Lecture Notes in Computer Science 2505

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Tokyo

Andrea Corradini Hartmut Ehrig
Hans-Jörg Kreowski Grzegorz Rozenberg (Eds.)

Graph Transformation

First International Conference, ICGT 2002
Barcelona, Spain, October 7-12, 2002
Proceedings



Springer

Series Editors

Gerhard Goos, Karlsruhe University, Germany
Juris Hartmanis, Cornell University, NY, USA
Jan van Leeuwen, Utrecht University, The Netherlands

Volume Editors

Andrea Corradini
Università di Pisa, Dipartimento di Informatica
Corso Italia 40, 56125 Pisa, Italy
E-mail: andrea@di.unipi.it

Hartmut Ehrig
Technical University of Berlin, Computer Science Department
Franklinstraße 28/29, 10587 Berlin, Germany
E-mail: ehrig@cs.tu-berlin.de

Hans-Jörg Kreowski
University of Bremen, Computer Science Department
Bibliothekstraße 1, 28359 Bremen, Germany
E-Mail: kreo@informatik.uni-bremen.de

Grzegorz Rozenberg
Leiden University, Leiden Institute of Advanced Computer Science
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands
E-mail: rozenber@liacs.nl

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Graph transformation : first international conference ; proceedings / ICGT 2002, Barcelona, Spain, October 7 - 12, 2002. Andrea Corradini ... (ed.). - Berlin ; Heidelberg ; New York ; Hong Kong ; London ; Milan ; Paris ; Tokyo : Springer, 2002
(Lecture notes in computer science ; Vol. 2505)
ISBN 3-540-44310-X

CR Subject Classification (1998): E.1, G.2.2, D.2.4, F.1, F.2,2, F.3, F.4.2-3

ISSN 0302-9743
ISBN 3-540-44310-X Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York,
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2002
Printed in Germany

Typesetting: Camera-ready by author, data conversion by Olgun Computergrafik
Printed on acid-free paper SPIN: 10870847 06/3142 5 4 3 2 1 0

Preface

ICGT 2002 was the first International Conference on Graph Transformation following a series of six international workshops on graph grammars with applications in computer science, held in Bad Honnef (1978), Osnabrück (1982), Warrenton (1986), Bremen (1990), Williamsburg (1994), and Paderborn (1998). ICGT 2002 was held in Barcelona (Spain), October 7–12, 2002 under the auspices of the European Association of Theoretical Computer Science (EATCS), the European Association of Software Science and Technology (EASST), and the IFIP Working Group 1.3, Foundations of Systems Specification.

The scope of the conference concerned graphical structures of various kinds (like graphs, diagrams, visual sentences and others) that are useful to describe complex structures and systems in a direct and intuitive way. These structures are often augmented by formalisms which add to the static description a further dimension, allowing for the modeling of the evolution of systems via all kinds of transformations of such graphical structures. The field of Graph Transformation is concerned with the theory, applications, and implementation issues of such formalisms.

The theory is strongly related to areas such as graph theory and graph algorithms, formal language and parsing theory, the theory of concurrent and distributed systems, formal specification and verification, logic, and semantics. The application areas include all those fields of Computer Science, Information Processing, Engineering, and the Natural Sciences where static and dynamic modeling by graphical structures and graph transformations, respectively, play an important role. In many of these areas tools based on graph transformation technology have been implemented and used.

The proceedings of ICGT 2002 consist of two parts. The first part comprises the extended abstracts of the invited talks followed by the contributed papers (in alphabetic order). The topics of the papers range over a wide spectrum, including graph theory and graph algorithms, theoretic and semantic aspects, modeling, applications, and tool issues. The second part contains a tutorial introduction to graph transformation from a software engineering point of view, and short presentations of the satellite events of ICGT 2002.

We would like to thank the members of the program committee and the secondary reviewers for their enormous help in the selection process. We are also grateful to Peter Knirsch and Sabine Kuske for their technical support in running the conference system and in editing the proceedings. Moreover, we would like to express our gratitude to the local organizers Nikos Mylonakis, Fernando Orejas (Chair), Elvira Pino, and Gabriel Valiente who did a great job. Finally, we would like to acknowledge the helpful use of the START conference system and the always excellent cooperation with Springer-Verlag, the publisher of the Lecture Notes in Computer Science.

August 2002

Andrea Corradini, Hartmut Ehrig
Hans-Jörg Kreowski, Grzegorz Rozenberg

Program Committee

Michel Bauderon	Bordeaux (France)
Paolo Bottoni	Rome (Italy)
Andrea Corradini (Co-chair)	Pisa (Italy)
Hartmut Ehrig	Berlin (Germany)
Gregor Engels	Paderborn (Germany)
Reiko Heckel	Paderborn (Germany)
Dirk Janssens	Antwerp (Belgium)
Hans-Jörg Kreowski (Co-chair)	Bremen (Germany)
Ugo Montanari	Pisa (Italy)
Manfred Nagl	Aachen (Germany)
Fernando Orejas	Barcelona (Spain)
Francesco Parisi-Presicce	Rome (Italy)
Mauro Pezzè	Milano (Italy)
John Pfaltz	Charlottesville (Virginia, USA)
Rinus Plasmeijer	Nijmegen (The Netherlands)
Detlef Plump	York (UK)
Azriel Rosenfeld	College Park (Maryland, USA)
Grzegorz Rozenberg	Leiden (The Netherlands)
Andy Schürr	Munich (Germany)
Gabriele Taentzer	Berlin (Germany)
Gabriel Valiente	Barcelona (Spain)

Secondary Referees

Charles J. Alpert	Jan Hendrik Hausmann	Mohamed Mosbah
Paolo Baldan	Berthold Hoffmann	Manfred Münch
Roswitha Bardohl	Zurab Khasidashvili	Julia Padberg
Boris Böhlen	Renate Klempien-Hinrichs	Rosario Pugliese
Jürgen Dassow	Olaf Kluge	Leila Ribeiro
Frank Drewes	Peter Knirsch	Géraud Sénizergues
Joost Engelfriet	Manuel Koch	Marta Simeoni
Ingrid Fischer	Barbara König	Dániel Varró
Felix Gatzemeier	Sabine Kuske	Nico Verlinden
Cyril Gavoille	Rugero Lanotte	Ingo Weinhold
Robert Godin	Andrea Lodi	Bernhard Westfechtel
Martin Gogolla	André Marburger	
Martin Große-Rhode	Giancarlo Mauri	
Annegret Habel	Jonathan Moffet	

Sponsoring Institutions

Ministerio de Ciencia y Tecnología (MCYT), Universitat Politecnica de Catalunya (UPC), Departament d'Universitats Recerca i Societat de l'Informació (DURSI)

Table of Contents

Invited Papers

Ubiquitous, Decentralized, and Evolving Software:
Challenges for Software Engineering 1
Carlo Ghezzi

Can Behavioral Requirements Be Executed?
(And Why Would We Want to Do So?) 6
David Harel

Bigraphs as a Model for Mobile Interaction 8
Robin Milner

Contributed Papers

Approximating the Behaviour of Graph Transformation Systems 14
Paolo Baldan and Barbara König

Transforming Specification Architectures by GenGED 30
Roswitha Bardohl, Claudia Ermel, and Julia Padberg

Decomposing Graphs with Symmetries 45
Michel Bauderon and Frédérique Carrère

Graph Transformations for the Vehicle Routing
and Job Shop Scheduling Problems 60
J. Christopher Beck, Patrick Prosser, and Evgeny Selensky

Call-by-Value λ-Graph Rewriting *Without Rewriting* 75
Maribel Fernández and Ian Mackie

Transformation: The Missing Link of MDA 90
Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood

Termination Detection of Distributed Algorithms
by Graph Relabelling Systems 106
Emmanuel Godard, Yves Métivier, Mohamed Mosbah, and Afif Sellami

Graph Transformation with Time: Causality and Logical Clocks 120
Szilvia Gyapay, Reiko Heckel, and Dániel Varró

Relabelling in Graph Transformation 135
Annegret Habel and Detlef Plump

Euler Graphs, Triangle-Free Graphs and Bipartite Graphs
in Switching Classes 148
Jurriaan Hage, Tero Harju, and Emo Welzl

Confluence of Typed Attributed Graph Transformation Systems 161
Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer

Abstraction and Control for Shapely Nested Graph Transformation 177
Berthold Hoffmann

Hyperedge Substitution in Basic Atom-Replacement Languages 192
Renate Klempien-Hinrichs

Distributed Graph Transformation Units 207
Peter Knirsch and Sabine Kuske

Describing Policies with Graph Constraints and Rules 223
Manuel Koch and Francesco Parisi-Presicce

Computer Aided Multi-paradigm Modelling to Process Petri-Nets
and Statecharts 239
Juan de Lara and Hans Vangheluwe

Using Graph Transformation as the Semantical Model
for Software Process Execution in the *APSEE* Environment 254
Carla A. Lima Reis, Rodrigo Quites Reis, Marcelo M. de Abreu, Heribert Schlebbe, and Daltro J. Nunes

Graph-Based Reengineering of Telecommunication Systems 270
André Marburger and Bernhard Westfechtel

Formalising Behaviour Preserving Program Transformations 286
Tom Mens, Serge Demeyer, and Dirk Janssens

Unparsing of Diagrams with DiaGen 302
Mark Minas and Frank Strüber

Linear Ordered Graph Grammars and Their Algebraic Foundations 317
Ugo Montanari and Leila Ribeiro

Rule Invariants in Graph Transformation Systems
for Analyzing Safety-Critical Systems 334
Julia Padberg and Bettina E. Enders

Incremental Transformation of Lattices:
A Key to Effective Knowledge Discovery 351
John L. Pfaltz

GraCAD – Graph-Based Tool for Conceptual Design 363
Janusz Szuba, Andy Schürr, and Adam Borkowski

A Formal Semantics of UML Statecharts by Model Transition Systems ... 378
Dániel Varró

Hierarchical Vertex Ordering ... 393
Sung-Ho Woo and Sung-Bong Yang

Tutorials and Workshops

Tutorial Introduction to Graph Transformation:
A Software Engineering Perspective ... 402
Luciano Baresi and Reiko Heckel

Tutorial on DNA Computing and Graph Transformation –
Computational Nature of Gene Assembly in Ciliates ... 430
Tero Harju, Ion Petre, and Grzegorz Rozenberg

TERMGRAPH 2002 – Workshop Survey ... 435
Detlef Plump

Workshop on Graph-Based Tools ... 440
Tom Mens, Andy Schürr, and Gabriele Taentzer

Workshop on Graph Transformation and Visual Modeling Techniques ... 445
Paolo Bottoni and Mark Minas

Workshop on Software Evolution through Transformations:
Towards Uniform Support throughout the Software Life-Cycle ... 450
Reiko Heckel, Tom Mens, and Michel Wermelinger

Workshop on Logic, Graph Transformations and Discrete Structures ... 455
Bruno Courcelle and Pascal Weil

Author Index ... 459

Ubiquitous, Decentralized, and Evolving Software: Challenges for Software Engineering

Carlo Ghezzi

Dipartimento di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci, 32, I-20133 Milano, Italy
Phone: +39-02-23993519, Fax: +39-02-23993411
ghezzi@elet.polimi.it

1 Introduction

The evolution of software engineering has been continuous over the past thirty years. Some major technological discontinuities, however, can be identified in this progress, which caused a more radical rethinking of the previous established approaches. This, in turn, generated research for new methods, techniques and tools to properly deal with the new challenges. This paper tries to identify some of these major evolutionary steps from a historical viewpoint, with the goal of understanding if common treats can be found to characterize them.

It is argued that major discontinuities indeed have an underlying common driving factor; namely, the goal of making software development increasingly more decentralized, flexible, and evolvable. Decentralization manifests itself in different forms: from the architecture viewpoint (from monolithic to fully distributed systems) to the process and business models.

After a brief outline of this evolution, the paper discusses the nature of the major discontinuity that we are currently facing, which is pushing decentralization, distribution and dynamic evolution to their extreme, and tries to identify the challenges to software engineering research that this is posing.

2 Architecting Software: Evolution and State of the Art

To better understand the situation that software engineering researchers need to face for the next years, it is instructive to look back at the evolution that occurred in the last decades.

In the beginning, software development methods typically assumed a stable environment (and hence stable requirements) and a fully centralized system architecture. These are the years of the waterfall lifecycle, advocated as a way to impose a much needed disciplined development style over the previous prevalent "code and fix" approach, which was recognized to be responsible for the lack of industrial quality of software.

From this rather static scenario, several evolutionary steps brought increasing degrees of dynamicity into software development. For instance, a first major shift was the recognition that requirements cannot be frozen before design and implementation, since the environment typically changes, and the requirements

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 1–5, 2002.
© Springer-Verlag Berlin Heidelberg 2002

with it. To cope with this problem, research addressed both methods and tools to support change. From the process viewpoint, this lead to flexible and incremental models, like the spiral model and prototyping-based processes. Regarding methods, this lead to fundamental principles like design for change, encapsulation and hiding of design decisions, clear distinction between interface and implementation, and between implementation and specification. Also, this led to modular and then object-oriented programming languages. Notably, the mechanisms devised to deal with software change where mostly static. A change would require modifying software, recompiling, and re-running the modified application. In a sense, dynamicity was introduced at the process level, although changes were still dealt statically at the product level.

Another significant step took place in application architectures, which was driven by changes in the underlying available physical architecture. From monolithic systems, we moved to distributed, client-server applications with multi-tier structure. This made it possible to accommodate changes due to the growth of the system, such as adding new clients, or splitting a server into a–say–two tier structure. Efforts were made to make the shift easy, by allowing interaction between clients and servers to look like as if they would coexist on the same physical machine. This was possible through mechanisms like remote procedure call and middleware that would support them. This solution supported a clear separation of the issues of components allocation on physical nodes, which affected such properties as system performance, and conceptual structure of the application, which could be designed without bothering about physical distribution. Techniques were also devised to support the ability to evaluate performance tradeoffs of the different solutions.

Another big leap was the need for decentralizing the responsibility to provide some needed functionality to pre-existing components or subsystems, and the ability of integrating in a distributed architecture such components or existing (legacy) subsystems. This was driven by two equally important issues, which both aim at decentralizing functionality and saving development costs. On the one side, the increasing availability of components-off-the-shelf to be used to build new applications, instead of starting developments from scratch. On the other, the need for reusing previously developed subsystems into new systems. This lead to the need for architectures that would support interoperability of different types of distributed components, via suitable middleware.

