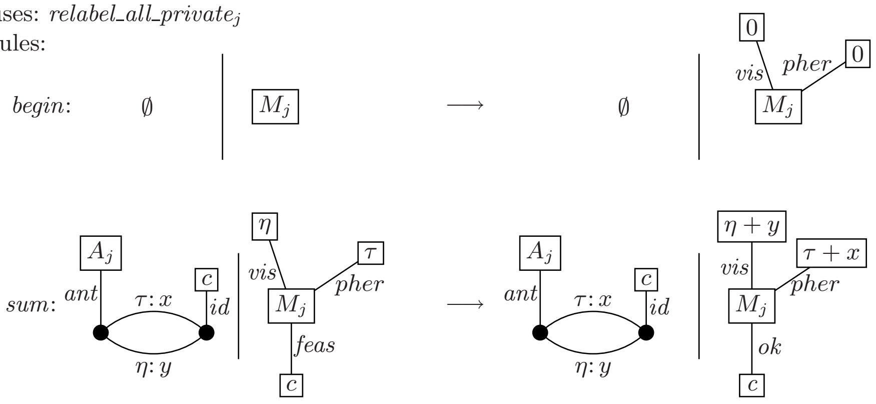
feasible_neighbours$_j$
uses: *delete_nonfeasible*
rules:



conds: *delete_nonfeasible* ; *feas*!

Fig. 6. The auxiliary unit *feasible_neighbours*$_j$

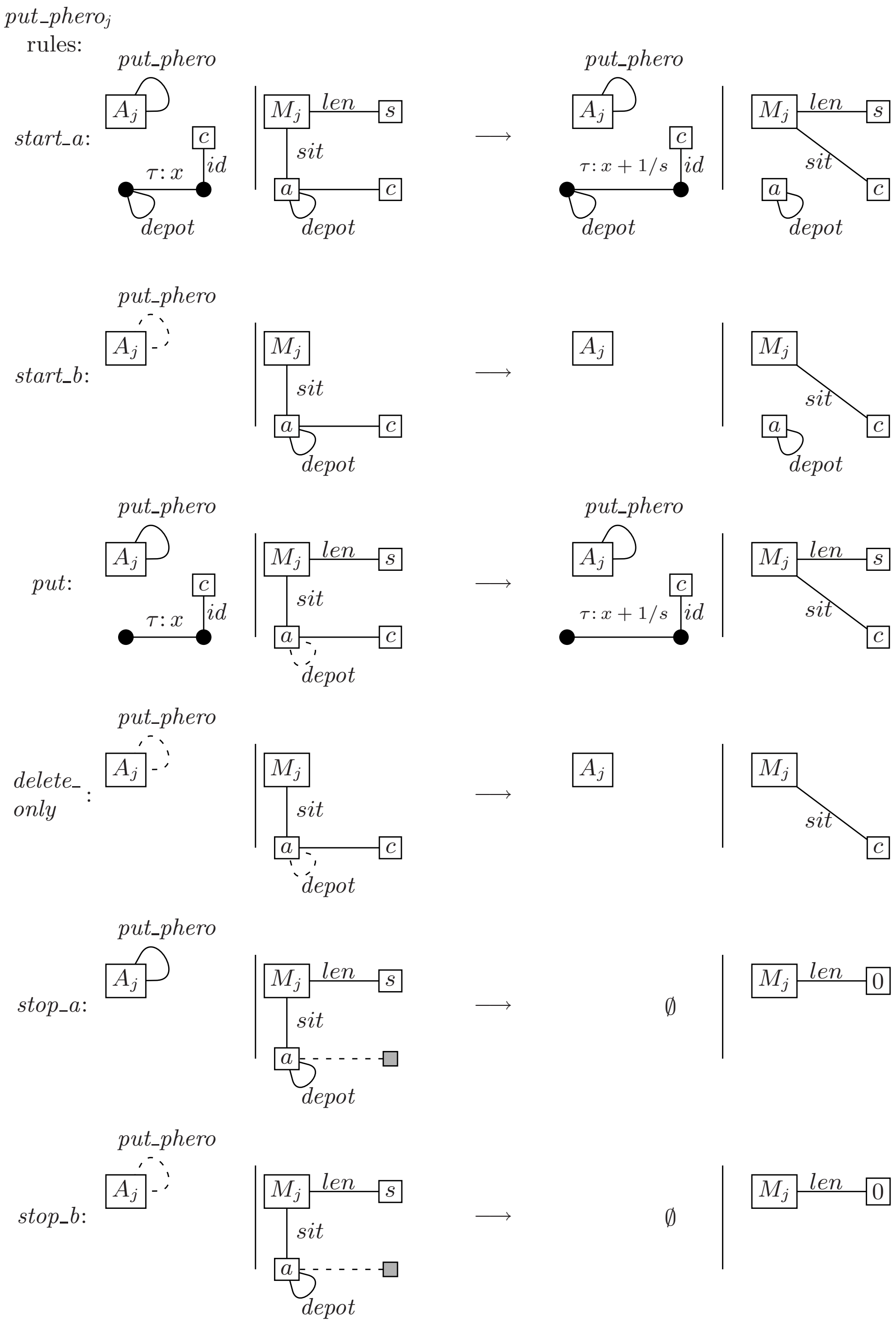
prob$_j$
uses: *relabel_all_private*$_j$
rules:



conds: *begin* ; *sum*! ; *relabel_all_private*$_j$ *(ok,feas)*

Fig. 7. The auxiliary unit *prob*$_j$

cases the ant traverses the solution path stored in its memory and meanwhile deletes it. (Because the path stored in the memory is shaped like a blossom with the depot in the middle, first the "petals" (subtours) are deleted and finally the depot.) This behaviour is represented by the rules *start_a* (resp. *start_b*) and *put* (resp. *delete_only*) and the subexpression of the control condition ((*start_a* + *start_b*) ; (*put*! + *delete_only*!))*. One application of a *start*-rule followed by applications of the rule *put* (resp. *delete_only*) as long as possible traverses one subtour of the found tour beginning and ending at the depot. The rules delete the traversed path from the memory (leaving the depot); *put* additionally leaves a pheromone trail in the common environment with the value $1/s$, where s is the length of the solution tour. Afterwards the remaining subtours are traversed until no further subtour is left in the memory. Then the respective *stop*-rule can be applied, which deletes the ant A_j from the common environment, the depot from the memory and resets the length of the traversed path to 0.



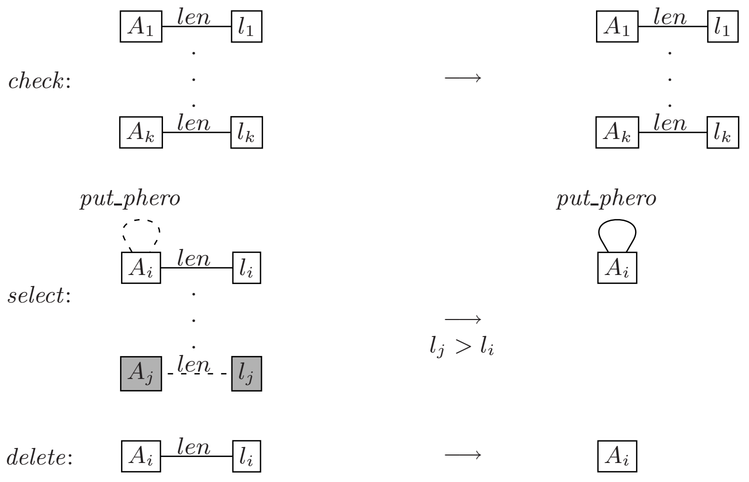
conds: $((start\_a + start\_b)\ ;\ (put! + delete\_only!))^*\ ;\ (stop\_a + stop\_b)$

Fig. 8. The auxiliary unit *put_phero*$_j$

5.3 The Unit *Evap&Select*

Evap&Select is given in Fig. 9. It is responsible for the evaporation of old pheromone trails, for the selection of the w best solutions provided by the ants, and for marking these w ants with a *put_phero*-label.

Evap&Select
uses: *relabel_all_global*
rules:



conds: *check* ; *relabel_all_global*($\tau\colon z, \tau\colon (1-\rho) * z$) ; $select^w$; *delete*!

Fig. 9. The autonomous unit *Evap&Select*

With the rule *check*, which is applied only once, the unit checks whether all ants have finished their search. This is the case if all ants have written the length of the found solution into the common environment. With the help of the unit *relabel_all_global* evaporation takes place by multiplying the pheromone value of every pheromone edge in the common environment with $(1-\rho)$, where $\rho \in (0,1]$ is a pheromone decay parameter. After that, the rule *select* is applied w times (in the control condition this is abbreviated by $select^w$). The rule *select* finds the ant with the best solution, marks it with a label *put_phero*, and deletes the information about the length of the ant's solution from the common environment. Each further application of *select* finds the next best solution. When the w best solutions are found, the rule *delete* is applied as long as possible to delete the remaining nodes and edges displaying the information about the lengths of the ants' solutions. This rank-based approach could be extended by the *elitist strategy* (see e.g. [DS04]). In this strategy the best solution so far is memorized and when pheromone update takes place, this tour gets additional pheromone.

(In our modeling of the CVRP, we do not consider this strategy because of space limitations.)

Remark. The presented modelization can be used to prove correctness properties a few of which are informally described here.

– In every transformation sequence of $tour_j$ a solution is constructed, i.e., a set of cycles of the construction graph is traversed by Ant_j and stored in its memory such that the depot belongs to every cycle, and every customer occurs exactly once in exactly one cycle.
– The unit put_phero_j deletes the constructed solution from the memory of Ant_j and increases the pheromone value of each edge in the solution by $\frac{1}{s}$ where s is the length of the solution.
– The unit feasible neighbours stores all nodes in the memory of Ant_j that are not visited, yet.
– Each execution of $(Ant_1||\dots||Ant_k||Evap\&Select)$ models an iteration of the corresponding ACO-Algorithm, i.e., (1) solution construction, (2) pheromone update, and (3) evaporation.

6 Conclusion

In this paper, we have modeled an ACO algorithm for the Capacitated Vehicle Routing Problem as a community of autonomous units. The autonomous behavior of every ant has been modeled as an autonomous unit, and global features of ACO algorithms such as the construction graph or the order in which solution construction, pheromone update, and evaporation take place have been modeled with global components of communities such as the initial environment specification or the global control condition. Since all ACO algorithms basically work according to the same underlying algorithm, we believe that they all can be modeled as communities of autonomous units in a natural way.

For solving ACO algorithms in a proper way, we have extended the parallel working autonomous units of [HKK09] by auxiliary units that allow to encapsulate auxiliary tasks in separate units and to manage large rule sets. We also have added a separate state for every autonomous unit in order to represent memories of ants. Furthermore, we have defined the syntax and a proactive semantics of a concrete class of control conditions that is adequate for units running in parallel. This class consists of regular expressions extended by a parallelism operator and an operator that prescribes to apply a rule as long as possible. We have given a construction that flattens the (hierarchical) import structure and the control conditions of autonomous units so that the parallel semantics of [HKK09] could be used for the extended units.

The modeling of ACO systems as communities of autonomous units has the following advantages. (1) The specification of ants as autonomous units provides the ants with a well-defined operational semantics. (2) The graph transformation rules of autonomous units allow for a visual specification of ant behavior instead of string-based pseudo code as it is often used in the literature. (3) The existing graph transformation systems (cf. e.g. [ERT99,GK08]) facilitate the visual simulation of ant colonies in a straightforward way (see also [Höl08]). (4) The formal semantics of communities of autonomous units constitutes a basis for proving correctness results by induction on the length of the transformation sequences or for examining other characteristics (such as termination) by making use of the wide theory of rule-based graph transformation (see [Roz97]). (5) Implementing ACO algorithms with graph transformational systems is useful for verification purposes, i.e., to check whether the algorithms behave properly for specific cases.

In the future, this and further case studies should be implemented with one of the existing graph transformation systems so that (1) the emerging behavior of ant colonies can be visually simulated, and (2) ACO algorithms can be verified. For the implementation purpose we plan to use GrGen [GK08] because it is one of the fastest and most flexible graph transformation systems. Further case studies could take into account more advanced elitist strategies as well as dynamic aspects (see e.g. [ES02,DS04,MGRV05,RDH04,RMLG07]). Another interesting task is to investigate how communities of autonomous units can serve as a modeling framework for swarm intelligence in general.

References

[CEH+97] Andrea Corradini, Hartmut Ehrig, Reiko Heckel, Michael Löwe, Ugo Montanari, and Francesca Rossi. Algebraic approaches to graph transformation part I: Basic concepts and double pushout approach. In Rozenberg [Roz97], pages 163–245.

[DS04] Marco Dorigo and Thomas Stützle. *Ant Colony Optimization.* MIT-Press, 2004.

[ERT99] Claudia Ermel, Michael Rudolf, and Gabriele Taentzer. The AGG-approach: Language and environment. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages and Tools*, pages 551–603. World Scientific, Singapore, 1999.

[ES02] Casper Joost Eyckelhof and Marko Snoek. Ant systems for a dynamic TSP - ants caught in a traffic jam. In M. Dorigo, G. Caro Di, and M. Sampels, editors, *Ant Algorithms - Third International Workshop, ANTS 2002*, volume 2462 of *Lecture notes in Computer Science*, pages 88–98, 2002.

[GK08] Rubino Geiß and Moritz Kroll. GrGen.NET: A fast, expressive, and general purpose graph rewrite tool. In A. Schürr, M. Nagl, and A. Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 568–569, 2008.

[HKK09] Karsten Hölscher, Hans-Jörg Kreowski, and Sabine Kuske. Autonomous units to model interacting sequential and parallel processes. *Fundamenta Informaticae*, 92(3):233–257, 2009.

[Höl08] Karsten Hölscher. *Autonomous Units as a Rule-based Concept for the Modeling of Autonomous and Cooperating Processes.* Logos Verlag, 2008. PhD thesis.

[KK07] Hans-Jörg Kreowski and Sabine Kuske. Autonomous units and their semantics - the parallel case. In J.L. Fiadeiro and P.Y. Schobbens, editors, *Recent Trends in Algebraic Development Techniques, 18th International Workshop, WADT 2006*, volume 4408 of *Lecture Notes in Computer Science*, pages 56–73, 2007.

[KK08] Hans-Jörg Kreowski and Sabine Kuske. Communities of autonomous units for pickup and delivery vehicle routing. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, *Proc. 3rd Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE '07)*, volume 5088 of *Lecture Notes in Computer Science*, pages 281–296, 2008.

[MGRV05] Roberto Montemanni, Luca Maria Gambardella, Andrea Emilio Rizzoli, and Alberto V.Donati. Ant colony system for a dynamic vehicle routing problem. *Journal of Combinatorial Optimization*, 10(4):327–343, 2005.

[RDH04] Marc Reimann, Karl Doerner, and Richard F. Hartl. D-ants: Savings based ants divide and conquer the vehicle routing problem. *Computers & OR*, 31(4):563–591, 2004.

[RMLG07] Andrea Emilio Rizzoli, Roberto Montemanni, Enzo Lucibello, and Luca Maria Gambardella. Ant colony optimization for real-world vehicle routing problems. *Swarm Intelligence*, 1(2):135–151, 2007.

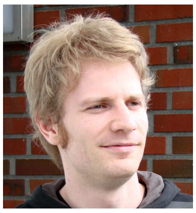
[Roz97] Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations.* World Scientific, Singapore, 1997.

Dr. Sabine Kuske
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
kuske@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~kuske

Sabine Kuske has been a member of Hans-Jörg's team since November 1991. He supervised her diploma and doctoral theses. Their common research interests concern all aspects of graph transformation; in particular, they have been working together on autonomous units.

Melanie Luderer
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
melu@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~melu

Melanie Luderer is a doctoral student of the International Graduate School for Dynamics in Logistics since 2006, and Hans-Jörg Kreowski is her supervisor. He was also the supervisor for her diploma thesis.



Hauke Tönnies
Fachbereich 3 – Informatik
Universität Bremen
D-28334 Bremen (Germany)
hatoe@informatik.uni-bremen.de
http://www.informatik.uni-bremen.de/~hatoe

Hauke Tönnies is a doctoral student supervised by Hans-Jörg Kreowski since 2008. He is supported by the Collaborative Research Centre 637: Autonomous Cooperating Logistic Processes.

On Judgements and Propositions

Bernd Mahr[*]

1 Introduction, Addressed to Hans-Jörg Kreowski

Dear Hans-Jörg! In the seventies and early eighties we have been colleagues in the 'Automatentheorie und Formale Sprachen' group at TU Berlin and shared many interests. Later we departed into different fields of research and followed different directions of thought. It is thirty years now that we have completed our PhD at TU Berlin, and this year is your sixtieth birthday. Since the times at ATFS we have not met often but have never lost the feeling of friendship and trust. What we have lost is to know much about each other. I therefore think that it is natural to ask, "What are you doing?" With the publication of this birthday volume I have the opportunity to briefly give you an answer and to honour you with a paper on a question which is presently twisting my mind: *What is the fundament of logic that admits the different views on it?*

It may come as a surprise that after all these years of teaching and research in logic I am unable to answer this question right away, but I must admit that exercising logic formally attracts the attention to the formal aspects of logic as a language and of logic as a calculus rather than to the fundamental question of its origins. It turned out that I could not avoid to answering this question: I teach a mandatory course to the first semester students, called 'Informatik Propädeutikum,' and I thought it would not suffice to start explaining logic with the words "assume you have a family of countably infinite sets of variables and a ..." But I thought it would be more appropriate to explain logic in this 'Propädeutikum' by the fundamental conceptions underlying its formalisation.

My hypothetical answer to the above question is that logic is rooted on conceptions of judgement and proposition. Both have been a matter of dispute since the beginning of Greek philosophy and are still today under debate. It was therefore only natural to look at some of their prominent conceptualisations and to try to gain a better understanding of what is meant by these and what is their relation. It turned out, however, that this is not easy at all as the notions of judgement and proposition are deeply involved. They touch on and relate fundamental questions of language, ontology, psychology, philosophy and mathematics, and their meaning is far from being common sense. By some authors the notion of proposition is even objected to be meaningful at all [vOQ80, pp. 331–401], and the word judgement is taken to express things of the most different kind, from the most elementary relation between the human mind and the world [Kan90], up to what is realised by natural deduction proofs in intuition-

[*] I thank Andrea Hillenbrand and Sebastian Bab for discussions and for their support.

istic type theory[1]. On the other hand my study of conceptualisations of what propositions and judgements may mean gave me interesting insights and shed some light on the different approaches to logic and to their relationships.

Here I come to the somewhat wider answer to the above question of what I am doing and why I am interested in the notion of judgement. For several and partly funny reasons I started, about ten years ago, to study the general notion of 'model' and wrote several articles on this topic[2]. I found that the key to resolve the problems in explaining the notion of 'model' in its full generality is to transform the question of *"what is a model"* from ontology to logic and to ask instead *"what justifies a judgement, that something is a model."* A deeper analysis of this new question led me back to thoughts on the notions of conception (in German *Auffassung*) and context I had developed years ago in a project on cognition and context, which grew out of our machine translation activities [Mah97]. The new thoughts on these notions of conception and context resulted in a 'model of conception.' This model is an axiomatisation of reflexive universes of things relativised by their subject dependent context. In a recent thesis [Wie08] set theoretic realisations of this axiomatisation have been studied and its consistency has been proven. So here is the other source to what motivates my interest in the notion of judgement: its close relation to the concepts of context and conception, which, I think, are not only fundamental in an explanation of the general notion of 'model' but are also of much wider interest in computer science and technology. However, to show this, there is much left to be done.

The following sections do not aim for a proven result but try to provide insight into some of the conceptualisations of judgement and proposition and their relationships. Hans-Jörg, I hope you enjoy reading them.

2 The Notions of Judgement and Proposition

By *"judgement"* one denotes both, the *act of judging* and the *result* of this act.[3] A judgement, as an act and as the result of this act, is always concerning something, *that which is judged*, *the judged*. A conventional though rather imprecise definition states that *that which is judged* is a proposition, and that a *proposition* is what is *true* or *false*.

The traditional views on judgement date back to the pre-Socratic philosopher Parmenides [Par69], and, among others, to Plato in his SOPHISTES, and

[1] See for example Göran Sundholm: *Proofs as Acts and Proofs as Objects: Some questions for Dag Prawitz*, as well as Prawitz' response to these questions, both in [The98, pp. 187–216, 283–337].

[2] See for example [Mah09].

[3] What actually is denoted by "judgement" depends heavily on what is considered to be an act of judging and what as the result of this act. See for example Göran Sundholm: *Proofs as Acts and Proofs as Objects: Some questions for Dag Prawitz*, as well as Prawitz' response to these questions, both in [The98].

predominantly to Aristoteles. Aristoteles developed in his writings[4] which were later collectively called *organon* (in German *Werkzeug*), in the sense of a *tool of the mind*, the first elaborated and most influential concept of judgement, and laid therewith the ground for what *logic* is about. It is common to the traditional views on judgement that that which is judged is affirmed or denied to *exist*, and that 'being,' 'the presence of being there,' [Tug03] and later also 'existence,' make the grounds for affirmation and denial. Judgements under these views concern things and matters in reality, and, accordingly, the notion of judgement under these views is based on reality as a fundament of truth. Despite many extensions and modifications much of the essence of these traditional views, namely of Aristoteles' approach to logic, has been maintained through the times until to the mid 19th century. And it can still be found today in the Tarski style of semantics, as it is commonly used in the model theoretic semantics of logic.

Major modifications on the traditional views, which finally lead to the formal treatment of logic today, originate from the work of George Boole on the mathematical theories of logic [Boo58], from the works of Bernhard Bolzano[5], namely in his WISSENSCHAFTSLEHRE and Franz Brentano [Bre08] in his PSYCHOLOGIE VOM EMPIRISCHEN STANDPUNKTE, and from the works of Brentano's students Kasimir Twardowski [Twa82], Alexius Meinong [Mei02] and Edmund Husserl [Hus93], on psychological theories of judgement. Maybe the strongest influence on modern logic had Gottlob Frege who, with his BEGRIFFSSCHRIFT (1879) [Fre07], laid the ground for a new understanding of quantification and predication, and developed basic principles of the notion of judgement, which have been widely adopted in the formal treatment of semantics. Based on Frege's work and other sources Alfred North Whitehead and Bertrand Russell wrote their PRINCIPIA MATHEMATICA [WR62] and Ludwig Wittgenstein responds in his TRACTATUS LOGICO PHILOSOPHICUS [Wit73] to Frege and to Russell's THEORY OF KNOWLEDGE [Rus92]. Later, in his Philosophical Investigations [Wit67], he recalls many of the thoughts he had put forward in the tractatus on the nature of language. The modern debates on the notion of judgement owe much to the theory of speech acts as it has been developed by John L. Austin [Aus02] and John R. Searle [Sea08]. In the course of this development the notions of judgement and proposition became a subject matter in ontology and philosophy rather than in classical formal logic where they are, so to say, banned to the meta-level of formalisation and only implicitly present in the interpretation of sentences. Explicit use of the notion of judgement, however, can be found in Per Martin-Löf's INTUITIONISTIC TYPE THEORY and in specification frameworks and formalisms inspired by him, like the calculus of constructions, LF, Coq or Isabelle. Martin-Löf insists on a formal distinction between propositions and judgements [ML84]. This distinction he elaborates further in his lectures ON THE MEANING OF THE LOGICAL CONSTANTS AND THE JUSTIFICATIONS OF THE LOGICAL LAWS and in a recent lecture on ASSERTIONS, ASSERTORIC

[4] These writings include CATEGORIAE, DE INTERPRETATIONE, ANALYTICA PRIORA, ANALYTICA POSTERIORI, TOPICA and DE SOPHISTIIS ELENCHIIS.

[5] [Bol81], see also [Ber92].

Contents and Propositions[6]. Martin-Löf's careful considerations also influenced some of the conceptualisations in epsilon-theory (see section 9 below), as it is being studied by the author and his co-workers.

3 Realistic Conceptions of Proposition and Judgement

At the very beginning of De Interpretatione Aristoteles explains his view on the relationship between language, mind and reality, which is essential for the understanding of his conception of judgements: *"that what is expressed* [logos, in German *Satz*, in English *sentence*] *is a symbol of the states of the soul, and that which is written is a symbol of that which is expressed"* [...] *"that, of which the states of the soul are images, are the things."* [TW04, p. 19] What is to observe here is that the states of the soul, which may be understood as *thoughts* in the sense of mental states, mediate between reality on the one side, and sentences being expressed and written on the other.

In his conception of the notion of judgement Aristoteles takes first of all a linguistic view, but combines it with a psychological and an ontological perspective. He describes a judgement as to being a particular type of sentence: *"Though a sentence is meant to denote, not every sentence is a judgement but only one in which the assertion of truth or falseness is present. It is, however, not present in every sentence since, for example, a wish is a sentence, but it is neither true nor false."* [Ari67b, p. 7] Today we would say that he distinguishes different kinds of sentences, a distinction which in speech act theory is made by the distinction of illocutionary forces of a proposition. As judgements he singles out *propositional sentences.* As the criterion for a sentence to be a judgement he states that the sentence is grammatically composed of two parts: a subject and a verb. Both parts have meaning in the sense that they both denote something by convention. And as the criterion for truth he states that the composition of subject and verb, the copula "is" or a derived form of it, has to reflect the relation of the things the subject and the verb denote: *"To affirm* [katáphasis] *is to express something towards something, and to deny* [apóphasis] *is to express something away from something"*; and concerning truth and falseness of a judgement made, Aristoteles writes: *"The one, who thinks as being separated what is separated and as being composed what is composed, thinks true; but he thinks false, whose thoughts are contrary to the things."* [Ari67c, p. 7] It is to note here that true and false are not atomic values but qualities of thinking. Other than logicians today Aristoteles restricts his observations to simple judgements of the subject-predicate form. He does not consider complex judgements which are composed of sub-judgements. For the meaning of simple judgements he follows a *principal of compositionality*, which postulates that the meaning of composed expressions is the composition

[6] See [ML96, 11–60]. His recent thoughts on judgements Martin-Löf presented in the lecture Assertions, Assertoric Contents and Propositions, which he gave at the workshop on *Judgements, Assertions, and Propositions - The Logical Semantics and Pragmatics of Sentences* at TU Berlin on January 11, 2008.

of the meanings of the individual expressions. Leibniz states a similar principle in his 'ars characteristica,' and Frege uses a variant of this principle in his functional interpretation of sentences.

In Aristoteles' view there are two qualities of judgements, namely affirmation and denial, and *"every judgement is either a judgement about what there is in reality, what there is by necessity or what there is by possibility"*. Aristoteles also distinguishes three kinds of judgement: *"a judgement* [...] *is either general, or particular or undetermined. General means that something applies*[7] *to all or none, particular means that something applies to a single, or to a single not, or not to all, and undetermined means that it applies or does not apply without determination of the general or the particular..."* [Ari67a] In modern logic where propositional sentences have a complex structure, the distinction between general, particular and undetermined judgements is less relevant or even meaningless.

4 Formalisation of Categorical Judgements

Today we would rephrase Aristoteles' view as follows: sentences and their written forms refer to thoughts created in a mental act, and these thoughts as mental states are images of existing things. Truth is assigned to thoughts, in the sense of a mental act, and requires the correspondence between that which is thought and the things of which the thoughts, in the sense of mental states, are images. Though thoughts, in Aristoteles' sense, correspond to what in speech act theory is called a propositional act, there is also a major difference: Aristoteles does not think of thoughts in terms of reference and predication in the way we do. Take as an example the sentence "all men are mortal," which, according to his grammatical criterion for judgements and his classification of kinds, is a general judgement. Aristoteles' reading of this sentence can be formalised as the type proposition

(all men) are (mortal)

in which the composition of two things is asserted. This sentence is true if in the real world *mortality* (which is the thing whose image is symbolised by the written expression 'mortal') applies to *all humans* (which is the thing whose image is symbolized by the written expression 'all men'). The famous 'problem of universals' in the middle ages was about the question if things as expressed by the words 'all men' and 'mortal,' have existence. Since Frege, motivated by the concept of a mathematical function in arithmetic and higher analysis, proposed predication as a form of function application which results truth values [Fre75, pp. 17-39], and since he introduced individual variables for the indication of individuals in quantification [Fre75, p. 33], today's conventional reading of the

[7] The English word 'applying' is used here to translate the Greek word 'hyparchein.'

respective sentence[8] can be expressed in the formalisation

$$\forall x.(\mathrm{men}(x) \rightarrow \mathrm{mortal}(x)).$$

If we write atomic predication in the form of a type proposition, we get

$$\forall x.(x : \mathrm{men} \rightarrow x : \mathrm{mortal})$$

This type propositional formalisation shows, on the one hand, a closer similarity to the Aristotelian view, though the sentence as a whole is not a simple judgement but has a complex structure, and it shows, on the other hand, a closer similarity to the truth condition for atomic predications in the Tarski style of model theoretic semantics, which may in a semi-formal way be phrased as

$$\text{For all } h \text{ it is true that, if } (h \in \mathrm{A}_{\mathrm{men}}) \text{ then } (h \in \mathrm{A}_{\mathrm{mortal}})$$

where h denotes an unspecified individual in the domain of interpretation, and $\mathrm{A}_{\mathrm{men}}$ and $\mathrm{A}_{\mathrm{mortal}}$ denote subsets of this domain. It is interesting to note that the model theoretic truth condition expresses the same idea as the truth condition in Aristoteles' view: "the presence of being there." The only difference is that the Aristotelian truth condition concerns things in reality while in the Tarski style of model theoretic semantics the condition is expressed relative to a domain of interpretation and expresses set theoretic membership. It is also interesting to note that the (semantic) reading of $\forall x.$ as *"for all h which instantiate x, it is true that"* turns the sentence

$$\forall x.(x : \mathrm{men} \rightarrow x : \mathrm{mortal})$$

into a symbolisation of a proposition in which an h-indexed family, not of propositions, but of judgements is expressed.

Following the conventional interpretation, the above sentence is also true in a world where humans cannot be found, since in this case the premises of the implication is false. This property indicates that the use of variables in quantification resolves the difficulty Parmenides saw in the notion of 'non-existence': He concluded that negation cannot be thought of because "what is not is not, and can therefore not be" [Par69]. In the modern understanding of 'non-existence,' instead, non-existence is the property of the domain of interpretation that the thing with the property in question cannot be found, i.e. is not there. The 'logical' reading [TW04] of a simple judgement as a complex sentence resolves this difficulty because it reads "existence" as "existence with some property." But this conception of existence as 'being there' has the consequence that there is always a universe of things needed to be there, whose entities fall under defined categories before their existence can be asserted. In conventional logic this is enforced in the inductive definition of formulas and the set theoretic structures for the interpretation of formulas. If we read, to use an example of Quine, the

[8] See also Russell's *On Denoting* from 1905, which in German translation appeared as [Rus00, pp. 3–22].

"existence of unicorns" as the "existence of something which unicorns," the question comes up of what the nature of this something is. At the linguistic level of conventional logic it is a variable and at the semantic level in the Tarski style interpretation it is possibly any element in the domain of interpretation. But what is it in a reality that can hardly be well-defined as a set, and can it be given some ontological status at all?[9]

5 Brentano's Notion of Judgement

Early conceptions of judgement and proposition with a particular emphasis on their roles in science and logic have been studied by Bernhard Bolzano in his WISSENSCHAFTSLEHRE (1837). They laid the ground for further investigations by Brentano and his students. A thorough account of these investigations can be found in the article AUSTRIAN THEORIES OF JUDGEMENT: BOLZANO, BRENTANO, MEINONG, AND HUSSERL by Robin D. Rollinger [Rol08, 233–261]. Of particular interest here is Brentano's conception of intentionality. It not only opened a new perspective on judgements but may also be seen as a major source of speech act theory. Speech act theory strongly influenced modern conceptions of linking mental acts and symbolic presentations, and is therefore also fundamental in intuitionistic approaches to proposition and judgement.

In Brentano's PSYCHOLOGY FROM AN EMPIRICAL STANDPOINT (1874), an act of judging is a case of a mental act. Mental acts contain an object intentionally within itself. What in a judgement is affirmed or denied is the existence of this object. And so we might say that in Brentano's conception a proposition is the existence of the object of a judgement, which may be true or false. *"Every mental phenomenon is characterized by what the Scholastics of the Middle Ages called the intentional (or mental) inexistence of an object, and what we might call, though not wholly unambiguously, reference to a content, direction towards an object (which might not to be understood here as meaning a thing), or immanent objectivity. Every mental phenomenon includes something as object within itself, although they do not all do so in the same way. In presentation* [in German 'Vorstellung'] *something is presented, in judgement something is affirmed or denied, in love loved, in hate hated, in desire desired and so on. This intentional in-existence is characteristic exclusively of mental phenomena. No physical phenomenon exhibits anything like it. We would, therefore, define mental phenomena by saying that they are phenomena which contain an object intentionally within themselves."* [Bre95, pp. 88–89]

Brentano's conception of *intention* has strongly influenced modern philosophy, namely logic, ontology, existential philosophy and theories of language and semantics. He claims that every mental act is a presentation or rests on a presentation, and that a distinction has to be made between the object presented and the content of its presentation. But at the same time he is convinced that there are presentations which, though they have content, have no object, like the

[9] I consider this a serious question. A somewhat unsatisfactory answer is given in [Gro92, pp. 106–119].

presentation of a golden mountain or the presentation of a round square. This belief, however, has the consequence that such things cannot be judged to not exist. But this appears to be against our intuition as we can think or even have an imagination of things which, we know, do not exist, and also can judge that they do not exist. We can think of a round square and even imagine unicorns and a flat earth, and we daily have the presentation of a user-friendly computer system. Even further, we seem to need a presentation of something before we can assert it to not exist. This observation, it turns out, touches at a major problem, the question of what exactly we mean by a proposition and by a judgement and how we understand the relation between a judgement and 'its' proposition. The disputes in analytic philosophy literature show that even today this problem has not yet found a commonly accepted solution.

6 Twardowski's Theory of Presentations

Kasimir Twardowski, one of Brentano's students in Vienna, addressed this problem in his Habilitation thesis ON THE CONTENT AND OBJECT OF PRESENTATIONS - A PSYCHOLOGICAL INVESTIGATION [Twa77] in 1894. He studied the concept of *presentation* (in German *Vorstellung*) and argues that presentations imply in what they present to the mind, two different objects rather than one: the object towards which a presentation is directed (in German *Gegenstand*), and the object which is its content (in German *Inhalt*). Though the focus of his thesis is on presentations, he also deals with the notion of *judgement* (in German *Urteil*) and sees a *"perfect analogy"* [Twa77, p. 7] between *presentations* and *judgements*. Both, he states, imply an act, both concern something, namely what is presented and what is judged, and in both this something which is presented or judged, is to be subdivided into object and content, the latter of which he calls the intentional object of the act.

While one and the same object can be presented as well as being judged, he finds the distinction between presentation and judgement in the intentional object of the act: *"When the object is presented and when it is judged, in both cases there occurs a third thing, besides the mental act and its object, which is, as it were, a sign of the object: its mental 'picture' when it is presented and its existence when it is judged."*[10] Here the 'object' of a judgement is the object about which the judgement is made, while the 'subject' of a judgement is that what is affirmed or denied, *the object's existence*. Twardowski insists that *"presentation and judgement are two separated classes of mental phenomena without intermediate forms of transition."* [Twa77, p. 6]

The distinction between object and content in what is presented or judged is most natural, though the question of what exactly an object is still remains unanswered. In the beginning of his treatise on objects in §7 of his investigation Twardowski gives a partial answer to it: *"According to our view, the object*

[10] See [Twa77, p. 7]; the conception of 'existence' as the content of a judgement is not obvious. See Grossmann's criticism on this conception in: Reinhard Grossmann: *Introduction*, in [Twa77, pp. VII - IVXXX, here pp. IX - XI].

of presentations, of judgements, of feelings, as well as of volitions [in German 'Wollungen']*, is something different from the thing as such* [in German 'Ding an sich']*, if we understand by the latter the unknown cause of what affects our senses. The meaning of the word 'object' coincides in this respect with the meaning of the word 'phenomenon' or 'appearance,' whose cause is either, according to Berkeley, God, or, according to the extreme idealists, our own mind, or, according to the moderate 'real-idealists' the respective things as such. What we have said so far about objects of presentation and what will come to light about them in the following investigations is claimed to hold no matter which one of the just mentioned viewpoints one may choose. Every presentation presents something, no matter whether it exists or not, no matter whether it appears as independent of us in our own imagination; whatever it may be, it is - insofar as we have a presentation of it - the object of these acts, in contrast to us and our activity of conceiving* [in German 'vorstellenden Tätigkeit']." [Twa77, p. 33]

And at the end of his treatise on objects he writes: *"Summarizing what was said, we can describe the object in the following way. Everything that is presented through a presentation, that is affirmed or denied through a judgement, that is desired or detested through an emotion* [in German 'Gemütsthätigkeit']*, we can call an object. Objects are either real or not real; they are either possible or impossible objects; they exist or do not exist. What is common to them all is that they are or that they can be the object (not the intentional object!) of mental acts* [in German 'psychischer Akte']*, that their linguistic designation is the name ..., and that considered as genus* [in German 'Gattung']*, they form a summum genus which finds its usual linguistic expression in the word 'something'* [in German 'etwas']*. Everything which is in the widest sense "something" is called "object," first of all in regard to a subject, but then also regardless of this relationship."*

An important term in this citation is the word "through." It assigns the mental act and its intentional object the role of a mediator: *through* the act and content of a presentation an object is presented, and, accordingly, *through* the acts of affirmation or denial of existence an object is judged. Every object, now, existing or not, can be seen to be the object of both, a presentation and a judgement. Twardowski's concept of object of a mental act solves the above mentioned problem of judging non-existing objects to not exist, and Brentano's belief in presentations which have no object turns out to be wrong: *"The confusion of the proponents of objectless presentations consists in that they mistook the non-existence of an object for its not being presented."* [Twa77, pp. 20–29, here p.22] But despite the fact that the general approach to the objects of a judgement seems to be most reasonable, the ontological status of objects in intentional relations is subject to controversies and not at all free from problems. It is therefore heavily debated in the literature. Progress has been made with the invention of 'states of affairs' and with the conception that judgements intend states of affairs rather than objects.[11] It seems to me that the invention of states of affairs has two sources: predications on the one hand, as they have been used by Frege in his BEGRIFFSSCHRIFT for the purpose of formalising arithmetic and

[11] See Grossmann's introduction to Twardowski in [Twa77] and [Gro92].

by Peano and Russell who applied and developed formal description techniques for other parts of mathematics, and *"Sachverhalte"* on the other, as certain types of (intentional) objects, studied by Meinong, Husserl and Reinach [Smi89]. This view is later also found in Wittgenstein's Tractatus, the first two sentences of which are: *"Die Welt ist alles, was der Fall ist. Die Welt ist die Gesamtheit der Tatsachen, nicht der Dinge"* ("The world is everything that is the case. The world is the total of what is the case, not of the things").[12]

7 Frege's Conception of Proposition and Judgement

An answer of pragmatic value for the question of what propositions and judgements are and how they are related seems possible only within a prescriptive deductive or semantic framework[13]. Conventional formal logic makes no clear distinction between the two concepts and avoids their conceptualisation at all. And a dedicated formal theory of propositions and judgements has not yet been proposed. However, there are considerations which aim at clarification.

In his article "ÜBER SINN UND BEDEUTUNG" (1892) Gottlob Frege discusses the meaning of verbal expressions, like names, denotations and sentences, and draws the well known distinctions between *sign* (in German *Zeichen*), *sense* (in German *Sinn*), *reference* (in German *Bedeutung*), and *presentation* (in German *Vorstellung*). *"A sign is the expression of some sense and it denotes or references its reference."* [Fre75, p. 46] A comparison of this distinction with Twardowski's distinction of *names*, *content* and *object* of a presentation shows many similarities, but also major differences: Frege's sense is not part of a mental state or act. It has objectivity. Therefore *presentations* are not *senses* and therefore Twardowski's *content* is not the same as Frege's *sense*, even though they play similar roles in the designation of an object. And *names* in Twardowski's conception do not designate matters of affairs but objects. In Frege's conception also sentences have a *sense* and a *reference*, and the *sense of a sentence* is what he calls a *thought* (in German *Gedanke*). Frege's concept of thought is what Husserl and (the early) Wittgenstein call *matter of affairs* [TW04, p. 17], and what Russell calls *proposition* for which he later uses the word *assertion.*[14] If a *sign* is a *sentence*, the question is what it references. In Frege's view, a sentence references a *truth value*, i.e. the value *true* or *false*. Accordingly, also truth values are objects (in German *Gegenstände*). Since in Twardowski's Habilitation there is no citation of Frege's work, we must conclude, that Frege's work was not known in Vienna at that time. Frege's conception of proposition was later adopted in formal logic, though in the hidden form of the recursive definition of interpretation and validity, which is derived from Frege's principles of compositionality and

[12] See [Wit73, p. 11], English translation by the author.

[13] To a certain degree this is done in Martin-Löf's intuitionistic type theory and formalisms following him (see below).

[14] See also [ML96, 11–60], where he gives an account on the development of the concepts of proposition and judgement in the light of his intuitionistic type theory.

truth functionality. In view of pragmatic language use and meaning, however, it has been strongly criticised.[15]

Frege also made an important contribution to the conceptualisation of judgement. What is being affirmed or denied in a judgement is that a proposition is true or false, or in other words, that a matter of affairs is a fact or not. Frege thereby frees the concept of judgement from its binding to object-existence. He also draws a clear distinction between proposition and judgement by saying that a judgement is not just the affirmation or denial of a proposition, but that the affirmation or denial is asserted. In his BEGRIFFSSCHRIFT (1879) Frege introduces a notation for assertions, the vertical stroke, which he later combined with the horizontal stroke to indicate assigning truth, and the negated horizontal stroke to indicate falseness. So, for example, the assertion that the earth is flat can then be expressed as

$$\vdash \text{flat (earth)}$$

and the assertion that the earth is not flat can be expressed as

$$|\!\!\top \text{flat (earth)}$$

Here the symbol $|\!\!\top$ is to be read as "it is not the case that," and not as "it is not asserted that" which would be the negation of the assertion. Frege's observation that a judgement is more than just the statement of a true or false proposition, because a statement could also mean an assumption, makes the distinction between different kinds of judgements, as it was customary in the traditional views on judgements, meaningless. If we respect this observation, a judgement is always an affirmation. In type propositional form the above assertions may be written as

$$\vdash \text{flat (earth) : true}$$

And, accordingly,

$$\vdash \text{flat (earth) : false}$$

Frege gives an impressive insight into his style of writing and the purpose and use of formal notations in mathematics in ÜBER DIE WISSENSCHAFTLICHE BERECHTIGUNG EINER BEGRIFFSSCHRIFT (1882). He motivates the notation of the judgement stroke $\vdash$ with the pragmatic needs in the writing of formal expressions and in the depiction of logical derivations on a sheet of paper. The question of *"how can we write?"* becomes prominent and the analysis of *"what can we write down?"* leads to the new view on judgements. Frege uses the judgement stroke in a given context of discourse, the context of a given system of axioms and rules or of a given model or theory. It is this context which justifies the assertion of truth.

8 Martin-Löf's Conception of Judgement and Proposition

In his INTUITIONISTIC TYPE THEORY, Martin-Löf makes the following distinction between proposition and judgement: *"Here the distinction between proposi-*

[15] See for example [Dum82].

tion (Ger. Satz) and assertion or judgement (Ger. Urteil) is essential. What we combine by means of the logical operations (falsum, implication, and, or, for all, there is) and hold to be true are propositions. When we hold a proposition to be true, we make a judgement:

$$((A : \textit{proposition}) \textit{ is true}) : \textit{judgement}$$

In particular, the premises and the conclusion of a logical inference are judgements. The distinction between proposition and judgement was clear from Frege to Principia. These notions have later been replaced by the formalistic notions of formula and theorem (in a formal system), respectively. Contrary to formulas, propositions are not defined inductively. So to speak, they form an open concept. In standard textbook presentations of first order logic, we can distinguish three quite separate steps:

1. *Inductive definition of terms and formulas*
2. *Specification of axioms and rules of inference*
3. *Semantical interpretation*

Formulas and deductions are given meaning only through semantics, which is usually done following Tarski and assuming set theory.

What we do here is meant to be closer to ordinary mathematical practice. We will avoid keeping form and meaning (content) apart. Instead we will at the same time display certain forms of judgement and inference that are used in mathematical proofs and explain them semantically. Thus we make explicit what is usually implicitly taken for granted. When one treats logic as any other branch of mathematics, as in the metamathematical tradition originated by Hilbert, such judgements and inferences are only partially and formally represented in the so-called object language, while they are implicitly used, as in any other branch of mathematics, in the so-called metalanguage.

Our main aim is to build up a system of formal rules representing in the best possible way informal (mathematical) reasoning." [ML84, pp. 3–4]

In Martin-Löf's informal reasoning by means of formal rules judgements are not viewed from a language perspective, as Aristoteles did and as we still do today, at least in most of the philosophical and formal logic accounts, but are closer to speech acts in the sense of Austin's *"how to do things with words."* Martin-Löf's informal reasoning is to be seen as a performing of acts of judging, which consist in the writing down of judgements. The writing down of judgements is justified by the rules of the type system, whose premises are again judgements. Some rules, however, have no premises. They are axioms. Judgements in Martin-Löf's type theory have one of the following written forms: A set, $A = B$, $a \in A$, or $a = b \in A$. The last two of these forms correspond closely to the judgements to be made in Cantor's criterion for a set to be 'well-defined,' which he phrased in 1882, with the study of powers, when he refined his notion of a set[16]: *"I call*

[16] The criterion is phrased in a letter by Cantor to Richard Dedekind in 1882; see for example [Dau79], cited in English from [Dau79, p. 83].

an aggregate (a collection, a set) of elements which belong to any domain of concepts [in German *Begriffssphäre*] *well-defined, if it must be regarded as internally determined on the basis of its definition and in consequence of the logical principle of the excluded middle. It must also be internally determined whether any object belonging to the same domain of concepts belongs to the aggregate in question as an element or not, and whether two objects belonging to the set, despite formal differences, are equal to one another or not."*

All forms of judgement in Martin-Löf's type theory propose a natural set theoretical interpretation. The given system of rules, however, admits also other readings of these forms. One of these readings corresponds to the well known concept of 'propositions as types,' also known as the Curry-Howard isomorphism, and reads the judgement $a \in A$ as *"a is a proof for the proposition A."* This reading is not only the basis of his system as an intuitionistic theory of types, but is also consistent with an intuitionistic interpretation of his approach as a whole: From a meta-level perspective the written forms of judgements symbolise propositions for which his system lays down what counts as a proof.[17] This is the way how he explains semantically these forms of judgements.

Concerning propositions Martin-Löf writes: *"Classically, a proposition is nothing but a truth value, that is, an element of the set of truth values, whose two elements are the true and the false. Because of the difficulties of justifying the rules for forming propositions by means of quantification over infinite domains, when a proposition is understood as a truth value, this explanation is rejected by the intuitionists and replaced by saying that*

> *A proposition is defined by laying down what counts as a proof of the proposition,*

and that

> *a proposition is true if it has a proof, that is, if a proof of it can be given.*

Thus, intuitionistically, truth is identified with provability, though of course not (because of Gödel's incompleteness theorem) with derivability within any particular formal system." [ML84, p. 11]

The conventional conception of formal logic leaves these notions of proposition and judgement out of its consideration. It treats these notions only implicitly in the recursive definitions of interpretation and avoids their explicit notation.

9 Logics with Propositional Variables

Also classical propositional and predicate logics can be seen as conceptions of propositions. They provide linguistic means, usually in terms of alphabets and inductive definitions, to write sentences which through interpretation become

[17] The status of a proof in the intuitionistic conception of truth has been a matter of discussion. See for example [Sun94], as well as Sundholm's and Prawitz' debate in the above mentioned volume 64 in Theoria [The98].

either true or false. Sentences in propositional logic are built up from propositional variables and propositional connectives like ‘and,’ ‘or,’ ‘not,’ and may be others. The interpretation of propositional sentences is based on a given truth-assignment which assigns truth-values ‘true’ or ‘false’ to propositional variables and is defined by an inductively defined evaluation function which assigns truth-values to propositional sentences. Here the principles of compositionality and truth functionality are maintained in their purest form. The ‘architecture’ of (first order) predicate logic is not much different, except that atomic formulas are not propositional variables but predications and equalities, that variables are object-variables taking values from a given semantic domain, and that expressive power and expressiveness are enriched by function symbols for object description, relation symbols for predications and quantifiers ranging over the elements of the carrier sets of the semantic domain.[18] Sentences in these logics are complex forms, which express, trough their interpretation for a given truth-assignment or in a given semantic domain, *sense*, to use Frege’s terminology. They may also be read as formal statements of a matters of affairs, which induced by their interpretation. But these matters of affairs are never made explicit and only hidden in the recursive interpretation of sentences. Interpretation only yields truth-values, and equivalence at the object level can only be expressed in terms of ‘having the same truth-value,’ rather than ‘stating the same matter of affairs.’ This is, how classical logic avoids the notions of proposition, and how it treats judgements only implicitly in its definition of the process of interpretation relative to a given truth-assignment or semantic domain.

There is a logic to explicitly express truth of propositions, quantification over propositional variables and propositional equivalence, which has been developed by Werner Sträter [Str92] and is called $\in_T$-logic. One of the motivations for its design was to avoid partial truth predicates and to admit formulations like the liar paradox

$$x \equiv x : \text{false}$$

to be treated as contradictions. $\in_T$-logic grew out of an extensional interpretation of types[19], which reads a *type proposition*

$$e : T$$

as a *statement of membership*

$$[[e]] \in [[T]]$$

The type proposition φ : true would then be read as a statement of membership with $[[\varphi]]$ denoting a proposition and [[true]] a set of true propositions.

$\in_T$-logic is equipped with propositional constants, variables and connectives, quantification over propositional variables, truth predicates and propositional equivalence. Its semantics is defined in the Tarski style, where the semantic domain is a domain of propositions and the interpretation function ensures the

[18] See for example [EMC$^+$01, pp. 221–455].
[19] See [MSU90] and [Mah93].

natural properties of propositional and of first order logic, as far as they apply. It fulfils the well known Tarski biconditionals in the sense that the sentence

$$\forall x.(x : \text{true} \leftrightarrow x) : \text{true}$$

is universally true. $\in_T$-logic has an impredicative nature and allows for intensional semantics of its sentences. Extensions of this logic have been defined and studied by Philipp Zeitz [Zei00], who introduced Parameterization, by Sebastian Bab [Bab07], who extended $\in_T$-logic by modal operators, and by Steffen Lewitzka [Lew09], who studied an intuitionistic variant of $\in_T$-logic.

Also Frege defines in his BEGRIFFSSCHRIFT[20] a logic with propositional variables. Frege's notations admit the reference to objects and to functions over objects, as well as to functions over functions. They allow for propositional variables ranging over truth values which are viewed as being objects like any other object, they admit to write operators which cover the classical propositional connectives, and they include propositional equality and quantification. The expressiveness of Frege's logic is closely related to a certain instance of Parameterized $\in_T$-logic in the sense of Zeitz. However, there is no perfect analogy. The major difference is in the use of quantification, and in the style of semantics.

Intuitively, there is good reason to also view classical logics as theories of propositions, no matter if in these logics propositions form a distinct and well defined category of entities to be dealt with or not. This is obvious in the case of logics which admit propositional variables, and it is even more obvious for $\in_T$-logic and its extensions, which explicitly support propositional quantification and equivalence, and assume propositions as elements of their semantic domains. Can the same be said for judgements? Classical logics introduce notations for judgements at their meta-level, usually in the form of a sign denoting validity of a sentence under a given interpretation, like for example the validity of φ under the truth-assignment B

$$B \models \varphi$$

But they do this in a rather propositional manner, as they also allow to denoting invalidity.

$$B \not\models \varphi$$

They treat judgements as propositions at the meta-level. Otherwise there is little difference between these signs and Frege's judgement stroke. Frege's notation is based on the assumption of a given model so that there is no need to indicate the truth assignment or the semantic domain of interpretation. And also the fact that the judgement stroke is written at the object level of formalisation is not of much relevance, since it is used at this level not as an operator but as an indicator and, in addition, only at the outermost position of the two dimensional expressions. The judgement stroke can be omitted but it cannot be negated. Negation of the judgement stroke would turn it into a propositional operator. This, by the way is the reason why the expression φ : true in $\in_T$-logic cannot reasonably be interpreted as a judgement. The judgement stroke is not

[20] See also [Her83, pp. IX–XV].

subject to interpretation but is a sign which has a purely pragmatic meaning. It indicates what is an answer to the question "What can I write?"

10 Summary

To turn to the question of how the notions of judgement and proposition discussed do relate to each other we try to answer the following questions:

1. Is there a distinction made between assertion and judgement?
2. Is there a distinction made between proposition and judgement?
3. What is that which is expressed by a judgement?
4. How is a judgement justified?

In his INTUITIONISTIC TYPE THEORY Martin-Löf speaks of judgements rather than assertions, and in his recent lecture on ASSERTIONS, ASSERTORIC CONTENTS AND PROPOSITIONS [ML08], he speaks of assertions rather than judgements. Not fully conform to other naming conventions he uses the term *assertion* to denote the verbal expression of a judgement, in the form of a spoken or written sentence, and with the use of some language or notational convention. But, as he argues, the interchangeable use of the terms judgement and assertion is justified since in logic both depend on rules, the focus of his interest, and rules are the same for both. Following the assumption that assertions are verbal expressions, judgements may be seen as the mental counterparts of assertions. But this view can hardly be maintained since also assertions include an act of judging. Aristoteles avoids this problem by distinguishing between mental states on the one side, which stand in an image-relation to the things, and judgements as symbolisations of these states on the other. He assumes, at least implicitly, that there are two acts: the act of thinking, which, as he says, can be true or false, in the sense of right or wrong, and the act of symbolising which produces a sentence or its written symbolisation. Brentano and Twardowski discuss judgements solely at the level of mental acts. Written forms are out of their interest. In their understanding contents are in the mind while objects are embedded in an intentional relation. The ontological status of objects, however, remains somewhat unclear[21]. They may be real or mental objects, like thoughts, and may exist or not. Frege's thoughts, instead, are explicitly thought of as being independent from some mind, as they have objectivity and can be shared by several subjects. In $\in_T$-Logic the distinction between judgement and assertion is mostly irrelevant, like in most formal logics with a set theoretic Tarski style of semantics. Sets in a set theoretic universe of interpretation have the same ontological status as thoughts in the sense of Frege [Gro92, pp. 106–119]. They are the means by which, through the application of rules of interpretation, sense and reference, in the sense of Frege, are being determined as elements of the given universe.

[21] See the introduction to [Twa77].

Not in all the conceptions discussed a distinction between proposition and judgement is being made. In Aristoteles' conception there is an act of thinking, which has all the ingredients of an act of judging, while the forms of sentences, which are called judgements, may be understood as the grammatical forms of propositions, and the mental states to which they refer, may be understood as propositions which can be true or false. In Brentano's and Twardowski's conception the concept of proposition cannot clearly be identified since that what a judgement is about, is an object which is not necessarily something that is to be true or false, but something that exists or not. Separated from this object to which the judgement refers, is its presentation on which the judgement relies and whose content is a mental image. Only the identification of a judgements object as a matter of affairs [Hus93], rather than an object of some other kind, shows the analogy between Frege's sense and the content of the presentation of the object which is judged to exist or not. Frege's sense is a thought, if the object in question represents a matter of affairs. And of a matter of affairs it can meaningfully be said to exist or not, depending on whether it is a fact or not. This is what Wittgenstein proposes in his TRACTATUS LOGICO PHILOSOPHICUS [Wit73, p. 11]. In his INTUITIONISTIC TYPE THEORY Martin-Löf draws a clear distinction between propositions and judgements and gives formal rules for the formation of judgements. In $\in_T$-Logic, however, the notion of judgement remains only implicit, like in other conventional logics. While the elements of the domain of interpretation are explicitly assumed to be propositions, whatever form they have, the notion of judgement, in the sense of Martin-Löf, is in $\in_T$-Logic only present at the meta-level and not part of the 'object language.' It appears that the notion of judgement, other than the notion of proposition, is not fully semantic in its nature, but has also a substantial pragmatic aspect. This pragmatic aspect is that what Frege expresses in his judgement stroke and what speech act theory identified as the illocutionary role or force of an assertoric act: the beholding of truth. Despite the truth predicates in $\in_T$-Logic and the fact that it obeys the Tarski biconditionals, the beholding of truth is not a feature of the language but an element of its use and as such a consequence of the choice of the universe of interpretation. In Martin-Löf's intuitionistic type theory this pragmatic aspect is part of the 'object language,' which gives it the pragmatic flavour expressed in the question "What can I write?"

One can generally say that that which is expressed in a judgement is the truth of some form of predication. This is obvious in Aristoteles' conception and in his choice of sentences which have the valid form of a judgement, and also, at least formally, in Frege's conception of a concept (in German *Begriff*) as a function whose application results a truth value. In Brentano's conception that which is expressed in a judgement is the existence or non-existence of an object, which in Twardowski's setting is the judgements content. Existence of an object, however, can only be seen as a form of predication if the object can be represented as a matter of affairs. The situation in Martin-Löf's INTUITIONISTIC TYPE THEORY is different. That what is expressed in a judgement is the provability of a proposition, or, in a different reading, the membership in a set.

There is no notion of truth but at the level of judgements in the correctness of the application of the rules. In what is expressed in a judgement, $\in_T$-Logic is not different to conventional formal logics, with the difference that the predications of truth and falseness differ slightly in their form.

If we ask, what justifies a judgement, major differences can be found. In Aristoteles' naturalistic conception justification comes objectively from the things and concerns the question of connectedness. Truth applies to thoughts as mental states and depends on a proper correspondence to the reality of things. In Brentano's conception justification has an epistemic nature and is obtained either from deductions or from inductive proofs. A different view is taken by Frege who sees the justification of a judgement to rest on necessity, which, according to him, corresponds to deduction, or empirical intuition. But truth, in its conception, is found *through* judgements, a conception which gives the judgement stroke not only a pragmatic aspect but, other than it appeared at first, turns it at the same time into a constituent of semantics. Despite similarities in the role of judgements, Frege's view differs in this respect from the conception of Martin-Löf, who sees the basis for justification in the system of rules and not in the beholding of truth. The judgement stroke in his conception is part of pragmatics and not of semantics. In a Tarski style of semantics, as it is applied in $\in_T$-Logic and other conventional logics, the justification comes from the choice of the semantic domain in the interpretation and from the correct application of the interpretation rules. This is not much different to Frege's view, since the choice of the semantic domain of interpretation is also a judgement, and therefore not fully free from subjective influence - but other than in Frege's conception, it avoids, so to say, the responsibility for this choice to be part of the interpretation. In the view of $\in_T$-Logic and other conventional logics, truth and the conditions for the justification of judgements can be said to be defined.

References

[Ari67a] Aristoteles. Analytica Priora, I 27. 43 a 25. In *Adolf Trendelenburg, Elemente der Aristotelischen Logik - griechisch und deutsch, p. 11 (translation by the author).* Rowohlt, 1967.

[Ari67b] Aristoteles. De Interpretatione, 4.17 a I. In *Adolf Trendelenburg, Elemente der Aristotelischen Logik - griechisch und deutsch, p. 7 (translation by the author).* Rowohlt, 1967.

[Ari67c] Aristoteles. Metaphysica, IX 10. 1051 b 3. In *Adolf Trendelenburg, Elemente der Aristotelischen Logik - griechisch und deutsch, p. 7 (translation by the author).* Rowohlt, 1967.

[Aus02] John L. Austin. *Zur Theorie der Sprechakte (How to do things with Words).* Reclam Stuttgart, 2002.

[Bab07] Sebastian Bab. *$\in_\mu$-Logik – Eine Theorie propositionaler Logiken.* Shaker Verlag Aachen, 2007.

[Ber92] Jan Berg. *Ontology without Ultrafilters and Possible Worlds - An examination of Bolzano's Ontology.* Academia Sankt Augustin, 1992.

[Bol81] Bernhard Bolzano. *Wissenschaftslehre, Band 1 - 4.* Scientia Verlag Aalen, 1981.

[Boo58] George Boole. *Investigation of The Laws of Thought On Which Are Founded the Mathematical Theories of Logic and Probabilities.* Dover New York, 1958.

[Bre95] Franz Brentano. *Psychology from an Empirical Standpoint, edited by Linda L. McAlister.* Routledge London, 1995.

[Bre08] Franz Brentano. *Psychologie vom Empirischen Standpunkte - Von der Klassifikation psychischer Phänomene.* Ontos Verlag Heusenstamm, 2008.

[Dau79] Joseph Warren Dauben. *Georg Cantor. His Mathematics and Philosophy of the Infinite.* Princeton University Press, 1979.

[Dum82] Michel Dummett. *Wahrheit.* Reclam Stuttgart, 1982.

[EMC$^+$01] Harmut Ehrig, Bernd Mahr, Felix Cornelius, Matrin Große-Rhode, and Philip Zeitz. *Mathematisch-strukturelle Grundlagen der Informatik.* Springer Berlin/Heidelberg, 2nd edition, 2001.

[Fre75] Gottlob Frege. Funktion und Begriff. In Günther Patzig, editor, *Funktion, Begriff, Bedeutung.* Vandenhoeck und Ruprecht Göttingen, 1975.

[Fre07] Gottlob Frege. Begriffsschrift, eine der arithmetischen nachgebildete Formelsprache des reinen Denkens. In *Begriffsschrift und andere Aufsätze.* Georg Olms Verlag Hildesheim, 2007.

[Gro92] Reinhard Grossmann. *The Existence of the World - An Introduction to Ontology.* Routledge London, 1992.

[Her83] Hans Hermes. Zur Begriffschrift und zur Begründung der Arithmetik. In *Gottlob Frege: Nachgelassene Schriften.* Meiner Verlag Hamburg, 1983.

[Hus93] Edmund Husserl. *Logische Untersuchungen.* Max Niemeyer Verlag Tübingen, 1993.

[Kan90] Immanuel Kant. *Kritik der Urteilskraft.* Meiner Hamburg, 1990.

[Lew09] Steffen Lewitzka. $\in_I$*: an intuitionistic logic without Fregean Axiom and with predicates for truth and falsity.* to appear, 2009.

[Mah93] Bernd Mahr. Applications of type theory. In *Proceedings of the International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 343–355. Springer Verlag, 1993.

[Mah97] Bernd Mahr. Gegenstand und Kontext - Eine Theorie der Auffassung. In K. Eyferth, B. Mahr, R. Posner, and F. Wysotzki, editors, *Prinzipien der Kontextualisierung.* 1997. KIT Report 141, Technische Universität Berlin, 1997.

[Mah09] Bernd Mahr. Die Informatik und die Logik der Modelle. *Informatik Spektrum*, 32(3):228–249, 2009.

[Mei02] Alexius Meinong. *Über Annahmen.* Johann Ambrosius Barth Leipzig, 1902.

[ML84] Per Martin-Löf. *Intuitionistic Type Theory.* Bibliopolis Napoli, 1984.

[ML96] Per Martin-Löf. *On he meaning of the Logical Constants and the Justifications of the Logical Laws, Nordic Journal of Philosophical Logic, volume 1, no. 1.* Scandinavian University Press, 1996.

[ML08] Per Martin-Löf. Workshop on *Judgements, Assertions, and Propositions - The Logical Semantics and Pragmatics of Sentences* at TU Berlin, January 11, 2008.

[MSU90] Bernd Mahr, Werner Sträter, and Carla Umbach. *Fundamentals of a theory of types and declarations.* Forschungsbericht, KIT-Report 82, Technische Universität Berlin, 1990.

[Par69] Parmenides. *Vom Wesen des Seienden - Die Fragmente.* Suhrkamp Frankfurt a. M., 1969.

[Rol08] Robin D. Rollinger. *Austrian Phenomenology: Brentano, Husserl, Meinong, and Others on Mind and Object.* Ontos Frankfurt, 2008.

[Rus92] Bertrand Russell. *Theory of Knowledge - The 1913 Manuscript.* Routledge London, 1992.

[Rus00] Bertrand Russell. Über das Kennzeichnen. In *Philosophische und politische Aufsätze.* Reclam Stuttgart, 2000.

[Sea08] John R. Searle. *Sprechakte - Ein sprachphilosophischer Essay.* Suhrkamp Frankfurt a. M., 2008.

[Smi89] Barry Smith. Logic and the Sachverhalt. *The Monist*, 72(1):52–69, Jan. 1989.

[Str92] Werner Sträter. $\in_T$ *Eine Logik erster Stufe mit Selbstreferenz und totalem Wahrheitsprädikat.* KIT-Report 98, 1992. Dissertation, Technische Universität Berlin.

[Sun94] Göran Sundholm. Existence, Proof and Truth-Making: A Perspective on the Intuitionistic Conception of Truth. *Topoi*, 13:117–126, 1994.

[The98] *Theoria - A Swedish Journal of Philosophy*, volume 64, issues 2–3. Wiley Interscience, 1998.

[Tug03] Ernst Tugendhat. *ti kata tinos - Eine Untersuchung zu Struktur und Ursprung Aristotelischer Grundbegriffe.* Alber Symposion Freiburg/München, 4th edition, 2003.

[TW04] Ernst Tugendhat and Ursula Wolf. *Logisch-semantische Propädeutik.* Reclam Stuttgart, 2004.

[Twa77] Kasimir Twardowski. *On the Content and Object of Presentations - A Psychological Investigation.* The Hague: Martinus Nijhoff, 1977. Translated and introduced by R. Grossmann.

[Twa82] Kasimir Twardowski. *Zur Lehre vom Inhalt und Gegenstand der Vorstellungen - Eine psychologische Untersuchung.* Philosophia Verlag München/Wien, 1982.

[vOQ80] Willard van Orman Quine. *Wort und Gegenstand.* Reclam Stuttgart, 1980.

[Wie08] Tina Wieczorek. *On Foundational Frames for Formal Modelling. Sets, ϵ-Sets and a Model of Conception.* Dissertation, Technische Universität Berlin, December 2008.

[Wit67] Ludwig Wittgenstein. *Philosophische Untersuchungen (Philosophical Investigations).* Suhrkamp Frankfurt a. M., 1967.

[Wit73] Ludwig Wittgenstein. *Tractatus logico-philosophicus.* Suhrkamp Frankfurt a. M., 1973.

[WR62] Alfred North Whitehead and Bertrand Russell. *Principia Mathematica.* Cambridge University Press, 1962.

[Zei00] Philip Zeitz. *Parametrisierte $\in_T$-Logik: Eine Theorie der Erweiterung abstrakter Logiken um die Konzepte Wahrheit, Referenz und klassische Negation.* Logos Verlag Berlin, 2000. Dissertation, Technische Universität Berlin, 1999.

Prof. Dr. Bernd Mahr

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin
D-10587 Berlin (Germany)
mahr@cs.tu-berlin.de
http://flp.cs.tu-berlin.de/ma/mahr.html

Bernd Mahr and Hans-Jörg Kreowski were colleagues when they were research associates and assistant professors at TU Berlin, sharing many interests. Although they later departed into different fields of research and followed somewhat different directions of thought, Bernd points out in his article that the feeling of friendship and trust never got lost.

On Teaching Logic and Algebraic Specification

Till Mossakowski

Courses on algebraic specification and logic have been important cornerstones of teaching theoretical computer science for many years. Moreover, algebraic specification and logic are applied in areas like software specification and verification, but also in ontologies and weak artificial intelligence[1], and other areas. During my studies, I myself was greatly influenced by courses on algebraic specification and logic. The logic courses mainly provided a very abstract and dry introduction to the formalities of logic — the motivation for logic needed to have arisen independently of the course. By contrast, Hans-Jörg Kreowski always has carefully motivated his courses on algebraic specification (and other subjects), has brought spirit into concepts by using a graphic and descriptive style of presentation, and activated students by insisting on letting them answer questions, discuss points and solve exercises, with room for developing own ideas (especially within so-called student projects, a specialty of Bremen university). This teaching greatly influenced my choice of research subject.

Dear Hans-Jörg, I wish you all the best for your 60th birthday, and please continue your mixture of brilliant research and excellent teaching even though facing the fact that our university system by far does not encourage and support the latter to the degree actually needed,[2] and also students often are not used to an activating teaching style.

In this work, I will report on some research and some teaching I have done in the context of the Common Algebraic Specification Language (CASL [3,4]). CASL is a common language for algebraic specification that has been initiated by the IFIP working group 1.3 "Foundations of systems specification" (see also the report [1]), which was founded and initially lead by Hans-Jörg Kreowski.

1 First-Order Logic

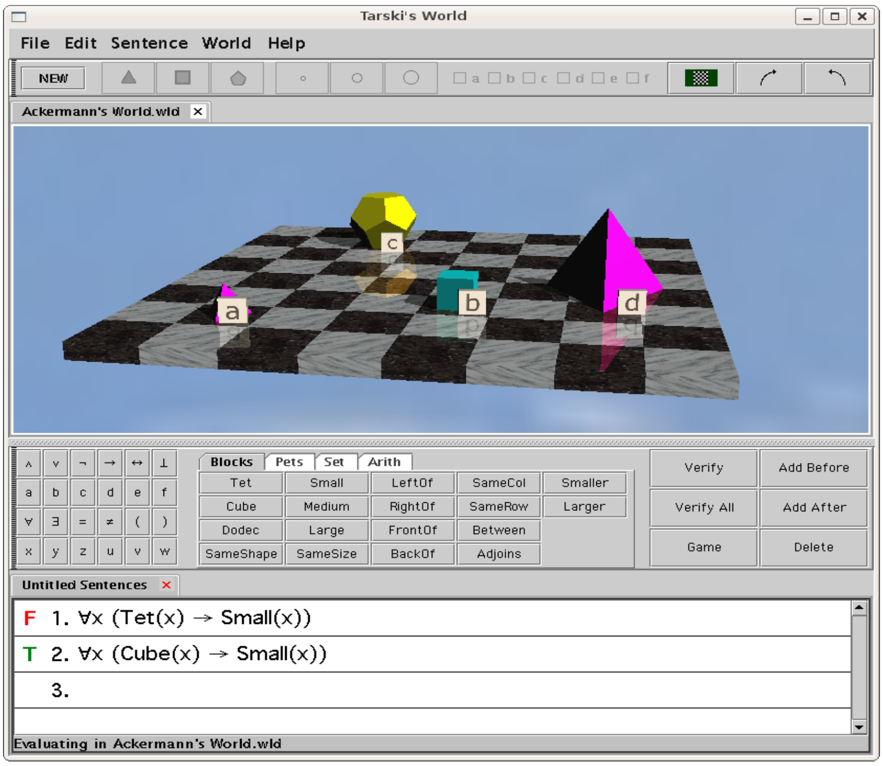
Basically, I regularly teach a course about logic that is quite popular (attended by roughly 100 students) and a course on more specialised subjects usually attended only by smaller groups of students.

1.1 Language, Proof and Logic

For teaching first-order logic, I use the book "Language, proof and logic" [2], abbreviated LPL. The most striking feature of LPL is the use of software tools

[1]Here, *weak* AI is used for systems that solve tasks in specialised domains using heuristics or learning, as opposed to *strong* AI, which aims at passing the Turing test.

[2]As a curiosity: I tried to buy a book about university didactic in the university's book shop — they had no such book directly available, only books about school didactic.


Tarski's World
File Edit Sentence World Help
NEW
Ackermann's World.wld
a
b
c
d
Blocks Pets Set Arith
Tet Small LeftOf SameCol Smaller
Cube Medium RightOf SameRow Larger
Dodec Large FrontOf Between
SameShape SameSize BackOf Adjoins
Verify Add Before
Verify All Add After
Game Delete
Untitled Sentences
Evaluating in Ackermann's World.wld

F 1. ∀x (Tet(x) → Small(x))

T 2. ∀x (Cube(x) → Small(x))

3.

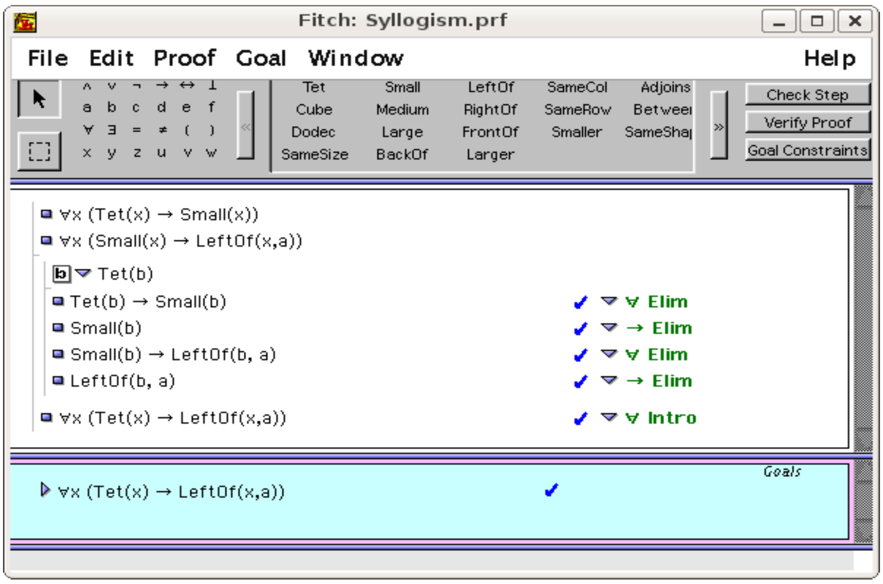
Fig. 1: Evaluating first-order sentences with the program *Tarski's world*.

supporting the students with their own exercises and experiments in logic. This goes as far that a server in Stanford can automatically evaluate some of the students' exercises and give detailed feedback, such that students can revise their solutions. This allows a far better activation of students than with lectures alone — in an ex-cathedra lecture with 100 students, only a small portion of them can actually participate.

However, the usefulness of the software tools should also not be overestimated: it is still very important to have handwritten exercises that are corrected by the teacher, as well as explanations of the students and discussions within the lecture.

In my view, the most important insight of LPL is the following: the notion of first-order structure (or model) is an advanced topic![1] (The same holds for notion of algebra used in algebraic specification.) Instead, LPL largely uses a

[1]It is only treated in part III of the book. Part I is about propositional logic, part II about first-order logic, and part III about advanced topics.


Fitch: Syllogism.prf
File Edit Proof Goal Window Help
∧ ∨ ¬ → ↔ ⊥
a b c d e f
∀ ∃ = ≠ ( )
x y z u v w
Tet Small LeftOf SameCol Adjoins
Cube Medium RightOf SameRow Betwee
Dodec Large FrontOf Smaller SameSha
SameSize BackOf Larger
Check Step
Verify Proof
Goal Constraints
Goals

```
∀x (Tet(x) → Small(x))
∀x (Small(x) → LeftOf(x,a))
  b  Tet(b)
     Tet(b) → Small(b)              ✓  ∀ Elim
     Small(b)                       ✓  → Elim
     Small(b) → LeftOf(b, a)        ✓  ∀ Elim
     LeftOf(b, a)                   ✓  → Elim
∀x (Tet(x) → LeftOf(x,a))           ✓  ∀ Intro

▷ ∀x (Tet(x) → LeftOf(x,a))         ✓
```

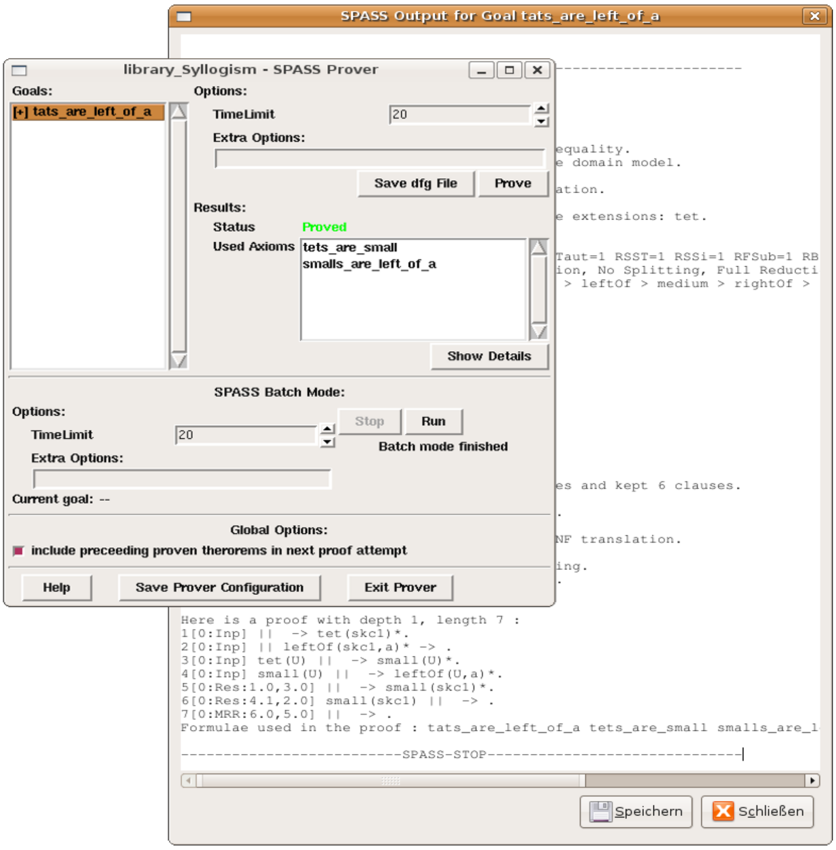
Fig. 2: Sample proof with the program *Fitch*.

fixed interpretation of first-order logic in a blocks world (see Fig. 1 showing a screenshot of the program *Tarski's world*).