Certainly, this summary of the history of software engineering is reductive. Still, it shows how the evolution of software development has been largely driven by the need to accommodate increasing degrees of dynamicity, decentralization, and decoupling. Future scenarios show that this tendency is further exacerbated. Components are not only decoupled, but more and more autonomous in nature. This is becoming true irrelevant of their size. Thus for instance, macrocomponents like clients for peer-to-peer file sharing are designed in a way that is largely independent of other components. At the same time, mobile code technology enables microcomponents of the size of a language module (e.g., Java classes) to get relocated into a totally different execution context.

Dynamicity is also increasingly permeating software development. For instance, technological factors like the surge of wireless communications and mobile computing, together with factors like the structure of businesses run over the Internet, are defining application scenarios that are extremely dynamic. In the latter case, software components need to be deployed in environments that may change from time to time, due to changes in the business structure, or to adapt to the preferences of the user. In the former case, dynamicity must be dealt at an even finer time scale, since mobility defines a fluid system configuration where interactions among components may become only transient.

In these scenarios, assumptions about the existence of a single point of control, persistency, or authorization are often inefficient, impractical, or simply not applicable. More and more, the developer loses global control over the components belonging to the system, since they are autonomously and independently created and managed. Components can disappear either in an announced or an unannounced manner. Hence, system design must increasingly strive towards applications that are constituted by highly decoupled, autonomous, self-reconfiguring components. To some extent, the focus is no longer on component *integration*, rather it is shifted towards component *federation*.

It is interesting to note how some of the principles enunciated in the past for conventional systems are still valid, albeit interpreted in the new scenarios. For instance, Brooks' advices of "buy versus build", and "incremental development—grow, don't build, software" acquire new meanings when transposed to the current environment. These statements have usually been associated with the need for reusing software components whenever possible, and with a software development process that favors evolution of the product across a number of small increments. Nowadays, "buy vs. build" is becoming even more crucial. For instance, often only a portion of a distributed system is totally under control of the designer; the rest is constituted by pre-existing components and services that is convenient—or mandatory—to exploit. The current interest for Web services, once filtered from hype, is symptomatic of this trend. Similarly, incremental development is now a necessity more than a choice, with the added complexity induced by the fact that more and more the growth of the system cannot be handled statically, rather it needs to be managed during the runtime.

The next section briefly surveys some of what we believe are the most challenging (and intellectually stimulating) issues that arise in the environment defined by modern distributed systems, in an effort to help shaping a software engineering research agenda.

3 Research Challenges

The scenario depicted so far raises a number of fundamental questions, affecting software engineering practices.

It was argued that applications are increasingly built out of highly decoupled components. Nevertheless, components need to interact somehow. An immediate question is then about the kind of interface components should provide. Tradi-

tionally, components define a list of features they export to and they import from other components. A type system is typically used to ensure the correct use of such features, and more recently also to govern the lookup for a given service and the setup of the binding towards it, e.g., in Jini. Nevertheless, the syntactic matching enabled by the type system is often not enough. Instead, one would like to match component services based on their semantics. Hence the increasing importance of languages for knowledge representation, and opportunities for synergies between the fields. Moreover, the services offered by a component are typically not known in advance. In this context, reflection techniques become a fundamental asset for allowing applications to discover dynamically the characteristics of a given component, as well as for allowing a given component to reconfigure itself dynamically. Finally, in such a fluid environment, one would have guarantees about the behavior of a given component. In this context, methods like "design by contract" could gain even more importance than in conventional environments, if properly adapted.

The immediately next problem is how to make the components actually interact. Distributed computing platforms typically rely heavily on establishing a binding between the client and the supplier of some service. This is the case of mainstream middleware like RPC or distributed object technology. Desirable properties, e.g., location transparency, are usually achieved by exploiting an additional step of indirection when establishing the binding, e.g., using a lookup service. Nevertheless, the question is whether such a tightly coupled model of interaction is still suitable in a fluid environment where components tend to be highly decoupled and their overall configuration is frequently changing. By and large, two approaches have emerged to date. On one hand, there are systems where interaction still occurs through a binding to another component, but dynamicity and reconfiguration is taken under account by allowing the targeted component to be changed transparently. In this approach, binding occurs in two steps: first, a service discovery step matches the request for a service to the available offers. Second, a choice is made, and the binding is set. For these steps, we can envisage different levels of complexity and sophistication. For example, one may think that among the available services, one can choose based on some offered quality of service (and maybe be ready to pay a certain price for it). One may even think of a negotiation that goes on to establish this. And even one may think of federating a number of subservices to match a service request.

The other approach completely decouples components by replacing the binding towards another component with a binding to some other external entity, that is assumed to be global to every component. This is the case of event-based systems, where components react to the emission of events through the event dispatcher, and of systems inspired by Linda where components communicate by exchanging information through a shared tuple space.

Clearly, the tension between the two solutions is ultimately resolved only in the application context, where the choice of either style is tied to a specific functionality at hand. Nevertheless, the challenge is to devise programming ab-

stractions, as well as methods for designing and reasoning about applications that naturally support and integrate both styles.

As the last two examples suggest, the problem of enabling component interaction in a highly dynamic environment can be regarded as a coordination problem. Coordination models and languages decouple sharply the internal behavior of components from the interactions they need to carry out. Typically, the latter is represented explicitly by using some kind of abstraction, like Linda tuple spaces. One reason why coordination approaches resonate with the problems we are concerned with in this paper is that, by modeling explicitly the space where component interactions take place, coordination models naturally represent the computational context for components. The notion of context is fundamental in dealing with the high dynamicity set by the scenarios we defined here. If components are to be decoupled from one another and yet able to interact with the rest of the world, they need to have some way to define what "rest of the world" means to them. Clearly, this is likely to change according to the application domain at hand, and/or different portion of the context need to be treated in a different way for different applications. We believe that one of the major research challenges for software engineering researchers is the definition of abstractions that are able to properly capture the essence of the notion of context and its inherent dynamicity, and allow application component to customize and access their contextual view.

A radical change also occurs in the business models and the software process that drive software developments. For several classes of mainstream applications, We are moving from a situation where application development is mostly done from scratch, and under control of a single authority who dictates the requirements, supervises design, development, and often even deployment, to a situation where components are integrated, maybe even dynamically, to form new applications. Components are made availbale through the network by independent authorities. Thus the network is evolving from an information bazaar to a service bazaar. The bazaar metaphor indicates that computational resources are made available in a largely unstructured showcase, with no centralized control authority, where averybody can have some form of access. In this new setting, systems are built by federating services available in the bazaar, rather than building everything from scratch, and by publishing local information to the global world. What kinds of business models are possible for this setting? How do these affect the process models we need to follow to guide developments?

4 Conclusions

In the limited space of this contribution I tried to provide an overview of the state of the art of software engineering by evidencing how old trends, like those towards dynamicity, decentralization, and decoupling, are now exacerbated by modern distributed computing. Moreover, I touched on a few key research challenges that I believe are fundamental for dealing with this kind of systems from a software engineering standpoint.

Can Behavioral Requirements Be Executed? (And Why Would We Want to Do So?)

David Harel

The Weizmann Institute of Science

A novel approach to behavioral requirements for reactive systems is described, in which highly expressive scenario-based requirements are "played in" directly from the system's GUI, or some abstract version thereof [2], and behavior can then be "played out" freely, adhering to all the requirements [3]. The approach, which is joint with Rami Marelly, is supported and illustrated by a tool we have built – the *play-engine*.

As the requirements are played in, the play-engine automatically generates a formal version of them, in an extended version of the language of live sequence charts (LSCs) [1]. The extension includes symbolic instances [5] and time constraints [6]. As behavior is played out, the engine causes the application to react according to the universal ("must") parts of the specification; the existential ("may") parts can be monitored to check for successful completion. See Figure 1, which extends the series of figures appearing in [2], so that the current work is shown incorporated within the conventional framework for the development of reactive systems described in [2].

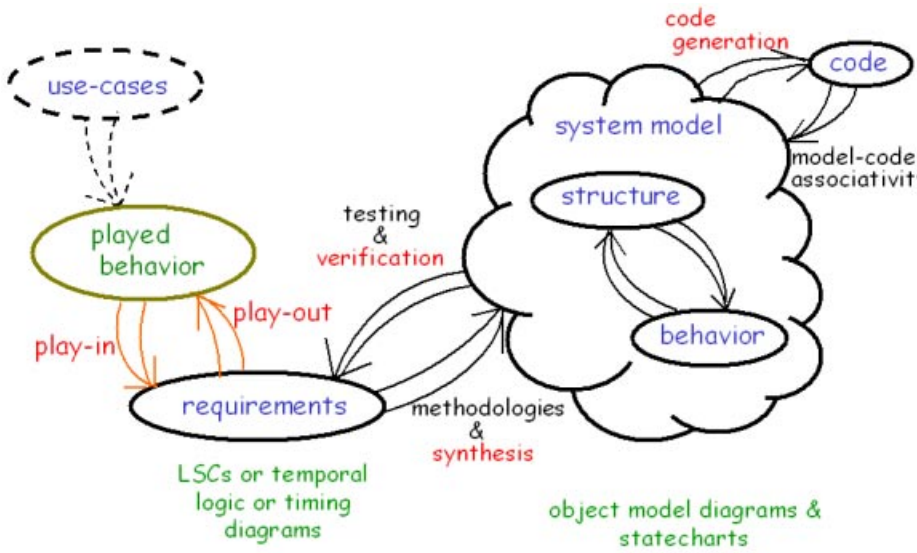


Fig. 1. Play-in/play-out within the development cycle; see Fig. 4 of [2]

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 6–7, 2002.
© Springer-Verlag Berlin Heidelberg 2002

We also describe our work on "smart" play-out, joint with Hillel Kugler and Amir Pnueli, whereby parts of the execution are driven by counterexamples produced using model-checking [4]. This makes it possible to avoid taking "bad" nondeterministic choices that may arise from the partial order within an LSC or the various interleavings of several LSCs. Thus, we employ formal verification techniques for driving the execution of the requirements, rather than for verifying the model's properties against those requirements later on. Smart play-out can also be used to find a way to satisfy an existential chart, i.e., to show how a scenario can be satisfied by the LSC specification, and is thus very useful for testing.

The entire play-in/out approach appears to be useful in many stages in the development of complex reactive systems, and could also pave the way to systems that are constructed directly from their requirements, without the need for intra-object or intra-component modeling or coding at all.

A particularly exciting application of the ideas is in the modeling of biological systems, where the information from experiments comes in the form of scenarios or scenario fragments. We are in the midst of a project involving modeling and analyzing parts of the development of the C. elegans worm using the play-engine, and will report on this effort separately.

References

1. W. Damm and D. Harel, "LSCs: Breathing Life into Message Sequence Charts", *Formal Methods in System Design* **19**:1 (2001).(Preliminary version in *Proc. 3rd IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems* (*FMOODS'99*), (P. Ciancarini, A. Fantechi and R. Gorrieri, eds.), Kluwer Academic Publishers, 1999, pp. 293–312.)
2. D. Harel, "From Play-In Scenarios To Code: An Achievable Dream", *IEEE Computer* **34**:1 (January 2001), 53–60. (Also, *Proc. Fundamental Approaches to Software Engineering* (*FASE*), Lecture Notes in Computer Science, Vol. 1783 (Tom Maibaum, ed.), Springer-Verlag, March 2000, pp. 22–34.)
3. D. Harel and R. Marelly, "Specifying and Executing Behavioral Requirements: The Play-In/Play-Out Approach", to appear.
4. D. Harel, H. Kugler, R. Marelly and A. Pnueli, "Smart Play-Out of Behavioral Requirements", *Proc. 4th Int. Conf. on Formal Methods in Computer-Aided Design* (FMCAD 2002), November 2002, to appear.
5. R. Marelly, D. Harel and H. Kugler, "Multiple Instances and Symbolic Variables in Executable Sequence Charts", *Proc. 17th Ann. AM Conf. on Object-Oriented Programming, Systems, Languages, and Applications* (OOPSLA '2002), November, 2002, to appear.
6. D. Harel and R. Marelly, "Time-Enriched LSCs: Specification and Execution", *Proc. 10th IEEE/ACM Int. Symp. on Modeling, Analysis and Simulation of Computer and Telecommunication Systems* (MASCOTS '02), October 2002, to appear.

Bigraphs as a Model for Mobile Interaction

Robin Milner

Cambridge University, The Computer Laboratory,
William Gates Building, J J Thomson Avenue, Cambridge CB3 0FD, UK

Abstract. A bigraphical reactive system (BRS) involves *bigraphs*, in which the nesting of nodes represents locality, independently of the edges connecting them. BRSs represent a wide variety of calculi for mobility, including the π-calculus. This short essay explains how bigraphs compose, and uses the π-calculus to illustrate how they already provide elements of a unifying theory for calculi of mobile interactive processes.

Bigraphs and Their Composition

This short essay presents the main features of *bigraphical reactive systems* (BRSs), a fully graphical model for mobile computation in which both location and connectivity are prominent. They address the long-term challenge to express as much as possible of world-wide distributed computing in a single mathematical model. They also aim to provide in graphical form a framework that subsumes several calculi for concurrent mobile processes, and to provide some elements of a unified theory for them.

The main feature of the BRS model is that it treats locality and connectivity as independent attributes of a distributed system. A simple example will motivate this independence. We may wish to model the worldwide web at an abstract level, where we reflect the spatial separation of agents, and their movement in space, but nevertheless wish to treat a communication between two separated agents as atomic. This amounts to saying that there are should be two notions of 'neighbour' in our model, so that 'where you are does not determine whom you may talk to'.

Thus a bigraph combines two independent graphical structures; hence the term *bigraph*. A typical bigraph is sketched in Figure 1, illustrating several features of bigraphs:

- The nodes of a bigraph may be nested; this is its *topographical* structure.
- The nodes possess *ports*, and may thereby be connected independently of their nesting; this is their *monographical* structure (a term chosen to contrast with *bigraph*).
- To each node is assigned a *control* $(K, L, \ldots)$; this dictates how many ports the node has, and how it behaves dynamically.
- The ports of a bigraph also include *names* like z_0, z_1 for interfacing; when the bigraph is inserted somewhere within another bigraph, then these names allow its nodes to be linked to those of the host bigraph.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 8–13, 2002.
© Springer-Verlag Berlin Heidelberg 2002

– A bigraph may contain a number of *sites* (indexed by $0, 1, \ldots$) at which insertions of bigraphs may be made. The sites indicate where these insertions occur topographically; the inserted bigraphs will be linked to the host monographically via *conames* such as y as shown.