Of course, with using a fixed domain of interpretation (carrier set) and fixed interpretation of predicates, one loses much of the "loose specification" approach used in both algebraic specification and logic. However, the essential gain over the traditional approach is that a fixed interpretation is much easier to grasp. Indeed, a useful didactic will proceed from the concrete to the abstract (and not vice versa), and the abstractness of the concept of (carrier) set (and of function and relation) is often underestimated — even if illustrated with useful example carrier sets from computer science like lists, strings or trees.[1] Moreover, fixing the carrier set and interpretation of predicates is not as harmful as it looks: in a blocks world, it is still possible to obtain some degree of looseness by using different configurations of the blocks. Students can then inspect the effect of different configurations on the evaluation of sentences, and use a game, a so-called Henkin-Hintikka game, to understand the evaluation in more detail. Some looseness of course is also essential to understand the concept of logical

[1]Let me further illustrate this point with some anecdotes about the concept of function. Vladimiro Sassone told me that he taught a course on recursive functions. After several weeks, he spent one lecture on students' questions. The first question was: "what is a function?". Michael Kohlhase regularly poses this question in his oral exams, and in spite of him announcing this question, only about 60% of students know the answer.

consequence — another concept that is surprisingly difficult to grasp for many students. The most difficult part to understand is that logical consequence does not imply the truth of the premises — it also holds in cases where the premises are always false.



```
----------------------

equality.
e domain model.

ation.

e extensions: tet.

Taut=1 RSST=1 RSSi=1 RFSub=1 RB
ion, No Splitting, Full Reducti
 > leftOf > medium > rightOf >

es and kept 6 clauses.
.
NF translation.

ing.
.
Here is a proof with depth 1, length 7 :
1[0:Inp] ||  -> tet(skc1)*.
2[0:Inp] || leftOf(skc1,a)* -> .
3[0:Inp] tet(U) ||  -> small(U)*.
4[0:Inp] small(U) ||  -> leftOf(U,a)*.
5[0:Res:1.0,3.0] ||  -> small(skc1)*.
6[0:Res:4.1,2.0] small(skc1) ||  -> .
7[0:MRR:6.0,5.0] ||  -> .
Formulae used in the proof : tats_are_left_of_a tets_are_small smalls_are_l

--------------------------SPASS-STOP------------------------------
```

Fig. 3: Sample proof with HETS and SPASS.

Here, the interplay of Tarski's world with *Fitch* greatly helps: Fitch is a program that can be used for the construction of a natural deduction proof, in case that a logical consequence actually holds, see Fig. 2. In the other cases, Tarski's world can be used to construct countermodels.

LPL offers a great deal of motivation and explanation of the natural deduction calculus (and Fitch) in terms of common natural language arguments. It must be noted though that students more often have difficulties with Fitch than with Tarski's world. The reason seems to be again the level of abstraction: while Tarski's world is about a blocks world that is still close to everyday's experience, Fitch is about proofs that follow certain rules which are quite common in mathematical arguments, but not in everyday's experience. Moreover, students often have difficulties with finding suitable rules to apply in a given situation, or with the development of a proof strategy. Therefore, the development of proof strategies is explicitly discussed in the lecture and supported with numerous exercises. However, I think that this still does not suffice. An interactive dialogue suggesting different strategies or heuristics might help to stimulate more experiments also for those students that do not grasp natural deduction so quickly.

1.2 Hets and State-of-the-Art Provers

This also brings me to another point: the relation of Fitch to state-of-the-art automated and interactive theorem provers. Some students are motivated to conduct larger proofs, but Fitch is not suited for this, since it is not possible to prove lemmas and theorems for later re-use. Here, I use CASL and the Heterogeneous Tool Set HETS [8,7], which offers the connection to a selection of resolution provers (SPASS, Vampire) and tableau provers (Isabelle), as well as to SAT solvers (zChaff, minisat) — all tools that are used in current research. However, these tools of course do not offer the special proof rule provided by Fitch that can be used to derive facts that are specific to the blocks world (this rule is called "AnaCon"). Actually, the rule AnaCon can be simulated with a suitable first-order axiomatisation of the blocks world in CASL. Then proofs can be conducted e.g. with the automated resolution prover SPASS [10]. A drawback is that the output format of resolution proofs is still rather cryptic, since the problem is first translated to clause form. A translation from resolution proofs to natural deduction (using tools like Tramp [5] or Metis [6]) could help here, but one should be careful not to provide an automatic tool that completely discourages students to build their own natural deduction proofs.

2 Structured Specification

While research in algebraic specification started with the application of methods from universal algebra and equational logic to the specification of abstract data types, later the algebraic nature was found more in the powerful constructs that are used to build larger specifications from smaller ones in a modular way. One such construct is the restriction to so-called initial and free models, a quite central but complex notion in the area of algebraic specification. While teaching this notion, I developed the idea to use propositional logic (instead of equational or first-order logic) to illustrate constructs for structuring specification. The advantage is that the logic is so simple that one can really concentrate on the structuring. Moreover, it is possible to display individual models: they are just rows in a truth table. Using this approach, the following subsections explain logical consequence, conservative extensions, and initial/free specifications. The development will be a bit more technical than above, and also will rely on mathematical notation. However, it will be intensively illustrated with results from HETS.

2.1 Logical Consequence

Logical consequence is the central notion of logic (and is also important for algebraic specification): what follows from what? As indicated above, logical consequence is a notion that is difficult to grasp for many students. Hence, with HETS, we provide an easy truth table approach for illustrating this notion.

Definition 1 (Signature). *A* propositional signature Σ *is a set (of propositional symbols, or variables).*

Definition 2 (Sentence). *Given a propositional signature* Σ*, a* propositional sentence *over* Σ *is one produced by the following grammar*

$$\phi ::= p \mid \bot \mid \top \mid \neg\phi \mid \phi \wedge \phi \mid \phi \vee \phi \mid \phi \rightarrow \phi \mid \phi \leftrightarrow \phi$$

with $p \in \Sigma$. *Sen*(Σ) *is the set of all* Σ*-sentences*

Definition 3 (Model). *Given a propositional signature* Σ*, a* Σ-model *(or* Σ*-valuation) is a function in* $\Sigma \rightarrow \{T, F\}$*. Mod*$(\Sigma)$ *is the set of all* Σ*-models.*

A Σ-model M can be extended to

$$M^{\#} : \mathrm{Sen}(\Sigma) \rightarrow \{T, F\}$$

using truth tables.

Definition 4. ϕ *holds in* M *(or* M satisfies ϕ*), written* $M \models_{\Sigma} \phi$ *iff*

$$M^{\#}(\phi) = T$$

Definition 5 (Logical consequence). *Given* $\Gamma \subseteq Sen(\Sigma)$ *and* $\phi \in Sen(\Sigma)$*,* ϕ *is a* logical consequence *of* Γ *(written as* $\Gamma \models \phi$*), if for all* $M \in Mod(\Sigma)$

$$M \models \Gamma \text{ implies } M \models \phi.$$

Example 6. An argument in natural language is tested for validity by translating it into propositional logic.

John plays tennis, if it's a sunny weekend day. If John plays tennis, then Mary goes shopping. It is Saturday. It is sunny. — Mary goes shopping	sunny $\wedge$ weekend $\rightarrow$ tennis tennis $\rightarrow$ shopping saturday sunny saturday $\rightarrow$ weekend — shopping

The set of premises has the sentence *shopping* as a logical consequence

```
logic Propositional

spec Weekend =
  props tennis, shop, sunny, sat, we
  . sunny /\ we => tennis   %(SWT)%
  . tennis => shop          %(TSh)%
  . sat                     %(sat)%
  . sat => we               %(satW)%
  . sunny                   %(sun)%
  . shop                    %(shop)% %implied
end
```

Listing 1: A simple logical consequence

```
Legend:
M = model of the premises
+ = OK, model fulfills conclusion
- = not OK, counterexample for logical consequence
o = OK, premises are not fulfilled, hence conclusion is irrelevant

   || sat | shop | sunny | tennis | we || SWT | TSh | sat | satW | sun || shop
===++=====+======+=======+========+====++=====+=====+=====+======+=====++=====
 o ||   F |    F |     F |      F |  F ||   T |   T |   F |    T |   F ||    F
 o ||   F |    F |     F |      F |  T ||   T |   T |   F |    T |   F ||    F
 o ||   F |    F |     F |      T |  F ||   T |   F |   F |    T |   F ||    F
 o ||   F |    F |     F |      T |  T ||   T |   F |   F |    T |   F ||    F
 o ||   F |    F |     T |      F |  F ||   T |   T |   F |    T |   T ||    F
 o ||   F |    F |     T |      F |  T ||   F |   T |   F |    T |   T ||    F
 o ||   F |    F |     T |      T |  F ||   T |   F |   F |    T |   T ||    F
 o ||   F |    F |     T |      T |  T ||   T |   F |   F |    T |   T ||    F
 o ||   F |    T |     F |      F |  F ||   T |   T |   F |    T |   F ||    T
 o ||   F |    T |     F |      F |  T ||   T |   T |   F |    T |   F ||    T
 o ||   F |    T |     F |      T |  F ||   T |   T |   F |    T |   F ||    T
 o ||   F |    T |     F |      T |  T ||   T |   T |   F |    T |   F ||    T
 o ||   F |    T |     T |      F |  F ||   T |   T |   F |    T |   T ||    T
 o ||   F |    T |     T |      F |  T ||   F |   T |   F |    T |   T ||    T
 o ||   F |    T |     T |      T |  F ||   T |   T |   F |    T |   T ||    T
 o ||   F |    T |     T |      T |  T ||   T |   T |   F |    T |   T ||    T
 o ||   T |    F |     F |      F |  F ||   T |   T |   T |    F |   F ||    F
 o ||   T |    F |     F |      F |  T ||   T |   T |   T |    T |   F ||    F
 o ||   T |    F |     F |      T |  F ||   T |   F |   T |    F |   F ||    F
 o ||   T |    F |     F |      T |  T ||   T |   F |   T |    T |   F ||    F
 o ||   T |    F |     T |      F |  F ||   T |   T |   T |    F |   T ||    F
 o ||   T |    F |     T |      F |  T ||   F |   T |   T |    T |   T ||    F
 o ||   T |    F |     T |      T |  F ||   T |   F |   T |    F |   T ||    F
 o ||   T |    F |     T |      T |  T ||   T |   F |   T |    T |   T ||    F
 o ||   T |    T |     F |      F |  F ||   T |   T |   T |    F |   F ||    T
 o ||   T |    T |     F |      F |  T ||   T |   T |   T |    T |   F ||    T
 o ||   T |    T |     F |      T |  F ||   T |   T |   T |    F |   F ||    T
 o ||   T |    T |     F |      T |  T ||   T |   T |   T |    T |   F ||    T
 o ||   T |    T |     T |      F |  F ||   T |   T |   T |    F |   T ||    T
 o ||   T |    T |     T |      F |  T ||   F |   T |   T |    T |   T ||    T
 o ||   T |    T |     T |      T |  F ||   T |   T |   T |    F |   T ||    T
M+ ||   T |    T |     T |      T |  T ||   T |   T |   T |    T |   T ||    T
```

Listing 2: Truth table for the logical consequence from Listing 1

Note that the formalisation contains an axiom `saturday` $\rightarrow$ `weekend` not present in the informal version. This axiom represents implicit background knowledge. The HETS input syntax for this example is shown in Listing 1.

With HETS, we can now construct the following truth table as shown in Listing 2. The truth table is divided into three parts, using `||`. The first part consists of the signature: all propositional letters are listed. Below the signature, you find all possible models, one per row. The second part consists of the theory (the axioms, also playing the role of *premises* of the argument): for each axiom, its truth value is listed. Only rows containing `T` for every axiom are models of the theory (indicated by an `M`). Finally, the third part contains the proof goal, or *conclusion* of the argument. The conclusion needs to be true for each row that is a model.

A simple non-example of a logical consequence (actually, we omitted the fact that saturday is a weekend day) is shown in Listing 3.

```
spec Weekend2 =
  props tennis, shop, sunny, sat, we
  . sunny /\ we => tennis   %(SWT)%
  . tennis => shop          %(TSh)%
  . sat                     %(sat)%
  . sunny                   %(sun)%
  . shop                    %(shop)% %implied
end
```

Listing 3: Example of a non-consequence

2.2 Conservative Extensions

A theory is *satisfiable*, if it has a model.[1] Satisfiability of theories is quite important for an axiomatic or loose approach to specification: it is easy to introduce unintentional inconsistencies, and an inconsistent (unsatisfiable) specification cannot be realised, hence it does not successfully model an aspect of reality.[2]

Satisfiability of large theories is hard to show. Actually, there are large first-order theories like the SUMO ontology for which satisfiability is an open question — indeed there is a prize set up for proving consistency of SUMO [9]. A modular way to satisfiability is opened up by conservative extensions: in a sense, these transport satisfiability.

[1]In some logics like equational logic, each theory is trivially satisfiable. In these cases, satisfiability should be replaced with satisfiability by a non-trivial model, where the latter is a model that falsifies at least one sentence.

[2]This is different for paraconsistent logics, which however will not be considered here.

```
Legend:
M = model of the premises
+ = OK, model fulfills conclusion
- = not OK, counterexample for logical consequence
o = OK, premises are not fulfilled, hence conclusion is irrelevant

   || sat | shop | sunny | tennis | we || SWT | TSh | sat | sun || shop
===++=====+======+=======+========+====++=====+=====+=====+=====++=====
 o ||   F |    F |     F |      F |  F ||   T |   T |   F |   F ||    F
 o ||   F |    F |     F |      F |  T ||   T |   T |   F |   F ||    F
 o ||   F |    F |     F |      T |  F ||   T |   F |   F |   F ||    F
 o ||   F |    F |     F |      T |  T ||   T |   F |   F |   F ||    F
 o ||   F |    F |     T |      F |  F ||   T |   T |   F |   T ||    F
 o ||   F |    F |     T |      F |  T ||   F |   T |   F |   T ||    F
 o ||   F |    F |     T |      T |  F ||   T |   F |   F |   T ||    F
 o ||   F |    F |     T |      T |  T ||   T |   F |   F |   T ||    F
 o ||   F |    T |     F |      F |  F ||   T |   T |   F |   F ||    T
 o ||   F |    T |     F |      F |  T ||   T |   T |   F |   F ||    T
 o ||   F |    T |     F |      T |  F ||   T |   T |   F |   F ||    T
 o ||   F |    T |     F |      T |  T ||   T |   T |   F |   F ||    T
 o ||   F |    T |     T |      F |  F ||   T |   T |   F |   T ||    T
 o ||   F |    T |     T |      F |  T ||   F |   T |   F |   T ||    T
 o ||   F |    T |     T |      T |  F ||   T |   T |   F |   T ||    T
 o ||   F |    T |     T |      T |  T ||   T |   T |   F |   T ||    T
 o ||   T |    F |     F |      F |  F ||   T |   T |   T |   F ||    F
 o ||   T |    F |     F |      F |  T ||   T |   T |   T |   F ||    F
 o ||   T |    F |     F |      T |  F ||   T |   F |   T |   F ||    F
 o ||   T |    F |     F |      T |  T ||   T |   F |   T |   F ||    F
M- ||   T |    F |     T |      F |  F ||   T |   T |   T |   T ||    F
 o ||   T |    F |     T |      F |  T ||   F |   T |   T |   T ||    F
 o ||   T |    F |     T |      T |  F ||   T |   F |   T |   T ||    F
 o ||   T |    F |     T |      T |  T ||   T |   F |   T |   T ||    F
 o ||   T |    T |     F |      F |  F ||   T |   T |   T |   F ||    T
 o ||   T |    T |     F |      F |  T ||   T |   T |   T |   F ||    T
 o ||   T |    T |     F |      T |  F ||   T |   T |   T |   F ||    T
 o ||   T |    T |     F |      T |  T ||   T |   T |   T |   F ||    T
M+ ||   T |    T |     T |      F |  F ||   T |   T |   T |   T ||    T
 o ||   T |    T |     T |      F |  T ||   F |   T |   T |   T ||    T
M+ ||   T |    T |     T |      T |  F ||   T |   T |   T |   T ||    T
M+ ||   T |    T |     T |      T |  T ||   T |   T |   T |   T ||    T
```

Listing 4: Truth table for the non-consequence from Listing 3

```
spec Sp =
    $\Sigma_1$
    $\Gamma_1$
then
    $\Sigma_\Delta$
    $\Gamma_\Delta$
end
```

```
spec Animals =
    props bird, penguin
    . penguin => bird
then
    prop can_fly
    . penguin => not can_fly
end
```

Listing 5: Theory extensions in CASL.

```
logic Propositional

spec Animal =
  props bird, penguin, living
  . penguin => bird  %(pb)%
  . bird => living   %(bl)%
then %cons
  prop animal
  . bird => animal   %(ba)%
  . animal => living %(al)%
end
```

```

spec Penguin =
  props bird, penguin
  . penguin => bird         %(pb)%
then
  prop can_fly
  . bird => can_fly         %(bc)%
  . penguin => not can_fly  %(pnc)%
end
```

Listing 6: Example of a conservative and a non-conservative extension in CASL

To illustrate the concept, consider the specification in Listing 6. Indeed, to formally underpin this, we introduce some notions that will be central for structured specification:

Definition 7. *Given two signatures Σ_1, Σ_2 a* signature morphism *is a function $\sigma : \Sigma_1 \to \Sigma_2$ (note that signatures are sets).*

Sentences can be translated along signature morphisms:

Definition 8. *A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ induces a* sentence translation *$\sigma : Sen(\Sigma_1) \to Sen(\Sigma_2)$, defined inductively by*

- $\sigma(\bot) = \bot$
- $\sigma(\top) = \top$
- $\sigma(\phi_1 \wedge \phi_2) = \sigma(\phi_1) \wedge \sigma(\phi_2)$
- *etc.*

Models are translated *against* signature morphisms. The intuition is that the translated model $M|_\sigma$ works as follows: interpret a symbol by first translating it along the signature morphism σ and then look up the interpretation in the original model M.

Definition 9. *A signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ induces a* model reduction *$\_|_\sigma$: $Mod(\Sigma_2) \to Mod(\Sigma_1)$. Given $M \in Mod(\Sigma_2)$ i.e. $M : \Sigma \to \{T, F\}$, then $M|_\sigma \in Mod(\Sigma_1)$ is defined as $M|_\sigma(\phi) := M(\sigma(\phi))$ i.e. $M|_\sigma = M \circ \sigma$*

Sentence and model translation interact well with each other:

Theorem 10 (Satisfaction condition). *Given a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$, $M_2 \in Mod(\Sigma_2)$ and $\phi_1 \in Sen(\Sigma_1)$, then:*

$$M_2 \models_{\Sigma_2} \sigma(\phi_1) \text{ iff } M_2|_\sigma \models_{\Sigma_1} \phi_1$$

("truth is invariant under change of notation.")

Definition 11. *A* theory morphism $(\Sigma_1, \Gamma_1) \to (\Sigma_2, \Gamma_2)$ *is a signature morphism $\sigma : \Sigma_1 \to \Sigma_2$ such that for $M_2 \in Mod(\Sigma_2, \Gamma_2)$ we have $M_2|_\sigma \in Mod(\Sigma_1, \Gamma_1)$*

Extensions (written in Casl with the keyword **then**; cf. Listing 5) always lead to a theory morphism (by definition). The semantics of the Casl specification is the theory morphism $\sigma : (\Sigma_1, \Gamma_1) \to (\Sigma_2, \Gamma_2)$, where $\Sigma_2 = \Sigma_1 \cup \Sigma_\Delta$ and $\Gamma_2 = \Gamma_1 \cup \Gamma_\Delta$, such that $\sigma : \Sigma_1 \to \Sigma_2$ is the inclusion.

We are now ready to define conservative extensions:

Definition 12. *A theory morphism $\sigma : T_1 \to T_2$ is* model-theoretically-conservative*, if any $M_1 \in Mod(T_1)$ has a σ-expansion to a T_2-model, that is, a model*

$$M_2 \in Mod(T_2) \text{ with } M_2|_\sigma = M_1.$$

We can now evaluate which of the extensions shown in Listing 6 are conservative. Actually, the first extension is conservative. In the truth table output by Hets (see Listing 7), we can see that each model (marked with an `M` in the leftmost column) has an expansion (marked with an `M` in the column right to the middle).

By contrast, the second extension is *not* conservative: the last model fails to have an expansion, see Listing 8.

The central theorem that allows us to transport satisfiability is the following:

Theorem 13. *If $T_1 \xrightarrow{\sigma_1} T_2 \xrightarrow{\sigma_2} \dots \xrightarrow{\sigma_{n-1}} T_n$ are model-theoretically conservative, and T_1 is satisfiable, then T_n is satisfiable.*

2.3 Initial and Free Specifications

Freeness and cofreeness constraints are a powerful mechanism at the level of structured specifications. They work for any logic. Propositional logic is a good starting point for learning about freeness and cofreeness, since things are much less complicated here when compared with other logics.

Consider the following two somewhat circular statements:

Harry: John tells the truth.
John: If Mary is right, then Harry does not tell the truth.

```
Legend:
M = model of the axioms
+ = OK, has expansion
- = not OK, has no expansion, hence conservativity fails
o = OK, not a model of the axioms, hence no expansion needed

   || bird | living | penguin ||  pb  |  bl  ||   || animal ||   ba   |   al
===++======+========+=========++======+======++===++========++========+=======
M+ ||    F |      F |       F ||    T |    T || M ||      F ||      T |      T
   ||      |        |         ||      |      ||   ||      T ||      T |      F
---++------+--------+---------++------+------++---++--------++--------+-------
 o ||    F |      F |       T ||    F |    T ||   ||        ||        |
---++------+--------+---------++------+------++---++--------++--------+-------
M+ ||    F |      T |       F ||    T |    T || M ||      F ||      T |      T
   ||      |        |         ||      |      || M ||      T ||      T |      T
---++------+--------+---------++------+------++---++--------++--------+-------
 o ||    F |      T |       T ||    F |    T ||   ||        ||        |
---++------+--------+---------++------+------++---++--------++--------+-------
 o ||    T |      F |       F ||    T |    F ||   ||        ||        |
---++------+--------+---------++------+------++---++--------++--------+-------
 o ||    T |      F |       T ||    T |    F ||   ||        ||        |
---++------+--------+---------++------+------++---++--------++--------+-------
M+ ||    T |      T |       F ||    T |    T ||   ||      F ||      F |      T
   ||      |        |         ||      |      || M ||      T ||      T |      T
---++------+--------+---------++------+------++---++--------++--------+-------
M+ ||    T |      T |       T ||    T |    T ||   ||      F ||      F |      T
   ||      |        |         ||      |      || M ||      T ||      T |      T
```

Listing 7: Truth table for a conservative extension from Listing 6

```
Legend:
M = model of the axioms
+ = OK, has expansion
- = not OK, has no expansion, hence conservativity fails
o = OK, not a model of the axioms, hence no expansion needed

   || bird | penguin ||  pb  ||   || can_fly ||   bc   | pnc
===++======+=========++======++===++=========++========+=====
M+ ||    F |       F ||    T || M ||       F ||      T |    T
   ||      |         ||      || M ||       T ||      T |    T
---++------+---------++------++---++---------++--------+-----
 o ||    F |       T ||    F ||   ||         ||        |
---++------+---------++------++---++---------++--------+-----
M+ ||    T |       F ||    T ||   ||       F ||      F |    T
   ||      |         ||      || M ||       T ||      T |    T
---++------+---------++------++---++---------++--------+-----
M- ||    T |       T ||    T ||   ||       F ||      F |    T
   ||      |         ||      ||   ||       T ||      T |    F
```

Listing 8: Truth table for a non-conservative extension from Listing 6

```
spec Liar0 =
    prop mary
    props harry, john
    . harry => john                %(whenjohn)%
    . john => (mary => not harry) %(whenharry)%
then %implies
  . harry %(harry)%
  . john  %(john)%
  . mary  %(mary)%
  . not harry %(notharry)%
  . not john  %(notjohn)%
  . not mary  %(notmary)%
end
```

Listing 9: A circular set of statements

```
Legend:
M = model of the premises
+ = OK, model fulfills conclusion
- = not OK, counterexample for logical consequence
o = OK, premises are not fulfilled, hence conclusion is irrelevant

   || harry | john | mary || whenjohn | whenharry || harry
===++=======+======+======++==========+===========++======
M- ||     F |    F |    F ||        T |         T ||     F
M- ||     F |    F |    T ||        T |         T ||     F
M- ||     F |    T |    F ||        T |         T ||     F
M- ||     F |    T |    T ||        T |         T ||     F
 o ||     T |    F |    F ||        F |         T ||     T
 o ||     T |    F |    T ||        F |         T ||     T
M+ ||     T |    T |    F ||        T |         T ||     T
 o ||     T |    T |    T ||        T |         F ||     T
```

Listing 10: Truth table for the circular statements from Listing 9

Let us formalise these statements and look at the logical consequences. We introduce three propositions telling us whether Harry, John, resp. Mary tell the truth.

Actually, when calling HETS with the truth table prover, the first goal cannot be proved, see Listing 10.

The other goals cannot be proved either. So this theory cannot decide the truth of the propositional letters, and it leaves open whether Harry, John or Mary tell the truth or lie, and indeed, we have five possible cases (indicated by the five models, i.e. those rows marked with `M` in Listing 10). A semantics that admits many possible interpretations and only constrains them by logical formulas is called *open world semantics*.

By contrast, a *closed world semantics* assumes some default, e.g. any propositional letter whose truth value cannot be determined is assumed to be false. Indeed, *free* or *initial semantics* imposes this kind of constraints. As a prerequisite, we need to define a partial order on propositional models:

Definition 14. *Given a propositional signature Σ and two Σ-models M_1 and M_2, then $M_1 \leq M_2$ if $M_1(p) = T$ implies $M_2(p) = T$ for all $p \in \Sigma$.*

Then, a free (or initial) specification, written $\texttt{free}\{SP\}$, selects the least model of a specification:

$$\mathrm{Mod}(\texttt{free}\{SP\}) = \{M \in \mathrm{Mod}(SP) \mid M \text{ least model in } \mathrm{Mod}(SP)\}$$

Note that a least model need not exist; in this case, the model class is empty, hence the free specification inconsistent. Coming back to our example, have a look at Listing 11. With the HETS truth table prover, we now get the truth table in Listing 12. That is, Harry, John and Mary all are lying! Actually, we are not forced by the specification to think that they tell the truth, so by minimality of the initial model, the propositional letters are all assigned false.

```
spec Liar1 =
  free {
    prop mary
    props harry, john
    . harry => john             %(whenjohn)%
    . john => (mary => not harry) %(whenharry)%
   }
then %implies
  . not harry %(notharry)%
  . not john  %(notjohn)%
  . not mary  %(notmary)%
end
```

Listing 11: Closed world assumption, specified as a free extension

```
    || harry | john | mary || notharry | notjohn | free || notmary
===++=======+======+======++==========+=========+======++========
M+  ||     F |    F |    F ||        T |       T |    T ||       T
 o  ||     F |    F |    T ||        T |       T |    F ||       F
 o  ||     F |    T |    F ||        T |       F |    F ||       T
 o  ||     F |    T |    T ||        T |       F |    F ||       F
 o  ||     T |    F |    F ||        F |       T |    F ||       T
 o  ||     T |    F |    T ||        F |       T |    F ||       F
 o  ||     T |    T |    F ||        F |       F |    F ||       T
 o  ||     T |    T |    T ||        F |       F |    F ||       F
```

Listing 12: Truth table for the specification of Listing 11

Of course, the assumption that propositional letters are false by default is somewhat arbitrary. We could have taken the opposite assumption. Indeed, this exactly is what final (or cofree) specifications do, see Listing 13. However, no greatest model exists in this case, hence the cofree specification is inconsistent, as shown in Listing 14.

$$\mathrm{Mod}(\texttt{cofree}\{SP\}) = \{M \in \mathrm{Mod}(SP) \,|\, M \text{ greatest model in } \mathrm{Mod}(SP)\}$$

```
spec Liar2 =
  cofree {
    prop mary
    props harry, john
    . harry => john             %(whenjohn)%
    . john => (mary => not harry) %(whenharry)%
   }
then %implies
  . false %(false)%
end
```

Listing 13: Closed world assumption, specified as a cofree extension

```
   || harry | john | mary || whenjohn | whenharry | cofree || false
===++=======+======+======++==========+===========+========++======
 o ||     F |    F |    F ||        T |         T |      F ||     F
 o ||     F |    F |    T ||        T |         T |      F ||     F
 o ||     F |    T |    F ||        T |         T |      F ||     F
 o ||     F |    T |    T ||        T |         T |      F ||     F
 o ||     T |    F |    F ||        F |         T |      F ||     F
 o ||     T |    F |    T ||        F |         T |      F ||     F
 o ||     T |    T |    F ||        T |         T |      F ||     F
 o ||     T |    T |    T ||        T |         F |      F ||     F
```

Listing 14: Truth table for the specification of Listing 13

We can also mix the open and closed world assumptions. Assume that we want to be unspecific about Mary, but use closed world assumption for Harry and John, see Listing 15.

The semantics is as follows:

$$\begin{aligned}&\mathrm{Mod}(SP_1\ \texttt{then}\ \texttt{free}\{SP_2\}) =\\ &\quad \{M \in \mathrm{Mod}(SP_1\ \texttt{then}\ SP_2) \mid\\ &\qquad M \text{ is the least model in } \{M' \in \mathrm{Mod}(SP_1\ \texttt{then}\ SP_2) \mid M|_\sigma = M'|_\sigma\}\ \}\end{aligned}$$

and as a result, we obtain that both Harry and John lie (independently of what Marry concerns!), see Listing 16.

The dual concept is cofreeness with mixed open and closed world semantics, see Listing 17. Also the semantics is obtained by dualising:

```
spec Liar3 =
  prop mary
then
  free {
    props harry, john
    . harry => john             %(whenjohn)%
    . john => (mary => not harry) %(whenharry)%
   }
then %implies
  . not harry %(harry)%
  . not john  %(john)%
end
```

Listing 15: Mixed open world and closed world semantics using free

```
   || harry | john | mary || whenjohn | whenharry | free || harry
===++=======+======+======++==========+===========+======++======
M+ ||     F |    F |    F ||        T |         T |    T ||     T
M+ ||     F |    F |    T ||        T |         T |    T ||     T
 o ||     F |    T |    F ||        T |         T |    F ||     T
 o ||     F |    T |    T ||        T |         T |    F ||     T
 o ||     T |    F |    F ||        F |         T |    F ||     F
 o ||     T |    F |    T ||        F |         T |    F ||     F
 o ||     T |    T |    F ||        T |         T |    F ||     F
 o ||     T |    T |    T ||        T |         F |    F ||     F
```

Listing 16: Truth table for the specification of Listing 15

```
spec Liar4 =
  prop mary
then
  cofree {
    props harry, john
    . harry => john             %(whenjohn)%
    . john => (mary => not harry) %(whenharry)%
   }
then %implies
  . harry \/ mary %(harrymary)%
  . john  %(john)%
end
```

Listing 17: Mixed open world and closed world semantics using cofree

$$\begin{aligned}&\mathrm{Mod}(SP_1\ \texttt{then cofree}\{SP_2\}) =\\ &\quad \{M \in \mathrm{Mod}(SP_1\ \texttt{then}\ SP_2) \mid\\ &\qquad M \text{ is the greatest model in } \{M' \in \mathrm{Mod}(SP_1\ \texttt{then}\ SP_2) \mid M|_\sigma = M'|_\sigma\}\ \}\end{aligned}$$

The result in the example is that John tells the truth, and at least either of Harry and Mary as well, see Listing 18.

```
   || harry | john | mary || whenjohn | whenharry | cofree || harrymary
===++=======+======+======++==========+===========+========++==========
 o ||     F |    F |    F ||        T |         T |      F ||         F
 o ||     F |    F |    T ||        T |         T |      F ||         T
 o ||     F |    T |    F ||        T |         T |      F ||         F
M+ ||     F |    T |    T ||        T |         T |      T ||         T
 o ||     T |    F |    F ||        F |         T |      F ||         T
 o ||     T |    F |    T ||        F |         T |      F ||         T
M+ ||     T |    T |    F ||        T |         T |      T ||         T
 o ||     T |    T |    T ||        T |         F |      F ||         T
```

Listing 18: Truth table for the specification of Listing 17

3 Conclusion

The overall picture is as follows: typically, I start with a course on first-order logic as described in Sect. 1, followed by a more special course on structuring and institutions, following Sect. 2. The second course starts with propositional logic, which keeps the examples simple, and then proceeds to description logics (used for ontologies and semantic web) and finally again to first-order logic.

Teaching algebraic specification and logic can really be fun, and there is much room for developing better ideas and tools supporting this. Feedback and improvements are welcome!

References

1. E. Astesiano, H.-J. Kreowski, and B. Krieg-Brückner. *Algebraic Foundations of Systems Specification.* Springer, 1999.
2. J. Barwise and J. Etchemendy. *Language, proof and logic.* CSLI publications, 2002.
3. Michel Bidoit and Peter D. Mosses. CASL *User Manual*, volume 2900 of *LNCS (IFIP Series).* Springer, 2004. With chapters by T. Mossakowski, D. Sannella, and A. Tarlecki.
4. CoFI (The Common Framework Initiative). CASL *Reference Manual*, volume 2960 of *LNCS (IFIP Series).* Springer, 2004.
5. Andreas Meier. System description: TRAMP: Transformation of machine-found proofs into ND-proofs at the assertion level. In David A. McAllester, editor, *Automated Deduction - CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 460–464. Springer, 2000.

6. Jia Meng, Claire Quigley, and Lawrence C. Paulson. Automation for interactive proof: First prototype. *Information and Computation*, 204(10):1575–1596, 2006.
7. T. Mossakowski. Heterogeneous specification and the heterogeneous tool set. Habilitation thesis, University of Bremen, 2005.
8. Till Mossakowski, Christian Maeder, and Klaus Lüttich. The Heterogeneous Tool Set. In Orna Grumberg and Michael Huth, editors, *TACAS 2007*, volume 4424 of *Lecture Notes in Computer Science*, pages 519–522. Springer-Verlag Heidelberg, 2007.
9. Adam Pease, Geoff Sutcliffe, Nick Siegel, and Steven Trac. The annual SUMO reasoning prizes at CASC. In Boris Konev, Renate A. Schmidt, and Stephan Schulz, editors, *Proceedings of the First International Workshop on Practical Aspects of Automated Reasoning, Sydney, Australia, August 10-11, 2008*, volume 373 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.
10. C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobalt, and D. Topic. SPASS version 2.0. In Andrei Voronkov, editor, *Automated Deduction – CADE-18*, LNCS 2392, pages 275–279. Springer-Verlag, 2002.

Dr. Till Mossakowski
DFKI GmbH Bremen
Safe & Secure Cognitive Systems
Enrique-Schmidt-Str. 5
D-28359 Bremen (Germany)
Till.Mossakowski@dfki.de
http://www.dfki.de/sks/till

During his studies, Till Mossakowski took several courses in Theoretical Computer Science held by Hans-Jörg Kreowski. Moreover, he was a member of his group from 1993 to 1998. Hans-Jörg also supervised Till's diploma and doctoral theses, and was an examiner of his habilitation thesis.

Algebraic Model Checking

Peter Padawitz

Abstract. Several more or less algebraic approaches to model checking are presented and compared with each other with respect to their range of applications and their degree of automation. All of them have been implemented and tested in our Haskell-based formal-reasoning system Expander2. Besides realizing and integrating state-of-the art proof and computation rules the system admits rarely restricted specifications of the models to be checked in terms of rewrite rules and functional-logic programs. It also offers flexible features for visualizing and even animating models and computations. Indeed, this paper does not present purely theoretical work. Due to the increasing abstraction potential of programming languages like Haskell the boundaries between developing a formal system and implementing it or making it 'user-friendly' as well as between systems developed in different communities become more and more obsolete. The individual topics discussed in the paper reflect this observation.

1 Introduction

Model checking means proving properties of labelled or unlabelled transition systems (TRS). Modal, temporal or dynamic logics have been developed to formalize the properties and provide methods for proving them (see e.g. [4, 13, 26]). In contrast to classical predicate logic, modal logics hide the relations (here: the transition systems) they are talking about. Translations of the latter into the former are well-known (see e.g. [1, 18]), but did not affect very much the direction of research in model checking. With the invention of *coalgebraic* logics (see e.g. [14, 25, 15, 8, 2]) the direction of translation is reversed: these logics generalize the 'relation-hiding' concept of modal logics from merely unstructured states and transitions to arbitrary *destructor-based* types and thus open up alternatives to classical predicate-logic-based data type verification. Moreover, the use of coalgebraic concepts reveals the intrinsic algebraic flavor of modal logics (usually called its *global* semantics): their formulas denote relations; the logical operators (including fixpoint operators!) are functions building relations from relations. The underlying data are either states (elements of a destructor-based type) or paths (which also form a destructor-based type).

We have investigated and implemented in our proof assistant Expander2 [20, 21, 22] four approaches to model checking. The first one may be called purely algebraic because proving a formula boils down to its complete *evaluation.* In the second one, formulas are proved by solving sets of regular equations represented by *data flow graphs.* The third technique uses *simplification* rules and must accompany the first one if, for instance, the underlying type has infinitely many elements (such as the set of paths of a TRS). The fourth method applies *co/Horn logic*, extends the others by powerful inference rules (mainly *parallel co/resolution* and *incremental co/induction*) and thus imposes the fewest restrictions on the formulas to be proved. On the other hand, this technique requires more manual control of the proof process than the others.

For lack of space the present paper skips the data flow approach. The other methods are illustrated mainly with a couple of axiomatic specifications of small Kripke structures and the verification of properties given by *path formulas*. All model representations and proof records given here were generated by Expander2. To a great extent, Expander2 specifications follow the syntax of the functional programming language Haskell (see haskell.org) with which we assume a little familiarity. We also use Haskell for some definitions that involve data structures like lists or trees. Neither a purely set-theoretical notation nor an - unfortunately still prevailing - imperative syntax can cope with the elegance and adequacy of Haskell.

Although it is long ago, the extremely inspiring work with Hans-Jörg Kreowski (and my supervisors Hartmut Ehrig and Dirk Siefkes) at the computer science department of the Technical University of Berlin, lasting from 1974 to 1983, have influenced the direction of my research over the entire subsequent 25 years. We worked in three areas: automata theory, graph grammars and algebraic software specification. In all of them, constructions and methods from universal algebra played the key rôle. My additional work on Horn logic and rewrite systems was also led by the algebraic viewpoint. Last not least, graph grammar concepts left their mark on the treatment of term graphs in Expander2.

2 Kripke structures in Expander2

Since we want to use the same techniques for several variants of transition systems and modal logics, the following definitions take into account deterministic *and* nondeterministic, labelled *and* unlabelled systems as well as state *and* path formulas:

A **Kripke structure** $K = (St, At, Lab, \rightarrow, val, valL)$ consists of a set St of **states**, a set At of **atoms**, a set Lab of **labels** (actions, input, output, etc.), a **transition relation** $\rightarrow\ \subseteq St \times St$ or $\overset{-}{\rightarrow}\ \subseteq St \times Lab \times St$ and **state valuations** $val \subseteq At \times St$ and $valL \subseteq At \times Lab \times St$. If Lab is nonempty, $\rightarrow$ denotes $\cup\{\overset{lab}{\rightarrow} \mid lab \in Lab\}$.