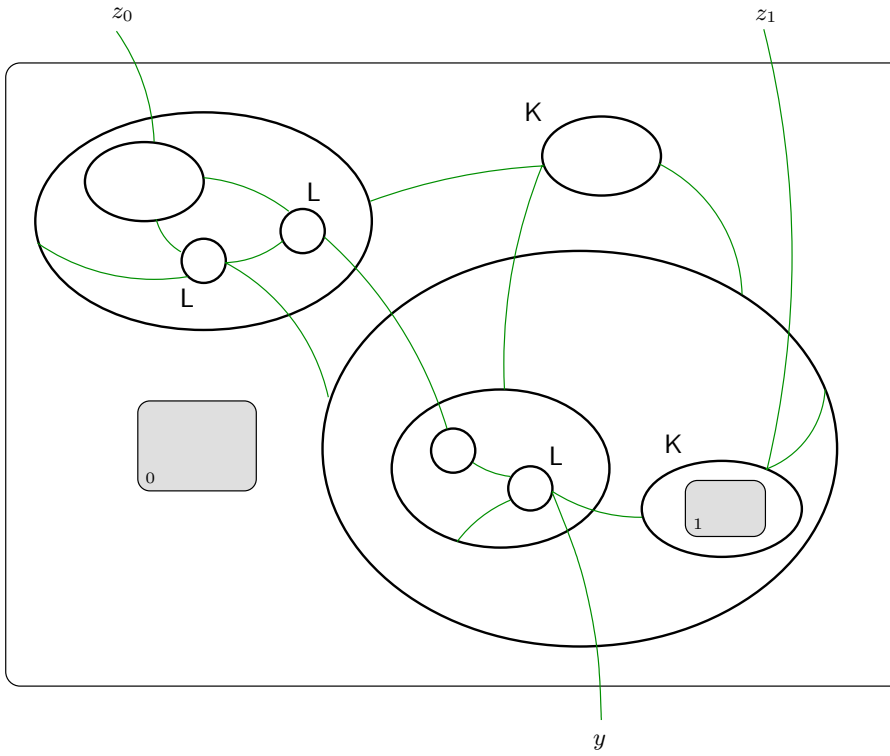


Fig. 1. An example of a bigraph, $H : \langle 2, \{y\}\rangle \to \langle 1, \{z_0, z_1\}\rangle$

We now turn briefly to mathematical representation. We have indicated how to compose bigraphs, by inserting one into another; thus they are modelled by a category Big, whose objects are *interfaces* $I, J, \ldots$ and whose arrows are bigraphs $G : I \to J$. Each interface takes the form $\langle m, X\rangle$, where m is an integer and X a finite set of names. The bigraph in Figure 1 is $H : \langle 2, \{y\}\rangle \to \langle 1, \{z_0, z_1\}\rangle$; it has two sites and one coname y, and one *region* and two names z_0, z_1.

Figure 2 shows another graph $G : \langle 3, \{x_0, x_1\}\rangle \to \langle 2, \{y\}\rangle$. The node set of G is $V = \{v_0, \ldots, v_3\}$, and the diagram shows how G can be resolved into two structures on V; a topograph G^{T} which is a forest with regions as roots and sites as leaves, and a monograph G^{M} which forgets the node nesting and retains only the port linkage. The figure also serves to illustrate composition: G shares the interface $\langle 2, \{y\}\rangle$ with H of Figure 1, so we can form the composite bigraph

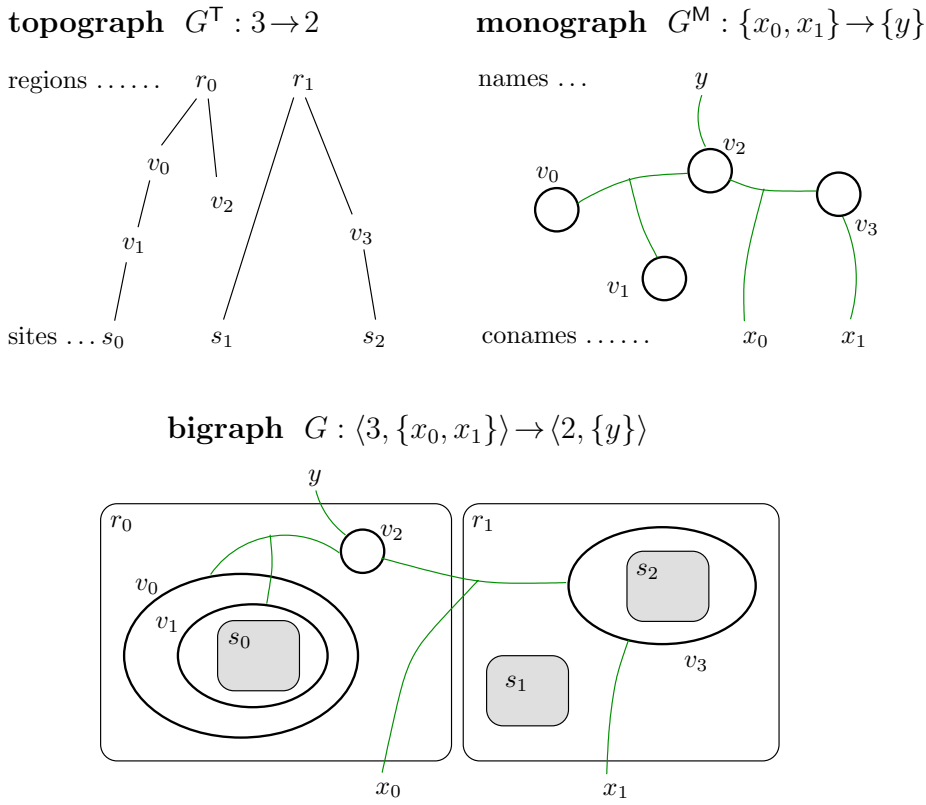


Fig. 2. Forming a bigraph from a topograph and a monograph

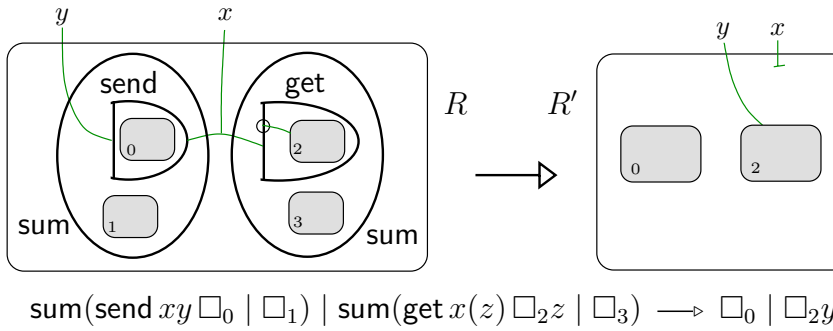


Fig. 3. Reaction rule for the π-calculus with summation

$H \circ G$; we insert the two regions of G in the sites of H (retaining all links between the regions), and then form a new link via y.

Dynamics of Bigraphs

The technical formulation of bigraphs is designed with one particular aim (among others): to represent the dynamics of process calculi, in such a way as to allow their theory to be developed uniformly. We shall illustrate this dynamics in terms of the familiar rule of reaction for the π-calculus, which is normally written

$$(\overline{x}y.P + M) \mid (x(z).Q + N) \longrightarrow\!\!\triangleright\, P \mid \{y/z\}Q \ .$$

In bigraphs, we represent the two sides of this rule by two bigraphs, R the *redex* and R' the *reactum*. This is shown in Figure 3. The four sites in R represent the parameters P, M, Q, N of the rule, as written above. We have already discussed (free) names; bigraphs can also be equipped with *binding* names, illustrated by the circled port on the get node, representing the binding name z.

The π-calculus requires three controls: send for message sending, get for message receipt and sum for summation. Note especially that nesting of nodes – topography – plays the role of *guarding*, which is essential for process calculi; activity is not permitted within these three controls. (Thus a reaction within P, say, becomes possible only after the present reaction.)[1]

Our presentation shows that naming is well represented by graphical linkage. Topography plays another rôle here; it naturally provides the *scope* of a bound name. As further evidence that our presentation of the π-calculus as bigraphs is faithful, we have found that two π-calculus terms are *structurally congruent* in the standard sense iff they become identical as bigraphs. But bigraphs do not commit us exclusively to graphical representation; their algebraic theory provides a term language for arbitrary bigraphs. Under the diagram in Figure 3 are shown the terms that represent our π-calculus reaction rule, in a mildly sugared form (using a square $\Box$ for each site); there is a close correspondence with standard π-calculus notation for the rule.

Importantly, it turns out that *labelled transition systems* (LTSs) for a large class of BRSs can be derived uniformly from reaction rules like the one we have shown. We summarise how this works. In a standard treatment of process calculi, reactions like $P \longrightarrow\!\!\triangleright\, P'$ (where P and P' are agents) are often refined into *labelled transitions* $P \xrightarrow{F}\!\!\triangleright\, P'$, where the labels F are drawn from some vocabulary expressing the interactions between an agent and its environment. These transitions – usually tailor-made for each calculus – lead naturally to the definition of behavioural preorders and equivalences, and if the label set is well-chosen these equivalences are found to be congruences. In bigraphs, following the general approach of Leifer and Milner [8], we are able to *derive* LTSs uniformly for

[1] Not all controls will be guarding. For example, a control to represent the *ambients* of Cardelli and Gordon [1] must not be guarding, since reactions are allowed to occur within ambients.

a substantial class of process calculi, in such a way that behavioural congruences (e.g. bisimilarity, failures equivalence) are ensured. Roughly speaking, in a derived transition $P \xrightarrow{F} P'$ the label F consists of a minimal graphical context that, with P inserted, enables a reaction.

Sources and Related Work

This work is ongoing. It originated with *action calculi* [9], but gains greater force and simplicity from the independent treatment of locality and connectivity. Work by Gardner [5] and Sewell [13] has contributed to the static and dynamic theory of bigraphs. Much of the basic theory appears in technical reports [2,11] (though the former is not applied directly to bigraphs), and an overview of bigraphs appears in [10]. However, the theory has become a lot simpler in the course of applying it to process calculi; the first results in this direction will appear in [6] (including more references to related work), and a more detailed technical report [7] is planned.

Our concern with process calculi and behavioural equivalences has led to an approach in bigraph theory somewhat different from the well-known tradition of graph rewriting, pioneered especially by Ehrig [3] with the double-pushout construction. But there are important correspondences, and an urgent challenge is to explore these. I thank Hartmut Ehrig for recent enlightening discussions in this direction, leading to a short report [4] that will appear soon.

References

1. Cardelli, L., Gordon, A.D.: Mobile ambients. Foundations of system specification and computational structures, LNCS 1378 (2000) 140–155
2. Cattani, G.L., Leifer, J.J., Milner, R.: Contexts and embeddings for closed shallow action graphs. University of Cambridge Computer Laboratory, Technical Report 496 (2000). [Submitted for publication.]
   Available at `http://pauillac.inria.fr/~leifer`
3. Ehrig, H.: Introduction to the theory of graph grammars. Graph Grammars and their Application to Computer Science and Biology, LNCS73, Springer Verlag (1979) 1–69
4. Ehrig, H.: Bigraphs meet double pushouts. To appear in Bulletin of EATCS (2002)
5. Gardner, P.A.: From process calculi to process frameworks. Proc. CONCUR 2000, 11th International Conference on Concurrency Theory (2000) 69–88
6. Jensen, O.H., Milner, R.: Bigraphs and transitions. Submitted for publication (2002).
7. Jensen, O.H., Milner, R.: Forthcoming Technical Report, University of Cambridge Computer Laboratory and Univerity of Aalborg Computer Science Department (2002)
8. Leifer, J.J., Milner, R.: Deriving bisimulation congruences for reactive systems. Proc. CONCUR 2000, 11th International Conference on Concurrency theory (2000) 243–258. Available at `http://pauillac.inria.fr/~leifer` .
9. Milner, R.: Calculi for interaction. Acta Informatica 33 (1996) 707–737

10. Milner, R.: Bigraphical reactive systems. CONCUR 2001,Proc. 12th International Conference in Concurrency Theory, LNCS2154 (2001) 16–35
11. Milner, R.: Bigraphical reactive systems: basic theory. Technical Report 503, University of Cambridge Computer Laboratory (2001). Available from `http://www.cl.cam.ac.uk/~rm135` .
12. Milner, R., Parrow, J., Walker D.: A calculus of mobile processes, Parts I and II. Journal of Information and Computation, Vol 100 (1992) 1–40 and 41–77
13. Sewell, P.: From rewrite rules to bisimulation congruences. Theoretical Computer Science, Vol 274(1–2) (2002) 183–230

Approximating the Behaviour of Graph Transformation Systems*

Paolo Baldan[1] and Barbara König[2]

[1] Dipartimento di Informatica, Università Ca' Foscari di Venezia, Italy
[2] Institut für Informatik, Technische Universität München, Germany
baldan@dsi.unive.it koenigb@in.tum.de

Abstract. We propose a technique for the analysis of graph transformation systems based on the construction of finite structures approximating the behaviour of such systems with arbitrary accuracy. Following a classical approach, one can construct a chain of finite under-approximations (*k-truncations*) of the Winskel's style unfolding of a graph grammar. More interestingly, also a chain of finite over-approximations (*k-coverings*) of the unfolding can be constructed and both chains converge (in a categorical sense) to the full unfolding. The finite over- and under-approximations can be used to check properties of a graph transformation system, like safety and liveness properties, expressed in (meaningful fragments of) the modal μ-calculus. This is done by embedding our approach in the general framework of abstract interpretation.

1 Introduction

Graph transformation systems (GTSs) [28] are a powerful specification formalism for concurrent and distributed systems [10], generalising Petri nets. Along the years their concurrent behaviour has been deeply studied and a consolidated theory of concurrency is now available [27,11]. Although several semantics of Petri nets, like process and unfolding semantics, have been extended to GTSs (see, e.g., [5,26,3,4]), concerning automated verification, the literature does not contain many contributions to the static analysis of GTSs (see [18,19]).

Most of the mentioned semantics for GTSs define an operational model of computation, which gives a concrete description of the behaviour of the system in terms of non-effective (e.g., infinite, non-decidable) structures. In this paper, generalising the work in [1], we provide a technique for constructing finite approximations of the behaviour for a class of (hyper)graph transformation systems. We show how one can construct under- and over-approximations of the behaviour of the system. The "accuracy" of such approximations can be fixed and arbitrarily increased in a way that the corresponding chain of (both under- and over-) approximations converges to the exact behaviour.