Let $s \in St$, $lab \in Lab$ and $sts \cup sts' \subseteq St$. $sucs(s) = \{s' \in St \mid s \rightarrow s'\}$ and $sucsL(s, lab) = \{s' \in St \mid s \xrightarrow{lab} s'\}$ denote the sets of all (direct) successors of s resp. all successors of s after input/execution of lab.

$$path(K) = \{p \in St^{\mathbb{N}} \mid \forall\, i \in \mathbb{N} : p_i \rightarrow p_{i+1} \vee (sucs(p_i) = \emptyset \wedge p_i = p_{i+1})\}$$

denotes the set of *paths* of K. Given a function $f : St \to \mathcal{P}(St)$,

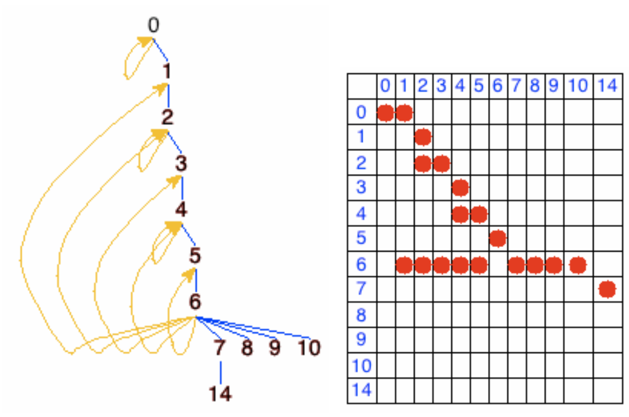
$$\begin{array}{rcl} imgsShares(sts)(f)(sts') &=& \{s \in sts \mid f(s) \cap sts' \neq \emptyset\}, \\ imgsSubset(sts)(f)(sts') &=& \{s \in sts \mid f(s) \subseteq sts'\} \end{array}$$

denote the sets of states $s \in sts$ such that at least one resp. all f-images of s are in sts'. Expander2 admits the specification of Kripke structures in terms of rewrite rules (axioms for `->`) as in the following example. It is small and not very practical, but involves a couple of frequently used functional or logical operators.

```
-- TRANS
constructs:  less SAT                   -- constructors
defuncts:    inits states atoms drawSF -- defined functions
fovars:      n x y                      -- first-order variables
axioms:      inits == [0]                                 &
             atoms == map(less)[0..10]                    &
             (n < 6 & n `mod` 2 = 0 ==> n -> n<+>n+1)     &
             (n < 6 & n `mod` 2 =/= 0 ==> n -> n+1)       &
             6 -> branch$[1,3,5]++[7..10]++[4,2]          &
             7 -> 14                                      &
             less(x) -> branch$filter(rel(y,y<x))$states  &
             drawSF == wtree$fun(SAT(x),rframe$text$x,
                                 x,x)
```

After the specification has been entered and the button *build Kripke model* has been pushed, Expander2 constructs the set `states` of states, the transition relation and the state valuation of the model from the axioms for the binary predicate `->`. For instance, `states` is the set of terms that are reachable from `inits` via the transitive closure of `->`. `&` and `|` denote conjunction resp. disjunction. Equational axioms involving `==` are used as simplification rules (see below). `<+>` and `branch` are constructors for building sets of successor states. The apply-operator `$` and the list functions `map`, `++` and `filter` are interpreted as in Haskell. `fun` and `rel` are the λ-abstraction operators for functions resp. relations. For instance, `fun(p,t,q,u)` denotes the function that, when applied to `v`, yields the corresponding instance of `t` resp. `u` if `v` matches `p` resp. `q`.

`wtree(f)(t)` turns each node n of the term `t` into a graphical widget by applying `f` to the term representation (!) of n. In the `drawSF` axiom of `TRANS`, nodes matching `SAT(x)` are framed by a rectangle, others are (re-)turned into their string representation. The result of simplifying `wtree(f)(t)` is interpreted by the painter module of Expander2 (see, e.g., Fig. 3). Another graphical interpreter of Expander2 turns binary or ternary relations into matrices:



	0	1	2	3	4	5	6	7	8	9	10	14
0	●	●										
1			●									
2			●	●								
3					●							
4					●	●						
5							●					
6		●	●	●	●	●		●	●	●	●	
7												●
8												
9												
10												
14												

Fig. 1. The term (graph) representing the transition relation of TRANS *and its interpretation by the matrix interpreter of Expander2*

The solver module of Expander2 always produces or transforms term graphs like the one on the left-hand side of Fig. 1. Basically, term graphs are trees, but they may involve additional edges (those with tips). The solver module computes further term representations of a binary or ternary relation: a list of pairs resp. triples and a conjunction of regular equations (see Fig. 2).

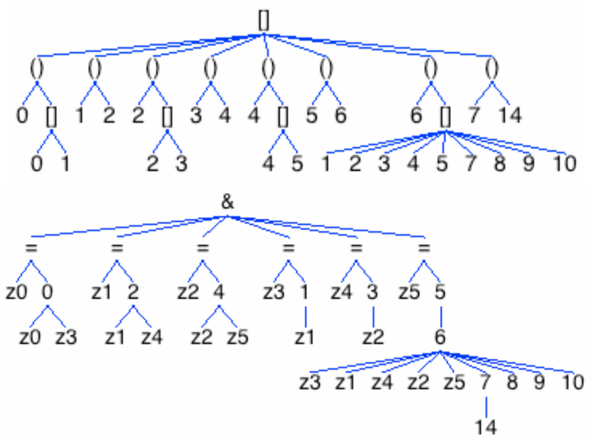


Fig. 2. A list of pairs and a conjunction of regular equations representing the transition relation of TRANS

3 Modal logic and algebra

Let *Var* be a set of variables denoting sets of states or paths (sequences of states). The words generated from sf resp. pf by the following context-free rules are called **state formulas** resp. **path formulas**: Let $at \in At$, $lab \in Lab$

and $x \in Var$.

$$\begin{array}{rcl}
sf & \rightarrow & at \mid true \mid false \mid \neg sf \mid sf \vee sf \mid sf \wedge sf \mid sf \Rightarrow sf \\
(1)\ sf & \rightarrow & EX\ sf \mid AX\ sf \mid \langle lab \rangle sf \mid [lab]sf \\
(2)\ sf & \rightarrow & x \mid \mu x.sf \mid \nu x.sf \\
sf & \rightarrow & EF\ sf \mid AF\ sf \mid EG\ sf \mid AG\ sf \mid sf\ EU\ sf \mid sf\ AU\ sf \\
pf & \rightarrow & at \mid true \mid false \mid \neg pf \mid pf \vee pf \mid pf \wedge pf \mid pf \Rightarrow pf \\
(3)\ pf & \rightarrow & next\ pf \mid \langle lab \rangle pf \mid [lab]pf \\
(4)\ pf & \rightarrow & x \mid \mu x\ pf \mid \nu x\ pf \\
pf & \rightarrow & F\ pf \mid G\ pf \mid pf\ U\ pf
\end{array}$$

Some of the above operators are subsumed by others. This is intended because we favor *natural* deduction where the user is allowed to formalize conjectures as adequately as possible. The reduction to a minimal set of operators should be left to the model checker. Ours will turn all formulas into equivalent ones that consist of propositional, next-step ((1) resp. (3)) and fixpoint operators ((2) resp. (4)).

Like every context-free grammar the above one defines an algebraic **signature** $\Sigma = (PS, S, OP)$ with a set PS of *primitive sorts* (here: at, lab and x), a set S of further sorts, one for each nonterminal of the grammar, and a set OP of operators, one for each rule of the grammar: a rule $A \rightarrow w$ becomes an operator of type $v \rightarrow A$ where v is the product of the nonterminals of w. In the above case, Σ-terms represent formulas, and proving the latter means evaluating the former with respect to a suitable interpretation of Σ, i.e. a **Σ-algebra**, say A.

Each sort $s \in PS \cup S$ is interpreted by a 'carrier' set s^A and each operator f by a function f^A whose domain and range comply with the interpretation of the sorts involved in the type of f. The nature of primitive sorts is to have the same interpretation in every Σ-algebra A. Hence at, lab and x are always interpreted as the given sets At, Lab and Var of atoms, labels and variables, respectively. The interpretation of sf and pf reflects what is often called the a *global semantics* of modal logic:

$$sf^A = (Var \rightarrow \mathcal{P}(St)) \rightarrow \mathcal{P}(St)$$
$$pf^A = (Var \rightarrow \mathcal{P}(path(K))) \rightarrow \mathcal{P}(path(K))$$

In Σ, each atom at becomes a constant of sort sf and also a constant of sort pf. Both fixpoint operators (μ and ν) have the types $Var \times sf \rightarrow sf$ and $Var \times pf \rightarrow pf$. Analoguous *binding* operators occur in other term languages as well, e.g., the *abstraction* and least-fixpoint operators λ resp. μ for building higher-order functions or the quantification operators $\forall$ and $\exists$ that come with an algebraic view on predicate logic.

Fixpoint operators are the main model builders. Be it single objects (including functions of arbitrary order), types (sets of objects) or relations (predicates) of arbitrary arity, whatever cannot be constructed by simply combining given objects (resp. sets) conjunctive- or disjunctively, is defined as a solution of a system of regular equations between variables on the left- and terms/formulas on the right-hand side, i.e. as a fixpoint of the function induced by the equations. From the classical theory of recursive functions via the semantics of logic programming languages up to domain theory and universal co/algebra, fixpoints provide the link between description, computation and proof in all these approaches.

The existence of a fixpoint requires the monotonicity of the functions used in the equations to be solved. Its stepwise constructability requires the stronger property of (upward or downward) continuity. In the case of a modal formula φ, monotonicity is ensured if each free occurrence of $x \in Var$ in φ has *positive polarity*, i.e. the number of negations on the path from the binder of x (μ or ν) to the occurrence is even. Continuity is guaranteed if, in addition to the monotonicity requirement, the transition relation is *image finite*, i.e. for all $s \in St$ and $lab \in Lab$, $sucs(s)$ resp. $sucsL(lab)(s)$ is finite. Hence, if St is finite, the global semantics of a modal formula is stepwise computable if all free variable occurrences in φ have positive polarity.

Given a Kripke structure K, the above interpretations of sf and pf extend to a Σ-algebra, called the **modal algebra over** K: Let $s \in St$, $lab \in Lab$, $\varphi, \psi \in sf^A \cup pf^A$, $b : Var \to \mathcal{P}(St)$ and $c : Var \to \mathcal{P}(path(K))$.

$$\begin{array}{ll}
at^A(b) & =_{def} val(at) \\
at^A(c) & =_{def} \{p \in path(K) \mid p_0 \in val(at)\} \\
true^A(b) & =_{def} St \\
false^A(b) & =_{def} \emptyset \\
\neg^A(\varphi)(b) & =_{def} St \setminus \varphi(b) \\
(\varphi \vee^A \psi)(b) & =_{def} \varphi(b) \cup \psi(b) \\
(\varphi \wedge^A \psi)(b) & =_{def} \varphi(b) \cap \psi(b) \\
\varphi \Rightarrow^A \psi & =_{def} \neg^A(\varphi) \vee^A \psi \\
EX^A(\varphi) & =_{def} imgsShares(St)(sucs) \circ \varphi \\
AX^A(\varphi) & =_{def} imgsSubset(St)(sucs) \circ \varphi \\
\langle lab \rangle^A(\varphi) & =_{def} imgsShares(St)(sucsL(lab)) \circ \varphi \\
[lab]^A(\varphi) & =_{def} imgsSubset(St)(sucsL(lab)) \circ \varphi \\
x^A(b) & =_{def} b(x) \\
next^A(\varphi)(c) & =_{def} \{p \in path(K) \mid \lambda i.p_{i+1} \in \varphi(c)\} \\
(\mu x)^A(\varphi)(b) & =_{def} \mathtt{up}(\varphi(\lambda y.b[y/x]))(\emptyset) \\
(\nu x)^A(\varphi)(b) & =_{def} \mathtt{down}(\varphi(\lambda y.b[y/x]))(St)
\end{array}$$

$f[a/x]$ denotes an update of (the valuation or substitution) f: $f[a/x](x) = a$ and for all $y \neq a$, $f[a/x](y) = f(y)$.

The synonymous operators on path formulas are interpreted analogously: just replace the state valuation b by the path valuation c. The functions `up` and `down` are defined (in Haskell) as follows:

```
up, down :: Eq a => ([a] -> [a]) -> [a] -> [a]
up f   = g where g s = if all (‘elem‘ s) fs then s else g fs
                       where fs = f s
down f = g where g s = if all (‘elem‘ fs) s then s else g fs
                       where fs = f s
```

They transform a finite set by repeatedly applying f until it does not change any more. If applied to $s = \emptyset$ resp. $s = St$ and provided that St is finite, the iteration terminates and—by Kleene's fixpoint theorem—return the least resp. greatest solution of the equation $x = \varphi$ in $\mathcal{P}(St)$.

All operators of Σ that are not interpreted directly in the modal algebra over K can be reduced to fixpoints:

$$\begin{array}{lll}
EF(\varphi) & = \mu x(\varphi \vee EX(x)) & \textit{finally} \\
AF(\varphi) & = \mu x(\varphi \vee (EX(\mathit{true}) \wedge AX(x))) & \\
EG(\varphi) & = \nu x(\varphi \wedge (AX(\mathit{false}) \vee EX(x))) & \textit{generally} \\
AG(\varphi) & = \nu x(\varphi \wedge AX(x)) & \\
\varphi\ EU\ \psi & = \mu x(\psi \vee (\varphi \wedge EX(x))) & \textit{until} \\
\varphi\ AU\ \psi & = \mu x(\psi \vee (\varphi \wedge AX(x))) & \\
F(\varphi) & = \mu x(\varphi \vee \mathit{next}(x)) & \textit{finally} \\
G(\varphi) & = \nu x(\varphi \wedge \mathit{next}(x)) & \textit{generally} \\
\varphi\ U\ \psi & = \mu x(\psi \vee (\varphi \wedge \mathit{next}(x))) & \textit{until}
\end{array}$$

Provided that the Kripke structure K has only finitely many states, each state formula φ can be completely evaluated in the modal algebra over K. For this purpose Expander2 derives K from a specification like `TRANS` and thus makes the following simplification rules applicable to state formulas φ resp. state sets *sts*:

State formula evaluation

$$\frac{\varphi(s)}{\mathit{True}}\ s \in \varphi^A \qquad \frac{\varphi(s)}{\mathit{False}}\ s \in St \setminus \varphi^A \qquad \frac{\mathit{sols}(\varphi)}{\varphi^A} \quad (1)$$

$$\frac{\mathit{embed}(\mathit{sts})}{\textit{transition graph with each state } s \in \mathit{sts} \textit{ replaced by } SAT(s)} \quad (2)$$

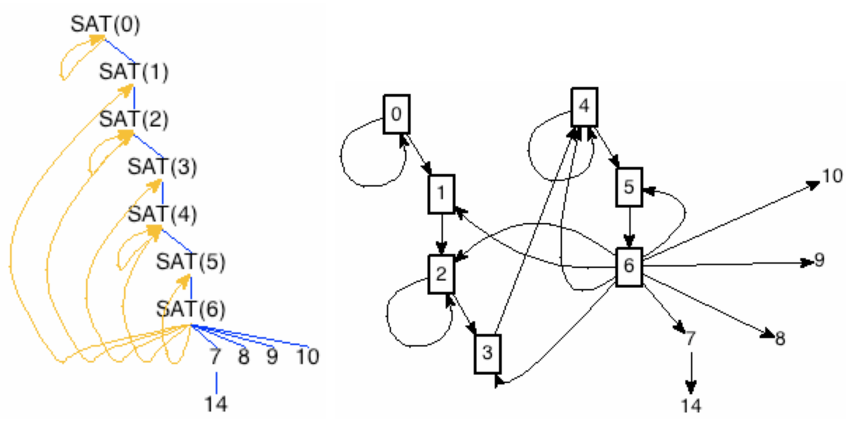


Fig. 3. The result of applying (1), (2) and the function `drawSF` *of* `TRANS` *to* `sols$EF$atom$less$4`

4 Model checking by simplification

A path formula like $\forall\, pa : \varphi(pa)$ quantifies over the *infinite* set of paths of the underlying Kripke structure K and thus cannot be proved by simply evaluating it in the modal algebra over K: the implementation of the fixpoint operators μ and ν with the functions `upWith` and `downWith` will not terminate. However, as fixpoint operators are ubiquitous in model design, so are the key proof rules *induction*, *coinduction* and *expansion* for properties of a fixpoint, say $a = (a_1, \dots, a_n)$. If a solves the equation $(x_1, \dots, x_n) = t(x_1, \dots, x_n)$, expanding a term or formula φ means replacing all occurrences of a (or components thereof) in φ by (the corresponding projections on) $t(a)$. Expansion is sound for all solutions of the equation, induction and coinduction only for the least resp. greatest one.

Expansion Let *op* be a fixpoint operator, $u = (t_1, \dots, t_n)$ and $1 \leq i \leq n$.

$$\frac{op\ x_1 \dots x_n.t}{t[\pi_i(op\ x_1 \dots x_n.t)/x_i \mid 1 \leq i \leq n]} \qquad \frac{\pi_i(op\ x_1 \dots x_n.u)}{t_i[\pi_j(op\ x_1 \dots x_n.u)/x_j \mid 1 \leq j \leq n]}$$

π_i, $1 \leq i \leq n$, denotes the projection of an n-tuple on its i-th component. In the case of unary fixpoints (like the modal operators μ and ν), projections do not occur and we only need the first rule. In general, non-unary fixpoints arise from mutually recursive definitions of several functions or relations.

For reducing the danger of non-termination Expander2 applies expansion rules only to formulas that lack redices for other simplification rules. The simplifier traverses a formula tree depthfirst (leftmost-outermost) or breadthfirst (parallel-outermost) when searching for the next rule redex. The strategy of parallel-outermost simplification that postpones expansion steps as far as possible is a fixpoint strategy, i.e. terminates whenever *any* strategy terminates [16]. This suggests why the evaluation of path formulas in the modal algebra may not terminate: evaluation in an algebra always proceeds bottom-up and thus follows an *innermost* strategy!

Expansion rules are applied to the fixpoint itself (or a component thereof). The redices of induction and coinduction, however, are implications with the fixpoint as its premise resp. conclusion:

Induction and coinduction

$$\frac{\mu x_1 \dots x_n.\varphi \Rightarrow \psi}{\varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n] \Rightarrow \psi} \Uparrow \qquad \frac{\psi \Rightarrow \nu x_1 \dots x_n.\varphi}{\psi \Rightarrow \varphi[\pi_i(\psi)/x_i \mid 1 \leq i \leq n]} \Uparrow$$

The arrow $\Uparrow$ indicates that the succedent of the rule, i.e., the formula below the horizontal line, implies the antecedent, but not necessarily vice versa. Anyhow, we write the conclusion of the implication above the line because the rule syntax should reflect the order in which the rules are applied in a proof.

Hence it may happen that induction or coinduction is applicable to a valid formula, but the rule succedent does not hold true. Then the co/induction hypothesis, which is given by ψ, was too weak (resp. too strong). ψ must then be *generalized*, i.e. extended to some δ by adding a factor (resp. summand). It follows from the incompleteness of second-order logic that the candidates for δ cannot be enumerated. The following rule shows the boundaries within which δ must be searched for:

Second-order induction and coinduction

$$\frac{\mu x_1 \dots x_n.\varphi \Rightarrow \psi}{\exists \delta : \varphi[\pi_i(\delta)/x_i \mid 1 \le i \le n] \Rightarrow \delta \Rightarrow \psi} \Updownarrow \qquad \frac{\psi \Rightarrow \nu x_1 \dots x_n.\varphi}{\exists \delta : \psi \Rightarrow \delta \Rightarrow \varphi[\pi_i(\delta)/x_i \mid 1 \le i \le n]} \Updownarrow$$

The soundness of (first-order) co/induction is easy to show: $\mu x_1 \dots x_n.\varphi$ and $\nu x_1 \dots x_n.\varphi$ denote solutions of the equation $(x_1, \dots, x_n) = \varphi$ in the modal algebra A (see section 3). Since the operators of φ^A are monotone, the fixpoint theorem of Knaster and Tarski tells us that the least resp. greatest solution of $(x_1, \dots, x_n) = \varphi$ in A is the least resp. greatest tuple $B = (B_1, \dots, B_n)$ of sets such that (1) $\varphi[B_i/x_i \mid 1 \le i \le n]^A \subseteq B$ or (2) $B \subseteq \varphi[B_i/x_i \mid 1 \le i \le n]^A$, respectively. Since $\Rightarrow$ is interpreted in A by set inclusion, the conclusion of the co/induction rule is valid iff (1)/(2) with B_i replaced by $\pi_i(\psi)^A$ holds true. Consequently, the rule antecedent follows from the minimality resp. maximality of B with respect to (1)/(2).

Since co/induction is part of the simplifier of Expander2, the system takes care of not destroying co/induction redices. For instance, the following simplification rules are applied only to formulas that are *not* such redices:

Implication splitting Suppose that φ and ψ are simplified.

$$\frac{\varphi \;\Rightarrow\; \psi_1 \wedge \dots \wedge \psi_n}{\varphi \Rightarrow \psi_1 \;\wedge\; \dots \;\wedge\; \varphi \Rightarrow \psi_n} \Updownarrow \qquad \frac{\varphi_1 \vee \dots \vee \varphi_n \;\Rightarrow\; \psi}{\varphi_1 \Rightarrow \psi \;\wedge\; \dots \;\wedge\; \varphi_n \Rightarrow \psi} \Updownarrow$$

The above statements on (the necessity of) generalizations should convince the reader that the co/inductive provability of the premise of a splitting rule does not imply the co/inductive provability of its conclusion! On the other hand, if implication splitting does not interfere with co/induction, it *should* be applied because, as a hidden distribution of $\wedge$ over $\vee$, it is a step towards a disjunctive normal form. More crucial than Boolean simplifications is the simplifier's handling of quantified variables. Here the aim is to move quantifiers such that most of them occur in existentially quantified conjunctions of equations or, dually, universally quantified disjunctions of inequations. Such subformulas are then treated separately by term replacement, atom splitting and atom removal, which often reduces the number of variables or even eliminates all of them.

At first, the simplifier of Expander2 treats a formula as a term to be evaluated bottom-up by applying interpretations of the involved operators in a suitable algebra, say B. In contrast to the modal algebra, B is a term algebra, i.e. it consists of formulas, but usually smaller ones than the original equivalent ones. For instance, an existential quantifier is (1) merged with directly following ones, (2) distributed over a subsequent implication or disjunction and (3) restricted to those variables that have free occurrences in the quantified formula.

If a formula has been evaluated in this way, the simplifier applies rules (including the ones presented in this and the previous section) only to outermost redices as described above. Since path formulas cannot be evaluated in the modal algebra, we specify temporal operators in terms of further simplification rules that will be used in proofs together with expansion and co/induction.

```
-- LTLS
preds:      P Q true false hatom not \/ /\ `then` F G `U`
                                    -- predicates
constructs: blink                   -- the stream 010101...
fovars:     at s                    -- first-order variables
hovars:     X P Q                   -- higher-order variables
axioms:
   (true$s <==> True)
 & (false$s <==> False)
 & (hatom(at)$s <==> at -> head$s)
 & (not(P)$s <==> Not(P$s))
 & ((P\/Q)$s <==> (P$s | Q$s))
 & ((P/\Q)$s <==> (P$s & Q$s))
 & ((P`then`Q)$s <==> (P$s ==> Q$s))
 & (F$P <==> MU X.(P\/X.tail))          -- finally
 & (G$P <==> NU X.(P/\X.tail))          -- generally
 & ((P`U`Q) <==> MU X.(Q\/(P/\X.tail)))   -- until
 & head$blink == 0
 & tail$blink == 1:blink   -- coalgebraic specification of blink
```

Except for the fixpoint operators, the axioms directly implement the interpretation of temporal operators in the modal algebra. State operators could be axiomatized analogously. However, this is not needed if the underlying Kripke structure is finite. The formula `hatom(at)$s` checks whether the head of the path s satisfies $at \in At$ (see section 2). The functions `head` and `tail` are defined as in Haskell. They provide the destructors of the data type of paths and are used here for specifying the stream 010101... A point in terms denotes function composition. The conjecture

```
 s = blink | s = 1:blink ==> G(F$(=0).head)$s                    (1)
```

says that the streams `blink` and `1:blink` are fair insofar as they contain infinitely many zeros. By the `G`-axiom of `LTLS`, (1) simplifies to:

```
 s = blink | s = 1:blink ==> NU X.(F((=0).head)/\X.tail)$s      (2)
```

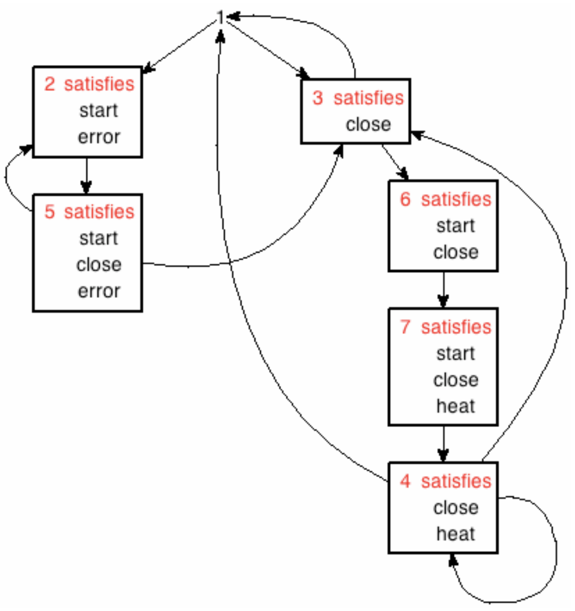
(2) is an instance of the antecendent of coinduction (see above). Applying the rule yields:

```
 All s:(s = blink | s = 1:blink ==>
        (F((=0).head)/\(rel(s,s=blink|s=1:blink).tail))$s)      (3)
```

47 further simplification steps including three expansion steps turn (3) into *True*. The entire proof goes through automatically.

The second sample proof is based on a model of a microwave controller [4]:

```
-- MICROS
specs:          LTLS                  -- imported specifications
constructs:     start close heat error SAT
defuncts:       inits states atoms drawK
fovars:         ats
axioms:
inits == [1] & atoms == [start,close,heat,error] &
1 -> branch[2,3] & 2 -> 5 & 3 -> branch[1,6] &
4 -> branch[1,3,4] & 5 -> branch[2,3] & 6 -> 7 & 7 -> 4
& start -> branch[2,5,6,7]
& close -> branch[3,4,5,6,7]
& heat -> branch[4,7]
& error -> branch[2,5]
& drawK == wtree$fun(x`sat`ats,rframe$matrix[x,satisfies(ats)],
                     x,x)
```



The Kripke model that Expander2 derives from `MICROS` *and the function* `drawK`

The conjecture

`G(hatom$error)$s ==> G(not$hatom$heat)$s` (1)

says that a path consisting of error states never contains heat states. By the `G`-axiom of `LTLS`, (1) simplifies to:

```
NU X.(hatom(error)/\X.tail)$s ==>
NU X.(not(hatom$heat)/\X.tail)$s                               (2)
```

Applying the coinduction rule yields:

```
 All pa:(NU X.(hatom(error)/\X.tail)$s ==>
         (not(hatom$heat)/\
          (rel(s,NU X.(hatom(error)/\X.tail)$s).tail))$s)      (3)
```

41 further simplification steps lead (3) to *True*. Three expansion steps are needed, and the entire proof goes through automatically.

5 Model checking by co/resolution and co/induction

Both evaluation and simplification regard modal formulas as representations of data, namely (tuples of) sets. This is the actual reason for the algebraic flavor of modal logics: their operators denote *functions* that generate or transform data. Fixpoint operators are no exception. They map the left-hand sides of regular equations to the equations' solutions (see section 3). First-order predicate logic as well as logic programming follow a different view. Their formulas do not denote data, but propositions or statements *about* data. Set membership takes us from the (sets-as-)data view to the propositional one, set comprehension back from the propositional to the data view. So where is the difference? It comes with the fixpoint property that cannot be expressed within first-order logic. Instead, we axiomatize *co/predicates* in terms of (generalized) *co/Horn clauses* and fix their interpretation as the least resp. greatest relations (on a given data model) that satisfy the axioms. Details of this approach and its connection with relational and functional programming can be found in [18, 19].

For checking Kripke structures with co/Horn logic we replace the modal algebra of section 3 and the specification LTLS of section 4 by the following one:

```
-- LTL
preds:    P Q true false hatom not \/ /\ `then` F `U`
copreds:  G                                         -- copredicates
fovars:   s
hovars:   P Q
axioms:
   (F(P)$s <=== P$s | F(P)$tail$s)                 -- finally
 & (G(P)$s ===> P$s & G(P)$tail$s)                 -- generally
 & ((P`U`Q)$s <=== Q$s | P$s & (P`U`Q)$tail$s)    -- until
```

In addition, the propositional operators are specified in the same way as in `LTLS` in terms of simplification rules. The modal operators, however, now have co/Horn axioms. The direction of the implication arrow (`<===` or `===>`) indicates whether the axiom is called a Horn or a co-Horn clause and the relational expression on its left-hand side a predicate or a copredicate and thus interpreted as the least or greatest relation satisfying the axiom(s). When applied in a logical derivation, a co/Horn clause is always applied from left to right. Besides premise and/or conclusion a clause may contain a *guard* that confines redices to formulas that unify with the left-hand side (premise resp. conclusion) *and* satisfy the guard (see the co/resolution rules given below).

An expansion step is replaced by the simultaneous application of all axioms with the same relational expression on the left-hand side:

Parallel resolution upon the predicate p

$$\frac{p(t)}{\bigvee_{i=1}^{k} \exists Z_i : (\varphi_i\sigma_i \wedge \boldsymbol{x} = \boldsymbol{x}\sigma_i)} \Updownarrow$$

where $\gamma_1 \Rightarrow (p(t_1) \Longleftarrow \varphi_1), \dots, \gamma_n \Rightarrow (p(t_n) \Longleftarrow \varphi_n)$ are the (Horn) axioms for p (with guards $\gamma_1, \dots, \gamma_n$).

Parallel coresolution upon the copredicate p

$$\frac{p(t)}{\bigwedge_{i=1}^{k} \forall Z_i : (\boldsymbol{x} = \boldsymbol{x}\sigma_i \Rightarrow \varphi_i\sigma_i)} \Updownarrow$$

where $\gamma_1 \Rightarrow (p(t_1) \Longrightarrow \varphi_1), \dots, \gamma_n \Rightarrow (p(t_n) \Longrightarrow \varphi_n)$ are the (co-Horn) axioms for p (with guards $\gamma_1, \dots, \gamma_n$).

In both rules, $\boldsymbol{x}$ is a vector of 'new' variables, for all $1 \leq i \leq k$, $t\sigma_i = t_i\sigma_i$, $\gamma_i\sigma_i \vdash \mathit{True}$ and $Z_i = \mathit{var}(t_i, \varphi_i)$, and for all $k < i \leq n$, t is not unifiable with t_i.

As in section 4, co/induction can only be applied to implications with a predicate (the first-order analog of a variable bound by μ) in the premise or a copredicate (the first-order analog of a variable bound by ν) in the conclusion. In contrast to co/induction as a simplification rule, we may now start a proof with the original conjecture and generalize it later—when simplification rules are no longer applicable and generalization candidates have emerged from preceding proof steps. Restricted to the proof of bisimilarities (relations describing behavioral equality), this incremental procedure is also known as *circular coinduction* [7, 11].

Incremental induction upon the predicate p

$$\frac{p(x) \;\Rightarrow\; \psi(x)}{\bigwedge_{p(t)\Leftarrow\varphi\in AX_p} (\varphi[p'/p] \Rightarrow \psi(t))} \Uparrow \quad \frac{p'(x) \;\Rightarrow\; \delta(x)}{\bigwedge_{p(t)\Leftarrow\varphi\in AX_p} (\varphi[p'/p] \Rightarrow \delta(t))} \Uparrow \quad p \not\in \psi \cup \delta$$

AX_p denotes the set of axioms for p. When the first rule is applied, p' is stored as a new copredicate with the axiom $p'(x) \Rightarrow \psi(x)$. When the second rule is applied, the axiom $p'(x) \Rightarrow \delta(x)$ is added.

Incremental coinduction upon the copredicate p

$$\frac{\psi(x) \;\Rightarrow\; p(x)}{\bigwedge_{p(t)\Rightarrow\varphi\in AX_p} (\psi(t) \Rightarrow \varphi[p'/p])} \Uparrow \quad \frac{\delta(x) \;\Rightarrow\; p'(x)}{\bigwedge_{p(t)\Rightarrow\varphi\in AX_p} (\delta(t) \Rightarrow \varphi[p'/p])} \Uparrow \quad p \not\in \psi \cup \delta$$

When the first rule is applied, p' is stored as a new predicate with the axiom(s) $p'(x) \Leftarrow \psi(x)$ and, if p is behavioral equality, Horn clauses that establish p' as an equivalence relation. When the second rule is applied, the axiom $p'(x) \Leftarrow \delta(x)$ is added.

Co/resolution and co/induction complement each other in the way axioms work together with conjectures in proofs. Roughly said, co/resolution applies axioms to conjectures and the proof proceeds with the modified conjectures. Conversely, co/induction applies conjectures to axioms and establishes the modified axioms as new conjectures.

The generalization of the above rules to several co/predicates (the first-order analog of a fixpoint formula with several bound variables) is straightforward.

In contrast to section 4, incremental coinduction allows us to start a proof that the stream `blink` is fair with the conjecture

$$\psi = \texttt{G(F\$(=0).head)\$blink}$$

and derive the factor $\delta =$ `G(F$(=0).head)$1:blink` of the generalized conjecture $\psi \wedge \delta$ within the proof of ψ. Indeed, applying incremental coinduction to ψ yields the new conjecture

```
All P s:(P = F((=0).head) & s = blink ===>
         P(s) & G0(P)$tail$s)                                 (1)
```

`G0` is the predicate p' created during rule application (see above). Its axiom is:

```
G0(z0)$z1 <=== z0 = F((=0) . head) & z1 = blink             (ax1)
```

Six simplification steps transform (1) into:

```
F((=0).head)$blink & G0(F((=0).head))$(1:blink)               (2)
```

Parallel resolution upon F and subsequent simplification steps remove the first factor of (2). The second factor is a redex for the second rule of incremental coinduction. Hence (2) is turned into:

```
All P s:(P = F((=0).head) & s = 1:blink ===>
         P(s) & G0(P)$tail(s))                                (3)
```

and a further axiom for $G0$ is created:

```
G0(z2)$z3 <=== z2 = F((=0) . head) & z3 = 1:blink           (ax2)
```

Five simplification steps transform (3) into:

```
F((=0).head)$(1:blink) & G0(F((=0).head))$blink               (4)
```

Three resolution and subsequent simplification steps turn (4) into *True*.

The conjecture

```
G(hatom$error)$s ==> G(not$hatom$heat)$s                      (1)
```

(see `MICROS` in section 4) can also be proved by incremental coinduction and co/resolution. The coinduction rule adds

```
G0(z0)$s <=== G(hatom$error)$s & z0 = not$hatom$heat
```

to the set of axioms. Subsequent coresolution and simplification steps automatically lead to:

```
All s:(G(hatom$error)$s ==> G0(not$hatom$heat)$tail$s)        (2)
```

(2) admits both coresolution upon `G` and resolution upon `G0`. The first step would lead the proof into a cycle because the only axiom for `G` (see `LTL`) is recursive (`G` occurs on both sides of the axiom). The axiom for `G0`, however, is non-recursive—as axioms introduced by co/induction steps always are. Hence we choose the resolution step and obtain after simplification:

```
All s:(G(hatom$error)$s ==> G(hatom$error)$tail$s)            (3)
```

Coresolution upon G and subsequent simplification turn (3) into *True*.

6 Conclusion

We have presented three approaches to the verification of Kripke structures based on a labelled or unlabelled transition system (also called Kripke frame) or a mixture thereof. The first method consists in evaluating modal formulas in an algebra of sets of states or paths. For state formulas, the evaluation procedure is part of the simplification component of Expander2. Since fixpoint computations are involved, model checking by evaluation is restricted to models with a finite set of states.

The second technique uses simplification rules, which extend the modal algebra of the first approach by expansion, induction and coinduction. This allows us to prove also path formulas and to verify Kripke models with infinitely many states. We have described and illustrated a strategy of applying expansion, co/induction and other simplification rules that is complete: it terminates whenever any other strategy would also terminate.

The third approach is based on our previous work [17, 18] on co/Horn logic where co/Horn clauses axiomatize least resp. greatest relational fixpoints and parallel co/resolution provides the counterpart of expansion in pure simplification proofs. Co/induction as used in the second approach is replaced by incremental co/induction, a proof rule that admits the automatic—and often inevitable—generalization of the respective conjecture. Incremental co/induction was inspired by the method of circular coinduction [7, 11] that, however, is tailored to the proof of equations.

The first method may be compared with other model checkers, which also hide all logical inference involved from the user by turning both the Kripke structure and the formula to be proved into some efficiently processible internal representation and then running a deterministic algorithm that checks the formula in a single visible step. The second and the third method work on both the Kripke structure's internal representation—if there any—*and* its specification given by rewrite rules (Horn clause axioms for `->`) and thus admit the treatment of infinite-state systems. The formula to be proved, however, is processed in its original form. With the second method, the proof goes through automatically—provided that co/inductive subconjectures appear as suitable implications and generalizations are not needed (see section 4). Similar co/induction rules were implemented in Isabelle [6, 24], but their use needs manual control. PVS [10, 12] and CLAM [5] also admit coinduction, but—like circular coinduction (see section 5)—only for proving bisimilarities. Manual control is needed for our third method, but this is offset by more general co/induction redices and the possibility to generalize co/inductive conjectures during proof construction.

If proof assistants for Kripke structures were put on a line, starting from the most efficient to the most powerful ones, model checkers would occupy one end and established theorem provers the other. Our methods distribute over the whole line and their integration in Expander2 shows that model checking and (modal-)theorem proving can be performed simultaneously.

More and greater examples can be found in [23] and the *Examples* directory of Expander2. We are also about to integrate inductive techniques such as those for reasoning about systems communicating between a varying number of processes [3, 4]. Last not least, the `blink` example should indicate the actual goal of our research on model checking, namely to adapt its techniques to the more general ones used for proving properties of co/algebraic data types.