* Research partially supported by the EC TMR Network GETGRATS, by the IST Project AGILE, and by the MURST project TOSCA.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 14–29, 2002.
© Springer-Verlag Berlin Heidelberg 2002

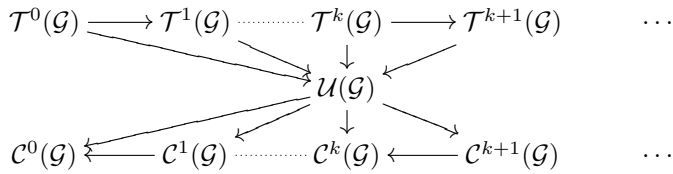
We concentrate on the unfolding semantics of GTSs, one reason for referring to a concurrent semantics being the fact that it allows to avoid to check all the interleavings of concurrent events. The unfolding construction for GTSs produces a static structure which fully describes the concurrent behaviour of the system, including all possible rewriting steps and their mutual dependencies, as well as all reachable states [26,4]. However, as already mentioned, the unfolding, being infinite for any non-trivial system, cannot be used directly for verification purposes. Given a *graph grammar*, i.e., a GTS with a start hypergraph, we show how to construct finite structures which can be seen as approximations of the full unfolding of the grammar, at a chosen level k of accuracy.

Under-approximations (k-truncations). The unfolding of a graph grammar $\mathcal{G}$ can be defined categorically as the colimit of its prefixes of finite causal depth. Hence "under-approximations" of the behaviour of $\mathcal{G}$ can be trivially produced by stopping the construction of the unfolding at a finite causal depth k, thus obtaining the so-called *k-truncation* $\mathcal{T}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$. In the case of Petri nets this is at the basis of the *finite prefix approach*: if the system is finite-state and if the stop condition is suitably chosen, the prefix turns out to be *complete*, i.e., it contains the same information as the full unfolding [22,12]. In general, for infinite-state systems, any truncation of the unfolding will be just an under-approximation of the behaviour of the system, in the sense that any computation in the truncation can be really performed in the original system, but not vice versa. Nevertheless, finite truncations can still be used to check interesting properties of the grammar, e.g., some liveness properties of the form "eventually A" for a predicate A (see Section 5).

Over-approximations (k-coverings). A more challenging issue is to provide (sensible) over-approximations of the behaviour of a grammar $\mathcal{G}$, i.e., finite approximations of the unfolding which "represent" all computations of the original system (but possibly more). To this aim, generalising [1], we propose an algorithm which, given a graph grammar $\mathcal{G}$, produces a finite structure, called *Petri graph*, consisting of a hypergraph and of a P/T net (possibly not safe or cyclic) over it, which can be seen as an over-approximation of the unfolding. Differently from [1], one can require the approximation to be exact up to a certain causal depth k, thus obtaining the so-called *k-covering* $\mathcal{C}^k(\mathcal{G})$ of the unfolding of $\mathcal{G}$.

The covering $\mathcal{C}^k(\mathcal{G})$ over-approximates the behaviour of $\mathcal{G}$ in the sense that every computation in $\mathcal{G}$ is mapped to a valid computation in $\mathcal{C}^k(\mathcal{G})$. Moreover every hypergraph reachable from the start graph can be mapped homomorphically to (the graphical component of) $\mathcal{C}^k(\mathcal{G})$ and its image is reachable in the Petri graph. Therefore, given a property over graphs reflected by graph morphisms, if it holds for all graphs reachable in the covering $\mathcal{C}^k(\mathcal{G})$ then it also holds for all reachable graphs in $\mathcal{G}$. Important properties of this kind are the non-existence and non-adjacency of edges with specific labels, the absence of certain paths (for checking security properties) or cycles (for checking deadlock-freedom). Temporal properties, such as several safety properties of the form "always A", can be proven directly on the Petri net component of the coverings (see Section 5).

The fact that the unfolding can be approximated with arbitrary high accuracy is formalised by proving that both under- and over-approximations of the unfolding, converge to the full (exact) unfolding. In categorical terms, the unfolding $\mathcal{U}(\mathcal{G})$ of a graph grammar $\mathcal{G}$ can be expressed both as the colimit of the chain of k-truncations $\mathcal{T}^k(\mathcal{G})$ and as the limit of the chain of k-coverings $\mathcal{C}^k(\mathcal{G})$:



The idea that finite under- and over-approximations can be used for checking properties of a graph grammar $\mathcal{G}$ is enforced by identifying significant fragments of the μ-calculus for which the validity of a formula in some approximation implies the validity of the same formula in the original grammar. Nicely, this is done by viewing our approach as a special case of the general paradigm of abstract interpretation.

2 Hypergraph Rewriting, Petri Nets and Petri Graphs

In this section we first introduce the class of (hyper)graph transformation systems considered in the paper. Then, after recalling some basic notions for Petri nets, we will define Petri graphs, the structure combining hypergraphs and Petri nets, which will be used to represent and approximate the behaviour of GTSs.

2.1 Graph Transformation Systems

Given a set A we denote by A^* the set of finite strings of elements of A. For $u \in A^*$ we write $|u|$ for the length of u. Moreover, if $f : A \to B$ is a function then $f^* : A^* \to B^*$ denotes its extension to strings. Throughout the paper Λ denotes a fixed set of *labels* and each label $l \in \Lambda$ is associated with an *arity* $ar(l) \in \mathbb{N}$.

Definition 1 (hypergraph). *A* (Λ-)hypergraph *G is a tuple (V_G, E_G, c_G, l_G), where V_G is a set of nodes, E_G is a set of edges, $c_G : E_G \to {V_G}^*$ is a connection function and $l_G : E_G \to \Lambda$ is the labelling function for edges satisfying $ar(l_G(e)) = |c_G(e)|$ for every $e \in E_G$. Nodes are not labelled. A node $v \in V_G$ is called* isolated *if it is not connected to any edge.*

We use rules as in the double-pushout approach [9], with some restrictions.

Definition 2 (rewriting rule). *A graph rewriting rule is a span of injective hypergraph morphisms $r = (L \xhookleftarrow{\varphi_L} K \xhookrightarrow{\varphi_R} R)$, where L, K, R are finite hypergraphs. The rule is called* simple *if (i) K is discrete, i.e. it contains no edges, (ii) no two edges in the left-hand side L have the same label, (iii) the morphism φ_L is bijective on nodes, (iv) V_L does not contain isolated nodes.*

Hereafter we will restrict to simple rules. A simple rule can delete and produce but not preserve edges, while nodes cannot be deleted (conditions (i) and (iii)). Moreover, it cannot consume two edges with the same label and its left-hand side must be connected (conditions (ii) and (iv)). These restrictions are mainly aimed at simplifying the presentation. Only (iii), which allows to apply a rule without checking the dangling condition, could require serious technical complications to be removed (but observe that deletion of nodes can be simulated considering graphs up to isolated nodes and leaving a node isolated instead of deleting it).

To simplify the notation, in the following we will assume that for any rule $r = (L \overset{\varphi_L}{\hookleftarrow} K \overset{\varphi_R}{\hookrightarrow} R)$, the morphisms φ_L and φ_R are (set-theoretical) inclusions and that $K = L \cap R$ (componentwise). Furthermore the components of a rule r will be denoted by L_r, K_r and R_r.

Definition 3 (hypergraph rewriting). *Let r be a rewriting rule. A* match *of r in a hypergraph G is any morphism $\varphi : L_r \to G$. In this case we write $G \Rightarrow_{r,\varphi} H$ or simply $G \Rightarrow_r H$, if there exists a double-pushout diagram*

$$\begin{array}{ccccc} L_r & \hookleftarrow & K_r & \hookrightarrow & R_r \\ {\scriptstyle\varphi}\downarrow & & \downarrow & & \downarrow \\ G & \hookleftarrow & D & \hookrightarrow & H \end{array}$$

Given a graph transformation system (GTS)*, i.e., a finite set of rules $\mathcal{R}$, we write $G \Rightarrow_{\mathcal{R}} H$ if $G \Rightarrow_r H$ for some $r \in \mathcal{R}$. Moreover $\Rightarrow_{\mathcal{R}}^*$ denotes the transitive closure of $\Rightarrow_{\mathcal{R}}$. A* GTS *with a (finite) start graph $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ is called a* graph grammar*.*

2.2 Petri Nets

We fix some basic notation for Petri nets [25,23]. Given a set A we will denote by $A^{\oplus}$ the free commutative monoid over A (*multisets* over A). Given a function $f : A \to B$, by $f^{\oplus} : A^{\oplus} \to B^{\oplus}$ we denote its monoidal extension.

Definition 4 (Petri net). *Let A be a finite set of action labels. An A-*labelled Petri net *is a tuple $N = (S, T, {}^\bullet(), ()^\bullet, p)$ where S is a set of places, T is a set of transitions, ${}^\bullet(), ()^\bullet : T \to S^{\oplus}$ assign to each transition its pre-set and post-set and $p : T \to A$ assigns an action label to each transition.*

The Petri net is called irredundant *if there are no distinct transitions with the same label and pre-set, i.e., if for any $t, t' \in T$*

$$p(t) = p(t') \ \wedge \ {}^\bullet t = {}^\bullet t' \quad \Rightarrow \quad t = t'. \tag{1}$$

A marked Petri net *is a pair (N, m_N), where N is a Petri net and $m_N \in S^{\oplus}$ is the initial marking.*

The irredundancy condition (1) aims at avoiding the presence of multiple events, indistinguishable for what regards the behaviour of the system. Hereafter *all* the considered Petri nets will be assumed irredundant, unless stated otherwise.

Definition 5 (causality relation). *Let N be a (marked) Petri net. The* causality relation $<_N$ *over N is the least transitive relation such that, for any $t \in T$, $s \in S$, we have (i) $s <_N t$ if $s \in {}^\bullet t$ and (ii) $t <_N s$ if $s \in t^\bullet$.*

2.3 Petri Graphs

Petri graphs, as introduced in [1], are structures consisting of a hypergraph and of a Petri net whose places are the edges of the graph.

Definition 6 (Petri graph). *Let $\mathcal{R}$ be a* GTS. *A Petri graph (over $\mathcal{R}$) is a tuple $P = (G, N, \mu)$ where G is a hypergraph, $N = (E_G, T_N, {}^\bullet(), ()^\bullet, p_N)$ is an $\mathcal{R}$-labelled Petri net with edges of G as places, and μ associates to each transition $t \in T_N$, with $p_N(t) = r$, a hypergraph morphism $\mu(t) : L_r \cup R_r \to G$ such that*

$$ {}^\bullet t = \mu(t)^\oplus(E_{L_r}) \ \wedge \ t^\bullet = \mu(t)^\oplus(E_{R_r}) \tag{2} $$

A Petri graph for a grammar $(\mathcal{R}, G_\mathcal{R})$ is a pair (P, ι) where $P = (G, N, \mu)$ is a Petri graph for $\mathcal{R}$ and $\iota : G_\mathcal{R} \to G$ is a graph morphism. The multiset $\iota^\oplus(E_{G_\mathcal{R}})$ is called initial marking *of the Petri graph. A marking $m \in {E_G}^\oplus$ is called* reachable (coverable) *in (P, ι) if it is reachable (coverable) in the underlying Petri net.*

Condition (2) allow to interpret transitions in the net as "occurrences" of rules in $\mathcal{R}$. More precisely, if $p_N(t) = r$ and $\mu(t) : L_r \cup R_r \to G$ is the morphism associated to the transition, then $\mu(t)_{|L} : L_r \to G$ must be a match of r in G such that the image of the edges of L_r in G coincides with the pre-set of t. Then, the graph items resulting from the application of r must be already in G, and the corresponding edges must coincide with the post-set of t. This is formalised by the condition over the image through $\mu(t)$ of the edges of R_r. For an example see Section 5, where Fig. 2 presents two Petri graphs for the GTS in Fig. 1.

A safe marking m of a Petri graph $P = (G, N, \mu)$ is intended to represent the subgraph of G consisting of the edges in m and of the nodes attached to these edges. For a general non-safe marking edges with k tokens will result in k "parallel" edges. This is formalised in the next definition.

Definition 7. *Let $P = (G, N, \mu)$ be a Petri graph. Given a hypergraph morphism $\varphi : G' \to G$ injective on nodes, we say that the marking $\varphi^\oplus(E_{G'})$* generates *the graph G'.*

In the following we will often confuse a marking of a Petri graph with its generated graph, and say, e.g., that a given graph is reachable in a Petri graph.

Every hypergraph G can be considered as a Petri graph $[G] = (G, N, \mu)$ for $\mathcal{R}$, by taking N as the net with $S_N = E_G$ and no transitions. Similarly, $G_\mathcal{R}$ can be seen as a Petri graph for $(\mathcal{R}, G_\mathcal{R})$ by taking as $\iota : G_\mathcal{R} \to G_\mathcal{R}$ the identity.

Definition 8 (category of Petri graphs). *A* Petri graph morphism *is a pair $\psi = (\varphi, \tau) : (G, N, \mu) \to (G', N', \mu')$ where*

- *$\varphi : G \to G'$ is a hypergraph morphism;*
- *$\tau : T_N \to T_{N'}$ is a mapping such that for every $t \in T_N$, ${}^\bullet\tau(t) = \varphi^\oplus({}^\bullet t)$ and $\tau(t)^\bullet = \varphi^\oplus(t^\bullet)$, and $p_{N'} \circ \tau = p_N$.*
- *for every $t \in T_N$, $\mu'(\tau(t)) = \varphi \circ \mu(t)$.*

The category of Petri graphs and Petri graph morphisms is denoted by PG.

It is possible to show that the category PG is finitely cocomplete, i.e. it contains all finite colimits. In particular we will later make use of pushouts and coequalizers to define unfolding and folding operations.

3 Unfolding and Under-approximations

In this section we define the unfolding of a graph grammar. Following a common approach in the literature (see, e.g., [26,29]) the unfolding is defined as the limit (actually, the categorical colimit) of the chain of its finite prefixes, each of which can be seen as an *under-approximation* of the behaviour of the system.

The finite prefixes of the unfolding are constructed inductively beginning from the start graph of the grammar and performing, at each stage, all the possible basic unfolding steps, until the given causal depth is reached. A basic step roughly consists of the "partial" application of a rule to a match, which does not delete the left-hand side, but only records the new graph item generated in the rewriting process and the rule occurrence.

To formally define a basic step we need to fix some notation. Given a transition t and a rule r we will denote by $P(t,r)$ the Petri graph $(L_r \cup R_r, N, \mu)$ where $N = (E_{L_r \cup R_r}, \{t\}, {}^\bullet t = E_{L_r}, t^\bullet = E_{R_r}, p_N(t) = r)$ and $\mu(t) = id_{L_r \cup R_r}$. By $\emptyset$ we denote a function with an empty set as domain.

Definition 9 (unfolding operation). *Let $P = (G, N, \mu)$ be a Petri graph for a* GTS $\mathcal{R}$. *Let $r \in \mathcal{R}$ be a rule and let $\varphi : L_r \to G$ be a match of r in G. The* unfolding *of P with rule r at match φ, denoted by $\mathsf{unf}(P, r, \varphi)$, is the Petri graph obtained as pushout of $(\varphi, \emptyset) : [L_r] \to P$ and $(id_{L_r}, \emptyset) : [L_r] \to P(t,r)$.*

If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define $\mathsf{unf}((P,\iota), r, \varphi) = (P', \psi \circ \iota)$ where $(\psi, \tau) : P \to P'$ is the PG morphism generated by the pushout.

We need to define the depth of an item in a Petri graph. We start with a definition of depth over Petri nets. To deal with the presence of causal cycles it is convenient to define several depth functions, each one measuring the depth of an item up to a fixed level k. Consider the monoid $\mathbb{M}_k = (\{0, \ldots, k\}, +)$, where for $m, n \in \{0, \ldots, k\}$, $m + n$ is ordinary addition if $m + n \leq k$ and $m + n = k$ otherwise.

Definition 10 (depth in a Petri net). *Let N be a Petri net. We define a function $D : (S_N \cup T_N \to \mathbb{M}_k) \to (S_N \cup T_N \to \mathbb{M}_k)$ as follows:*

$$D(d)(x) = \max\{d(s) \mid s \in S_N \ \wedge \ s < x\} + 1.$$

Then the function $depth_k : S_N \cup T_N \to \mathbb{M}_k$, assigning depth information to every Petri net item is the least fixed point of D.

The function $depth_k$ assigns to each item x of a Petri net its causal depth, i.e., the length h of the maximal chain of causally related items leading from the initial marking to x, when $h \leq k$ and k otherwise. Note that an item x located in a causality cycle has always maximal depth, i.e., $depth_k(x) = k$ for any k.

The definition generalises to Petri graphs in a straightforward way: places become edges and the depth of a node v is defined as the maximal depth of rules r where v appears in $R_r \setminus L_r$ (intuitively, of rules which can "generate" node v).

Definition 11 (depth of items in a Petri graph). *Let (P, ι) be a Petri graph with $P = (G, N, \mu)$. For any k the function $depth_k : E_G \cup T_N \to \mathbb{M}_k$ is defined as in Definition 10. This function is extended to nodes by defining, for $v \in V_G$*

$$depth_k(v) = \max\{depth_k(t) \mid p_N(t) = r \ \wedge \ v \in \mu(t)(V_{R_r} \backslash V_{L_r})\}$$

The prefixes of the unfolding of a graph grammar up to a given causal depth k are defined by the following algorithm.

Definition 12 (k-truncation). *Let $k \in \mathbb{N}$ and let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar. The algorithm generates a sequence $(P_i, \iota_i)_{i \in \mathbb{N}}$ of Petri graphs.*

(Step 0) *Initialise $(P_0, \iota_0) = ([G_{\mathcal{R}}], id_{G_{\mathcal{R}}})$.*

(Step $i+1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i.*

$\star$ *Unfolding: Find a rule r in $\mathcal{R}$ and a match $\varphi : L_r \to G_i$ such that*
- *$\varphi^{\oplus}(E_{L_r})$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^\bullet t = \varphi^{\oplus}(E_{L_r})$ and $p_{N_i}(t) = r$;*
- *for all $x \in \varphi(L_r)$ it holds that $depth_k(x) \not= k$.*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

If no unfolding step can be performed, the algorithm stops. The resulting Petri graph (P_i, ι_i) is called k-truncation *of the unfolding of $\mathcal{G}$ and denoted by $\mathcal{T}^k(\mathcal{G})$.*

It can be easily proven that the unfolding procedure described above is terminating and confluent, and thus that $\mathcal{T}^k(\mathcal{G})$ is well-defined. Furthermore, $\mathcal{T}^{k+1}(\mathcal{G})$ can be obtained from $\mathcal{T}^k(\mathcal{G})$ by performing only the unfolding steps which involve items of depth k. This gives a uniquely determined embedding $\lambda_k : \mathcal{T}^k(\mathcal{G}) \to \mathcal{T}^{k+1}(\mathcal{G})$ for any $k \in \mathbb{N}$. The diagram $\mathcal{T}^0(\mathcal{G}) \stackrel{\lambda_0}{\to} \ldots \mathcal{T}^k(\mathcal{G}) \stackrel{\lambda_k}{\to} \mathcal{T}^{k+1}(\mathcal{G}) \stackrel{\lambda_{k+1}}{\to} \ldots$ is called the *truncation tower*.

The next definition introduces the full unfolding of a graph grammar as colimit of its finite truncations (which can be shown to exist).

Definition 13 (unfolding as colimit of the k-truncations). *The* (full) unfolding *$\mathcal{U}(\mathcal{G})$ of a graph grammar $\mathcal{G}$ is the colimit of the truncation tower.*

The proposition below clarifies in which sense the unfolding represents the behaviour of the original grammar: any graph reachable in a graph grammar can be mapped *isomorphically* to a reachable subgraph of its unfolding, and, vice versa, any reachable subgraph of the unfolding is the isomorphic image of a reachable graph in the original grammar. Furthermore steps in the original grammar correspond to steps in the unfolding [26,4].

Proposition 14. *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and let $\mathcal{U}(\mathcal{G}) = (U, N, \mu)$ be its unfolding. Then for every graph G we have $G_{\mathcal{R}} \Rightarrow_{\mathcal{R}}^* G$ iff there exists an injective morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{U}(\mathcal{G})$. Furthermore, in the situation above if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \stackrel{t}{\to} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{U}(\mathcal{G})$. And if ${\varphi_G}^{\oplus}(E_G) \stackrel{t}{\to} m$ for some marking m, then there exists a graph G' such that $G \Rightarrow_{\mathcal{R}} G'$ and $m = {\varphi_{G'}}^{\oplus}(E_{G'})$.*

Clearly, k-truncations provide, in general, only under-approximations of the behaviour of the original grammar $\mathcal{G}$, i.e., only one implication of Proposition 14 holds: any graph reachable in $\mathcal{T}^k(\mathcal{G})$ is mapped isomorphically to a graph reachable in $\mathcal{G}$ and any valid computation in $\mathcal{T}^k(\mathcal{G})$ corresponds to a valid derivation sequence in $\mathcal{G}$, but, in general, not vice versa. Still, as we will see in Section 5, k-truncations can be useful for proving properties of the original grammar.

4 Folding and Over-approximations

In this section we define an algorithm which, given a graph grammar $\mathcal{G}$ and a level of accuracy k, produces a finite Petri graph $\mathcal{C}^k(\mathcal{G})$, called *$k$-covering*, which can be seen as an over-approximation of the behaviour of the grammar $\mathcal{G}$.

We have already mentioned that the full unfolding is usually infinite, also for finite-state systems. To obtain a finite over-approximation we modify the unfolding procedure by considering, besides the *unfolding rule*, also a *folding rule* which allows us to "merge" two occurrences of the left-hand side of a rule whenever they are, in a sense made precise later, one causally dependent on the other. Intuitively, the presence of such two occurrences of a left-hand side reveals a cyclic behaviour and applying the folding rule one avoids to unfold the corresponding infinite path. While guaranteeing finiteness, the folding operation causes a loss of information in a way that the resulting structure over-approximates the behaviour of the original system: every graph reachable in the original grammar $\mathcal{G}$ corresponds to a marking which is reachable in the covering and every valid derivation in $\mathcal{G}$ corresponds to a valid firing sequence in the covering (but not vice versa).

In order to compute better over-approximations of the behaviour the idea is to delay folding steps, constraining the algorithm to apply only unfolding steps until a given causal depth is reached. Roughly, this is obtained by "freezing" an initial part of the approximated unfolding, up to a given causal depth k, and by allowing only unfolding and no folding steps to affect that part. The resulting over-approximation $\mathcal{C}^k(\mathcal{G})$ is "exact" up to causal depth k, in the sense that any graph reachable in $\mathcal{G}$ in less than k steps will have a reachable *isomorphic* image in $\mathcal{C}^k(\mathcal{G})$. Instead, graphs which are reachable in a larger number of steps, in general, will be mapped homomorphically in $\mathcal{C}^k(\mathcal{G})$ (still to a reachable graph).

In this way one can obtain arbitrarily accurate approximations, a fact which is enforced by proving that the chain of k-coverings of a grammar $\mathcal{G}$ converges to the full (possibly infinite) unfolding $\mathcal{U}(\mathcal{G})$. In categorical terms, $\mathcal{U}(\mathcal{G})$ turns out to be the limit of the chain of coverings in a suitable subcategory of Petri graphs.

4.1 Computing k-Coverings

A basic definition needed to introduce k-coverings is that of a folding operation. Intuitively, it allows to merge two matches of the same rule in a Petri graph.

Definition 15 (folding operation). *Let $P = (G, N, \mu)$ be a Petri graph for a* GTS $\mathcal{R}$. *Let $r \in \mathcal{R}$ be a rule and let $\varphi', \varphi : L_r \to G$ be matches of r in G. The* folding *of P at the matches φ', φ, denoted $\mathsf{fold}(P, r, \varphi', \varphi) = P'$, is the Petri graph P' obtained as the coequalizer of $(\varphi, \emptyset), (\varphi', \emptyset) : [L_r] \to P$ in category* PG.

If (P, ι) is a Petri graph for a graph grammar $(\mathcal{R}, G_{\mathcal{R}})$, in the same situation, we define $\mathsf{fold}((P, \iota), r, \varphi', \varphi) = (P', \psi \circ \iota)$ where $(\psi, \tau) : P \to P'$ is the PG *morphism generated by the coequalizer.*

The algorithm which produces the k-covering $\mathcal{C}^k(\mathcal{G})$ generates a sequence of Petri graphs, beginning from the start graph of $\mathcal{G}$ and applying, non-deterministically, at each step, a folding or unfolding operation, until none of such steps is admitted. Folding steps will be applied only at depth k or greater. Note that as soon as folding steps are applied, the Petri graph will contain cycles.

Definition 16 (k-covering). *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and let $k \in \mathbb{N}$. The algorithm generates a sequence $(P_i, \iota_i)_{i \in \mathbb{N}}$ of Petri graphs, as follows.*

(Step 0) *Initialise $(P_0, \iota_0) = ([G_{\mathcal{R}}], id_{G_{\mathcal{R}}})$.*

(Step $i+1$) *Let (P_i, ι_i), with $P_i = (G_i, N_i, \mu_i)$, be the Petri graph produced at step i. Choose non-deterministically one of the following actions*

⋆ *Folding: Find a rule r in $\mathcal{R}$ and two different matches $\varphi', \varphi : L_r \to G_i$ of r such that*

- *$\varphi^{\oplus}(E_{L_r})$ is a coverable marking in P_i;*
- *there exists a transition $t \in T_{N_i}$ such that*

$$p_{N_i}(t) = r \ \wedge \ {}^\bullet t = {\varphi'}^{\oplus}(E_{L_r}) \ \wedge \ \forall e \in \varphi^{\oplus}(E_{L_r}) : (e \in {}^\bullet t \ \vee \ t <_{N_i} e) \quad (3)$$

- *for every edge or node $x \in E_{L_r} \cup V_{L_r}$ it holds that*

$$\varphi(x) = \varphi'(x) \ \vee \ depth_k(\varphi(x)) = depth_k(\varphi'(x)) = k. \quad (4)$$

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{fold}((P_i, \iota_i), r, \varphi', \varphi)$.

⋆ *Unfolding: Find a rule r in $\mathcal{R}$ and a match $\varphi : L_r \to G_i$ such that*

- *$\varphi^{\oplus}(E_{L_r})$ is a coverable marking in P_i;*
- *there is no transition $t \in T_{N_i}$ such that ${}^\bullet t = \varphi^{\oplus}(E_{L_r})$ and $p_{N_i}(t) = r$;*
- *there is no other match $\varphi' : L_r \to G_i$ satisfying the folding condition.*

Then set $(P_{i+1}, \iota_{i+1}) = \mathsf{unf}((P_i, \iota_i), r, \varphi)$.

If no folding or unfolding step can be performed, the algorithm terminates. The resulting Petri graph (P_i, ι_i) is called k-covering *of the unfolding of $\mathcal{G}$ and denoted by $\mathcal{C}^k(\mathcal{G})$.*

Condition (3) basically states that we can fold two matches of a rule r whenever the first one has been already unfolded producing a transition t, and the second match depends on the first one, in the sense that any edge in the second match

is already in the first one or causally depends on t. Roughly, the idea is that we should not unfold a left-hand side again, if we have already done the same unfolding step in its past, since this might lead to infinitely many steps. There are some similarities, to be further investigated, with the work in [14] where the sets of descendants and of normal forms of term rewriting systems are approximated by constructing an approximation automaton. Additionally, by Condition (4) only items of depth k can be merged, in a way that the prefix up to depth k of the unfolding is not involved in any folding operations. Actually some items of depth less than k can be part of a folding operation, but they must be left unchanged by the step.

4.2 Correctness, Termination and Confluence

We first show that the computed Petri graph $\mathcal{C}^k(\mathcal{G})$ gives an over-approximation of the behaviour of the given graph grammar, exact up to causal depth k. More precisely we prove that for any graph reachable in $\mathcal{G}$, there is a morphism into the covering $\mathcal{C}^k(\mathcal{G})$ such that the image of its edge set corresponds to a reachable marking. Furthermore, if a graph is reachable in $\mathcal{G}$ in less than k steps, then it will be mapped isomorphically to to (the graphical component) of $\mathcal{C}^k(\mathcal{G})$.

Proposition 17 (correctness). *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$ be a graph grammar and assume that the algorithm computing the k-covering terminates producing the Petri graph $\mathcal{C}^k(\mathcal{G}) = ((U, N, \mu), \iota)$. Then for every graph G*

i) if $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ there exists a morphism $\varphi_G : G \to U$ and the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{C}^k(\mathcal{G})$. Furthermore, if $G \Rightarrow_{\mathcal{R}} G'$ then ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} {\varphi_{G'}}^{\oplus}(E_{G'})$ for a suitable transition t in $\mathcal{C}^k(\mathcal{G})$.

ii) If $G_{\mathcal{R}} \Rightarrow^*_{\mathcal{R}} G$ with a (possibly parallel) derivation of length less than k then there exists an injective morphism $\varphi_G : G \to U$ such that the marking ${\varphi_G}^{\oplus}(E_G)$ is reachable in $\mathcal{C}^k(\mathcal{G})$ and $\max\{depth_k(x)) \mid x \in G\} < k$, and vice versa. Furthermore if ${\varphi_G}^{\oplus}(E_G) \xrightarrow{t} m$ for some transition t, then there exists a graph G' such that $G \Rightarrow_{\mathcal{R}} G'$ and $m = {\varphi_{G'}}^{\oplus}(E_{G'})$.