References

[1] J. van Benthem, J. Bergstra, *Logic of Transition Systems*, J. Logic, Language and Information 3 (1995) 247-283

[2] C. Cirstea, A. Kurz, D. Pattinson, L. Schröder, Y. Venema, *Modal Logics are Coalgebraic*, The Computer Journal, to appear

[3] E.M. Clarke, O. Grumberg, S. Jha, *Verification of Parameterized Networks*, ACM TOPLAS 19 (1997) 726-750

[4] E.M. Clarke, O. Grumberg, D.A. Peled, *Model Checking*, The MIT Press 1999

[5] L.A. Dennis, A. Bundy, I. Green, *Making a productive use of failure to generate witnesses for coinduction from divergent proof attempts*, Annals of Mathematics and Artificial Intelligence 29, Springer (2000) 99-138

[6] J. Frost, *A Case Study of Co-induction in Isabelle*, Report, Computer Laboratory, University of Cambridge 1995

[7] J. Goguen, K. Lin, G. Rosu, *Conditional Circular Coinductive Rewriting with Case Analysis*, Proc. WADT'02, Springer LNCS 2755 (2003) 216-232

[8] H. P. Gumm, *Universal Coalgebras and their Logics*, AJSE-Mathematics, to appear

[9] J. Goguen, G. Malcolm, *A Hidden Agenda*, Theoretical Computer Science 245 (2000) 55-101

[10] H. Gottliebsen, *Co-inductive Proofs for Streams in PVS*, Report, Queen Mary, University of London 2007

[11] D. Hausmann, T. Mossakowski, L. Schröder, *Iterative Circular Coinduction for CoCasl in Isabelle/HOL*, Proc. FASE'05, Springer LNCS 3442 (2005) 341-356

[12] U. Hensel, B. Jacobs, *Coalgebraic Theories of Sequences in PVS*, J. Logic and Computation 9 (1999) 463-500

[13] M. Huth, M. Ryan, *Logic in Computer Science: Modelling and Reasoning about Systems*, 2nd Ed. Cambridge University Press 2004

[14] B. Jacobs, J. Rutten, *A Tutorial on (Co)Algebras and (Co)Induction*, EATCS Bulletin 62 (1997) 222-259
[15] A. Kurz, *Specifying Coalgebras with Modal Logic*, Theoretical Computer Science 260 (2001) 119-138
[16] Z. Manna, Mathematical Theory of Computation, McGraw-Hill 1974
[17] P. Padawitz, *Proof in Flat Specifications*, in: Algebraic Foundations of Systems Specification, IFIP State-of-the-Art Report, Springer (1999) 321-384
[18] P. Padawitz, *Swinging Types = Functions + Relations + Transition Systems*, Theoretical Computer Science 243 (2000) 93-165
[19] P. Padawitz, *Dialgebraic Specification and Modeling*, draft, fldit-www.cs.tu-dortmund.de/~peter/Dialg.pdf
[20] P. Padawitz, *Expander2: A Formal Methods Presenter and Animator*, fldit-www.cs.tu-dortmund.de/~peter/Expander2.html
[21] P. Padawitz, *Expander2: Towards a Workbench for Interactive Formal Reasoning*, in: Formal Methods in Software and Systems Modeling: Essays Dedicated to Hartmut Ehrig, Springer LNCS 3393 (2005) 236-258
[22] P. Padawitz, *Expander2: Program verification between interaction and automation*, Proc. 15th Workshop on Functional and (Constraint) Logic Programming, Elsevier ENTCS 177 (2007) 35-57
[23] P. Padawitz, *Algebraic Model Checking and more*, slides in German, fldit-www.cs.tu-dortmund.de/~peter/Haskellprogs/CTL.pdf
[24] L. C. Paulson, *Mechanizing Coinduction and Corecursion in Higher-Order Logic*, J. Logic and Computation 7 (1997) 175-204
[25] J. Rutten, *Universal Coalgebra: A Theory of Systems*, Theoretical Computer Science 249 (2000) 3-80
[26] C. Stirling, *Modal and Temporal Logics*, in: Handbook of Logic in Computer Science, Clarendon Press (1992) 477-563

Prof. Dr. Peter Padawitz
Fakultät für Informatik
Technische Universtität Dortmund
D-44221 Dortmund (Germany)

From 1976 to 1982, Peter Padawitz was a colleague of Hans-Jörg Kreowski at TU Berlin. Due to their common interest in algebraic specification, they are coauthors of several publications and continued to meet at various occasions, e.g., the meetings of the IFIP Working Group 1.3 (Foundations of System Specification) and the WADT conferences.

Checking Graph-Transformation Systems for Confluence[⋆] (Extended Abstract)

Detlef Plump

Abstract. In general, it is undecidable whether a terminating graph-transformation system is confluent or not. We introduce the class of *coverable* hypergraph-transformation systems and show that confluence is decidable for coverable systems that are terminating. Intuitively, a system is coverable if its typing allows to extend each critical pair with a non-deletable context that uniquely identifies the persistent nodes of the pair. The class of coverable systems includes all hypergraph-transformation systems in which hyperedges can connect arbitrary sequences of nodes, and all graph-transformation systems with a sufficient number of unused edge labels.

1 Introduction

Confluent sets of graph-transformation rules can be executed without backtracking since all terminating derivations produce the same result for a given input graph. Applications of confluence include the efficient recognition of graph classes by graph reduction [1, 5, 3], the parsing of languages defined by graph grammars [7, 15], and the deterministic input/output behaviour of programs in graph-transformation languages such as AGG [16], FUJABA [11], GrGen [8] or GP [14].

In the settings of string and term rewriting, confluence is decidable for terminating systems [6, 2, 4]: one computes all *critical pairs* $t \leftarrow s \rightarrow u$ of rewrite steps and checks whether t and u are *joinable* in that they reduce to a common string resp. term. In contrast, confluence is undecidable in general for terminating graph-transformation systems [13]. The problem is, in brief, that the joinability of all critical pairs need not imply confluence of a system. To guarantee confluence, one has to impose extra conditions on the joining derivations, leading to the notion of a *strongly joinable* critical pair. However, strong joinability of all critical pairs is not a necessary condition for confluence and hence, in general, cannot be used to decide confluence.

In this paper, we introduce *coverable* hypergraph-transformation systems and show that confluence is decidable for coverable systems that are terminating. Intuitively, a system is coverable if its typing allows to extend each critical pair with a non-deletable context—a *cover*—that uniquely identifies the persistent nodes of the pair. We give a decision procedure for confluence that processes each extended critical pair $\widehat{\Gamma}\colon \widehat{U}_1 \Leftarrow \widehat{S} \Rightarrow \widehat{U}_2$ by reducing $\widehat{U}_1$ and $\widehat{U}_2$ to normal

[⋆] Dedicated to Hans-Jörg Kreowski on the occasion of his 60th birthday.

forms X_1 and X_2, and checking whether X_1 and X_2 are isomorphic. If this is the case, then the critical pair underlying $\widehat{\Gamma}$ is strongly joinable; otherwise, a counterexample to confluence has been found.

Roughly speaking, a cover for a critical pair can be constructed if the signature of the hypergraph-transformation system under consideration contains (hyper-)edge labels that do not occur in rules and that can be used to connect the persistent nodes of the critical pair by edges. Such a cover cannot be deleted by rules. Moreover, there must be a unique surjective morphism from the cover to each of its images under a graph morphism. We give different conditions under which covers can be constructed and show, in particular, that the class of coverable systems includes all hypergraph-transformation systems in which hyperedges can connect arbitrary sequences of nodes.

The rest of this paper is organised as follows. The next section recalls some terminology for binary relations and defines hypergraphs and their morphisms. Section 3 reviews the double-pushout approach to (hyper-)graph transformation in a setting where rules are matched injectively and can have non-injective right-hand morphisms. We define confluence of hypergraph-transformation systems and recall the fact that confluence is undecidable for terminating systems. In Section 4 we review the role of critical pairs in establishing confluence. Section 5 introduces covers for critical pairs and coverable systems, discusses our main result and the associated decision procedure for confluence, and presents special cases where confluence is decidable. In Section 6, we conclude and discuss a topic for future work.

2 Preliminaries

We recall some terminology for binary relations (consistent with [2, 4]) and define hypergraphs and their morphisms.

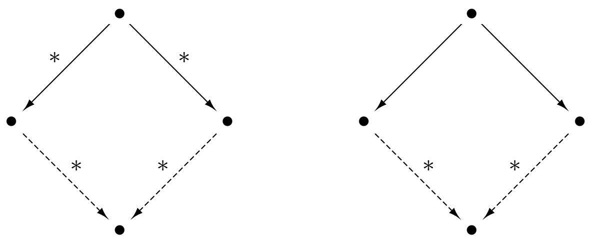
2.1 Relations

Let $\to$ be a binary relation on a set A. The inverse relation of $\to$ is denoted by $\leftarrow$. We write $\to^+$ for the transitive closure of $\to$ and $\to^*$ for the transitive-reflexive closure of $\to$. Two elements $a, b \in A$ have a *common reduct* if $a \to^* c \leftarrow^* b$ for some c. If $a \to^* c$ and there is no d such that $c \to d$, then d is a *normal form* of a.

The relation $\to$ is (1) *terminating* if there is no infinite sequence $a_1 \to a_2 \to a_3 \to \ldots$, (2) *confluent* if for all a, b and c with $b \leftarrow^* a \to^* c$, elements b and c have a common reduct (see Figure 1(a)), (3) *locally confluent* if for all a, b and c with $b \leftarrow a \to c$, elements b and c have a common reduct (see Figure 1(b)).

By the following well-known result, local confluence and confluence are equivalent in the presence of termination.

Lemma 1 (Newman's Lemma [10]). *A terminating relation is confluent if and only if it is locally confluent.*



(a) confluence (b) local confluence

Fig. 1. Confluence properties

2.2 Hypergraphs

We deal with directed, labelled hypergraphs and use a simple type system where the label of a hyperedge restricts the number of incident nodes and their labels. A *signature* $\Sigma = \langle \Sigma_{\mathrm{V}}, \Sigma_{\mathrm{E}}, \mathrm{Type}\rangle$ consists of a set Σ_{V} of *node labels*, a set Σ_{E} of *hyperedge labels* and a mapping Type assigning to each $l \in \Sigma_{\mathrm{E}}$ a set $\mathrm{Type}(l) \subseteq \Sigma_{\mathrm{V}}^*$. Unless stated otherwise, we denote by Σ an arbitrary but fixed signature over which all hypergraphs are labelled.

A *hypergraph* over Σ is a system $G = \langle \mathrm{V}_G, \mathrm{E}_G, \mathrm{mark}_G, \mathrm{lab}_G, \mathrm{att}_G\rangle$ consisting of two finite sets V_G and E_G of *nodes* (or *vertices*) and *hyperedges*, two labelling functions $\mathrm{mark}_G \colon \mathrm{V}_G \to \Sigma_{\mathrm{V}}$ and $\mathrm{lab}_G \colon \mathrm{E}_G \to \Sigma_{\mathrm{E}}$, and an attachment function $\mathrm{att}_G \colon \mathrm{E}_G \to \mathrm{V}_G^*$ such that $\mathrm{mark}_G^*(\mathrm{att}_G(e)) \in \mathrm{Type}(\mathrm{lab}_G(e))$ for each hyperedge e. (The extension $f^* \colon A^* \to B^*$ of a function $f \colon A \to B$ maps the empty string to itself and $a_1 \dots a_n$ to $f(a_1) \dots f(a_n)$.) We write $\mathcal{H}(\Sigma)$ for the set of all hypergraphs over Σ.

In pictures, nodes and hyperedges are drawn as circles and boxes, respectively, with labels inside. Lines represent the attachment of hyperedges to nodes, where numbers specify the left-to-right order in the attachment string. For example, Figure 2 shows a hypergraph with four nodes (all labelled with •) and three hyperedges (labelled with B and S).

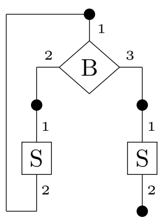


Fig. 2. A hypergraph

A hypergraph G is a *graph* if each hyperedge e is an ordinary edge, that is, if $\mathrm{att}_G(e)$ has length two. Ordinary edges may be drawn as arrows with labels written next to them.

Given hypergraphs G and H, a *hypergraph morphism* (or *morphism* for short) $f\colon G \to H$ consists of two functions $f_\mathrm{V}\colon \mathrm{V}_G \to \mathrm{V}_H$ and $f_\mathrm{E}\colon \mathrm{E}_G \to \mathrm{E}_H$ that preserve labels and attachment to nodes, that is, $\mathrm{mark}_H \circ f_\mathrm{V} = \mathrm{mark}_G$, $\mathrm{lab}_H \circ f_\mathrm{E} = \mathrm{lab}_G$ and $\mathrm{att}_H \circ f_\mathrm{E} = f_\mathrm{V}^* \circ \mathrm{att}_G$. A morphism $incl\colon G \to H$ is an *inclusion* if $incl_\mathrm{V}(v) = v$ and $incl_\mathrm{E}(e) = e$ for all $v \in \mathrm{V}_G$ and $e \in \mathrm{E}_G$. In this case G is a *subhypergraph* of H which is denoted by $G \subseteq H$. Every morphism $f\colon G \to H$ induces a subhypergraph of H, denoted by $f(G)$, which has nodes $f_\mathrm{V}(\mathrm{V}_G)$ and hyperedges $f_\mathrm{E}(\mathrm{E}_G)$. Morphism f is *injective* (*surjective*) if f_V and f_E are injective (surjective). If f is surjective, then H is an *image* of G. If f is both injective and surjective, then it is an *isomorphism*. In this case G and H are *isomorphic*, which is denoted by $G \cong H$.

The *composition* of two morphisms $f\colon G \to H$ and $g\colon H \to M$ is the morphism $g \circ f\colon G \to M$ consisting of the composed functions $g_\mathrm{V} \circ f_\mathrm{V}$ and $g_\mathrm{E} \circ f_\mathrm{E}$. The composition is also written as $G \to H \to M$ if f and g are clear from the context.

A *partial hypergraph morphism* $f\colon G \to H$ is a hypergraph morphism $S \to H$ such that $S \subseteq G$. Here S is the *domain of definition* of f, denoted by $\mathrm{Dom}(f)$.

3 Graph Transformation

We briefly review the *double-pushout approach* to graph transformation. In our setting, rules are matched injectively and can have non-injective right-hand morphisms. (See [9] for a comparison with other variants of the double-pushout approach.)

3.1 Rules and derivations

A *rule* $r\colon \langle L \leftarrow K \to R\rangle$ consists of two hypergraph morphisms with a common domain, where $K \to L$ is an inclusion. The hypergraphs L and R are the *left-* and *right-hand side* of r, and K is the *interface*. The rule is *injective* if the morphism $K \to R$ is injective.

Let G and H be hypergraphs, $r\colon \langle L \leftarrow K \to R\rangle$ a rule and $f\colon L \to G$ an injective morphism. Then G *directly derives* H by r and f, denoted by $G \Rightarrow_{r,f} H$, if there exist two pushouts of the following form:

$$\begin{tikzcd} L \arrow[d, "f"'] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\ G & D \arrow[l] \arrow[r] & H \end{tikzcd} \qquad (1)$$

Given a set of rules $\mathcal{R}$, we write $G \Rightarrow_\mathcal{R} H$ to express that there exist $r \in \mathcal{R}$ and a morphism f such that $G \Rightarrow_{r,f} H$.

We refer to [13] for the definition and construction of hypergraph pushouts. Intuitively, the left pushout corresponds to the construction of D from G by removing the items in $L - K$, and the right pushout to the construction of H from D by merging items according to $K \to R$ and adding the items in R that are not in the image of K.

A double-pushout as in diagram (1) is called a *direct derivation* from G to H and may be denoted by $G \Rightarrow_{r,f} H$ or just by $G \Rightarrow_r H$ or $G \Rightarrow H$. A *derivation* from G to H is a sequence of direct derivations $G = G_0 \Rightarrow \ldots \Rightarrow G_n = H$, $n \geq 0$, and may be denoted by $G \Rightarrow^* H$.

Given a rule $r\colon \langle L \leftarrow K \rightarrow R \rangle$, an injective morphism $f\colon L \to G$ satisfies the *dangling condition* if no hyperedge in $\mathrm{E}_G - f_{\mathrm{E}}(\mathrm{E}_L)$ is incident to a node in $f_{\mathrm{V}}(\mathrm{V}_L - \mathrm{V}_K)$. It can be shown that, given r and f, a direct derivation as in diagram (1) exists if and only if f satisfies the dangling condition [9].

With every derivation $\Delta\colon G_0 \Rightarrow^* G_n$ a partial hypergraph morphism can be associated that tracks the items of G_0 through the derivation: this morphism is undefined for all items in G_0 that are removed by Δ at some stage, and maps all other items to the corresponding items in G_n.

Definition 1 (Track morphism). Given a direct derivation $G \Rightarrow H$ as in diagram (1), the *track morphism* $\mathrm{tr}_{G \Rightarrow H}\colon G \to H$ is the partial hypergraph morphism defined by

$$\mathrm{tr}_{G \Rightarrow H}(x) = \begin{cases} c'(c^{-1}(x)) & \text{if} \quad x \in c(D), \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Here $c\colon D \to G$ and $c'\colon D \to H$ are the morphisms in the lower row of (1) and $c^{-1}\colon c(D) \to D$ maps each item $c(x)$ to x.

The track morphism of a derivation $\Delta\colon G_0 \Rightarrow^* G_n$ is defined by $\mathrm{tr}_\Delta = \mathrm{id}_{G_0}$ if $n = 0$ and $\mathrm{tr}_\Delta = \mathrm{tr}_{G_1 \Rightarrow^* G_n} \circ \mathrm{tr}_{G_0 \Rightarrow G_1}$ otherwise, where id_{G_0} is the identity morphism on G_0.

Definition 2 (Hypergraph-transformation system). A *hypergraph-transformation system* $\langle \Sigma, \mathcal{R} \rangle$ consists of a signature Σ and a set $\mathcal{R}$ of rules over Σ. The system is *injective* if all rules in $\mathcal{R}$ are injective. It is a *graph-transformation system* if for each label l in Σ_{E}, all strings in $\mathrm{Type}(l)$ are of length two.

As graph-transformation systems are special hypergraph-transformation systems, results for the latter also apply to the former. In particular, Theorem 2, Theorem 3 and Corollary 1 below hold for graph-transformation systems, too.

Example 1. Figure 3 shows hypergraph-transformation rules for reducing control-flow graphs (see also [13]). The associated signature contains a single node label • and two hyperedge labels which are graphically represented by hyperedges formed as squares and rhombs. Instead of using numbers to represent the attachment function, we use an arrow to point to the second attachment node of a square and define the order among the links of a rhomb to be "top-left-right". The rules are shown in a shorthand notation where only the left- and right-hand sides are depicted, the interface and the morphisms are implicitly given by the node names x,y,z. This example will be continued as Example 2, where it is shown that the system is confluent. □

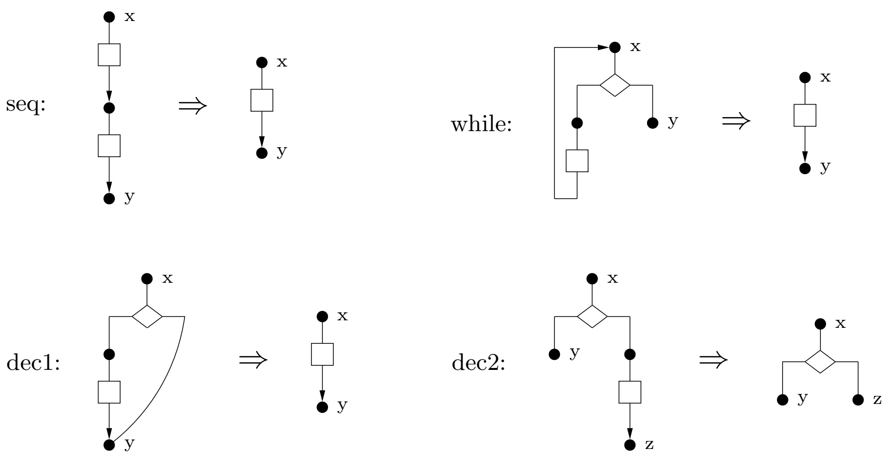


Fig. 3. Hypergraph-transformation system for flow-graph reduction

3.2 Independence and confluence

Two direct derivations $H_1 \Leftarrow_{r_1} G \Rightarrow_{r_2} H_2$ do not interfere with each other if, roughly speaking, the intersection of the left-hand sides of r_1 and r_2 in G consists of common interface items. If one of the rules is not injective, however, an additional injectivity condition is needed. For $i = 1, 2$, let r_i denote a rule $\langle L_i \leftarrow K_i \rightarrow R_i \rangle$.

Definition 3 (Independence). Direct derivations $H_1 \Leftarrow_{r_1} G \Rightarrow_{r_2} H_2$ as in Figure 4 are *independent* if there are morphisms $L_1 \to D_2$ and $L_2 \to D_1$ such that the following holds:

Commutativity: $L_1 \to D_2 \to G = L_1 \to G$ and $L_2 \to D_1 \to G = L_2 \to G$.

Injectivity: $L_1 \to D_2 \to H_2$ and $L_2 \to D_1 \to H_1$ are injective.

If r_1 and r_2 are injective, the direct derivations of Figure 4 are independent if and only if the intersection of the two left-hand sides coincides with the intersection of the two interfaces.

Lemma 2 (Independence for injective rules). *Let r_1 and r_2 be injective rules. Then direct derivations $H_1 \Leftarrow_{r_1,g_1} G \Rightarrow_{r_2,g_2} H_2$ are independent if and only if $g_1(L_1) \cap g_2(L_2) \subseteq g_1(K_1) \cap g_2(K_2)$.*

$$\begin{tikzcd}
R_1 \arrow[d] & K_1 \arrow[l] \arrow[r] \arrow[d] & L_1 \arrow[dr] \arrow[drrr, dashed] & & L_2 \arrow[dl] \arrow[dlll, dashed] & K_2 \arrow[l] \arrow[r] \arrow[d] & R_2 \arrow[d] \\
H_1 & D_1 \arrow[l] \arrow[rr] & & G & & D_2 \arrow[ll] \arrow[r] & H_2
\end{tikzcd}$$

Fig. 4. Independent direct derivations

To define confluence, we consider hypergraph transformation "up to isomorphism", that is, the transformation of isomorphism classes of hypergraphs. Given a hypergraph G, we denote by $[G]$ the isomorphism class $\{G' \mid G' \cong G\}$.

Definition 4 (Transformation modulo isomorphism). Given a hypergraph-transformation system $\langle \Sigma, \mathcal{R} \rangle$, the relation $\Rightarrow_{\mathcal{R},\cong}$ on isomorphism classes of hypergraphs over Σ is defined by: $[G] \Rightarrow_{\mathcal{R},\cong} [H]$ if there are hypergraphs G' and H' such that $G \cong G' \Rightarrow_{\mathcal{R}} H' \cong H$. We refer to $\Rightarrow_{\mathcal{R},\cong}$ as *hypergraph-transformation modulo isomorphism.*

By pushout properties, we have $[G] \Rightarrow_{\mathcal{R},\cong} [H]$ if and only if $G \Rightarrow_{\mathcal{R}} H$. But $[G] \Rightarrow^*_{\mathcal{R},\cong} [H]$ need not imply $G \Rightarrow^*_{\mathcal{R}} H$ since G and H may be distinct in the case $[G] = [H]$. As a consequence, confluence of $\Rightarrow_{\mathcal{R},\cong}$ need not imply confluence of $\Rightarrow_{\mathcal{R}}$.

Definition 5 (Confluence of $\langle \Sigma, \mathcal{R} \rangle$). A hypergraph-transformation system $\langle \Sigma, \mathcal{R} \rangle$ is *confluent* (*locally confluent*) if the relation $\Rightarrow_{\mathcal{R},\cong}$ is confluent (locally confluent).

By the following lemma, confluence and local confluence of $\Rightarrow_{\mathcal{R},\cong}$ are only slightly more general than the corresonding properties of $\Rightarrow_{\mathcal{R}}$: joining derivations need not meet in a common graph but in isomorphic graphs.

Lemma 3 ([13]). *Let $\langle \Sigma, \mathcal{R} \rangle$ be a hypergraph-transformation system.*

(1) *$\langle \Sigma, \mathcal{R} \rangle$ is confluent if and only if for all $G, G_1, G_2 \in \mathcal{H}(\Sigma)$, $G_1 \Leftarrow^*_{\mathcal{R}} G \Rightarrow^*_{\mathcal{R}} G_2$ implies that there are $H_1, H_2 \in \mathcal{H}(\Sigma)$ such that $G_1 \Rightarrow^*_{\mathcal{R}} H_1 \cong H_2 \Leftarrow^*_{\mathcal{R}} G_2$.*

(2) *$\langle \Sigma, \mathcal{R} \rangle$ is locally confluent if and only if for all $G, G_1, G_2 \in \mathcal{H}(\Sigma)$, $G_1 \Leftarrow_{\mathcal{R}} G \Rightarrow_{\mathcal{R}} G_2$ implies that there are $H_1, H_2 \in \mathcal{H}(\Sigma)$ such that $G_1 \Rightarrow^*_{\mathcal{R}} H_1 \cong H_2 \Leftarrow^*_{\mathcal{R}} G_2$.*

A system $\langle \Sigma, \mathcal{R} \rangle$ is *terminating* if $\Rightarrow_{\mathcal{R},\cong}$ is terminating. Since $[G] \Rightarrow_{\mathcal{R},\cong} [H]$ if and only if $G \Rightarrow_{\mathcal{R}} H$, we have that $\langle \Sigma, \mathcal{R} \rangle$ is terminating if and only if $\Rightarrow_{\mathcal{R}}$ is terminating. The following result follows directly from Newman's Lemma.

Lemma 4. *A terminating hypergraph-transformation system is confluent if and only if it is locally confluent.*

In general, confluence is undecidable even for terminating graph-transformation systems. The precise result is as follows.

Theorem 1 ([13]). *The following problem is undecidable in general:*

Instance: *An injective and terminating graph-transformation system $\langle \Sigma, \mathcal{R} \rangle$ such that Σ_{V} is a singleton and Σ_{E} and $\mathcal{R}$ are finite.*
Question: *Is $\langle \Sigma, \mathcal{R} \rangle$ confluent?*

Note that since graph-transformation systems are special hypergraph-transformation systems, the result also applies to the latter.

4 Critical Pairs

Critical pairs consist of direct derivations of minimal size that are not independent. We recall their definition from [12, 13].

Definition 6 (Critical pair). Let $r_i \colon \langle L_i \leftarrow K_i \rightarrow R_i \rangle$ be rules, for $i = 1, 2$. A pair of direct derivations $U_1 \Leftarrow_{r_1, g_1} S \Rightarrow_{r_2, g_2} U_2$ is a *critical pair* if

(1) $S = g_1(L_1) \cup g_2(L_2)$ and
(2) the steps are not independent.

Moreover, we require $g_1 \neq g_2$ in case $r_1 = r_2$.

Two critical pairs $U_1 \Leftarrow_{r_1, g_1} S \Rightarrow_{r_2, g_2} U_2$ and $U'_1 \Leftarrow_{r_1, g'_1} S' \Rightarrow_{r_2, g'_2} U'_2$ are *isomorphic* if there is an isomorphism $f \colon S \rightarrow S'$ such that for $i = 1, 2$, $g'_i = f \circ g_i$. In the sequel, we equate isomorphic critical pairs so that condition (1) guarantees that a finite set of rules has only a finite number of critical pairs.

Given a critical pair $\Gamma \colon U_1 \Leftarrow S \Rightarrow U_2$, let $\mathrm{Persist}_\Gamma$ be the subhypergraph of S consisting of all nodes v such that both $\mathrm{tr}_{S \Rightarrow U_1}(v)$ and $\mathrm{tr}_{S \Rightarrow U_2}(v)$ are defined.

Definition 7 (Joinability). Let $\langle \Sigma, \mathcal{R} \rangle$ be a hypergraph-transformation system. A critical pair $\Gamma \colon U_1 \Leftarrow S \Rightarrow U_2$ is *joinable* if there are derivations $U_i \Rightarrow^*_{\mathcal{R}} X_i$, for $i = 1, 2$, and an isomorphism $f \colon X_1 \rightarrow X_2$. Moreover, Γ is *strongly joinable* if, in addition, for each node v in $\mathrm{Persist}_\Gamma$,

(1) $\mathrm{tr}_{S \Rightarrow U_1 \Rightarrow^* X_1}(v)$ and $\mathrm{tr}_{S \Rightarrow U_2 \Rightarrow^* X_2}(v)$ are defined and
(2) $f_{\mathrm{V}}(\mathrm{tr}_{S \Rightarrow U_1 \Rightarrow^* X_1}(v)) = \mathrm{tr}_{S \Rightarrow U_2 \Rightarrow^* X_2}(v)$.

In [13] it is shown that a hypergraph-transformation system is locally confluent if all its critical pairs are strongly joinable. Combining this result with Newman's Lemma yields a sufficient condition for the confluence of terminating systems.

Theorem 2 ([13]). *A terminating hypergraph-transformation system is confluent if all its critical pairs are strongly joinable.*

Example 2. The hypergraph-transformation system of Figure 3 is terminating since each of the rules reduces the size of a hypergraph it is applied to. Figure 5 shows that all critical pairs of the system are strongly joinable. (We indicate track morphisms by node names.) Thus, by Theorem 2, the system of Figure 3 is confluent. □

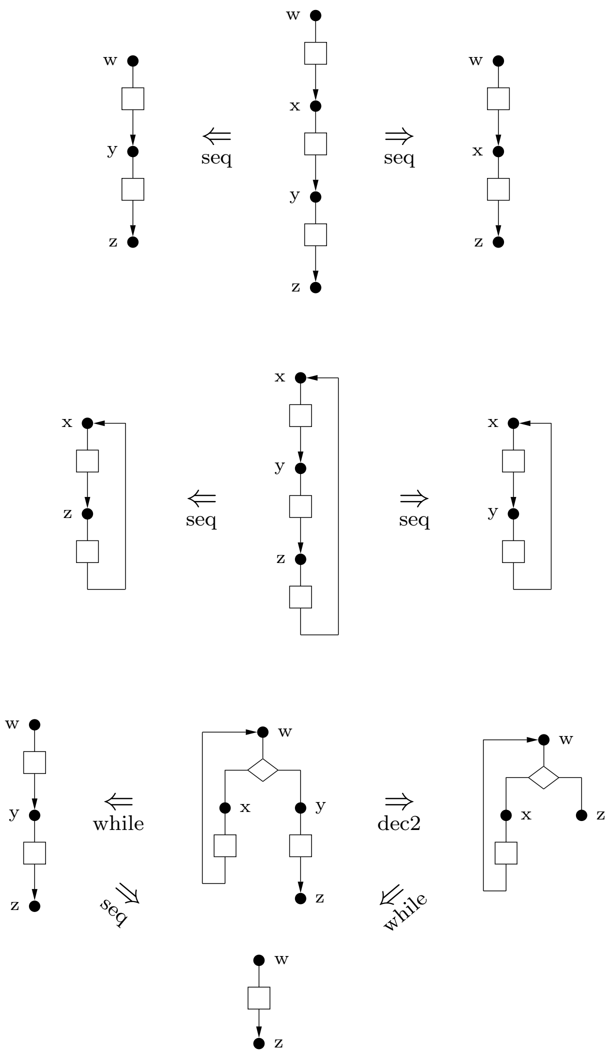
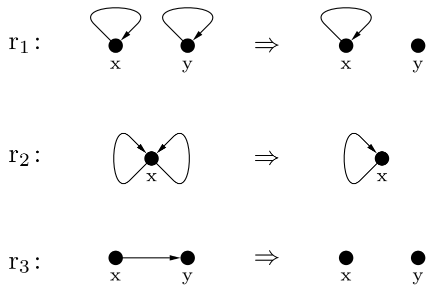


Fig. 5. The critical pairs of the system of Figure 3

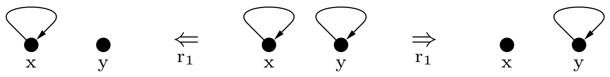
5 Coverable Systems

In general, by Theorem 1, confluence of a terminating hypergraph-transformation system $\langle \Sigma, \mathcal{R} \rangle$ cannot be decided by checking whether all critical pairs are strongly joinable. For, suppose we encounter a critical pair $U_1 \Leftarrow S \Rightarrow U_2$ that is joinable but not strongly joinable, that is, there are hypergraphs X_1 and X_2 such that $U_1 \Rightarrow^*_{\mathcal{R}} X_1 \cong X_2 \Leftarrow^*_{\mathcal{R}} U_2$ but no isomorphism $X_1 \to X_2$ is compatible with the track morphisms $\mathrm{tr}_{S \Rightarrow U_i \Rightarrow^* X_i}$. Then, assuming that all other critical pairs are joinable, $\langle \Sigma, \mathcal{R} \rangle$ may or may not be confluent. This is demonstrated by the following example.

Example 3. Consider the graph-transformation system $\langle \Sigma, \mathcal{R} \rangle$ consisting of singletons Σ_{V} and Σ_{E}, and the following rules:



This system is terminating as every rule application reduces the number of edges. It is also confluent since whenever $H_1 \Leftarrow^*_{\mathcal{R}} G \Rightarrow^*_{\mathcal{R}} H_2$, there are derivations $H_1 \Rightarrow^*_{\mathcal{R}} H'_1 \cong H'_2 \Leftarrow^*_{\mathcal{R}} H_2$ where H'_1 and H'_2 consist of $|V_G|$ nodes and either no edges (if G is loop-free) or one loop and no other edges. However, despite confluence, the critical pair

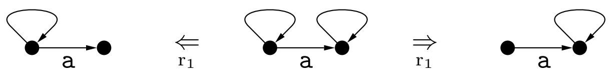


is not strongly joinable because the outer graphs are normal forms[1] and the isomorphism between them is not compatible with the track morphisms as required by condition (2) of Definition 7.

Thus, we cannot report non-confluence if we encounter a joinable critical pair that is not strongly joinable. On the other hand, joinability of all critical pairs does not guarantee confluence. Suppose, for instance, that we add an edge label a to Σ_{E}. Then all critical pairs are still joinable but confluence breaks down, as

[1] A graph G is a *normal form* with respect to a system $\langle \Sigma, \mathcal{R} \rangle$ if there is no graph H such that $G \Rightarrow_{\mathcal{R}} H$.

witnessed by the following counterexample:



□

This example also shows that signature extensions need not preserve confluence. In particular, hyperedge labels that do not occur in rules turn out to be crucial for ensuring that local confluence implies strong joinability of all critical pairs.

Given a hyperedge e in a hypergraph G, the pair $\langle \mathrm{lab}_G(e), \mathrm{mark}^*_G(\mathrm{att}_G(e))\rangle$ is the *profile* of e. If $\mathcal{R}$ is a set of hypergraph-transformation rules, we write $\mathrm{Prof}(\mathcal{R})$ for the set of all hyperedge profiles occurring in $\mathcal{R}$ and $\mathrm{Mark}(\mathcal{R})$ for the set of all node labels occurring in $\mathcal{R}$.

Definition 8 ($G^{\mathcal{R}}$ and $G^{\ominus}$). Let $\langle \Sigma, \mathcal{R}\rangle$ be a hypergraph-transformation system and $G \in \mathcal{H}(\Sigma)$. We define subhypergraphs $G^{\mathcal{R}}$ and $G^{\ominus}$ as follows:

(1) $G^{\mathcal{R}}$ consists of all hyperedges with profile in $\mathrm{Prof}(\mathcal{R})$ and all nodes with label in $\mathrm{Mark}(\mathcal{R})$.
(2) $G^{\ominus}$ consists of all hyperedges in $\mathrm{E}_G - \mathrm{E}_{G^{\mathcal{R}}}$, all attachment nodes of these hyperedges, and all nodes in $\mathrm{V}_G - \mathrm{V}_{G^{\mathcal{R}}}$.

It follows that $G = G^{\mathcal{R}} \cup G^{\ominus}$, where $G^{\mathcal{R}}$ and $G^{\ominus}$ may share some attachment nodes of edges in $G^{\ominus}$. These shared nodes cannot be removed by any rule in $\mathcal{R}$, by the dangling condition for direct derivations.

Definition 9 (Cover). Given a critical pair Γ of a hypergraph-transformation system $\langle \Sigma, \mathcal{R}\rangle$, a *cover* for Γ is a hypergraph $C \in \mathcal{H}(\Sigma)$ such that

(1) $\mathrm{Persist}_\Gamma \subseteq C$,
(2) $C^{\ominus} = C$, and
(3) for every image $\tilde{C}$ of C, there is a unique surjective morphism $C \to \tilde{C}$.

Remarks.

1. By condition (2), the profiles of the hyperedges in C are distinct from those in $\mathrm{Prof}(\mathcal{R})$. Also, since all node labels in $\mathrm{Persist}_\Gamma$ belong to $\mathrm{Mark}(\mathcal{R})$, (1) and (2) imply that each node in $\mathrm{Persist}_\Gamma$ is incident to some hyperedge in C.
2. Intuitively, C uniquely identifies the nodes in $\mathrm{Persist}_\Gamma$ in that for every image $\tilde{C}$ of C, each node in $\mathrm{Persist}_\Gamma$ corresponds to a unique node in $\tilde{C}$. Moreover, the rules in $\mathcal{R}$ can affect C at most by merging some nodes in $\mathrm{Persist}_\Gamma$.
3. By condition (3), C does not possess nontrivial automorphisms. That is, the identity $\mathrm{id}_C \colon C \to C$ is the only isomorphism on C.

Example 4. Consider a critical pair $\Gamma\colon U_1 \Leftarrow S \Rightarrow U_2$.

1. If $\mathrm{Persist}_\Gamma = \emptyset$, then the empty hypergraph is a cover for Γ.
2. Let $\mathrm{Persist}_\Gamma$ consist of a single node v with label m. If there is some $l \in \Sigma_\mathrm{E}$ such that $m \in \mathrm{Type}(l)$ and $\langle l, m\rangle \notin \mathrm{Prof}(\mathcal{R})$, then the hypergraph C consisting of v and an hyperedge e with $\mathrm{lab}_C(e) = l$ and $\mathrm{att}_C(e) = v$ is a cover for Γ. Alternatively, if $mm \in \mathrm{Type}(l)$ and $\langle l, mm\rangle \notin \mathrm{Prof}(\mathcal{R})$, then the graph C consisting of v and an edge e with $\mathrm{lab}_C(e) = l$ and $\mathrm{att}_C(e) = vv$ is a cover for Γ.
3. Let $\mathrm{Persist}_\Gamma$ consist of nodes $v_1,\ldots,v_n$ with $n \geq 2$ and $\mathrm{mark}_S(v_i) = m_i$, for $i = 1,\ldots,n$. If there is $l \in \Sigma_\mathrm{E}$ such that $m_1\ldots m_n \in \mathrm{Type}(l)$ and $\langle l, m_1\ldots m_n\rangle \notin \mathrm{Prof}(\mathcal{R})$, then C consisting of $v_1,\ldots,v_n$ and an hyperedge e with $\mathrm{lab}_C(e) = l$ and $\mathrm{att}_C(e) = v_1\ldots v_n$ is a cover for Γ. Alternatively, suppose that there are distinct labels $l_1,\ldots,l_{n-1} \in \Sigma_\mathrm{E}$ such that for $i = 1,\ldots,n-1$, $m_i m_{i+1} \in \mathrm{Type}(l_i)$ and $\langle l_i, m_i m_{i+1}\rangle \notin \mathrm{Prof}(\mathcal{R})$. Then a graph cover C for Γ is given by $v_1,\ldots,v_n$ and edges $e_1,\ldots,e_{n-1}$ where for $i = 1,\ldots,n-1$, $\mathrm{lab}_C(e_i) = l_i$ and $\mathrm{att}_C(e_i) = v_i v_{i+1}$. (For instance, the critical pair discussed in Example 3 can be covered in this way after the edge label a with Type(a) $= \{\bullet\bullet\}$ has been added to Σ_E.) □

Figure 6 shows the alternative covers of Example 4.3 for a critical pair with n persistent nodes. Note that $l_1,\ldots,l_{n-1}$ need to be distinct as otherwise condition (3) of Definition 9 may be violated.

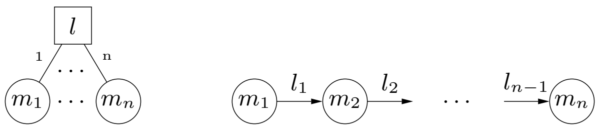


Fig. 6. Alternative covers for a critical pair with n persistent nodes

Definition 10 (Coverable system). A hypergraph-transformation system is *coverable* if for each of its critical pairs there exists a cover.

Our main result is that for coverable systems, local confluence is equivalent to the strong joinability of all critical pairs.

Theorem 3. *A coverable hypergraph-transformation system is locally confluent if and only if all its critical pairs are strongly joinable.*

Theorem 2 establishes the "if"-direction of this result. We outline the proof for the converse, which is based on extending critical pairs with their covers. Consider a critical pair $\Gamma\colon U_1 \Leftarrow S \Rightarrow U_2$ and a cover C for Γ such that $S \cap C = \mathrm{Persist}_\Gamma$. Then there are extended direct derivations $\widehat{U}_1 \Leftarrow \widehat{S} \Rightarrow \widehat{U}_2$, where $\widehat{S} = S \cup C$. By local confluence, there are hypergraphs X_1 and X_2 such that

$\widehat{U}_1 \Rightarrow^* X_1 \cong X_2 \Leftarrow^* \widehat{U}_2$. The derivations $\widehat{S} \Rightarrow \widehat{U}_i \Rightarrow^* X_i$, $i = 1, 2$, preserve the nodes in $\mathrm{Persist}_\Gamma$ because the latter are incident to edges in C . Hence, after taking the cover C off, one obtains restricted derivations $S \Rightarrow U_i \Rightarrow^* \overline{X}_i$, $i = 1, 2$, that satisfy condition (1) of Definition 7. Moreover, one can show that $\overline{X}_1 = X_1^{\mathcal{R}} \cong X_2^{\mathcal{R}} = \overline{X}_2$. Restricting the morphisms $\mathrm{tr}_{\widehat{S} \Rightarrow \widehat{U}_i \Rightarrow^* X_i}$, $i = 1, 2$, to $\widehat{S}^\ominus$ and $X_i^\ominus$ yields surjective morphisms $t_i \colon \widehat{S}^\ominus \to X_i^\ominus$. Also, given an isomorphism $f \colon X_1 \to X_2$, its restriction $f^\ominus \colon X_1^\ominus \to X_2^\ominus$ is an isomorphism. Hence both $f^\ominus \circ t_1 \colon \widehat{S}^\ominus \to X_2^\ominus$ and $t_2 \colon \widehat{S}^\ominus \to X_2^\ominus$ are surjective morphisms. Since $\widehat{S}^\ominus = C$, condition (3) of Definition 9 implies $f \circ t_1 = t_2$. It then follows that condition (2) of Definition 7 is satisfied. Thus Γ is strongly joinable.

Assumption. For the rest of this section, we consider hypergraph-transformation systems $\langle \Sigma, \mathcal{R} \rangle$ in which Σ_{V}, Σ_{E} and $\mathcal{R}$ are finite.

As a consequence of Theorem 3, confluence of terminating coverable systems is equivalent to the strong joinability of all critical pairs. This allows to decide confluence by testing for the latter property.

Corollary 1. *Confluence is decidable for coverable hypergraph-transformation systems that are terminating.*

Given a terminating and coverable system, Algorithm 1 checks whether all critical pairs are strongly joinable by extending critical pairs with covers and then testing for simple joinability of all "covered pairs". By the proof of Theorem 3, joinability of a covered pair implies strong joinability of the underlying critical pair. Given a covered pair $\widehat{\Gamma} \colon \widehat{U}_1 \Leftarrow \widehat{S} \Rightarrow \widehat{U}_2$, one nondeterministically computes a normal form X_i of $\widehat{U}_i$, for $i = 1, 2$, and checks whether X_1 and X_2 are isomorphic. If they are, then the critical pair Γ underlying $\widehat{\Gamma}$ is strongly joinable, otherwise a counterexample to confluence has been found.

Algorithm 1 Decision procedure for confluence

```
Input: a terminating and coverable hypergraph-transformation system $\langle \Sigma, \mathcal{R} \rangle$ and its
  set of critical pairs CP
  for all $\Gamma \colon U_1 \Leftarrow_{r_1,g_1} S \Rightarrow_{r_2,g_2} U_2$ in CP do
    {let $C$ be a cover for $\Gamma$ such that $S \cap C = \mathrm{Persist}_\Gamma$}
    $\widehat{S} := S \cup C$
    {for $i = 1, 2$, let $\widehat{g}_i$ be the extension of $g_i$ to $\widehat{S}$}
    for $i = 1$ to 2 do
      construct a derivation $\widehat{S} \Rightarrow_{r_i,\widehat{g}_i} \widehat{U}_i \Rightarrow^*_{\mathcal{R}} X_i$ such that $X_i$ is a normal form
    end for
    if $X_1 \ncong X_2$ then
      return "non-confluent"
    end if
  end for
  return "confluent"
```

Example 5. Consider again the hypergraph-transformation system of Example 1. Suppose that its typing allows a rhomb hyperedge to have two attachment nodes, besides the version with three attachment nodes used in the rules. Then each critical pair of this system can be covered and Algorithm 1 determines that the system is confluent. For example, Figure 7 shows the extended version of the bottom critical pair of Figure 5 and its joining derivations.

The graph-transformation system of Example 3, on the other hand, is not coverable. It becomes coverable after the edge label a has been added to the signature, when Algorithm 1 determines that the resulting system is non-confluent. □

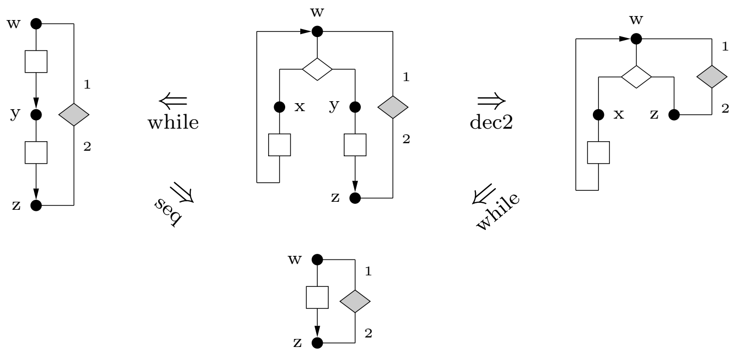


Fig. 7. An extended critical pair of the system of Figure 3

Particular classes of hypergraph- and graph-transformation systems for which confluence is decidable can be obtained by specialising Corollary 1 with the conditions given in Example 4.3 or with similar conditions. For instance, in the case of graph transformation, another sufficient condition for terminating systems is that for each critical pair Γ with persistent nodes $v_1, \ldots, v_n$, there are distinct labels $l_1, \ldots, l_n \in \Sigma_\mathrm{E}$ such that for $i = 1, \ldots, n$, $\mathrm{mark}_S(v_i)\mathrm{mark}_S(v_i) \in \mathrm{Type}(l_i)$ and $\langle l_i, \mathrm{mark}_S(v_i)\mathrm{mark}_S(v_i)\rangle \notin \mathrm{Prof}(\mathcal{R})$. In this case a cover can be constructed by attaching to $v_1, \ldots, v_n$ loops labelled with $l_1, \ldots, l_n$.

In the case of hypergraph transformation, a sufficient condition for the decidability of confluence (of terminating systems) can be given purely in terms of the signature Σ. We call a signature Σ *universal* if for each $l \in \Sigma_\mathrm{E}$, $\mathrm{Type}(l) = \Sigma_\mathrm{V}^*$.

Corollary 2. *Confluence is decidable for terminating hypergraph-transformation systems with universal signatures.*

For, if hyperedges can have arbitrary sequences of attachment nodes, we can cover critical pairs with hyperedges that have longer attachment sequences than any hyperedges in rules by using repeated nodes in the attachment.

6 Conclusion

Confluence is an undecidable property of terminating graph- and hypergraph-transformation systems. We have identified coverable systems as a subclass that comes with a decision procedure for confluence. The class is nontrivial and properly includes all hypergraph-transformation systems with universal signatures.

A topic for future work is to extend Algorithm 1 such that it decides confluence for certain non-coverable systems. The idea is to add to the signature of an input system a hyperedge label whose typing allows to cover all critical pairs. One then runs the algorithm as before: if all extended pairs are joinable, one can conclude that the underlying critical pairs are strongly joinable and hence that the system is confluent. However, if a non-joinable extended pair is encountered whose underlying critical pair is joinable, then the procedure has to give up because the input system may or may not be confluent.

References

1. Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An algebraic theory of graph reduction. *Journal of the ACM*, 40(5):1134–1164, 1993.
2. Franz Baader and Tobias Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.
3. Adam Bakewell, Detlef Plump, and Colin Runciman. Specifying pointer structures by graph reduction. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 30–44. Springer-Verlag, 2004.
4. Marc Bezem, Jan Willem Klop, and Roel de Vrijer, editors. *Term Rewriting Systems.* Cambridge University Press, 2003.
5. Hans L. Bodlaender and Babette van Antwerpen-de Fluiter. Reduction algorithms for graphs of small treewidth. *Information and Computation*, 167(2):86–119, 2001.
6. Ronald V. Book and Friedrich Otto. *String-Rewriting Systems.* Texts and Monographs in Computer Science. Springer-Verlag, 1993.
7. Rodney Farrow, Ken Kennedy, and Linda Zucconi. Graph grammars and global program data flow analysis. In *Proc. 17th Annual Symposium on Foundations of Computer Science*, pages 42–56. IEEE, 1976.
8. Rubino Geiß, Gernot Veit Batz, Daniel Grund, Sebastian Hack, and Adam M. Szalkowski. GrGen: A fast SPO-based graph rewriting tool. In *Proc. International Conference on Graph Transformation (ICGT 2006)*, volume 4178 of *Lecture Notes in Computer Science*, pages 383–397. Springer-Verlag, 2006.
9. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
10. M.H.A. Newman. On theories with a combinatorial definition of "equivalence". *Annals of Mathematics*, 43(2):223–243, 1942.
11. Ulrich Nickel, Jörg Niere, and Albert Zündorf. The FUJABA environment. In *Proc. International Conference on Software Engineering (ICSE 2000)*, pages 742–745. ACM Press, 2000.

12. Detlef Plump. Hypergraph rewriting: Critical pairs and undecidability of confluence. In Ronan Sleep, Rinus Plasmeijer, and Marko van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 15, pages 201–213. John Wiley, 1993.
13. Detlef Plump. Confluence of graph transformation revisited. In Aart Middeldorp, Vincent van Oostrom, Femke van Raamsdonk, and Roel de Vrijer, editors, *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*, volume 3838 of *Lecture Notes in Computer Science*, pages 280–308. Springer-Verlag, 2005.
14. Detlef Plump. The graph programming language GP. In *Algebraic Informatics, Third International Conference (CAI 2009), Revised Selected and Invited Papers*, volume 5725 of *Lecture Notes in Computer Science*. Springer-Verlag, 2009. To appear.
15. Jan Rekers and Andy Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1997.
16. Gabriele Taentzer. AGG: A graph transformation environment for modeling and validation of software. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*, volume 3062 of *Lecture Notes in Computer Science*, pages 446–453. Springer-Verlag, 2004.

Dr. Detlef Plump
Department of Computer Science
The University of York
York YO10 5DD (United Kingdom)
det@cs.york.ac.uk
http://www.cs.york.ac.uk/~det

Detlef Plump did his diploma thesis on *jungle evaluation* with Hans-Jörg Kreowski in 1986. He was a member of his group from 1991 to 2000. As a research associate, he worked in the DFG-funded project *Jungle Rewriting*. Supervised by Hans-Jörg Kreowski, he received his doctoral degree in 1993. He obtained habilitation in 1999, and moved to the University of York in 2000.

Refactoring Object-Oriented Systems

Christoph Schulz, Michael Löwe, and Harald König

Abstract. Refactoring of information systems is hard, for two reasons. On the one hand, large databases exist which have to be adjusted. On the other hand, many programs access that data. These programs all have to be migrated in a consistent manner such that their semantics does not change. It cannot be relied upon, however, that no running processes exist during such a migration. Consequently, a refactoring of an information system needs to take care of the migration of data, programs, *and* processes. This paper introduces a model for complete object-oriented systems, describing the schema level with classes, associations, operations, and inheritance as well as the instance level with objects, links, methods, and messages. Methods are expressed by special double-pushout graph transformations. Homomorphisms are used for the typing of the instance level as well as for the description of refactorings which specify the addition, folding, and unfolding of schema elements. Finally, a categorial framework is presented which allows to derive instance migrations from schema transformations in such a way that programs and processes to the old schema are correctly migrated into programs and processes to the new schema.

1 Introduction

During the engineering and use of information systems, data and software undergo many modifications. These modifications can be divided into two categories. The first category contains all modifications that have a direct and externally visible impact on the functionality of the software or on the information content of the database. The second category consists of modifications which only *prepare* modifications of the first category and which, by themselves, do not lead to changes in the behaviour of the software or in the meaning of the data under transformation. Modifications of the second category are called "refactorings" [1]. They provide a major method to quickly adapt software to constantly changing requirements.

Refactorings are expected to be applied multiple times in different but similar situations. This is comparable to *design patterns* in software engineering which have emerged in the last twenty years [2, 3]. Consequently, a suitably general specification of a refactoring is necessary. This, however, requires a certain level of *abstraction* for the software and the data to be transformed. Such an abstraction is often called *schema* or *model* and describes important structural aspects of the data and software, which are *instances* of, or *typed* in, this schema. Today, the "object-oriented view of life" dominates the field of software engineering. Therefore, models are typically object-oriented and try to capture the structure by grouping similar objects into *classes* and describing relations between them by various types of *associations*.

Two typical object-oriented refactorings are "Introduce a new superclass" (Fig. 1) and "Move the origin of an association from a subclass to a superclass", as shown in Fig. 2. A combined application of these two refactorings on the schema in Fig. 3a could be used to prepare the model for an extension by an additional subclass of *Customer*, e. g. *CorporateCustomer* (Fig. 3b and 3c).[1]

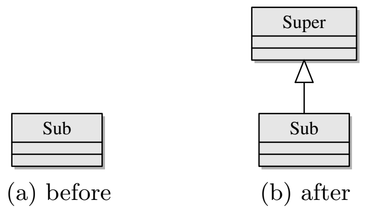


Fig. 1: Refactoring "Introduce a new superclass"

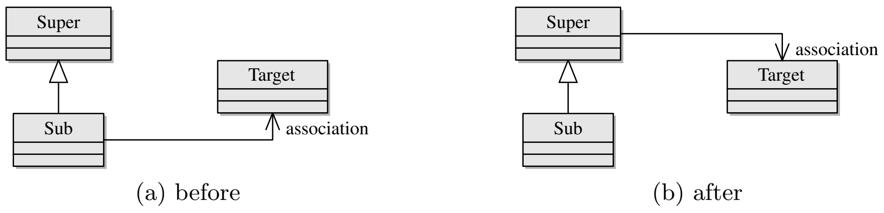
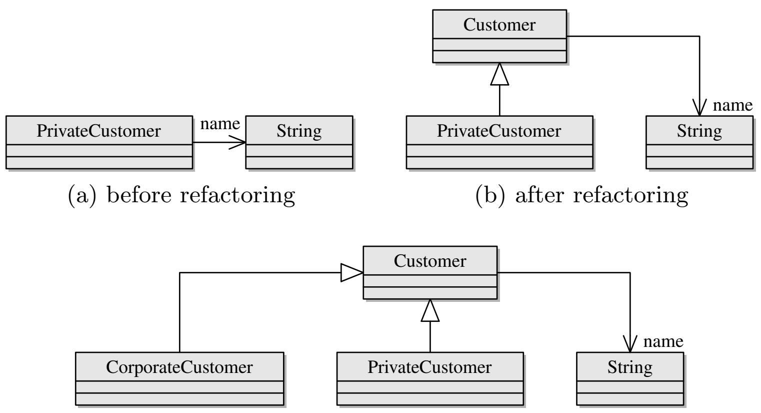


Fig. 2: Refactoring "Move the origin of an association from a subclass to a superclass"

It is important to consider the *consequences* of a refactoring. Obviously, the more general the structures are which are about to be transformed, the more instances are likely to be affected. Changing a data schema may not only require the data typed in this schema to be adjusted, but may also affect the software which uses the schema structures to access and manipulate the data. Changing a software model may have no consequences on the data but will probably influence programs (which can be considered *implementations* of the software model) and processes (which are programs under execution). We call the instance changes that follow from a model refactoring the *migration* induced by that refactoring. For the time being, little has been written about how refactoring data models results in migrations of dependent programs, and even less has been written

[1] All class diagrams are specified in the UML [4].



(a) before refactoring (b) after refactoring

(c) final aim

Fig. 3: Refactoring an exemplary object-oriented model

about refactoring and induced migration of *whole systems*, which we define to consist of data, programs, and processes, all typed in the same schema.[2]

This paper contributes to this topic by providing a graph-like mathematical model which allows to specify object-oriented systems as well as schema refactorings. To describe data together with their schema, *graph structures* conforming to the model are used. Nodes of such graph structures represent classes (schema) or objects (instance), edges represent associations (schema) or links (instance). *Homomorphisms* between such graph structures express *typings*, (parts of) *refactorings*, and *migrations*.

These graph structures can be used to describe software models and processes, as well: An operation is simply a special node within the graph structure representing the schema, and edges originating from an operation constitute parameters. Analogously, at the instance level, messages and arguments are special nodes and edges, which are typed in operations and parameters at the schema level by an appropriate homomorphism. In the mathematical description of the model, the data and software constructs are separated through the use of special *predicates*.

Programs are somewhat different, as they do not specify a single state but rather a state *transition* which is performed when the program is executed. In this paper, programs are considered to consist of a (possibly large) set of methods, where each method describes a single state transition. Each such transition speficies how a message of a certain type is processed when all necessary preconditions are met; examples are assignments, method calls, or evaluation of expressions. As program states are described by (parts of) graph structures at the instance level, it follows that state transitions can be adequately specified by

[2] See [5] and especially the bibliography contained therein for a general overview on software refactoring.

the use of *graph structure transformations*. This paper chooses the DPO approach for describing and applying graph structure transformations[3]. Consequently, a method is represented by a span of homomorphisms, and applying a method to a given program state is computed by two pushout diagrams.

Results of category theory are used to compute induced migrations from schema refactorings. It will be shown, however, that certain restrictions must be obeyed in order to guarantee reasonable results. Fortunately, these restrictions are met by the practical examples.

The paper is organized as follows. Section 3 incrementally introduces a graph structure specification *MP* with positive Horn formulas which constitutes the foundation of the mathematical description of data and software. The category $\mathbf{Alg}(MP)$ of all *MP*-systems and *MP*-homomorphisms, as well as the (sub-)categories $\mathbf{Alg}(MP){\downarrow}S$ and $\mathbf{Sys}(S)$ with a fixed schema S, represent the universe of discourse for the following sections. Section 4 explains how methods are represented as DPO rules and introduces requirements that are necessary to use DPO graph structure transformations successfully in the categories mentioned above. Section 5 addresses the migration of data and processes. Section 6 discusses the migration of programs and contains the main result of this paper, namely that the migration of methods preserves their semantics for new processes as well as for old processes reviewed under the transformed schema. Section 7 outlines three main directions for future research.

Due to lack of space, this paper does not contain any proofs. All the proofs can be found in [7].

2 Related Work

There exist approaches for modelling programs as algebraic graph transformation rules [8–10]. However, they fail in various ways to be suitable for our purposes. The approach in [8] does not support inheritance. Furthermore, program execution is "destructive", i. e., repetitive control flow constructs as loops cannot be modelled directly but have to be simulated through recursion, a work-around which is not necessary in our approach as the control flow structures are not modified by program execution. The approach presented in [9, 10] does not have a notion of a schema in which programs and processes are typed. This missing link makes it hard if not impossible to compute induced migrations for programs and processes when the data schema is changed. Finally, both approaches consider objects to be opaque, whereas in our approach, each object is decomposed into parts called "particles" which reflect the class hierarchy. This rich object structure makes it possible to type the instance level in a schema without resorting to special typing morphisms or type graph flattening as proposed in [6, 11, 12]. Finally, our approach is unique in the respect that it combines a program and process model with a model for schema transformations and induced migrations.

[3] DPO stands for "Double Pushout"; the approach is presented in e. g. [6].

3 Models and Instances

The schema and the instance level of object-oriented systems are modelled by systems wrt. an extended specification.[4] An *extended specification* $\mathit{Spec} = (\Sigma, H(X))$ is an extended signature together with a set of positive Horn formulas $H(X)$ over a set of variables X. An *extended signature* $\Sigma = (S, \mathit{OP}, P)$ consists of a set of *sorts* S, a family of *operation symbols* $\mathit{OP} = (\mathit{OP}_{w,s})_{w \in S^*, s \in S}$, and a family of *predicates* $P = (P_w)_{w \in S^*}$ such that $=_s \in P_{s\,s}$ for each sort $s \in S$. A *system* A wrt. an extended signature $\Sigma = (S, \mathit{OP}, P)$, short Σ-system, consists of a family of *carrier sets* $(A_s)_{s \in S}$, a family of *operations* $(op^A \colon A_w \to A_s)_{w \in S^*, s \in S, op \in \mathit{OP}_{w,s}}$, and a family of *relations* $(p^A \subseteq A_w)_{w \in S^*, p \in P_w}$ such that $=_s^A \subseteq A_s \times A_s$ is the diagonal relation for each sort s.[5] A *system* A wrt. an extended specification $\mathit{Spec} = (\Sigma, H(X))$ is a Σ-system such that all axioms are valid in A. A *Σ-homomorphism* $h \colon A \to B$ between two Σ-systems A und B wrt. an extended signature $\Sigma = (S, \mathit{OP}, P)$ is a family of mappings $(h_s \colon A_s \to B_s)_{s \in S}$, such that the mappings are compatible with the operations and relations, i. e., $h_s \circ op^A = op^B \circ h_w$ for all operation symbols $op \colon w \to s$ and $h_w(p^A) \subseteq p^B$ for all predicates $p \colon w$ where $w = s_1 s_2 \ldots s_n \in S^*$.[6] Each Σ-homomorphism $h \colon A \to B$ between two *Spec*-systems A und B wrt. an extended specification $\mathit{Spec} = (\Sigma, H(X))$ is called a *Spec*-homomorphism.

The (first) model version for classes and associations is just graphs as depicted in Fig. 4. Nodes correspond to classes and edges correspond to associations. In Fig. 5a, an exemplary UML schema is presented. The underlying graph for this schema is shown in Fig. 5b. Figure 5c illustrates the resulting *Graph* system.

Graph =
sorts

N	(nodes)
E	(edges)

opns

$s \colon E \to N$	(source node of an edge)
$t \colon E \to N$	(target node of an edge)

Fig. 4: The *Graph* signature

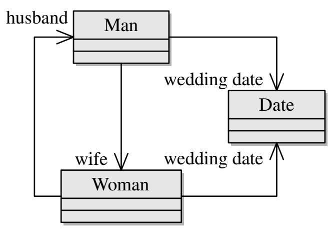
An instance of a given schema S is represented as a system I wrt. the same signature, together with a typing homomorphism $\mathit{type} \colon I \to S$. At the instance level, nodes represent objects and edges constitute links.

The next model version provides the possibility to model inheritance relations between classes by an additional binary predicate *under*: If, in a system S, a

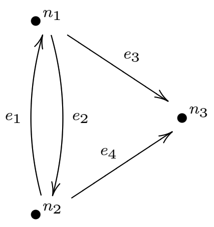
[4] See [13] for the special case when signatures consist of only one sort.

[5] Given $w = s_1 s_2 \ldots s_n$, A_w is an abbreviation for the product set $A_{s_1} \times A_{s_2} \times \cdots \times A_{s_n}$.

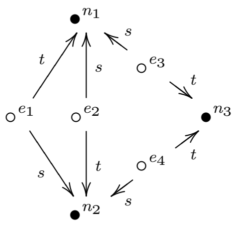
[6] Given $w = s_1 s_2 \ldots s_n$, $h_w(x_1, x_2, \ldots, x_n)$ is an short-hand notation for the term tuple $(h_{s_1}(x_1), h_{s_2}(x_2), \ldots, h_{s_n}(x_n))$.



(a) UML schema



(b) Representing graph



(c) Representing algebra

Fig. 5: A system for the *Graph* signature

class A is "under" a class B, i. e., if it is a subclass of B, then the relation $under^S$ contains the pair (A, B). The specification MP_1 is shown in Fig. 6.[7] As inheritance is hierarchical and, therefore, a partial order, it is reasonable to formulate corresponding requirements for the *under* relation.

$MP_1 = Graph +$
prds
$under\colon N\ N$ (subnode of)
axms
inheritance

$$x \in N : under(x, x) \qquad \text{(reflexivity)}$$

$$x, y \in N : under(x, y) \wedge under(y, x) \Rightarrow x = y \qquad \text{(antisymmetry)}$$

$$x, y, z \in N : under(x, y) \wedge under(y, z) \Rightarrow under(x, z) \qquad \text{(transitivity)}$$

Fig. 6: The MP_1 specification including the predicate *under*

While the use of the predicate *under* is quite natural at the schema level, the question arises how it is to be interpreted at the instance level. Typically, objects are seen as monolithic entities even if they are mapped to multiple types in the class hierarchy. In this paper, we follow a different approach and consider objects to consist of a set of interconnected parts called *particles*. Each particle is represented by a node and is typed in a specific class in the schema. The advantage of this approach is that the structure of an object is made visible and

[7] The "MP" stands for "Model Part" and describes the fact that systems for this model represent only a part (schema or instance) of the whole object-oriented system.

resembles the object's type hierarchy at the schema level allowing proper typing of links.[8] Figure 7 shows an exemplary instance level for the schema in Fig. 3b.[9]

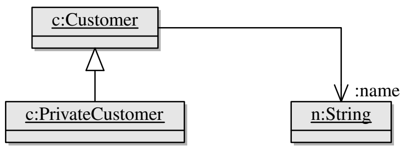


Fig. 7: Objects represented by particles

The model currently allows an object to contain more than one particle for the same type. This is typically forbidden by object-oriented languages.[10] In order to implement this requirement, we want to specify something like that:

$$(1) \qquad x, y \in N : \mathit{rel}(x, y) \wedge \mathit{type}(x) = \mathit{type}(y) \Rightarrow x = y \qquad \text{(unique particles)}$$

Here, another predicate *rel* has been used which shall be fulfilled when two particles belong to the same object and, therefore, are *rel*ated. Obviously, this predicate describes an equivalence relation as it is reflexive, symmetric, and transitive. Furthermore, the equivalence comprises the *under* relation, because each two particles connected by the *under* relation belong to the same object and, consequently, are part of the *rel* relation.[11] The resulting specification MP_2 is shown in Fig. 8.[12]

Another issue currently not resolved is association multiplicity: In our model, all associations are many-to-many, because the number of links at the instance level is not restricted in any way. However, many-to-one associations are often necessary in object-oriented schemas to allow at most one linked target object for any given association and source object. To achieve this, a formula like the following one is necessary which disallows the existence of two links which are instances of the same association and start at the same particle:[13]

[8] For the purpose of typing, simple homomorphisms are sufficient; there is no need to introduce homomorphisms "up to inheritance".

[9] We do not use a different notation for schema inheritance and the relationship between particles because it can be easily deduced from the context which relation is meant.

[10] An exception to this rule is the programming language *C++* which explicitly allows this behaviour [14].

[11] Note, however, that the *rel* relation might not be *generated* by the *under* relation. That means that there may exist related particles that do *not* belong to the same object. However, this is avoided in all practical examples.

[12] Note that reflexivity of the *rel* relation need not be specified by an axiom as it is a consequence from the combination of the first inheritance axiom and the last component axiom.

[13] Note that this axiom disallows multi-valued associations completely. This is desired, however, as only single-valued associations can be dereferenced at the instance level

$MP_2 = MP_1 +$
prds
$rel\colon N\ N$ (related to)
axms
components

$$x, y \in N : rel(x, y) \Rightarrow rel(y, x) \quad \text{(symmetry)}$$

$$x, y, z \in N : rel(x, y) \wedge rel(y, z) \Rightarrow rel(x, z) \quad \text{(transitivity)}$$

$$x, y \in N : under(x, y) \Rightarrow rel(x, y) \quad \text{(components)}$$

Fig. 8: The MP_2 specification including the predicate *rel*

(2)

$$x, y \in E : source(x) = source(y) \wedge type(x) = type(y) \Rightarrow x = y \quad \text{(at most one)}$$

The axioms (1) and (2) are called *typing axioms*.

The last issue is the integration of software constructs, namely operations, parameters, messages, arguments, and methods. In order to model operations and messages, the specification MP_2 is extended by a unary predicate called *software* which distinguishes between class nodes and operation nodes in schemas and between object nodes and message nodes in instances. The distinction between association edges and parameter edges on the one hand and between link edges and argument edges on the other hand is deduced from the context: If an edge starts at a class/object it is considered an association/link, otherwise it constitutes a parameter/argument. The resulting specification is shown in Fig. 9.[14]

An example of an operation is displayed in Fig. 10a, a message for this operation is shown in Fig. 10b. The modelling of methods builds upon the mapping of messages and arguments into the model and is described in the next section.

We use the following notation: $\mathbf{Alg}(MP)$ denotes the category of all *MP*-systems and *MP*-homomorphisms. The arrow category $\mathbf{Alg}(MP)^2$ consists of all typed instances which do not necessarily fulfil the typing axioms. The full subcategory $\mathbf{Sys} \subseteq \mathbf{Alg}(MP)^2$ restricts the arrow category to those typed instances conforming to these axioms. Given a fixed schema system S, the slice category $\mathbf{Alg}(MP){\downarrow}S$ expresses the category of all typed instances for the system S, and the category $\mathbf{Sys}(S)$ denotes the full subcategory of $\mathbf{Alg}(MP){\downarrow}S$ whose objects fulfil the typing axioms.[15]

in a well-defined way. Multi-valued associations need further information (e. g. an index) when accessing links, which does not fit well in our graph structure model.

[14] The model allows parameters to point to operations; this is reasonable as it enables to model basic statements like *if-then-else* as operations.

[15] Obviously, $\mathbf{Sys}(S)$ is also a subcategory of $\mathbf{Sys}$.

MP =
sorts

N	(nodes)
E	(edges)

opns

$s\colon E \to N$	(source node of an edge)
$t\colon E \to N$	(target node of an edge)

prds

$\mathit{under}\colon N\ N$	(subnode of)
$\mathit{rel}\colon N\ N$	(related to)
$\mathit{software}\colon N$	(software part vs. data part)

axms

inheritance

(3) $x \in N : \mathit{under}(x, x)$ (reflexivity)

(4) $x, y \in N : \mathit{under}(x, y) \wedge \mathit{under}(y, x) \Rightarrow x = y$ (antisymmetry)

(5) $x, y, z \in N : \mathit{under}(x, y) \wedge \mathit{under}(y, z) \Rightarrow \mathit{under}(x, z)$ (transitivity)

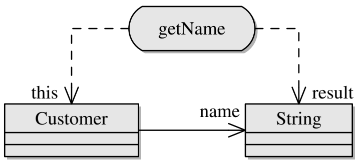
components

(6) $x, y \in N : \mathit{rel}(x, y) \Rightarrow \mathit{rel}(y, x)$ (symmetry)

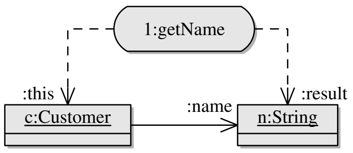
(7) $x, y, z \in N : \mathit{rel}(x, y) \wedge \mathit{rel}(y, z) \Rightarrow \mathit{rel}(x, z)$ (transitivity)

(8) $x, y \in N : \mathit{under}(x, y) \Rightarrow \mathit{rel}(x, y)$ (components)

Fig. 9: The complete *MP* specification



(a) Operation with parameters



(b) Message with arguments

Fig. 10: Software constructs

Furthermore, there exists a functor $\mathcal{F}\colon \mathbf{Alg}(\mathit{MP})^2 \to \mathbf{Sys}$ which transforms any typed instance by factoring through the congruence generated by the axioms such that the resulting typed instance fulfils the typing axioms. This functor is an epireflector because of the freeness property of the factorisation. This functor never changes the schema:

Lemma 3.1 ([7, Lemma 13.13]). *Let* $D ::= I \xrightarrow{\mathit{type}_I} S$ *be a typed instance, and let* $\mathcal{F}_{\mathrm{Ob}}(D)$ *be the typed instance* $I' \xrightarrow{\mathit{type}_{I'}} S'$*. Then* $S \cong S'$ *holds.* □

From this lemma, it follows that the functor $\mathcal{F}$ can be restricted to a slice category for some fixed schema S, resulting in a family of epireflectors $\mathcal{F}^S\colon \mathbf{Alg}(\mathit{MP}){\downarrow}S \to \mathbf{Sys}(S)$ for each possible schema S.

Summarising our results so far, an object-oriented schema is modelled as an *MP*-system S. An instance of this schema consists of an *MP*-system I and a typing *MP*-homomorphism $\mathit{type}\colon I \to S$ such that $I \xrightarrow{\mathit{type}} S$ is an object of the category $\mathbf{Sys}(S)$. Every schema instance $\mathit{type}\colon I \to S$ in $\mathbf{Alg}(\mathit{MP}){\downarrow}S$ can be uniquely transformed into an object of the category $\mathbf{Sys}(S)$ by the epireflector $\mathcal{F}^S$.

4 Methods

A *method* is part of a program and specifies how the program reacts on a message for a certain operation. It constitutes an *implementation* of an operation. Here, the set of operations consists not only of "user-defined" operations but also of operations for evaluating expressions and for representing statements.[16] In other words, for each construct which influences the behaviour of a process, there exists a corresponding operation. A *program* is then a collection of methods such that all operations for which messages exist are implemented.

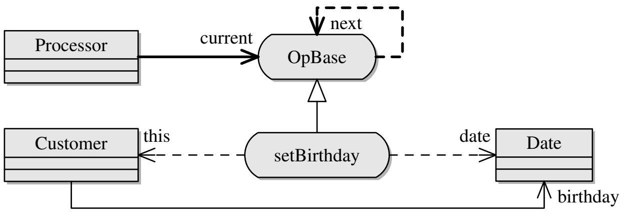
Each method is implemented by a single *DPO rule* [6] which is properly typed in the schema S. A typed DPO rule is a span $L \xleftarrow{l} K \xrightarrow{r} R$ together with the typings $L \xrightarrow{\mathit{type}_L} S$, $K \xrightarrow{\mathit{type}_K} S$, and $R \xrightarrow{\mathit{type}_R} S$, where L, K, R, and S are *Graph* systems and l and r are *Graph* homomorphisms such that $\mathit{type}_L \circ l = \mathit{type}_K = \mathit{type}_R \circ r$. The left part of the rule describes the required process state necessary for executing this method and contains at least a message typed in the operation this method implements. The remainder of the rule consists of the gluing part and the right part and specifies how this state is changed by method execution. Generally, the gluing part is the common subgraph of both the left and the right part of the rule.[17]

In order to be able to determine which message is ready to be processed, a special "marker" object called *processor* is used. A message referenced by a processor through a special *current* link is called *active*. Methods are formulated

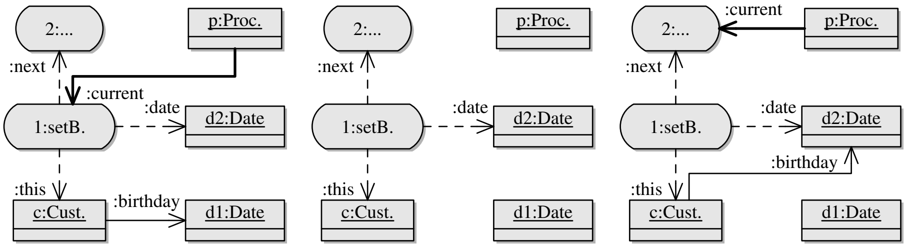
[16] For example, the addition of two integer values or the *if-then-else* statement are both represented by suitable operations.

[17] This means that both morphisms of the DPO rule are injective.

such that their left part requires an active message. Additionally, each method moves the processor object to the next message according to the flow of control. This next message is determined by a special argument called *next*.[18] Multiple processor objects can be used to model multi-threaded processing. Figure 11 displays the DPO rule for a method changing the target of a link.[19]



(a) Schema



(b) Left part (c) Gluing part (d) Right part

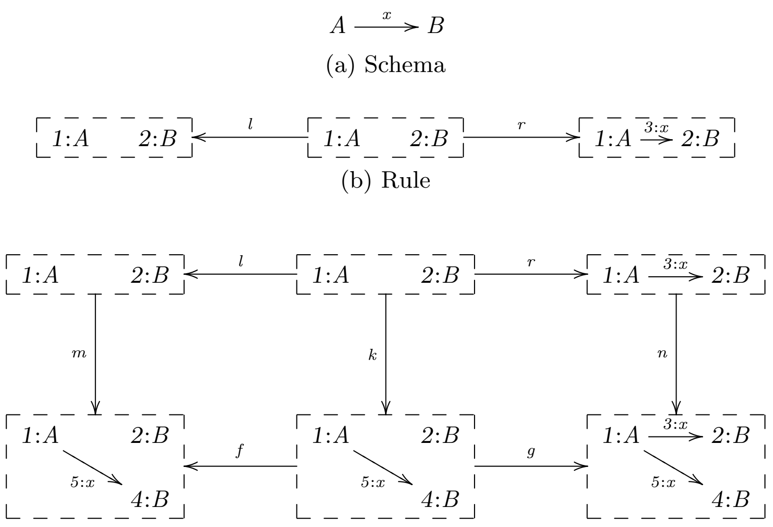
Fig. 11: Example method "Change target of *birthday* link"

A method is executed by applying the underlying DPO rule along a match to the graph structure describing the instance world, i. e., objects, links, messages, and arguments. According to the DPO model, in the first step a pushout complement has to be computed to complete the left side. In the second step, the right side is built by a pushout. However we cannot do this in our category $\mathbf{Sys}(S)$ as neither do pushout complements exist nor are pushouts along monomorphisms van-Kampen squares [6] in all cases. Therefore, we perform DPO transformations which are typed in a schema S in the slice category $\mathbf{Alg}(\mathit{MP*}){\downarrow}S$, where $\mathit{MP*}$ is the signature obtained by removing all axioms from MP, and provide sufficient conditions that guarantee the fulfilment of the axioms after transformation. These conditions are necessary as not all DPO transformations yield typed instances

[18] Only very few messages do not have a *next* argument. This includes the *end* message which terminates process execution and the *if-then-else* message which contains a *then* and a *else* argument instead.