It is not obvious at first glance that the algorithm computing the k-covering always terminates. To prove termination we rely on the corresponding result in [1] where we show that it is not possible to perform infinitely many unfolding steps, without having the folding condition satisfied at some stage.

Proposition 18 (termination). *The algorithm computing the k-covering (see Definition 16) terminates for every graph grammar $\mathcal{G}$ and every $k \in \mathbb{N}$.*

In order to prove that the algorithm produces a uniquely determined result, independently of the order in which folding and unfolding steps are applied, we can show that the rewriting relation on Petri graphs induced by folding and unfolding steps is locally confluent. By the Diamond Lemma [8], for a rewriting system local confluence and termination imply confluence.

Proposition 19 (confluence). *For any input grammar $\mathcal{G}$ and $k \in \mathbb{N}$ the algorithm computing the k-covering terminates with a result $\mathcal{C}^k(\mathcal{G})$ unique up to isomorphism.*

4.3 Full Unfolding as Limit of the Coverings

The fact that folding and unfolding operations are given in terms of colimits allows us to define, for any k, a (uniquely determined) Petri graph morphism $\upsilon_k : \mathcal{C}^{k+1}(\mathcal{G}) \to \mathcal{C}^k(\mathcal{G})$. The diagram $\mathcal{C}^0(\mathcal{G}) \overset{\upsilon_0}{\leftarrow} \ldots \mathcal{C}^k(\mathcal{G}) \overset{\upsilon_k}{\leftarrow} \mathcal{C}^{k+1}(\mathcal{G}) \overset{\upsilon_{k+1}}{\leftarrow} \ldots$ is called the *covering tower*.

The next proposition presents a central result of this paper. For technical reasons we consider the full subcategory PG^* of PG having as objects Petri graphs in which every edge is coverable and every transition can be fired.

Proposition 20 (unfolding as limit of the coverings). *The limit in the category PG^* of the covering tower $\mathcal{C}^0(\mathcal{G}) \overset{\upsilon_0}{\leftarrow} \ldots \mathcal{C}^k(\mathcal{G}) \overset{\upsilon_k}{\leftarrow} \mathcal{C}^{k+1}(\mathcal{G}) \overset{\upsilon_{k+1}}{\leftarrow} \ldots$ is the full unfolding $\mathcal{U}(\mathcal{G})$ of the graph grammar.*

5 Checking Temporal Properties

In this section we illustrate how our technique can be seen as a specific instance of abstract interpretation [7,17]. Embedding our work into this context we can resort to some results from [20], thus identifying classes of temporal properties (μ-calculus formulae) which, being preserved/reflected by abstractions, can be studied over suitable approximations of a GTS.

We recall some concepts from [20], the more basic one being the formalisation of abstraction given in terms of Galois connections (over powerset lattices).

Definition 21 (Galois connection). *Let Q_1 and Q_2 be two sets of states. A* Galois connection *from $\mathcal{P}(Q_1)$ to $\mathcal{P}(Q_2)$ is a pair of monotonic functions (α, γ), with $\alpha : \mathcal{P}(Q_1) \to \mathcal{P}(Q_2)$* (abstraction) *and $\gamma : \mathcal{P}(Q_2) \to \mathcal{P}(Q_1)$* (concretization)*, such that $id_{Q_1} \subseteq \gamma \circ \alpha$ and $\alpha \circ \gamma \subseteq id_{Q_2}$.*

Next we introduce $\langle\alpha, \gamma\rangle$-simulations which turn out to coincide with simulations in the sense of Milner (see [20] for details).

Definition 22 ($\langle\alpha, \gamma\rangle$-simulation). *Let $T_i = (Q_i, \to_i)$ with $i \in \{1, 2\}$ be transition systems, where Q_i is a set of* states *and $\to_i \subseteq Q_i \times Q_i$ is the* transition relation*. Let furthermore (α, γ) be a Galois connection from $\mathcal{P}(Q_1)$ to $\mathcal{P}(Q_2)$.*

We say that T_2 $\langle\alpha, \gamma\rangle$-simulates T_2, written $T_1 \sqsubseteq_{\langle\alpha,\gamma\rangle} T_2$, if $\alpha \circ pre[\to_1] \circ \gamma \subseteq pre[\to_2]$, where the function $pre[\to_i] : \mathcal{P}(Q_i) \to \mathcal{P}(Q_i)$ is defined by $pre[\to_i](Q) = \{q \in Q_i \mid \exists\, q' \in Q : q \to_i q'\}$.

Let T_1, T_2 be transition systems and let $\varphi : T_1 \to T_2$ be a *transition system morphism*, i.e., a function $\varphi : Q_1 \to Q_2$ such that such that for any $q, q' \in Q_1$ if $q \to_1 q'$ then $\varphi(q) \to_2 \varphi(q')$ (in other words, φ is a special kind of simulation).

Then, it can be easily seen that the pair (φ, φ^{-1}) is a Galois connection and furthermore $T_1 \sqsubseteq_{\langle\varphi,\varphi^{-1}\rangle} T_2$.

We next discuss how our under- and over-approximations of the behaviour of a graph grammar can be interpreted in this context. First observe that, a Petri graph (P, ι), with $P = (G, N, \mu)$, can be associated with a transition system $\mathbb{M}_{(P,\iota)}$, having reachable markings (multi-sets of edges) as states and the firing relation of the underlying Petri net N as transition relation. Alternatively we can consider the transition system, $\mathbb{G}_{(P,\iota)}$, where states are graphs (generated by the reachable markings, in the sense of Definition 7) and the transition relation is again induced by the firing relation of N.

Let $\mathcal{G}$ be a graph grammar and consider the full unfolding $\mathcal{U}(\mathcal{G})$, the k-truncations $\mathcal{T}^k(\mathcal{G})$ and the k-coverings $\mathcal{C}^k(\mathcal{G})$. Since by Definition 13 and Proposition 20 the full unfolding is the colimit of the truncations and the limit of the coverings, we have (unique) morphisms $\eta_k : \mathcal{T}^k(\mathcal{G}) \to \mathcal{U}(\mathcal{G})$ and $\theta_k : \mathcal{U}(\mathcal{G}) \to \mathcal{C}^k(\mathcal{G})$, which can be regarded as functions from sets of markings to sets of markings and furthermore they are morphisms between the transition systems of the underlying Petri nets. Hence we have the following result.

Proposition 23. *Let $\mathcal{G}$ be a graph grammar. Then (η_k, η_k^{-1}) and $(\theta_k, \theta_k^{-1})$ are Galois connections and $\mathbb{M}_{\mathcal{T}^k(\mathcal{G})} \sqsubseteq_{\langle\eta_k,\eta_k^{-1}\rangle} \mathbb{M}_{\mathcal{U}(\mathcal{G})} \sqsubseteq_{\langle\theta_k,\theta_k^{-1}\rangle} \mathbb{M}_{\mathcal{C}^k(\mathcal{G})}$.*

Modal μ-calculus. One of the central results of [20] is the preservation and reflection of modal μ-calculus formulae on transitions systems. Recall that the modal μ-calculus is a temporal logic enriched with fixed-point operators. The syntax of μ-calculus formulae is the following:

$$f ::= \ A \mid X \mid \Diamond f \mid \Box f \mid \neg f \mid f_1 \vee f_2 \mid f_1 \wedge f_2 \mid \mu X.f \mid \nu X.f$$

where $A \in \mathcal{A}$ are *atomic propositions* and $X \in \mathcal{X}$ are *proposition variables*. The formulae are evaluated over a transition system $T = (Q, \to)$, with respect to an *interpretation* $I : \mathcal{A} \to \mathcal{P}(Q)$, associating to any atomic proposition $A \in \mathcal{A}$ the set of states $I(A)$ where it holds. A formula $\Diamond f$ / $\Box f$ holds in a state q if some / any single step leads to a state where f holds. The connectives $\neg$,$\vee$, $\wedge$ are interpreted in the usual way. The formulae $\mu X.f$ and $\nu X.f$ represent the *least* and *greatest fixed point*, respectively. We write $q \models^I f$ to mean that the (closed) formula f holds in the state q, under the interpretation I. We say that a transition system T *satisfies a (closed) formula f under an interpretation I*, written $T \models^I f$, if $q_0 \models^I f$ where q_0 is the initial state of T.

The fragment of the modal μ-calculus without negation and box operator is denoted by $\Diamond L_\mu$. By dropping negation and the diamond operator we obtain the fragment $\Box L_\mu$. Some typical *liveness* properties of the form "eventually A" (i.e., $\mu X.(A \vee \Diamond X)$) can be expressed in the fragment $\Diamond L_\mu$, whereas some typical *safety* properties of the form "always A" (i.e., $\nu X.(A \wedge \Box X)$) can be expressed in the fragment $\Box L_\mu$. However, while for linear time there exists a syntactic characterization of liveness and safety properties [24], in the case of branching time there is not yet any established definition of liveness and safety [21].

Let us come back to graph transformation systems, where atomic propositions stand for graph properties, i.e., for sets of graphs. Let (P, ι) be a Petri graph and let f be a μ-calculus formula over a set of atomic propositions $\mathcal{A}$. Assume that I_m and I_g are interpretations of $\mathcal{A}$ over $\mathbb{M}_{(P,\iota)}$ and $\mathbb{G}_{(P,\iota)}$, respectively, such that, for any $A \in \mathcal{A}$, any marking m and graph $G(m)$ generated by m

$$m \in I_m(A) \quad \textit{iff} \quad G(m) \in I_g(A). \tag{5}$$

Then it is immediate to see that $\mathbb{M}_{(P,\iota)} \models^{I_m} f$ if and only if $\mathbb{G}_{(P,\iota)} \models^{I_g} f$. Furthermore, given a graph grammar $\mathcal{G}$, seen as a transition system in the obvious way, by Proposition 14 it follows that $\mathcal{G} \models^{I_g} f$ if and only if $\mathbb{G}_{\mathcal{U}(\mathcal{G})} \models^{I_m} f$.

Using the above observations and exploiting the preservation and reflection properties in [20] we can obtain the following result. We say that a set Gr of hypergraphs is *preserved* by graph morphisms whenever the existence of a morphism $\varphi : G \to G'$ with $G \in Gr$ implies $G' \in Gr$. Symmetrically, Gr is *reflected* by graph morphisms whenever the existence of a morphism $\varphi : G \to G'$ with $G' \in Gr$ implies $G \in Gr$.

Corollary 24. *Let $\mathcal{G} = (\mathcal{R}, G_{\mathcal{R}})$, let f be a μ-calculus formula over a set of atomic propositions $\mathcal{A}$. Let I_m and I_g be interpretations satisfying (5). Then*

- *If $f \in \Diamond L_\mu$, $\mathbb{M}_{\mathcal{T}^k(\mathcal{G})} \models^{I_m} f$ and every set $I_g(A)$ is preserved by hypergraph morphisms, then $\mathcal{G} \models^{I_g} f$.*
- *If $f \in \Box L_\mu$, $\mathbb{M}_{\mathcal{C}^k(\mathcal{G})} \models^{I_m} f$ and every set $I_g(A)$ is reflected by hypergraph morphisms, then $\mathcal{G} \models^{I_g} f$.*

We have shown how to reduce the analysis of the full transition system of a graph grammar to the analysis of simpler transition systems, generated by Petri nets (underlying Petri graphs). These transition systems might still have infinitely many states, but there are several decidability results for the modal μ-calculus and other forms of temporal logics [13,15,16].

Example. Let us consider the simple graph grammar $\mathcal{S}$ in Fig. 1, where edge labels have the following meaning: C (connections), S_{pub} (public servers), S_{prv} (private servers), P_{int} (internal processes) and P_{ext} (external processes). Internal processes can wander around the network and public servers can extend the network by creating new connections. Our aim is to show that the external process is never connected to a private server and thus has access to classified data. That is, we want to show that the following logical formula is satisfied by the graph transformation system: $f = \nu X.(A \wedge \Box X)$ where the atomic proposition A holds for all the graphs in $Gr_A = \{G \mid \forall e_1, e_2 \in E_G.(l_G(e_1) = S_{prv} \wedge l_G(e_2) = P_{ext} \Rightarrow c_G(e_1) \not= c_G(e_2))\}$ (it always holds that whenever a private server and an external process appear in a graph, then they are not connected to the same node). One can easily show that Gr_A is reflected by hypergraph morphisms.

Applying the algorithm in Definition 16 to the graph grammar $\mathcal{S}$ to compute the 0-covering $\mathcal{C}^0(\mathcal{S})$, we obtain the left-hand Petri graph in Fig. 2. Observe that the formula f is not satisfied by this covering, since A is invalid already

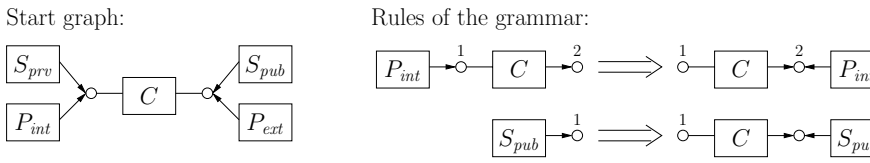


Fig. 1. The example graph grammar $\mathcal{S}$.

for the initial marking. Hence this gives us no indication whether or not the formula holds for $\mathcal{S}$. Therefore we try and compute the 1-covering and get the Petri graph on the right-hand side of Fig. 2. Now we can establish that f holds just by looking at the graph structure of the 1-covering $\mathcal{C}^1(\mathcal{S})$: edges of the form S_{prv} and of the form P_{ext} do not share a common node.

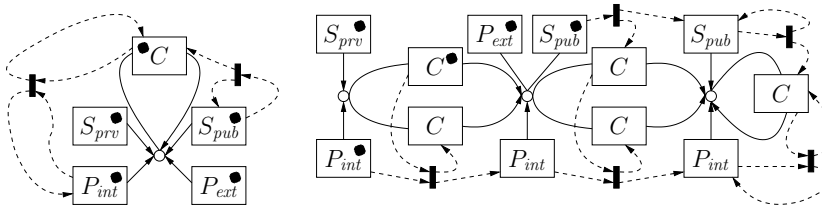


Fig. 2. The 0-covering $\mathcal{C}^0(\mathcal{S})$ and the 1-covering $\mathcal{C}^1(\mathcal{S})$ of grammar $\mathcal{S}$ in Fig. 1.