[19] The example shows that operations and messages can also be specialised and possess a particle structure (the particle structure of the messages is not displayed for clarity). This is used to allow processor objects to point to any message.

which fulfil all the axioms. The following figures demonstrate two such counter examples: adding a link violates axiom (2) (Fig. 12), and eliminating inheritance violates axiom (5) (Fig. 13). In the figures, the element-wise mapping of the homomorphisms is indicated by equally named nodes and edges, and frames are used to group the elements belonging to a single graph.



(a) Schema

(b) Rule

(c) Induced transformation

Fig. 12: Adding a link violates axiom (2) on page 328

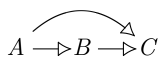
In order to rule out situations as depicted in Fig. 12 we need DPO rules that *pull back edges*. Such rules only add an edge at the right side if no other edge exists which starts at the same node.

Definition 4.1 (DPO rules pulling back edges). *Let $\Sigma = (S, OP, P)$ be an extended signature, and let $L \xleftarrow{l} K \xrightarrow{r} R$ be a DPO rule. Then the DPO rule* pulls back edges *if for each edge $e_R \in R_E$ there is a node $k \in K_N$ and an edge $e_L \in L_E$, such that the equations*

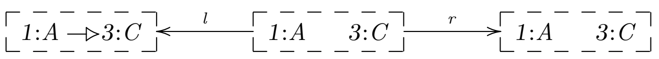
$$source^L(e_L) = l_N(k)$$
$$source^R(e_R) = r_N(k)$$
$$type_{L,E}(e_L) = type_{R,E}(e_R)$$

hold.

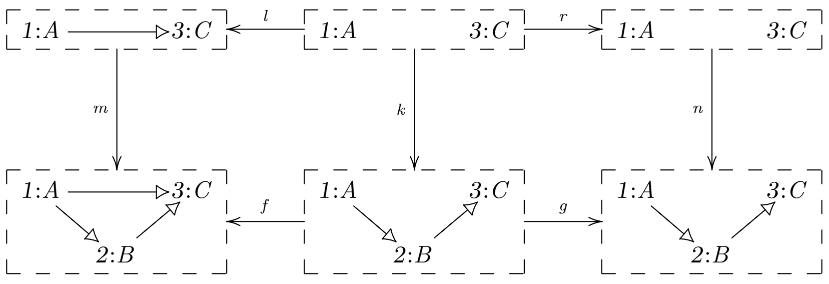
In order to rule out situations as depicted in Fig. 13, we restrict DPO rules to *completing* homomorphisms which "pull back" relations. Completing



(a) Schema



(b) Rule



(c) Induced transformation

Fig. 13: Eliminating inheritance violates axiom (5) on page 329

homomorphisms are an extension to strictly full homomorphisms:[20] While a strictly full homomorphism h only "pulls back" a relation if all related elements are known to be in the range of h, a completing homomorphism h "pulls back" relations even if only a *part* of the related elements is known to be reached.

Definition 4.2 (Completing homomorphism). *Let $\Sigma = (S, OP, P)$ be an extended signature, let $h\colon A \to B$ be a Σ-homomorphism between the Σ-systems A and B, and let $p \in P_w$ be a predicate over a sort word $w \in S^*$. Then h is* completing on *p if for every non-empty sort word w' resulting from eliminating arbitrary sorts from w and for each two tuples $x \in B_w$ and $x' \in A_{w'}$ the implication*

$$h_{w'}(x') = \langle x \rangle_{w'} \wedge x \in p^B \Rightarrow \exists y \in A_w : \langle y \rangle_{w'} = x' \wedge h_w(y) = x \wedge y \in p^A$$

holds, where the notation $\langle x \rangle_{w'}$ stands for the projection of the tuple x onto the elements of the sorts in w'. h is completing *if h is completing on all predicates.*

Now we are able to define *valid* DPO rules:

Definition 4.3 (Valid rule). *A DPO rule $L \xleftarrow{l} K \xrightarrow{r} R$ is* valid *iff it pulls back edges and l and r are completing homomorphisms.*

These restrictions do not have much impact on the expressiveness of methods. The first restriction requiring completing homomorphisms disallows changing the inner structure of objects by adding or removing particles. However, this is an unusual way of dealing with objects at runtime at best. The second restriction

[20] A homomorphism $h\colon A \to B$ is *strictly full* if $h_w(x) \in p^B \Rightarrow x \in p^A$ for all $x \in A_w$ and all predicates $p \in P_w$.

allows to add a link on the right side of a rule only if a similar link has previously been removed on the left side of the same rule. This is unproblematic if it can be ensured that there always exists a link for each (object, association) pair, which, for example, can initially point to a "null" object to indicate an uninitialised link.[21]

Now we can state the main theorem of this section:

Theorem 4.4 (Transformation preserves axioms [7, Theorem 14.29]). *Let S be an MP-system. Let $L \xleftarrow{l} K \xrightarrow{r} R$ be a valid rule in* $\mathbf{Sys}(S)$*, G a* $\mathbf{Sys}(S)$*-object, and $m\colon L \to G$ a match in* $\mathbf{Sys}(S)$*, such that the rule is applicable according to the DPO model. Let $G \xleftarrow{f} D \xrightarrow{g} H$ be the resulting transformation after applying the rule in* $\mathbf{Alg}(MP^*){\downarrow}S$*. Then D and H fulfil all axioms and are, therefore,* $\mathbf{Sys}(S)$*-objects.* □

5 Model Transformation and Data Migration

So far we can describe object-oriented systems, consisting of typed data, programs, and processes. In this section we introduce schema transformations that can be uniquely extended to migrations of corresponding data and proceseses (the migration of programs is handled in the next section).[22]

Definition 5.1 (Transformation, Refactoring). *A transformation $t\colon S \overset{S^*}{\rightsquigarrow} S'$ in the category* $\mathbf{Alg}(MP)$ *is a* span $S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$*. Such a transformation is called a* refactoring *iff l^t is surjective.*

A general transformation allows reduction and unfolding as well as extension and folding through the use of non-surjective homomorphisms (reduction and extension) and non-injective homomorphisms (unfolding and folding) on the left and right side of the span, respectively. Refactorings are special transformations which are constrained to surjective homomorphisms on the left side of the span. This constraint comes from the fact that refactorings are not allowed to delete schema objects because such a deletion almost always causes information (data, programs and/or processes) at the instance level to be lost, which does not meet the intuitive requirement that a refactoring preserve information. In the following we use the term *schema transformation* if the span consists of schema objects, and *migration* if the span consists of typed instances.

Given a typed instance $I \xrightarrow{type_I} S$ and a schema transformation $t\colon S \overset{S^*}{\rightsquigarrow} S'$, the migration is performed as follows:

(1) $\mathcal{P}^{l^t}$, the pullback functor along l^t, is applied to $I \xrightarrow{type_I} S$, resulting in the typed instance $I^* \xrightarrow{type_{I^*}} S^*$. This part of the transformation is responsible

[21] [7] shows in full detail how this can be done.

[22] See [15–17] for precursor material on data migration induced by schema transformations.

for unfolding instance elements if l^t is not injective, and for deleting elements if l^t is not surjective.

(2) $\mathcal{F}^{r^t}$, the composition functor along r^t, is applied to $I^* \xrightarrow{type_{I^*}} S^*$, resulting in the typed instance $I^* \xrightarrow{r^t \circ type_{I^*}} S'$. This part of the transformation is used to retype instance elements and to add new types without any instances.

(3) $I^* \xrightarrow{r^t \circ type_{I^*}} S'$ may violate the typing axioms. Therefore, the epireflector $\mathcal{F}^{S'}$ into the subcategory $\mathbf{Sys}(S')$ is applied to it, resulting in the typed instance $I' \xrightarrow{type_{I'}} S'$ (the schema is left unchanged due to Lemma 3.1). This part of the transformation is responsible for identifying instance elements due to retyping.

These three steps are visualised in Fig. 14.

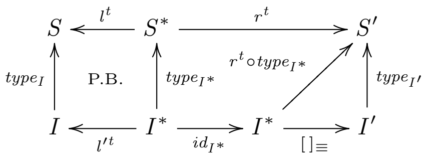

$$\begin{tikzcd}
S & S^* \arrow[l, "l^t"'] \arrow[rr, "r^t"] & & S' \\
I \arrow[u, "type_I"] \arrow[ur, phantom, "\text{P.B.}"] & I^* \arrow[l, "l'^t"] \arrow[u, "type_{I^*}"'] \arrow[r, "id_{I^*}"'] & I^* \arrow[ur, "r^t \circ type_{I^*}"] \arrow[r, "{[\,]_\equiv}"'] & I' \arrow[u, "type_{I'}"']
\end{tikzcd}$$

Fig. 14: Schema transformation and instance migration

The composition of the three functors results in the migration functor defined below:

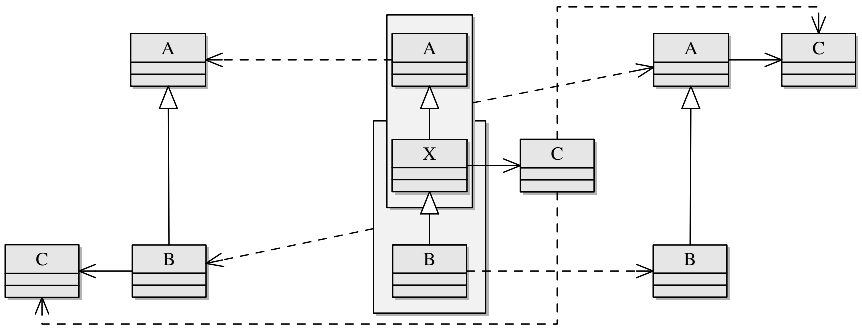
Definition 5.2 (Migration functor). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S'$ *be a transformation. The* migration functor $\mathcal{M}^t\colon \mathbf{Sys}(S) \to \mathbf{Sys}(S')$ *is then defined as:*

$$\mathcal{M}^t ::= \mathcal{F}^{S'} \circ \mathcal{F}^{r^t} \circ \mathcal{P}^{l^t} \;,$$

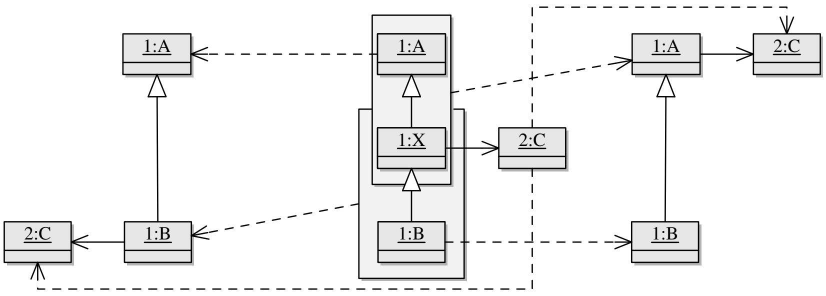
where the functor $\mathcal{P}^{l^t}: \mathbf{Alg}(MP){\downarrow}S \to \mathbf{Alg}(MP){\downarrow}S^*$ *is the pullback functor along* l^t*, the functor* $\mathcal{F}^{r^t}: \mathbf{Alg}(MP){\downarrow}S^* \to \mathbf{Alg}(MP){\downarrow}S'$ *is the composition functor along* r^t*, and the functor* $\mathcal{F}^{S'}: \mathbf{Alg}(MP){\downarrow}S' \to \mathbf{Sys}(S')$ *is the epireflector into the subcategory* $\mathbf{Sys}(S')$*.*

Note that due to Lemma 3.1, the migration functor results in an instance that is correctly typed in the target schema S'.

The example in Fig. 15 shows a transformation which moves the origin of an association one level upwards the inheritance hierarchy and the induced migration of an exemplary instance. On the left side the class B is unfolded, yielding the two classes B and X in the middle, and the origin of the association is moved to the temporary class X. On the right side the class X is folded with the class A, such that the association starts at the class A after the transformation. The modification of objects and links by the induced migration is performed analogously. Note that the unfolding on the left is due to the pullback construction, and the folding on the right side is due to the epireflector which takes care that axiom (1) is satisfied.



(a) Schema transformation



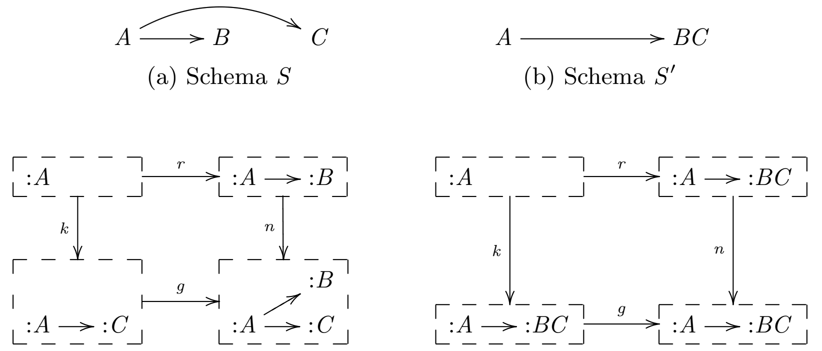
(b) Induced migration

Fig. 15: Moving the origin of an association upwards the inheritance hierachy

6 Method Migration

The migration of methods is performed in the same way as the migration of data and processes. But as methods are valid DPO rules according to Def. 4.3, it has to be ensured that their properties are preserved by a migration. Additionally, methods already executed which are represented by two pushout diagrams shall be transformed so that the resulting diagrams are again pushouts. This ensures that processes that have already been executed are compatible to the new schema after migration. However, this does not hold for arbitrary transformations. In Fig. 16, two classes B and C of a schema S are merged, resulting in the class BC in the schema S'. At the instance level, the right pushout of a method adding a link is presented. The migrated diagram is a pushout in the subcategory $\mathbf{Sys}(S')$ of all typed instances conforming to the typing axioms, but not a pushout in the category $\mathbf{Alg}(MP*)\downarrow S'$ in which the migration is computed. This can be deduced from the elemental properties of pushouts (see e. g. [6]).

In order to migrate DPO rules and DPO diagrams properly we need to restrict the allowed transformations. We can show that if transformations are disallowed to fold associations on the right side, DPO rules can be migrated correctly in all cases. This results in the following definition of a proper transformation:



(a) Schema S (b) Schema S'

(c) Right side of an applied DPO rule before the migration

(d) Right side of an applied DPO rule after the migration

Fig. 16: Pushout in $\mathbf{Alg}(MP^*)\downarrow S$ is not preserved by a migration

Definition 6.1 (Proper transformation). *A transformation* $S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$ *in* $\mathbf{Alg}(MP)$ *is* proper *if* r^t *is injective on associations, i. e., if* $r^t_E(x) = r^t_E(y) \Rightarrow x = y$ *holds for all* $x, y \in S^*_E$.

The correct migration of valid DPO rules is guaranteed by the following proposition:

Proposition 6.2 (Migration preserves valid DPO rules [7, Proposition 15.23]). *Given a proper transformation* $t\colon S \overset{S^*}{\rightsquigarrow} S'$*, let*

$$(I^1 \xrightarrow{type_{I^1}} S) \xleftarrow{l} (I^2 \xrightarrow{type_{I^2}} S) \xrightarrow{r} (I^3 \xrightarrow{type_{I^3}} S)$$

be a valid DPO rule. Then

$$\mathcal{M}^t(I^1 \xrightarrow{type_{I^1}} S) \xleftarrow{\mathcal{M}^t(l)} \mathcal{M}^t(I^2 \xrightarrow{type_{I^2}} S) \xrightarrow{\mathcal{M}^t(r)} \mathcal{M}^t(I^3 \xrightarrow{type_{I^3}} S)$$

is a valid DPO rule as well. □

The migration of DPO diagrams is ensured by the following proposition:

Proposition 6.3 (Migration preserves pushouts [7, Proposition 15.35]). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S' \mathrel{\hat{=}} S \xleftarrow{l^t} S^* \xrightarrow{r^t} S'$ *be a proper transformation and* $(L \xrightarrow{type_L} S) \xleftarrow{l} (K \xrightarrow{type_K} S) \xrightarrow{r} (R \xrightarrow{type_R} S)$ *be a valid DPO rule. Let*

$$(D \xrightarrow{type_D} S) \xrightarrow{g} (H \xrightarrow{type_H} S) \xleftarrow{n} (R \xrightarrow{type_R} S)$$

be a pushout of

$$(D \xrightarrow{type_D} S) \xleftarrow{k} (K \xrightarrow{type_K} S) \xrightarrow{r} (R \xrightarrow{type_R} S)$$

in $\mathbf{Alg}(MP_*)\downarrow S$*, where all typed instances are in* $\mathbf{Sys}(S)$*. Then*

$$\mathcal{M}^t(D \xrightarrow{type_D} S) \xrightarrow{\mathcal{M}^t(g)} \mathcal{M}^t(H \xrightarrow{type_H} S) \xleftarrow{\mathcal{M}^t(n)} \mathcal{M}^t(R \xrightarrow{type_R} S)$$

is a pushout of

$$\mathcal{M}^t(D \xrightarrow{type_D} S) \xleftarrow{\mathcal{M}^t(k)} \mathcal{M}^t(K \xrightarrow{type_K} S) \xrightarrow{\mathcal{M}^t(r)} \mathcal{M}^t(R \xrightarrow{type_R} S)$$

in $\mathbf{Alg}(MP_*)\downarrow S'$*, where all typed instances are in* $\mathbf{Sys}(S')$*.* □

Both propositions can be combined, yielding the following theorem:

Theorem 6.4 (Correctness of the migration of programs [7, Theorem 15.36]). *Let* $t\colon S \overset{S^*}{\rightsquigarrow} S'$ *be a proper transformation. Then the migration functor* $\mathcal{M}^t$ *transfers the validity of non-applied methods (DPO rules) and applied methods (DPO transformations) from the category* $\mathbf{Alg}(MP)\downarrow S$ *into the category* $\mathbf{Alg}(MP)\downarrow S'$*.*

Proof. Direct consequence of Proposition 6.2 and Proposition 6.3. □

7 Outlook

With the framework presented above, a major step towards migration of complete object-oriented systems is proposed. Certainly, the framework is not universal as it is subject to some (reasonable) constraints. Migrations are considered to be instances of transformations. The innovative part of the theory described consists of the *automatic* transformation of a migration source, computing the target with the help of a functor on slice categories. This functor is composed of three factors: Generally, the pullback functor $\mathcal{P}^{l^t}$ is right-adjoint where the second factor—the composition functor $\mathcal{F}^{r^t}$—is its left-adjoint. But the third factor—the construction $\mathcal{F}^S$ into the subcategory $\mathbf{Sys}(S)$—yields an adjunction as well. Thus, the whole migration enjoys well-understood universal properties which can further be pursued into three different directions.

The first direction for future research will be the development of tools that support migration induced by refactoring rules. If transformation rules can be captured ergonomically in an appropriate application, migrations can automatically and *uniquely* (by adjointness) be computed. Thus, content migration of databases is possible as well as migration of running processes in a software system. These tools should discover potential for composition, as well: Bigger refactorings should be decomposable into elementary changes, atomic steps must be proved to combine to more comprehensive procedures. This is another facet for future research.

Theorem 6.4 states that dynamical semantics is preserved by refactorings where semantics is based on valid DPO rules. Hence the second direction is to find a comparable correctness criterion for data. This must include a formal specification of "information" to distinguish between semantics-preserving refactorings and information-distorting transformations.

The third direction consists of abstracting away from pure graph structures. It has to be investigated to what extent the results can be generalised to elementary topoi or even to adhesive categories [6, 18]. An approach can be found in [15] which covers data migration only. Hence, an extension to method migration is desirable.

References

[1] Fowler, M.: Refactoring: Improving the Design of Existing Code. Addison-Wesley (1999)

[2] Fowler, M.: Patterns of Enterprise Application Architecture. Addison-Wesley (2002)

[3] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns. Addison-Wesley Professional (1995)

[4] Fowler, M., Scott, K.: UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley (2003)

[5] Mens, T., Tourwé, T.: A survey of software refactoring. IEEE Transactions On Software Engineering **30**(2) (2004) 126–139

[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamentals of Algebraic Graph Transformation. Springer (2006)

[7] Schulz, C.: Refactoring objektorientierter Systeme. Forschungsberichte der FHDW Hannover **2** (2009)

[8] Corradini, A., Dotti, F.L., Foss, L., Ribeiro, L.: Translating java code to graph transformation systems. In: ICGT. (2004) 383–398

[9] Kastenberg, H., Kleppe, A.G., Rensink, A.: Defining object-oriented execution semantics using graph transformations. In Gorrieri, R., Wehrheim, H., eds.: Proceedings of the 8th IFIP International Conference on Formal Methods for Open-Object Based Distributed Systems, Bologna, Italy. Volume 4037 of Lecture Notes in Computer Science., London, Springer Verlag (June 2006) 186–201

[10] Kastenberg, H., Kleppe, A.G., Rensink, A.: Engineering object-oriented semantics using graph transformations. Technical Report CTIT Technical Report 06-12, University of Twente (2006)

[11] Bardohl, R., Ehrig, H., de Lara, J., Runge, O., Taentzer, G., Weinhold, I.: Node type inheritance concept for typed graph transformation. Technical Report Technical Report 2003-19, Technical University, Berlin (2003)

[12] de Lara, J., Bardohl, R., Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Attributed graph transformation with node type inheritance. Theoretical Computer Science **376**(3) (2007) 139–163

[13] Mal'cev, A.I.: Algebraic systems. Springer (1973)

[14] International Organization for Standardization: ISO/IEC 14882:2003: Programming languages – C++, Genf, Schweiz. (2003)

[15] König, H., Löwe, M., Schulz, C.: Functor semantics for refactoring-induced data migration. Forschungsberichte der FHDW Hannover **1** (2007)

[16] Löwe, M., König, H., Schulz, C., Peters, M.: Refactoring information systems – a formal framework. In: Proceedings WMSCI 2006. Volume 1. (2006) 75–80

[17] Löwe, M., König, H., Schulz, C., Peters, M.: Refactoring information systems – handling partial composition. In: Electronic Communications of the EASST. Volume 3. (2006)
[18] Goldblatt, R.: Topoi: The Categorical Analysis of Logic. Dover Publications (1984)

Christoph Schulz
Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
christoph.schulz@fhdw.de

Prof. Dr. Michael Löwe
Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
michael.loewe@fhdw.de

Hans-Jörg Kreowski supervised Michael Löwe's diploma thesis at TU Berlin in 1981, and was the external examiner of his doctoral thesis in 1990.

Prof. Dr. Harald König
Fachhochschule für die Wirtschaft Hannover
Freundallee 15
D-30173 Hannover (Germany)
harald.koenig@fhdw.de

Das Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF) e.V. – Impressionen aus 25 Jahren –

Ralf E. Streibl

„*Was mich als Informatiker besonders betroffen macht, ist die Tatsache, dass fast alle Pläne für eine verstärkte Überwachung aller Bürgerinnen und Bürger und für die Einschränkung der Grundrechte auf Informations- und Kommunikationstechnik aufbauen und ohne diese so gar nicht denkbar wären. Technik insgesamt und gerade auch die I&K-Technik schaffen wichtige Voraussetzungen für eine gedeihliche gesellschaftliche Entwicklung. Das wird aber ins Gegenteil verkehrt, wenn die Technik gegen die Menschen gerichtet und damit Misstrauen gegen Technik geweckt wird*“ (Kreowski 2007, Seite 11).

Am 2. Juni 1984 wurde das *Forum Informatiker für Frieden und gesellschaftliche Verantwortung* in Bonn als Verein gegründet. (Seit 1988 lautet der Name *Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung*. Abbildung 1 zeigt das FIfF-Logo.) Anlass war die Sorge über die zunehmende Verflechtung von Informationstechnik und Rüstung und die damit einhergehende Militarisierung des Fachgebietes. Vorangegangen war bereits im Sommer 1983 eine Initiative *Informatiker warnen vor dem programmierten Atomkrieg*. Ferner existierten dezentral fachbezogene Friedensinitiativen, beispielsweise – ebenfalls seit 1983 – die Friedensinitiative am Fachbereich Mathematik und Informatik der Universität Bremen. Als Vorbild für die Gründung des FIfF diente die amerikanische Vereinigung *Computer Professionals for Social Responibility* (CPSR). Joseph Weizenbaum und Alan Borning waren als Vertreter von CPSR bei der Gründungsversammlung des FIfF mit dabei.

„*Der Name des Vereins verweist einerseits auf Gründungsanlass und aktuellen Arbeitsschwerpunkt und macht andererseits deutlich, dass seine Mitglieder kritische Analysen nicht auf militärische Anwendungen der Informatik beschränken wollen. Dass der Informatiker bereit ist, seiner gesellschaftlichen Verantwortung gerecht zu werden, muss sich gerade auch in Bereichen wie Automatisierung und Datenschutz zeigen*“ (Löhr 1984, Seite 18).

FIfF

Abb. 1. Das FIfF-Logo in seiner aktuellen Form

F. Drewes, A. Habel, B. Hoffmann, D. Plump (Eds.): Manipulation of Graphs, Algebras and Pictures. Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday, pp. 341–352, 2009.

Die Bundesrepublik Deutschland war 1983/84 stark gekennzeichnet von den Massenprotesten gegen den *NATO-Doppelbeschluss*, der die Stationierung atomar bestückter Mittelstreckenraketen und Cruise Missiles in Deutschland beinhaltete. Dagegen protestierten – mit Höhepunkten im *heißen Herbst* der Friedensbewegung 1983 – Millionen von Bundesbürgern mit Unterschriftenlisten, Großdemonstrationen, Blockadeaktionen, Menschenketten und vielen anderen Aktionen. Gleichzeitig verkündete der damalige US-Präsident Ronald Reagan seine Pläne einer *Strategic Defense Initiative* (SDI) für eine weltraumgestützte Raketenabwehr.

In der Folge dieser Entwicklungen entstanden eine Reihe berufsständischer Friedensinitiativen, darunter:

- 1982: IPPNW Deutschland – Internationale Ärzte für die Verhütung des Atomkrieges, Ärzte in sozialer Verantwortung (der internationale Verband IPPNW wurde bereits 1980 gegründet)
- 1982: Pädagoginnen und Pädagogen für den Frieden (PPF) (bereits 1981 gab es als Vorläufer *Pädagogen gegen den Rüstungswahnsinn*)
- 1982: Forum Friedenspsychologie (zunächst unter dem Namen: Friedensinitiative Psychologie – Psychosoziale Berufe)
- 1983: NaturwissenschaftlerInnen-Initiative *Verantwortung für Frieden und Zukunftsfähigkeit* (beim Kongress Anfang Juli 1983 in Mainz protestierten mehr als 3000 Naturwissenschaftlerinnen und Naturwissenschaftler aus dem In- und Ausland gegen Atomrüstung)
- 1984: Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF)
- 1984: KulturwissenschaftlerInnen-Initiative für Frieden und Abrüstung
- 1984: Arbeitskreis *Historische Friedensforschung*
- 1989: Juristinnen und Juristen gegen atomare, biologische und chemische Waffen (Deutsche Sektion der IALANA: International Association of Lawyers Against Nuclear Arms)

1983 erschien auf Initiative des *Bund demokratischer Wissenschaftler und Wissenschaftlerinnen* das erste Heft des INFORMATIONSDIENST WISSENSCHAFT UND FRIEDEN, dessen Herausgeberschaft ab 1985 von RepräsentantInnen verschiedener berufsständischer Friedensorganisationen übernommen wurde und die seit 1992 als interdisziplinäre Fachzeitschrift WISSENSCHAFT UND FRIEDEN viermal im Jahr erscheint. (Siehe Abbildung 2.) Zu den Mitherausgebern gehören viele berufsständische Friedensorganisationen – darunter natürlich auch das FIfF.

Ein weiterer Aspekt, der – insbesondere für die kritische Informatik – mit dieser Zeit verbunden ist, ist der Widerstand gegen die für das Jahr 1983 geplante Volkszählung, die aufgrund der Beschwerden beim Bundesverfassungsgericht zunächst ausgesetzt und dann mit dem sogenannten *Volkszählungsurteil* vom 15. Dezember 1983 ganz untersagt wurde. In seinem Urteil formulierte das Verfassungsgericht ein Grundrecht auf informationelle Selbstbestimmung, basierend auf Artikel 1 des Grundgesetzes (Schutz der Menschenwürde), welches seither zu einer wesentlichen Grundlage aller Datenschutz- und Überwachungsdiskussionen wurde (vgl. Büllesbach & Garstka 2005; Steinmüller 2007). Es mag als eine Ironie

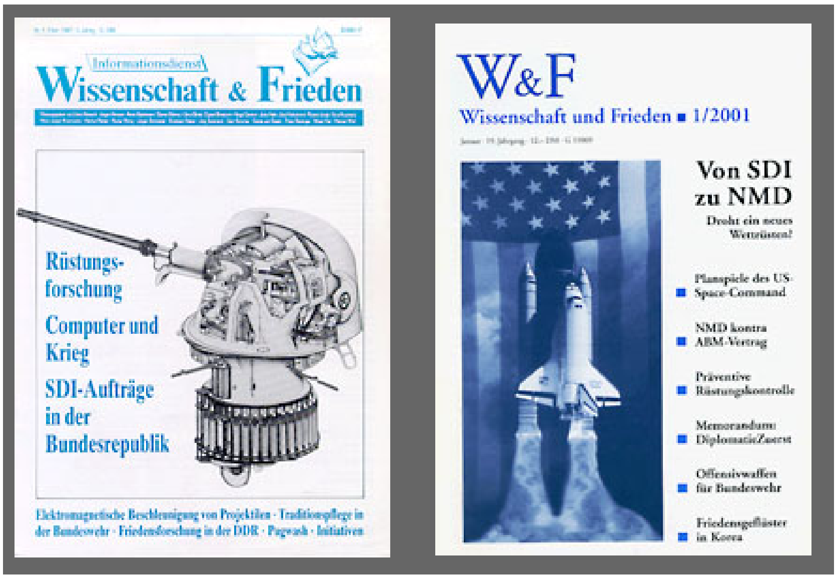

Informationsdienst
Wissenschaft & Frieden
Rüstungs-forschung
Computer und Krieg
SDI-Aufträge in der Bundesrepublik
Elektromagnetische Beschleunigung von Projektilen · Traditionspflege in der Bundeswehr · Friedensforschung in der DDR · Pugwash · Initiativen
W&F
Wissenschaft und Frieden ▪ 1/2001
Von SDI zu NMD
Droht ein neues Wettrüsten?
Planspiele des US-Space-Command
NMD kontra ABM-Vertrag
Präventive Rüstungskontrolle
Memorandum DiplomatieZuerst
Offensivwaffen für Bundeswehr
Friedensgeflüster in Korea

Abb. 2. *Wissenschaft und Frieden* – vom Informationsdienst zur interdisziplinären friedenswissenschaftlichen Fachzeitschrift

der Literaturgeschichte angesehen werden, dass George Orwell für seine bekannte Dystopie just dieses Jahr als Titel wählte, indem er die Ziffern des Entstehungsjahres 1948 vertauschte.

Dieses also in mehrfacher Hinsicht gesellschaftlich brisante Klima im Vorfeld der Gründung des FIfF wird auch bei einer Betrachtung von Titelbildern des Magazins Der Spiegel aus dem Jahr 1983 offenkundig: Vier Titelbilder bezogen sich auf die Volkszählung sowie Überwachungthemen, fünf auf die Raketenstationierung in Westdeutschland und den Widerstand der Friedensbewegung (vgl. Abbildung 3).

Im ersten *FIfF-Rundbrief*, erschienen im August 1984 (Abbildung 4), schrieb Hans-Jörg Kreowski unter der Überschrift *Aufbruch zu einer anderen Informatik* über die Vereinsgründung:

> *„Zweck des Forums ist dabei, die Verwendung von Computer- und Informationstechnik in der Rüstung und bei der Kriegsvorbereitung und -führung in Gegenwert und Zukunft zu untersuchen, zu kritisieren, öffentlich zu machen und Alternativen aufzuzeigen. Außerdem sollen regionale Gruppen, Initiativen und Einzelpersonen mit ähnlichen Zielen unterstützt werden. Perspektivisch jedoch wird die gesellschaftliche Verantwortung der Informatiker umfassender in das Wirken des Forums einbezogen werden. Vor allem soll dem Computer als Instrument der Arbeitsplatzvernichtung und der staatlichen Kontrolle gebührend Aufmerksamkeit gewidmet werden“* (Kreowski 1984).

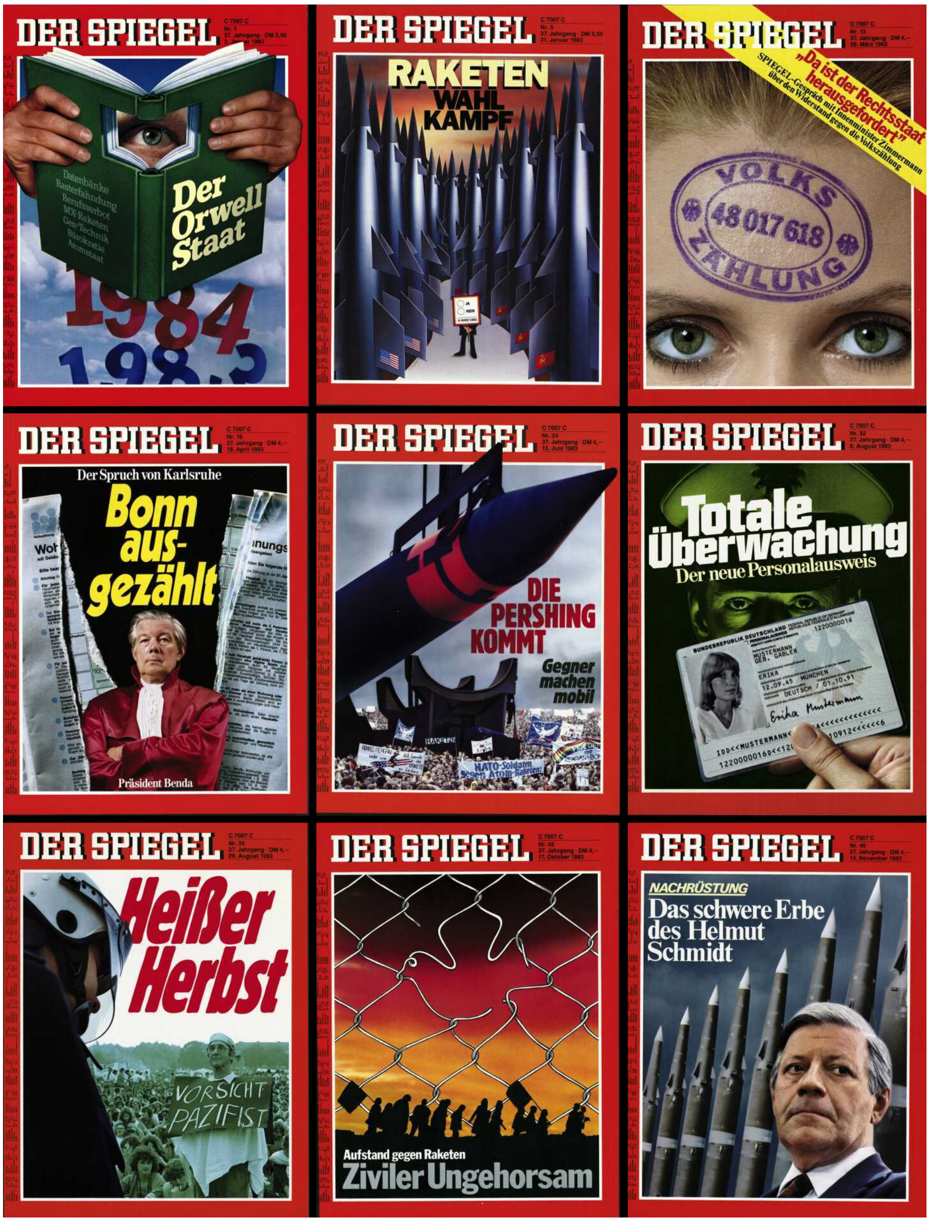
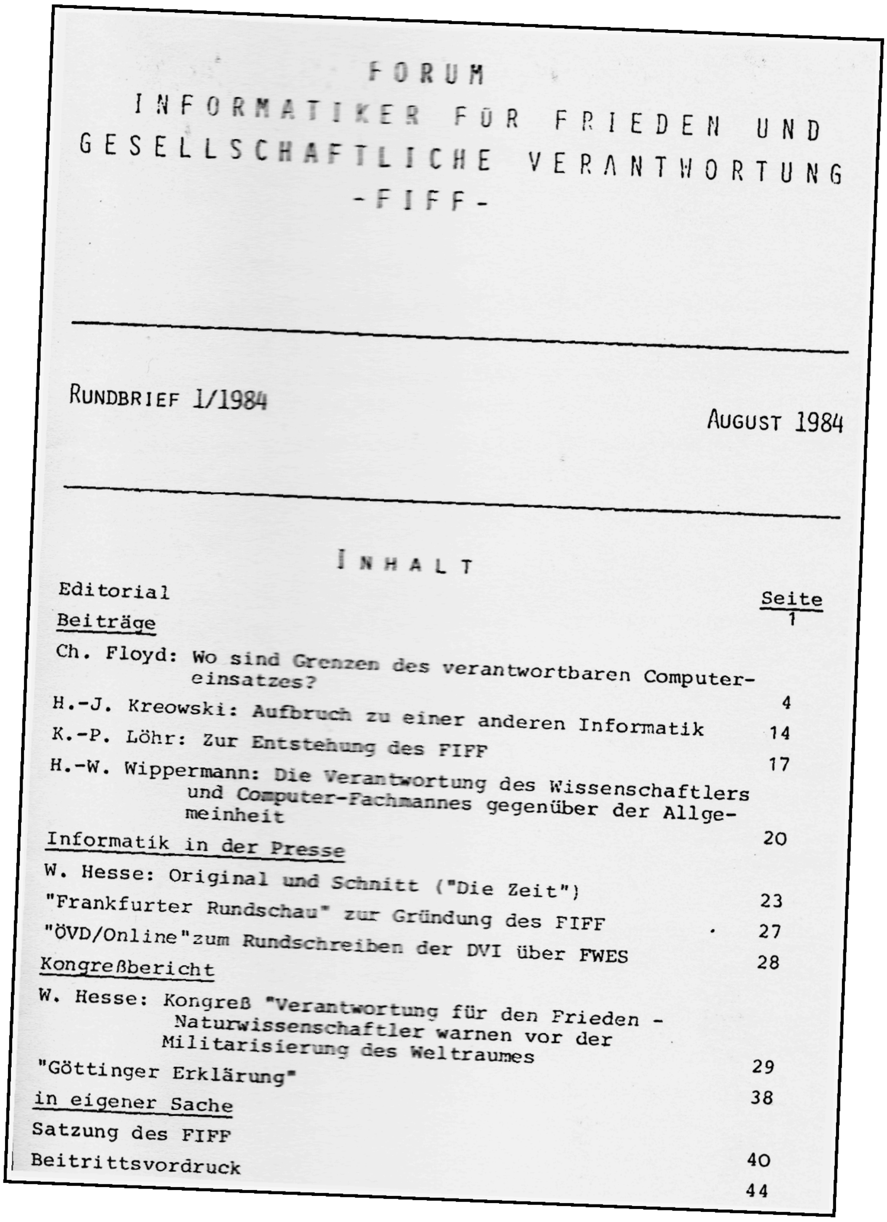


Abb. 3. Die Themenkomplexe Überwachung und Frieden im Spiegel von Titelbildern des Jahres 1983: *Der Orwell-Staat* (Nr. 1), *Raketenwahlkampf* (Nr. 5), *Volkszählung* (Nr. 13), *Bonn ausgezählt* (Nr. 16), *Die Pershing kommt* (Nr. 24), *Totale Überwachung. Der neue Personalausweis* (Nr. 32), *Heißer Herbst* (Nr. 35), *Aufstand gegen Raketen: Ziviler Ungehorsam* (Nr. 42), *Nachrüstung – Das schwere Erbe des Helmut Schmidt* (Nr. 46)



FORUM
INFORMATIKER FÜR FRIEDEN UND
GESELLSCHAFTLICHE VERANTWORTUNG
-FIFF-

RUNDBRIEF 1/1984

AUGUST 1984

INHALT

Abb. 4. Der erste Rundbrief des FIfF (August 1984)

Die im Namen des FIfF vorkommenden Begriffe Frieden und Verantwortung können als zwei eng verbundene Aspekte professionellen Handelns betrachtet werden. Ute Bernhardt, Helga Genrich und Ingo Ruhmann machten dies in ihrem Beitrag *Der Prozess Verantwortung* deutlich, erschienen in dem von Hans-Jörg Kreowski herausgegebenen Informatik-Fachbericht Nr. 309. Dieser Band enthält Beiträge zur Erinnerung an Reinhold Franck, Bremer Hochschullehrer und FIfF-Vorsitzender von 1987 bis 1990.

> *„Die Frage nach der Verantwortung von InformatikerInnen ist also keine Frage nach dem Ob, sondern nach dem Wie. (...) Dieser Prozess Verantwortung umfasst die eigenverantwortliche – Erkenntnis und Interesse vereinende – Selbstreflexion über Ziele und Interessen eigenen Tuns, die Beteiligung an der oder Offenheit für die Entwicklung einer angemessenen Theorie der Informatik und das Sich-Verantworten vor der Gesellschaft in einem offenen, von Tabus befreiten Diskurs. Das FIfF, dessen Vorsitzender Reinhold Franck bis zu seinem Tode war, wurde gegründet als ein offenes Forum für InformatikerInnen, um eine Möglichkeit für einen solchen Diskurs zu bieten. Frieden und gesellschaftliche Verantwortung im Namen des FIfF stehen nicht nebeneinander, sondern sind in einem Zusammenhang zu sehen. Zusammen sind sie Handlungsperspektive für tägliche Praxis im Forschungs- und Entwicklungsbereich. (...) Eine friedliche Informatik kann sich erst dann entwickeln, wenn von InformatikerInnen ganz bewusst fachliche und gesellschaftliche Verantwortung übernommen wird.“* (Bernhardt, Genrich, Ruhmann 1992).

Das FIfF setzt sich für eine Technikgestaltung ein, welche die Menschenwürde achtet und dazu beiträgt, die Demokratie und die Grundrechte weiterzuentwickeln. Es arbeitet für eine Informationstechnik, die Menschen in den Mittelpunkt stellt. Das FIfF versteht sich als Forum für eine kritische und lebendige Auseinandersetzung – offen für alle, die daran mitwirken wollen. Über seine Publikationen, Vorträge und Tagungen, Stellungnahmen und Presseerklärungen informiert das FIfF und regt an, Positionen zu beziehen. Zur Professionalisierung der FIfF-Arbeit wurde Anfang 1987 in Bonn das FIfF-Büro gegründet und blieb dort bis Ende 1998 ansässig. Danach übersiedelte die Geschäftsstelle für zwei Jahre nach Medemstade-Ihlienworth. Anfang 2001 zog das FIfF-Büro dann nach Bremen, wo es seither in der Villa Ichon angesiedelt ist, einem Friedens- und Kulturhaus am Goetheplatz 4 (Abbildung 5).

Ein wesentliches Medium des FIfF ist die Quartalszeitschrift FIfF-Kommunikation (Abbildung 6). Zwischen 1984 und 1987 verschickte das FIfF zunächst einen Rundbrief im Din-A5-Format, zusammengestellt von einem Team aus Kaiserslautern. Dieser wurde ab 1988 von der FIfF-Kommunikation abgelöst, damals betreut von einer Redaktion in München. Ab 1995 übernahm dann für einige Zeit ein schwerpunktmäßig in Paderborn angesiedeltes Team diese Arbeit. Inzwischen hat sich seit mehreren Jahren eine verteilte Redaktionsarbeit etabliert: Zu jedem Heft gibt es eine wechselnde Schwerpunktredaktion, die sich um die Zusammenstellung von Beiträgen zum zentralen Heftthema kümmert.

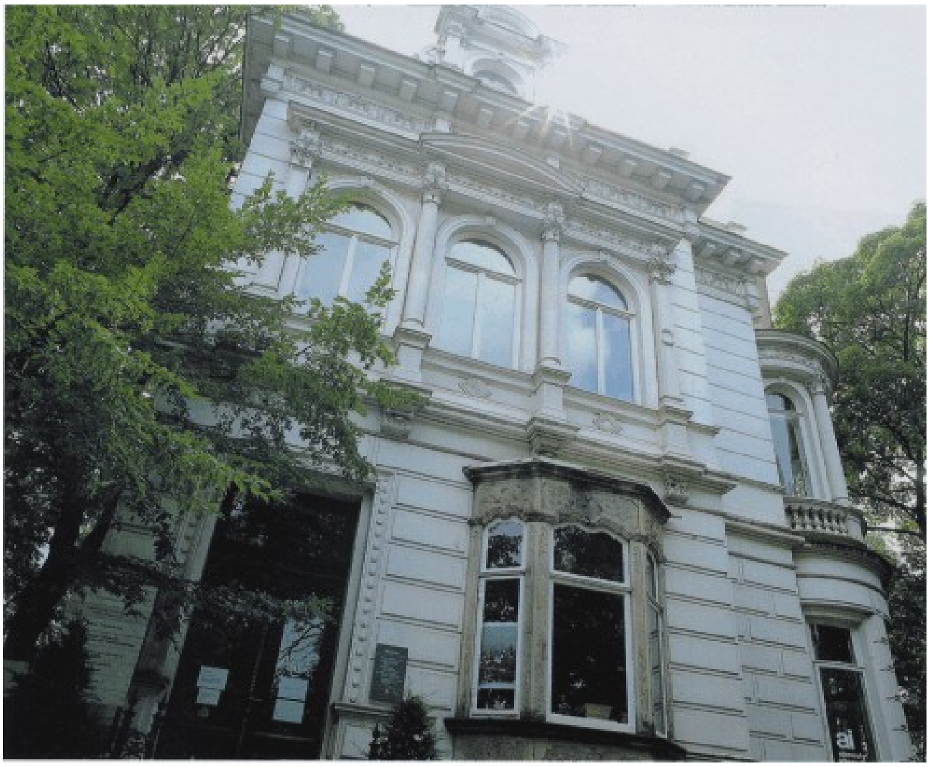


Abb. 5. Die Villa Ichon in Bremen: Das FIfF-Büro befindet sich rechts im ersten Stock.

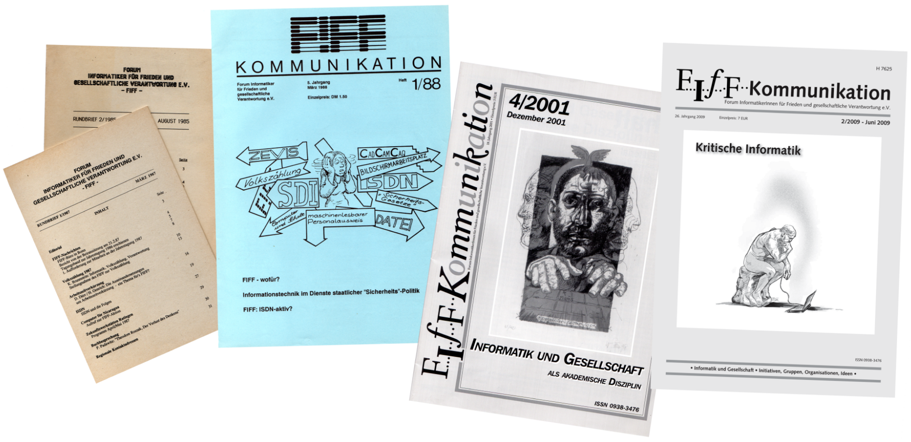


Abb. 6. Die FIfF-Kommunikation im Wandel

Diese kooperiert mit der Hauptredaktion, die den allgemeinen Teil organisiert, die Schwerpunktredaktionen unterstützt und die Gesamtplanung und -abläufe koordiniert. Beide Redaktionen sind heute nicht mehr an einen Ort gebunden.

Neben der FIfF-Kommunikation veröffentlicht das FIfF immer wieder Bücher und Broschüren, teilweise in Kooperation mit anderen Verlagen. Jüngstes Beispiel ist der von Hans-Jörg Kreowski (2008) herausgegebene Band *Informatik und Gesellschaft. Verflechtungen und Perspektiven.*

Die Auswirkungen von Informatik bzw. Informations- und Kommunikationstechnik sind heute mehr denn je in vielfältigen Gesellschaftsbereichen wirksam und sind natürlich auch nicht auf einen nationalen Raum beschränkt. Das FIfF kooperiert daher bei seiner Arbeit mit vielen Organisationen, Initiativen und Vereinigungen, die sich in Deutschland, Europa und darüber hinaus mit derartigen Fragen auseinandersetzen. Diese im einzelnen hier vorzustellen würde den Rahmen dieses Beitrages sprengen. Es sei hier auf das Heft 2/2009 der FIfF-Kommunikation (*Kritische Informatik*) verwiesen, welches diesbezüglich einen guten Überblick bietet.

Neben vielen anderen Aktivitäten kommt den Jahrestagungen des FIfF eine große Bedeutung zu – sie bilden ein Forum, wo sich Mitglieder und Interessierte in Vorträgen und Arbeitsgruppen zu aktuellen Themen informieren und miteinander diskutieren. Vom 13. bis 15. November 2009 wird die 25. FIfF-Jahrestagung hoffentlich wieder eine große Zahl von Interessierten anziehen. Es ist dann das vierte Mal (vgl. Abbildung 7), dass solch eine Tagung in Bremen stattfindet – jeweils vorbereitet von Organisations- und Programmkomitees, an denen Hans-Jörg Kreowski wesentlichen Anteil hatte (er hat darüber hinaus aber auch bei der Vorbereitung mehrerer Jahrestagungen an anderen Orten mitgewirkt).

1994 bot die erste Jahrestagung in Bremen Gelegenheit, Rückschau auf 10 Jahre FIfF zu halten. *„Wir wollten Partei ergreifen für die Menschen, für Gerechtigkeit, für Entwicklung – Partei gegen blinden, technikfixierten Fortschrittsglauben"*, erklärte damals die frühere FIfF-Vorsitzende Helga Genrich. Das FIfF hat sich von Beginn an gleichermaßen an die Öffentlichkeit und die Fachkollegen als Adressaten gewandt. In der Informatik – so wurde auf der Tagung konstatiert – sei inzwischen die Beschäftigung mit den gesellschaftlichen Auswirkungen Teil der Disziplin geworden. So gab es Mitte der 1990er Jahre einige Universitäten an denen „Informatik und Gesellschaft" in das Studium integriert und teilweise auch durch Hochschullehrer/innen vertreten war. Die Arbeit des FIfF habe – so das Resümee der Tagung – hier durchaus Früchte getragen. Gleichzeitig wurde der Blick in die Zukunft gerichtet: Mit neuen Konzepten und Entwicklungen im Bereich der Informationstechnik müssten auch aufs neue kritische Fragen gestellt und Tabus aufgebrochen werden. *„FIfF soll politisch weiterhin die Wächterrolle spielen, die es immer gespielt hat"*, ermunterte Wolfgang Coy als Sprecher des Fachbereichs 8 der Gesellschaft für Informatik die Zuhörer in seinem Grußwort. Viele Beiträge dieser Tagung wurden anschließend in einem Sammelband veröffentlicht (Kreowski et al. 1995).

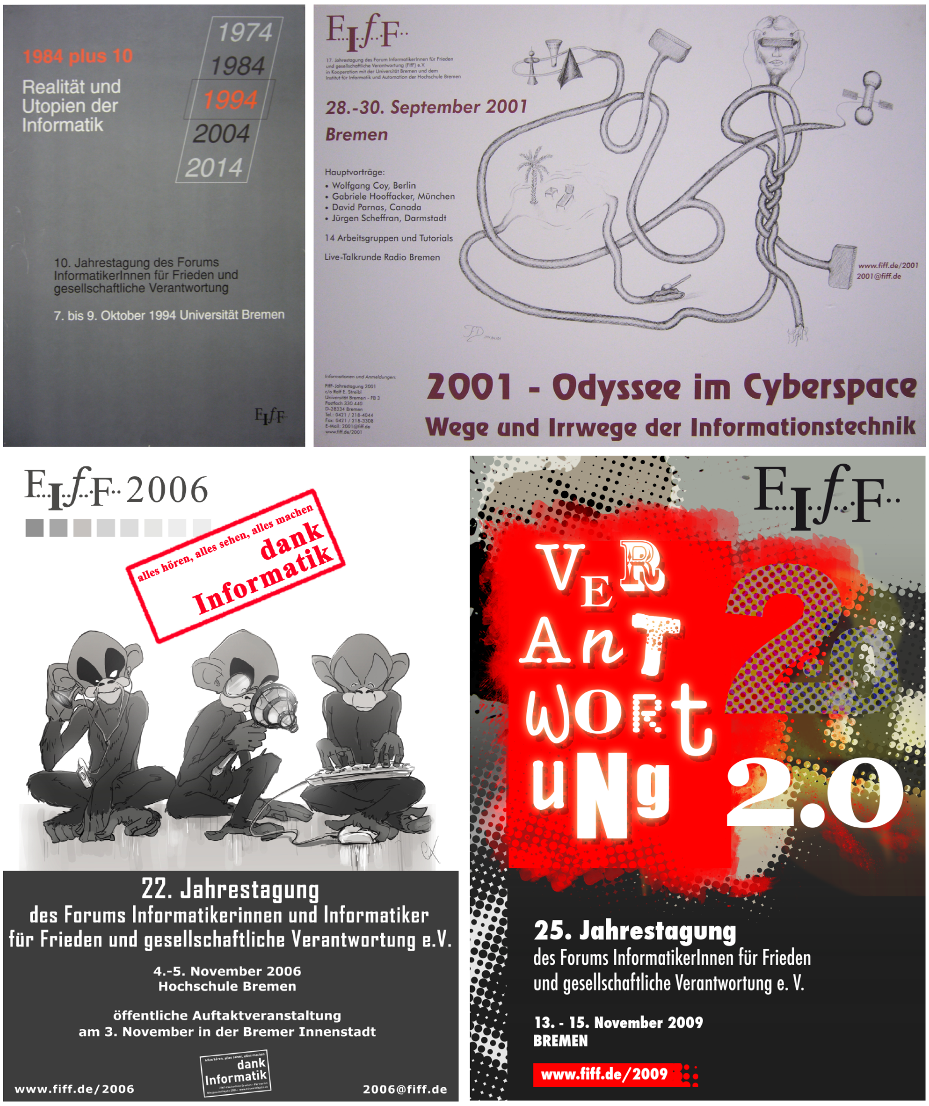
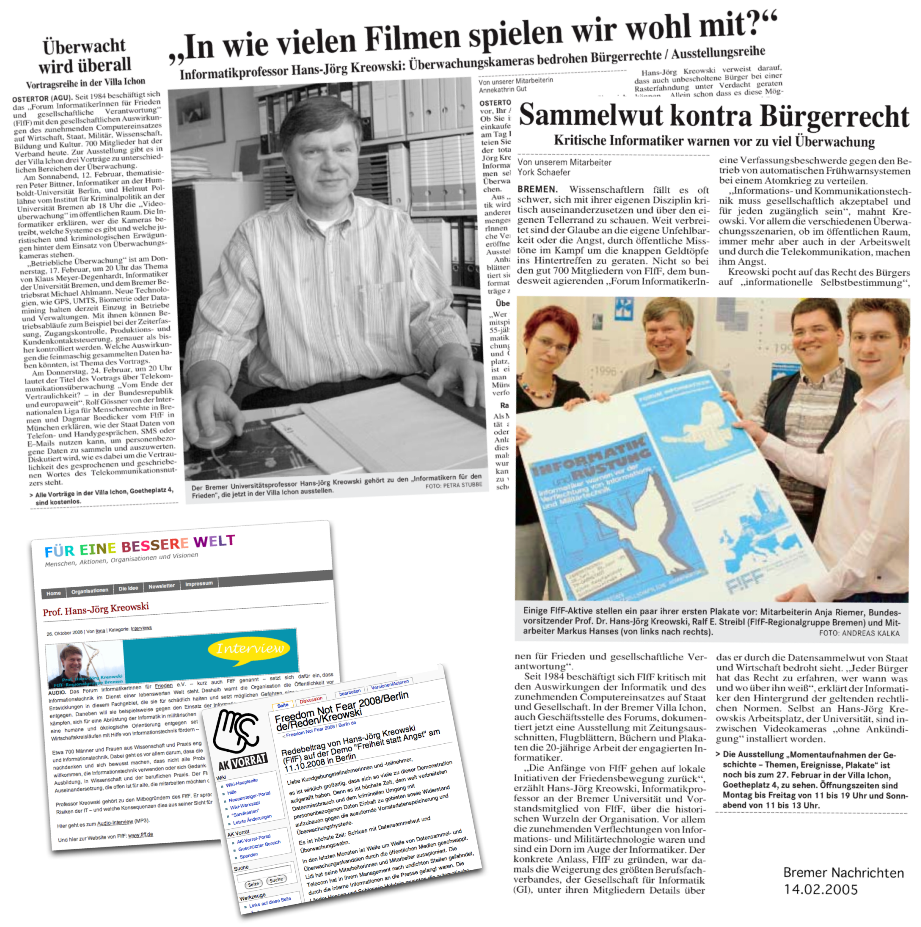

1984 plus 10
Realität und Utopien der Informatik
1974
1984
1994
2004
2014
10. Jahrestagung des Forums InformatikerInnen für Frieden und gesellschaftliche Verantwortung
7. bis 9. Oktober 1994 Universität Bremen
FIfF
17. Jahrestagung des Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF) e.V. in Kooperation mit der Universität Bremen und dem Institut für Informatik und Automation der Hochschule Bremen
28.-30. September 2001
Bremen
Hauptvorträge:
• Wolfgang Coy, Berlin
• Gabriele Hooffacker, München
• David Parnas, Canada
• Jürgen Scheffran, Darmstadt
14 Arbeitsgruppen und Tutorials
Live-Talkrunde Radio Bremen
www.fiff.de/2001
2001@fiff.de
2001 - Odyssee im Cyberspace
Wege und Irrwege der Informationstechnik
FIfF 2006
alles hören, alles sehen, alles machen
dank Informatik
22. Jahrestagung
des Forums Informatikerinnen und Informatiker für Frieden und gesellschaftliche Verantwortung e.V.
4.-5. November 2006
Hochschule Bremen
öffentliche Auftaktveranstaltung am 3. November in der Bremer Innenstadt
www.fiff.de/2006
2006@fiff.de
FIfF
VERANTWORTUNG 2.0
25. Jahrestagung
des Forums InformatikerInnen für Frieden und gesellschaftliche Verantwortung e. V.
13. - 15. November 2009
BREMEN
www.fiff.de/2009

Abb. 7. Die FIfF-Jahrestagungen in Bremen: *1984 plus 10: Realität und Utopien der Informatik* (10. FIfF-Jahrestagung 1994, Universität Bremen, Plakat: Ralf E. Streibl); *2001 – Odyssee im Cyberspace* (17. FIfF-Jahrestagung 2001, Universität Bremen, Plakat: Frank Drewes und Ralf E. Streibl); *alles hören, alles sehen, alles machen ... dank Informatik* (22. FIfF-Jahrestagung 2006, Hochschule Bremen, Plakat: Caro von Totth); *Verantwortung 2.0* (25. FIfF-Jahrestagung 2009, Universität Bremen, Plakat: Caro von Totth)



Überwacht wird überall

Vortragsreihe in der Villa Ichon

OSTERTOR (AGU). Seit 1984 beschäftigt sich das „Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung" (FIfF) mit den gesellschaftlichen Auswirkungen des zunehmenden Computereinsatzes auf Wirtschaft, Staat, Militär, Wissenschaft, Bildung und Kultur. 700 Mitglieder hat der Verband heute. Zur Ausstellung gibt es in der Villa Ichon drei Vorträge zu unterschiedlichen Bereichen der Überwachung.

Am Sonnabend, 12. Februar, thematisieren Peter Bittner, Informatiker an der Humboldt-Universität Berlin, und Helmut Pollähne vom Institut für Kriminalpolitik an der Universität Bremen ab 18 Uhr die „Videoüberwachung" im öffentlichen Raum. Die Informatiker erklären, wer die Kameras betreibt, welche Systeme es gibt und welche juristischen und kriminologischen Erwägungen hinter dem Einsatz von Überwachungskameras stehen.

„Betriebliche Überwachung" ist am Donnerstag, 17. Februar, um 20 Uhr das Thema von Klaus Meyer-Degenhardt, Informatiker der Universität Bremen, und dem Bremer Betriebsrat Michael Ahlmann. Neue Technologien, wie GPS, UMTS, Biometrie oder Datamining halten derzeit Einzug in Betriebe und Verwaltungen. Mit ihnen können Betriebsabläufe zum Beispiel bei der Zeiterfassung, Zugangskontrolle, Produktions- und Kundenkontaktsteuerung, genauer als bisher kontrolliert werden. Welche Auswirkungen die feinmaschig gesammelten Daten haben könnten, ist Thema des Vortrags.

Am Donnerstag, 24. Februar, um 20 Uhr lautet der Titel des Vortrags über Telekommunikationsüberwachung „Vom Ende der Vertraulichkeit? – in der Bundesrepublik und europaweit". Rolf Gössner von der Internationalen Liga für Menschenrechte in Bremen und Dagmar Boedicker vom FIfF in München erklären, wie der Staat Daten von Telefon- und Handygesprächen, SMS oder E-Mails nutzen kann, um personenbezogene Daten zu sammeln und auszuwerten. Diskutiert wird, wie es dabei um die Vertraulichkeit des gesprochenen und geschriebenen Wortes des Telekommunikationsnutzers steht.

> Alle Vorträge in der Villa Ichon, Goetheplatz 4, sind kostenlos.

„In wie vielen Filmen spielen wir wohl mit?"

Informatikprofessor Hans-Jörg Kreowski: Überwachungskameras bedrohen Bürgerrechte / Ausstellungsreihe

Der Bremer Universitätsprofessor Hans-Jörg Kreowski gehört zu den „Informatikern für den Frieden", die jetzt in der Villa Ichon ausstellen. FOTO: PETRA STUBBE

Von unserer Mitarbeiterin
Annekathrin Gut

Hans-Jörg Kreowski verweist darauf, dass auch unbescholtene Bürger bei einer Rasterfahndung unter Verdacht geraten

Sammelwut kontra Bürgerrecht

Kritische Informatiker warnen vor zu viel Überwachung

Von unserem Mitarbeiter
York Schaefer

BREMEN. Wissenschaftlern fällt es oft schwer, sich mit ihrer eigenen Disziplin kritisch auseinanderzusetzen und über den eigenen Tellerrand zu schauen. Weit verbreitet sind der Glaube an die eigene Unfehlbarkeit oder die Angst, durch öffentliche Misstöne im Kampf um die knappen Geldtöpfe ins Hintertreffen zu geraten. Nicht so bei den gut 700 Mitgliedern von FIfF, dem bundesweit agierenden „Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung".

Seit 1984 beschäftigt sich FIfF kritisch mit den Auswirkungen der Informatik und des zunehmenden Computereinsatzes auf Staat und Gesellschaft. In der Bremer Villa Ichon, auch Geschäftsstelle des Forums, dokumentiert jetzt eine Ausstellung mit Zeitungsausschnitten, Flugblättern, Büchern und Plakaten die 20-jährige Arbeit der engagierten Informatiker.

„Die Anfänge von FIfF gehen auf lokale Initiativen der Friedensbewegung zurück", erzählt Hans-Jörg Kreowski, Informatikprofessor an der Bremer Universität und Vorstandsmitglied von FIfF, über die historischen Wurzeln der Organisation. Vor allem die zunehmenden Verflechtungen von Informations- und Militärtechnologie waren und sind ein Dorn im Auge der Informatiker. Der konkrete Anlass, FIfF zu gründen, war damals die Weigerung des größten Berufsfachverbandes, der Gesellschaft für Informatik (GI), unter ihren Mitgliedern Details über eine Verfassungsbeschwerde gegen den Betrieb von automatischen Frühwarnsystemen bei einem Atomkrieg zu verteilen.

„Informations- und Kommunikationstechnik muss gesellschaftlich akzeptabel und für jeden zugänglich sein", mahnt Kreowski. Vor allem die verschiedenen Überwachungsszenarien, ob im öffentlichen Raum, immer mehr aber auch in der Arbeitswelt und durch die Telekommunikation, machen ihm Angst.

Kreowski pocht auf das Recht des Bürgers auf „informationelle Selbstbestimmung", das er durch die Datensammelwut von Staat und Wirtschaft bedroht sieht. „Jeder Bürger hat das Recht zu erfahren, wer wann was und wo über ihn weiß", erklärt der Informatiker den Hintergrund der geltenden rechtlichen Normen. Selbst an Hans-Jörg Kreowskis Arbeitsplatz, der Universität, sind inzwischen Videokameras „ohne Ankündigung" installiert worden.

Einige FIfF-Aktive stellen ein paar ihrer ersten Plakate vor: Mitarbeiterin Anja Riemer, Bundesvorsitzender Prof. Dr. Hans-Jörg Kreowski, Ralf E. Streibl (FIfF-Regionalgruppe Bremen) und Mitarbeiter Markus Hanses (von links nach rechts). FOTO: ANDREAS KALKA

> Die Ausstellung „Momentaufnahmen der Geschichte – Themen, Ereignisse, Plakate" ist noch bis zum 27. Februar in der Villa Ichon, Goetheplatz 4, zu sehen. Öffnungszeiten sind Montag bis Freitag von 11 bis 19 Uhr und Sonnabend von 11 bis 13 Uhr.

Bremer Nachrichten
14.02.2005

FÜR EINE BESSERE WELT
Menschen, Aktionen, Organisationen und Visionen

Home | Organisationen | Die Idee | Newsletter | Impressum

Prof. Hans-Jörg Kreowski

26. Oktober 2008 | Von ilona | Kategorie: Interviews

AUDIO. Das Forum InformatikerInnen für Frieden e.V. – kurz auch FIfF genannt – setzt sich dafür ein, dass Informationstechnik im Dienst einer lebenswerten Welt steht. Deshalb warnt die Organisation die Öffentlichkeit vor Entwicklungen in diesem Fachgebiet, die sie für schädlich halten und setzt möglichen Gefahren
entgegen. Daneben will sie beispielsweise gegen den Einsatz der
kämpfen, sich für eine Abrüstung der Informatik in militärischen
eine humane und ökologische Orientierung entgegen set
Wirtschaftskreisläufen mit Hilfe von Informationstechnik fördern –

Etwa 700 Männer und Frauen aus Wissenschaft und Praxis eng
und Informationstechnik. Dabei geht es vor allem darum, dass die
nachdenken und sich bewusst machen, dass nicht alle Probl
willkommen, die Informationstechnik verwenden oder sich Gedank
Ausbildung, in Wissenschaft und der beruflichen Praxis. Der Fl
Auseinandersetzung, die offen ist für alle, die mitarbeiten möchten

Professor Kreowski gehört zu den Mitbegründern des FIfF. Er sprac
Risiken der IT – und welche Konsequenzen dies aus seiner Sicht für

Hier geht es zum Audio-Interview (MP3).

Und hier zur Website von FIfF: www.fiff.de

AK VORRAT

Seite | Diskussion | bearbeiten | Versionen/Autoren

Wiki
- Wiki-Hauptseite
- Hilfe
- Neueinsteiger-Portal
- Wiki-Werkstatt
- "Sandkasten"
- Letzte Änderungen

AK Vorrat
- AK-Vorrat-Portal
- Geschützter Bereich
- Spenden

Suche
Seite | Suche

Werkzeuge
- Links auf diese Seite

Freedom Not Fear 2008/Berlin de/Reden/Kreowski

< Freedom Not Fear 2008 | Berlin de

Redebeitrag von Hans-Jörg Kreowski (FifF) auf der Demo "Freiheit statt Angst" am 11.10.2008 in Berlin

Liebe Kundgebungsteilnehmerinnen und -teilnehmer,

es ist wirklich großartig, dass sich so viele zu dieser Demonstration aufgerafft haben. Denn es ist höchste Zeit, dem weit verbreiteten Datenmissbrauch und dem kriminellen Umgang mit personenbezogenen Daten Einhalt zu gebieten sowie Widerstand aufzubauen gegen die ausufernde Vorratsdatenspeicherung und Überwachungshysterie.

Es ist höchste Zeit: Schluss mit Datensammelwut und Überwachungswahn.

In den letzten Monaten ist Welle um Welle von Datensammel- und Überwachungsskandalen durch die öffentlichen Medien geschwappt. Lidl hat seine Mitarbeiterinnen und Mitarbeiter ausspioniert. Die Telecom hat in ihrem Management nach undichten Stellen gefahndet, durch die interne Informationen an die Presse gelangt waren. Die

Abb. 8. Hans-Jörg Kreowski „fiffig" öffentlich, u.a. in Zusammenhang mit einer Vortragsreihe und Ausstellung zu 20 Jahren FIfF in der Villa Ichon, bei einem Audio-Interview „für eine bessere Welt" sowie als Redner bei der Großdemonstration „Freiheit statt Angst" am 11.10.2008 in Berlin.

Die Jahrestagung 2001 fand vom 28. bis 30.9.2001 unter dem Eindruck der Terroranschläge vom 11. September statt. Viele Redner/innen nahmen direkt darauf Bezug und das Tagungsprogramm wurde kurzfristig um einen Beitrag eines Völkerrechtlers ergänzt. Eine vorher geplante Diskussionsrunde erhielt mit Blick auf den anstehenden *War against Terror* kurzfristig ein neues Motto: *Informatik und Krieg: Das Ende der Machbarkeiten*. Sie wurde live von der Tagung in Radio Bremen 2 übertragen, Teilnehmer waren Ralf Bendrath, Wolfgang Coy, Hans-Jörg Kreowski, David L. Parnas und Nazir Peroz, die Moderation übernahm Wolfgang Hagen, Programmleiter bei Radio Bremen. In einem von der Mitgliederversammlung auf der Tagung beschlossenen Appell für Frieden und Freiheit wurde – wie sich bereits kurz danach zeigte absolut zu Recht – vor indirekten Folgen gewarnt: *„Das FIfF ist ebenso besorgt, dass die Terroranschläge als Vorwand dienen werden, die Grund- und Freiheitsrechte einzuschränken, insbesondere auch das Recht auf informationelle Selbstbestimmung zu beschneiden, den Datenschutz auszuhöhlen, die ausländischen – vor allem die islamischen – Mitbürgerinnen und Mitbürger unter Generalverdacht zu stellen und die Überwachung aller zu forcieren.“*

Die Jahrestagung 2006 wollte im *Informatikjahr* 2006 ein kleines Gegengewicht zu den Danksagungen an die Informatik bilden und einige leisere Töne im verbreiteten Jubel anschlagen. Es ging einmal mehr darum, zur Vorsicht zu mahnen und den Diskurs auch über problematische Entwicklungen einzufordern. Der Tagungstitel *alles hören, alles sehen, alles machen – dank Informatik* nahm daher mit einem Augenzwinkern das Motto des Wissenschaftsjahres auf. Zum Auftakt der Tagung diskutierten im *Haus der Wissenschaft* Günter Dueck und Hans-Jörg Kreowski, moderiert von Andreas Spillner.

Nachdem die Jahrestagung 2008 in Aachen sich hauptsächlich um aktuelle Bezüge von Informatik zu Krieg und Frieden drehte, liegt es 2009 für die Tagung zum 25-jährigen Bestehen des FIfF nahe, den Fokus explizit auf die gesellschaftliche Verantwortung zu legen. Die Tagung beginnt erneut mit einer Auftaktveranstaltung im Haus der Wissenschaft: Unter dem Titel *Netz_aktiv* diskutieren Lars Reppesgaard und Hendrik Speck über Google, soziale Netzwerke, Datenschutz und neue Nutzungsformen der Informationstechnik. Im weiteren Verlauf der Tagung wird der scheidende Vorsitzende Hans-Jörg Kreowski einen Hauptvortrag halten – als Rückschau und Positionsbestimmung des FIfF.

Hans-Jörg Kreowski hat das FIfF von Anfang bis heute inhaltlich, organisatorisch, finanziell und strukturell intensiv unterstützt, längst nicht nur während seiner Zeiten im Vorstand (1993 bis 1997, sowie seit 2003 als Vorsitzender). Es wäre unmöglich, all seine Aktivitäten aufzuzählen zu wollen – im vorstehenden Artikel klingt nur ein Bruchteil davon an. So bleibt an dieser Stelle nur eines zu sagen: *Danke, Hans-Jörg!*

Bemerkung. Ein Grußwort des stellv. FIfF-Vorsitzenden Stefan Hügel findet sich auf Seite 29 dieser Festschrift.

Literatur

Bernhardt, U.; Genrich, H.; Ruhmann, I. (1992). Der Prozess Verantwortung. In H.-J. Kreowski (Hrsg.): *Informatik zwischen Wissenschaft und Gesellschaft. Zur Erinnerung an Reinhold Franck.* Informatik-Fachberichte Nr. 309. Berlin: Springer, Seite 242–254.

Büllesbach, A.; Garstka, H.-J. (2005). Computerrecht – Meilensteine auf dem Weg zu einer datenschutzgerechten Gesellschaft. In *Computer und Recht*, **21**(10), Seite 720–724.

Kreowski, H.-J. (1984). Aufbruch zu einer anderen Informatik. In *Rundbrief 1/1984 des FIfF*, Seite 14ff.

Kreowski, H.-J.; Risse, T.; Spillner, A.; Streibl, R.E.; Vosseberg, K. (Hrsg.) (1995). *Realität und Utopien der Informatik.* Münster: agenda.

Kreowski, H.-J. (2007). Demokratie sichern und entfalten. In *Wider den Zeitgeist.* Sonderbeilage zur FIfF-Kommunikation 3/2007,

Kreowski, H.-J. (Hrsg.) (2008). *Informatik und Gesellschaft. Verflechtungen und Perspektiven.* Reihe: Kritische Informatik, Bd. 4. Münster: Lit.

Löhr, K.-P. (1984). Zur Entstehung des FIfF. In *Rundbrief 1/1984 des FIfF*, Seite 17ff.

Steinmüller, W. (2007). Das informationelle Selbstbestimmungsrecht. Wie es entstand und was man daraus lernen kann. In *FIfF-Kommunikation*, **24**(3), Seite 15–19.

Ralf E. Streibl

Studienzentrum Informatik
Universität Bremen
Postfach 330 440
D-28334 Bremen (Germany)
res@informatik.uni-bremen.de

Ralf E. Streibl studied Psychology and Computer Science at the University of Erlangen. At present he works at the University of Bremen as a lecturer (mainly in the field of informatics and society) as well as in the student advisory service. Having been a FIfF member since the annual meeting 1988 in Hamburg (by the way: Hans-Jörg Kreowski had a panel discussion there, so there was the first short encounter), he joined the FIfF Regionalgruppe Bremen when starting to work at the University of Bremen in 1993. Hans-Jörg Kreowski decided to leave the board of FIfF in 1997. Ralf became his successor and stayed until 2003, when Hans-Jörg returned to the board as its chairman. Since many years Ralf is furthermore active in the editorial staff of the FIfF-KOMMUNIKATION.

Forum InformatikerInnen für Frieden und gesellschaftliche Verantwortung (FIfF) e.V.

- Geschäftsstelle -
Goetheplatz 4
D-28203 Bremen (Germany)
fiff@fiff.de
www.fiff.de

Author Index

Sources of Illustrations

Note: Numbers in square brackets refer to publications of Hans-Jörg Kreowski listed at the beginning of the Festschrift. The following additional references are used.

[Dre06] Frank Drewes: *Grammatical Picture Generation. A Tree-Based Approach.* Texts in Theoretical Computer Science. An EATCS Series. Springer (2006).

[DKH00] Frank Drewes, Renate Klempien-Hinrichs: Picking Knots from Trees. The Syntactic Structure of Celtic Knotwork. In M. Anderson, P. Cheng, V. Haarslev, Proc. Diagrams 2000, *Lecture Notes in Artificial Intelligence* 1889:89–104 (2000).

Unless otherwise mentioned, all syntactic pictures have been generated by Frank Drewes using TREEBAG, and all collages of photographs on the front pages of colour sheets have been arranged by Caro von Totth, August 2009.

Front cover: Designed by Caro von Totth, August 2009. The raven makes a sound that may be spelled *gra gra*, which is an abbreviation for *graph grammars.* This observation led Annegret Habel and Hans-Jörg Kreowski to design the raven logotype for the 4th International Workshop on Graph Grammars and Their Application to Computer Science in Bremen in March 1990. Since then, it has frequently occurred in the context of graph transformation, e.g., on the Proceedings of the International Conference on Graph Transformation in 2002, 2006, and 2008 (LNCS volumes 2505, 4178, and 5214).

First page: The photograph of Hans-Jörg Kreowski appears courtesy of Dorle Kreowski, 2009.

Hans-Jörg's scientific family tree on page 1: Illustration made by Caro von Totth, 2009.

o ei e o a aoaio ye (the lost characters)