It would also be possible to extend the example by adding rules that allow movement of external processes and verify the same property. However, in this case the 1-covering would get larger and harder to draw. In [2] we have shown how to analyse a more complex GTS.

6 Conclusion

We have presented a technique for computing under- and over-approximations of the behaviour of graph transformation systems and we have identified suitable classes of properties of a GTS which can be inferred by analysing its approximations. We envision a scenario where a property of a given GTS can be checked by computing better and better approximations and verifying the property for each of them. Because of undecidability issues, this process might never terminate and it could also be costly from a complexity point of view, but with appropriate heuristics and fine-tuning of the technique, it is conceivable that several interesting properties for non-trivial GTSs can be verified in such a way.

In order to test the applicability of our theory we plan to implement the presented algorithm and to apply it to practical examples.

On the theoretical side, there are still several open problems. First, it would be interesting to classify logical formulae on graphs which are preserved and

reflected by graph morphisms, via a kind of type system. This would enable us to extend the results of Section 5 to a logic in which one is able to reason specifically about graph transition systems (see also [6]). Additionally it would be necessary to detail how the verification of these formulae on Petri graphs can be reduced to the existing model-checking techniques for Petri nets.

Another relevant issue is the extension of the developed theory to GTSs having more general forms of rules. Particularly promising, in order to decrease the size of the approximations, is the case of GTSs where rules might have a non-discrete left-hand side. This extension would require to resort to contextual nets in order to represent the Petri net structure underlying a Petri graph.

An open and, as it seems, highly non-trivial question is the treatment of finite-state GTSs. It would be quite interesting to understand if for a given GTS with only finitely many reachable graphs (up to isomorphism), there is a way to construct—using folding and unfolding steps—a finite Petri graph which gives an exact representation of the original GTS, without any proper approximation. This would allow to reduce the analysis of finite-state GTSs to that of Petri nets.

Acknowledgements

The authors are grateful to Andrea Corradini for his insightful suggestions and to the anonymous referees for their comments on the submitted version of this paper.

References

1. P. Baldan, A. Corradini, and B. König. A static analysis technique for graph transformation systems. In *Proc. of CONCUR '01*, pages 381–395. Springer-Verlag, 2001. LNCS 2154.
2. P. Baldan, A. Corradini, and B. König. Static analysis of distributed systems with mobility specified by graph grammars—a case study. In *Proc. of IDPT '02 (World Conference on Integrated Design & Process Technology)*, 2002. to appear.
3. P. Baldan, A. Corradini, and U. Montanari. Concatenable graph processes: relating processes and derivation traces. In *Proc. of ICALP'98*, volume 1443 of *LNCS*, pages 283–295. Springer Verlag, 1998.
4. P. Baldan, A. Corradini, and U. Montanari. Unfolding and Event Structure Semantics for Graph Grammars. In W. Thomas, editor, *Proc. of FoSSaCS '99*, volume 1578 of *LNCS*, pages 73–89. Springer Verlag, 1999.
5. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26:241–265, 1996.
6. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1: Foundations*, chapter 5. World Scientific, 1997.
7. P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2), 1996.
8. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In Jan van Leeuwen, editor, *Formal Models and Semantics, Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.

9. H. Ehrig. Introduction to the algebraic theory of graph grammars. In V. Claus, H. Ehrig, and G. Rozenberg, editors, *Proc. of the 1st International Workshop on Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 1–69. Springer Verlag, 1979.
10. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3: Concurrency, Parallellism, and Distribution*. World Scientific, 1999.
11. H. Ehrig, J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Concurrency, Parallelism and Distribution*. World Scientific, 1999.
12. J. Esparza. Model checking using net unfoldings. *Science of Computer Programming*, 23(2–3):151–195, 1994.
13. J. Esparza. Decidability of model-checking for infinite-state concurrent systems. *Acta Informatica*, 34:85–107, 1997.
14. T. Genet. Decidable approximations of sets of descendants and sets of normal forms. In T. Nipkow, editor, *Proc. of 9th International Conference on Rewriting Techniques and Applications*, volume 1379 of *LNCS*, pages 151–165. Springer Verlag, 1998.
15. R.R. Howell, L.E. Rosier, and H.-C. Yen. A taxonomy of fairness and temporal logic problems for Petri nets. *Theoretical Computer Science*, 82:341–372, 1991.
16. P. Jančar. Decidability of a temporal logic problem for Petri nets. *Theoretical Computer Science*, 74:71–93, 1990.
17. N.D. Jones and F. Nielson. Abstract interpretation: a semantics-based tool for program analysis. In S. Abramsky, Dov M. Gabbay, and T.S.E. Maibaum, editors, *Handbook of Logic in Computer Science, Vol. 4: Semantic Modelling*, pages 527–636. Oxford University Press, 1995.
18. M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems*. PhD thesis, Technische Universität Berlin, 2000.
19. B. König. A general framework for types in graph rewriting. In *Proc. of FST TCS 2000*, volume 1974 of *LNCS*, pages 373–384. Springer-Verlag, 2000.
20. C. Loiseaux, S. Graf, J. Sifakis, A. Bouajjani, and S. Bensalem. Property preserving abstractions for the verification of concurrent systems. *Formal Methods in System Design*, 6:1–35, 1995.
21. P. Manolios and R.J. Trefler. Safety and liveness in branching time. In *Proc. of LICS '01*, 2001.
22. K.L. McMillan. *Symbolic Model Checking*. Kluwer, 1993.
23. J. Meseguer and U. Montanari. Petri nets are monoids. *Information and Computation*, 88:105–155, 1990.
24. A. Prasad Sistla. Safety, liveness and fairness in temporal logic. *Formal Aspects of Computing*, 6(5):495–512, 1994.
25. W. Reisig. *Petri Nets: An Introduction*. EACTS Monographs on Theoretical Computer Science. Springer Verlag, 1985.
26. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, Technische Universität Berlin, 1996.
27. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, 1997.
28. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation. Volume 1: Foundations*. World Scientific, 1997.
29. V. Sassone. *On the Semantics of Petri Nets: Processes, Unfolding and Infinite Computations*. PhD thesis, University of Pisa - Department of Computer Science, 1994.

Transforming Specification Architectures by GenGED*

Roswitha Bardohl, Claudia Ermel, and Julia Padberg

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
{rosi,lieske,padberg}@cs.tu-berlin.de

Abstract. This contribution concerns transformations of specification architectures which are diagrams of sub-specifications. The graph of a diagram presents the architecture: nodes correspond to sub-specifications and edges to specification morphisms. We do not fix a specific visual specification technique, so this approach is in the tradition of high-level replacement systems.
The main emphasis of this contribution is the specification and transformation of specification architectures using GenGED. In GenGED, a visual language (VL) is defined by a visual alphabet and a visual syntax grammar. We define a VL for specification architectures by composing VLs for graphs and Petri nets enhanced by Petri net morphisms. From this VL definition a syntax-directed editor is generated supporting the editing of consistent specification architectures where local and global changes can easily be defined as transformation rules in our VL and visualized in the GenGED environment.

1 Introduction

The need for continuous development of software systems results mainly from the changing demands of the market and the technological advances. We suggest a layered approach that allows the simultaneous description of a system on an architecture and a specification level. Both levels are presented using an adequate visual modeling technique. Based on this approach we suggest a rule-based description of the model's evolution.

Two-Level Visual Design of Distributed Systems. Software architectures describe the different ways a system can be built. The larger a system is the more important this level of description is. Otherwise the detailed specification of the system, namely the different models, are indispensable. Our approach integrates these two levels and hence shows their relation. We propose a two-level representation for both the architecture and the specification of subsystems. This

* This work is part of the joint research project "DFG- Forschergruppe Petri-netz-Technologie", supported by the German Research Council (DFG), and the GRAPHIT project, supported by DLR and CNPq.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 30–44, 2002.
© Springer-Verlag Berlin Heidelberg 2002

allows the abstract representation as a graph as well as the detailed specification of the subsystems based on some adequate visual specification technique. The architecture is given as a specific graph whereas at specification level the models are given in terms of a visual modeling technique. These visual models may be defined in terms of graphs (then we have distributed graphs in the sense of [16]), Petri nets, algebraic specifications etc. The meaning of such a specification architecture is the overall specification it is intended to describe. Hence we define the *semantics of a specification architecture* as the composition of the sub-specifications according to the architecture graph.

Rule-Based Model Evolution. In order to tackle the problem of changing large and complex systems we use rules to transform our two-level description. These rules can obviously describe two different kinds of transformation. Changing the architecture implies changes on the specification level as well. Changes of the models (at the specification level) may but need not induce changes of the architecture. Hence we introduce *global changes* that cause effects on both levels (cf. Fig. 1). We suggest to conceive global changes as evolution steps. *Local changes* are executed on the specification level only. Moreover, we can distinguish local rules that describe synchronous changes of several sub-specifications.

Our approach generalizes the advantages of graph transformation to other specification techniques. We sketch the basic ideas of rule-based modification in terms of high-level replacement systems. The left-hand side L of a rule specifies the parts to be deleted and the right-hand side R those to be added. Note that in contrast to the graph transformation approach rules and transformations are *not* used for the description of the system behavior but for the description of its *changes*. The architecture level is represented by diagrams, where entities describing the subsystems are related to the specification of the corresponding subsystems at the specification level. We can distinguish two kinds of rules: local and global rules. *Local rules* imply identities at the architectural level but changes at the specification level whereas *global rules* imply changes on both levels. Fig. 1 illustrates the basic idea: On the architectural level a node is deleted and correspondingly the specification of that subsystem is deleted at the specification level. Moreover, the specification S is replaced by S'.
The semantics of the specification architecture is preserved. This means that the composed specification from the resulting specification architecture is the same as transforming the semantics of the source architecture. This ensures the compatibility with the usual transformation of specifications in terms of high-level replacement systems.

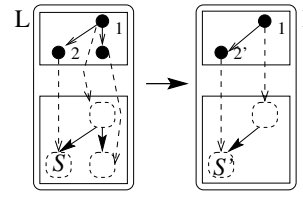


Fig. 1. Global Rule

Related Work. Several different research areas overlap with our work including architecture design techniques, architecture transformation, distributed systems engineering and evolutionary system development. As the main focus of our work lies in the evolution of visual models, the areas of software visualization and visual languages also relate to our approach. An overview on architecture

description languages (ADLs) based on components and connectors can be found in [13].

In Object Coordination Nets (OCoNs) [7] a UML architecture description is combined with Petri nets specifying the component behavior. The OCoN environment supports the visual development of OCoNs but not their relation to the architectural level. Other examples for visualizing architecture and (restricted forms of) their evolution can be found in ADL environments [4]. None of these tools allow a generic description of the visual model as we suggest in our approach using GenGED.

We model evolution steps by graph transformation which is also subject to considerable research. Software architecture reconfiguration based on graph transformation is presented by Wermelinger and Fiadeiro in [17]. They introduce a uniform algebraic framework based on category theory where an architecture is given as a graph whose nodes are refined to programs. Reconfiguration steps are modeled by conditional graph rewriting rules. In [16], Taentzer introduces Distributed Graph Transformation. In this formal specification technique an architecture level (network graph) and a component level (local graphs) are distinguished. This work is extended in [9] integrating distributed graph grammars and consistency checking rules. Our two-level approach is based on this work but allows more flexible visualization techniques than graphs for both levels.

Related approaches are given in [8,6,11,12] where software architecture graphs are transformed to adapt them to new requirements or to reduce the component interrelations. In our paper, we restrict to "editor transformation", i.e., changes are performed in the model editor. Another formal approach to software architecture transformation is given in [10]. Here, properties of component interrelations (i.e., invariants and dependencies) are formalized by a modal logic to enable consistent modifications in evolution steps. These invariants also might be expressed within graph rules for visual modeling.

2 Formal Foundations

Our approach of a *Two-Level Visual Design of Distributed Systems* is based on [16], i.e. we use the basic ideas of the algebraic approach to graph transformations in order to suggest a rule-based description of system changes and evolution. Here we concentrate on the basic concepts of our approach and do not discuss their formal representation which is described in [**?**] .

In [16] the dynamic network topology of a possibly open distributed system is described on the architectural level whereas the evolving data and system structures in the local subsystems are given at the specification level. The specification level is related to the architectural level via common interfaces. Again, different visual software architecture modeling languages [15] should be supported by our approach for the architectural level, as well as common visual modeling languages for the specification level. Therefore, we generalize [16] in allowing arbitrary specification techniques instead of graphs for the specification level. Analogously to distributed graph transformations [16] the architectural and the specification level can be related by functors. The architectural level is

represented by diagrams, where entities describing the subsystems can be related by corresponding diagram functors.

Main Concept 1 (Specification Architecture).
A specification architecture consists of an architecture graph $G = (G^N, G^E, s, t)$ and of specifications and specification morphisms. To be more precise we have for each node $i \in G^N$ a specification $\Delta(i)$ in a given specification category **Cat**. For each edge $e \in G^E$ from the source $s(e) = i$ to the target $t(e) = j$ there is a specification morphism $\Delta(e) : \Delta(i) \to \Delta(j)$ in the specification category **Cat**. This relation is expressed by a diagram functor $\Delta : \mathbf{FG} \to \mathbf{Cat}$ where **FG** is the category of finite graphs. $\Delta : \mathbf{FG} \to \mathbf{Cat}$ presents the diagram of the architecture graph G in the specification category **Cat**. △

The semantics of an architecture can be considered as the composition of the sub-specifications. The gluing of specifications is achieved by the colimit construction (the precise definition is given in [14]). The involved morphisms constitute the way the gluing is done.

Main Concept 2 (Semantics of a Specification Architecture).
The semantics of a specification architecture $\Delta : \mathbf{FG} \to \mathbf{Cat}$ is given by the gluing of all sub-specifications along the morphisms. This can be achieved in the following way: We first construct the disjoint union of all sub-specifications for each node. Then we glue recursively those parts of sub-specification that are the target of morphisms with the same source. △

Subsequently we discuss these two concepts in terms of an abstract example. That is we do not assume a specific specification technique; it might be graphs, Petri nets, algebraic specifications, COMMUNITY programs, or something else. The main feature is that we have distinct abstraction levels of representation. Namely, we present an architecture graph and corresponding diagrams of specifications, that is the left column in Fig. 2. Its formal denotation is depicted in the middle column and its functorial presentation in the right column. The rows denote the following: The top row depicts the architecture graph, the middle row the specification diagram and the bottom row the composed system, i.e. the semantics. In Fig. 2 there is a simple graph in the category **FG**. This graph consists of nodes $\{1, 2, 3\}$ and the edges in between. It is the architecture graph which is mapped by the diagram functor Δ to the specification diagram $\Delta(G)$ in category **Cat**. The specification diagram $\Delta(G)$

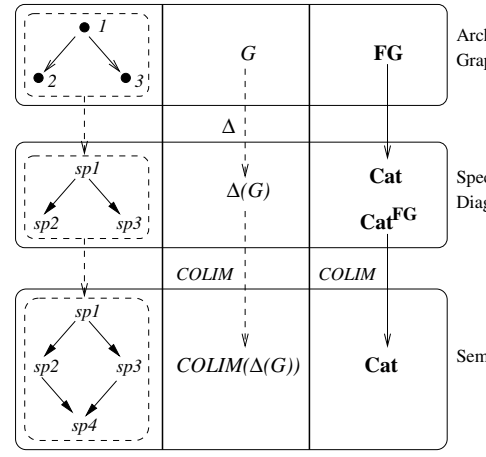


Fig. 2. Abstraction Levels of Specification Architectures

consists of specifications $sp1$, $sp2$ and $sp3$ and two specification morphisms in between. The semantics is given in the lowest row. It is the result of the gluing, namely it is the specification $sp4$. A concrete example, where Petri nets are used as specifications, is illustrated in Fig. 3.

The transfer to high-level replacement (HLR) systems causes a representation independence in the sense that we may choose the specification technique. As in [16] we need to ensure specific conditions in order to have pushouts of specification architectures or more precisely pushouts of diagram functors. Nevertheless, we have a HLR-category (namely HLR0), and we have transformations provided that the pushout conditions are satisfied. In [16] gluing condition and applicability are given explicitly. We obtain the well-known transformations in the double-pushout approach. We distinguish two levels, namely the level of the architecture graph and the level of the specifications. Hence we can qualify different types of rules; those that leave the architecture intact are called local rules. Those rules that change the architecture graph are called global. Global rules necessarily change the diagram of specifications and hence may change the specifications themselves. Analogously to [16] we can identify further those rules that describe synchronized changes of sub-specifications as special class of local rules. Examples are given subsequently in Section 3.

The compatibility with the semantics of the architecture, that is the composition of all sub-specifications, is ensured in [14]. This result is crucial as it relates our approach to the usual transformation of specifications. Hence it guarantees that the result of an architecture transformation is the same as the corresponding transformation of the composed specification.

Main Result in [14]

(Compatibility of Semantics Construction with Transformation). Given a transformation of a specification architecture $\Delta_G \stackrel{p}{\Longrightarrow} \Delta_H$ with $p = (\Delta_L \leftarrow \Delta_K \rightarrow \Delta_R)$ then we have as well a transformation

$$COLIM(\Delta_G) \stackrel{COLIM(p)}{\Longrightarrow} COLIM(\Delta_H)$$

with $COLIM(p) := (COLIM(\Delta_L) \leftarrow COLIM(\Delta_K) \rightarrow COLIM(\Delta_R))$. △

3 Example

As running example we use the well-known specification of a producer/consumer system. This example is (like the reader/writer protocol) one of the basic models for communication-based systems: Two independent agents (the *producer* P and the *consumer* C) communicate via a channel (the *buffer* B). The producer sends messages (writes) to the channel, and the consumer receives (reads) them from the channel[1].

[1] Please note that in our example the arrows describe *"being used"* relations.

In our example the sub-specifications are place/transition (P/T) nets, so the corresponding category is the category of P/T nets. The nets are the objects and the P/T nets morphisms are the edges in between. According to Concept 1 the specification architecture consists of an architecture graph, specifications and specification morphisms: Each node of the architecture graph indicates a sub-specification ($\Delta(i)$), namely a producer P, a consumer C and a buffer B, and each edge corresponds to a specification morphism in the corresponding category. The semantics of the specification architecture is given by the composition of the specifications along the morphisms.

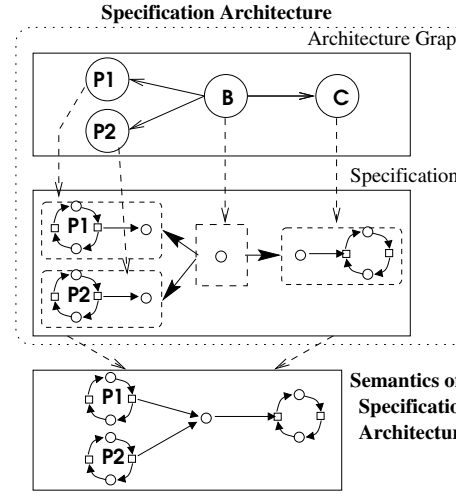


Fig. 3. A Producer/Consumer System

Now we extend the producer/consumer example by introducing a global and a local rule. As mentioned before local rules induce changes on the specification level only, whereas global rules imply changes on both levels, namely the specification level and the architecture level. Global rules usually are rules that allow for the insertion and connection of new producers, consumers and buffers.

Fig. 4 illustrates a DPO-rule for inserting a new consumer. Please note that the rule consists of architecture graphs (and graph morphisms) as well as specifications (and specification morphisms) according to Concept 1. The rule's left-hand side L demands for its application the existence of a buffer. This buffer is preserved by the rule indicated by the component I. By the right-hand side R a new consumer C is inserted and connected to the buffer.

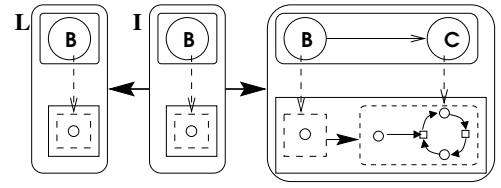


Fig. 4. Global Rule adding a new Consumer

Similar to the rule in Fig. 4 are rules for inserting a new buffer or a new producer. Deletion of components is achieved reversing the corresponding rules, i.e. exchanging the left- and the right-hand sides of the rules.

In Section 2 we mentioned the possibility to describe local changes of sub-specifications. Fig. 5 illustrates such a rule changing the specification of a producer. For its application the rule needs the existence of a specific producer specification presented in L (the producer loads in each production cycle arbitrarily one of the two buffers). The two buffers and a part of the producer are preserved indicated by I (one transition and some edges are removed). By applying this rule a new transition and edges are inserted as shown in R (in each cycle both buffers are loaded).

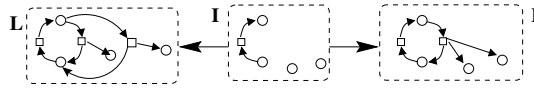


Fig. 5. Local Rule changing the Specification of a Producer

As stated in the main result in Section 2 we obtain the same result, independent whether we first construct the semantics and then apply the rules, or if we first apply the rules and then construct the semantics. For illustration let us consider Fig. 6 (a) showing the application of the global rule in Fig. 4 to the specification architecture in Fig. 3. This yields the insertion of a new consumer called $C2$. Fig. 6 (b) shows the corresponding semantics.

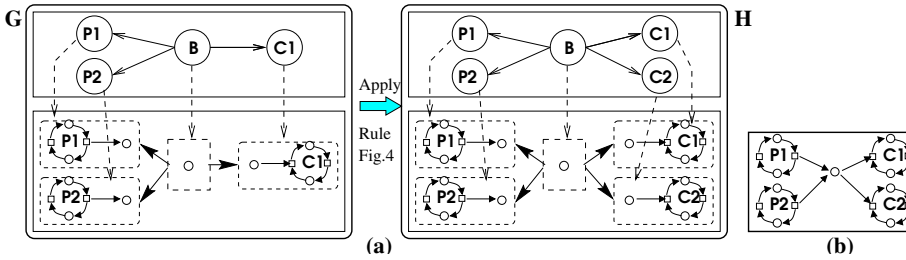


Fig. 6. Application of the Global Rule in Fig. 4 to G (a) and the Semantics of H (b)

The brief example of this section illustrates the formal basis of our approach. For examples using other specification techniques, the reader is referred to the discussion in [14]. In the following section we suggest a graphical editor for two-level specification architectures using GenGED .

4 Defining Two-Level Specification Architectures Using GenGED

The GenGED approach developed at the TU Berlin [1] allows for the generic description of visual languages (VLs) and visual environments based on VL specifications. The simplest form of a VL specification we consider here consists of a visual alphabet and a visual syntax grammar. The definition of a VL specification is based on algebraic graph transformation and graphical constraint solving techniques. VL sentences (diagrams) can be derived by applying the grammar rules in the syntax grammar to its start diagram.

In general, a diagram consists of a set of *symbol graphics* that are spatially related. We offer graphical constraints for these spatial relationships. Symbol graphics and graphical constraints concern the layout of diagrams, called *concrete syntax*. The logical part of a diagram (its symbols independent of the concrete layout) is called *abstract syntax*. The combination of both syntactical levels, called *visual syntax* level, is represented by attributed graphs.

4.1 The Visual Alphabet

A *visual alphabet* establishes a type system for *symbols* and *links*, i.e. it defines the vocabulary of a VL. It can be represented as a type graph. Here as well we distinguish the abstract and the concrete syntax level. Symbol graphics and graphical constraints specify layout conditions. In addition to logical (data) attributes like, e.g., a name for a place in a Petri net, symbol graphics define a further kind of attributes for all abstract symbol nodes.

Graphical constraints specify layout conditions. They are given by equations over constraint variables denoting the positions and sizes of graphical objects. The set of all constraint variables and constraints define a constraint satisfaction problem (CSP) that has to be solved by an adequate variable binding in a diagram conforming to the alphabet.

We now develop the visual alphabet for the two-level specification architecture language. We begin by defining two separate alphabets, one for a special kind of graphs (the architectural level) and one for place/transition (P/T) nets according to the specification level. As a second step we enhance the P/T net alphabet in order to be able to express P/T net morphisms between different nets and to combine the two alphabets.

Definition 1 (Alphabet for Architecture Graphs).
In architecture graphs, we have the symbol types Node and Edge only. Nodes are drawn as circles and may be attributed by strings (their names) placed inside the node circles. Edges are directed arcs connecting two nodes. The visual alphabet of the architecture graph language is shown in Fig. 7 (a). The dashed arrows mark the connections of the abstract syntax and the concrete syntax level. Link constraints are illustrated by dotted arrows between the symbol layouts. △

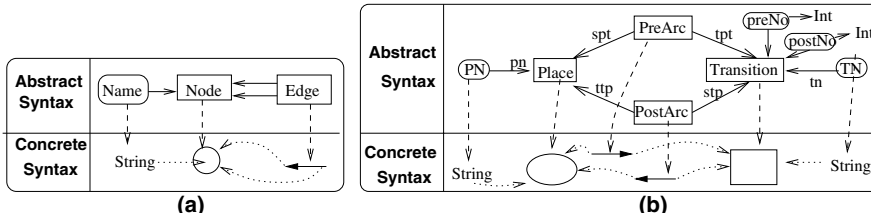


Fig. 7. Alphabets for Architecture Graphs (a) and for P/T Nets (b)

Definition 2 (Alphabet for P/T Nets).
The visual alphabet of the P/T net language, called P/T net alphabet, is illustrated in Fig. 7 (b). We defined attribute symbols for place names PN and for transition names TN which are linked to the symbol types Place resp. Transition. Each name is given by a String data type. To compute the number of places in a transition environment (pre and post domain), a transition symbol carries the integer attributes preNo and postNo. These numbers are needed later to control the insertion of mappings between transitions.

We distinguish arcs that run from places to transitions (PreArc) and arcs that run from transitions to places (PostArc). Both kind of arcs have a certain source

and target symbol where they are linked to (depicted by the edges spt, tpt, ttp, stp, short for source/target of place-transition arc resp. transition-place arc). To keep the alphabet simple, we restrict to unmarked P/T nets where the uniform arc weight is "1", and therefore arc inscriptions are omitted. (For a specification of marked P/T nets in GenGED including their firing behavior we refer to [5]).) The constraints force a specific layout of nets typed over the P/T net alphabet. For example, one constraint ensures that the place name is always "near" the ellipse (the symbol graphic for Place symbols). △

Sentences over the P/T net alphabet defined so far are unmarked P/T nets. The visual language we aim to specify, also should provide means to express morphisms between different P/T nets. This means, we have to enhance the alphabet from Def. 2 to include sentences consisting of more than one net, and to allow morphisms between different nets. We call such a relation on different P/T nets *P/T net systems*. In order to obtain an alphabet for the two-level specification architecture language, we combine the alphabets for architecture graphs and for P/T net systems.

Definition 3 (Alphabet for the Two-Level Language).
A suitable combination of the alphabets for architecture graphs and for P/T nets yielding a two-level language alphabet is shown in Fig. 8[2]: In order to support *P/T net systems*, we introduce the symbol Net (dashed frame) into the P/T net alphabet. A Net frame encloses a concrete P/T net, hence, the symbols Place and Transition are linked to Net. We introduce the symbol Morphism (arrow with rounded arrow head) linking one Net symbol to another.

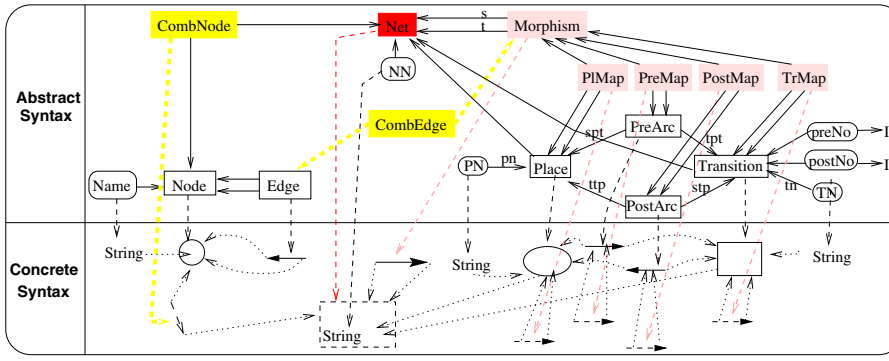


Fig. 8. Alphabet for the Two-Level Language

On the P/T net level a morphism is given by mappings from all net objects of the source net to net objects of the target net. The mappings have to be type compatible (places are mapped to places, transitions to transitions and arcs to arcs of the same type). Also, mappings have to be compatible with structure

[2] For illustrational reasons we omit to model the rectangles surrounding each of the two levels as, e.g. shown in Fig. 1.

(the source/target node of an arc is mapped to the source/target node of the arc's image), and they have to preserve the firing behavior (the transitions are mapped to transitions with the same number of ingoing and outgoing arcs only). We ensure the type compatibility by introducing the symbol types PlMap, TrMap, PreMap, PostMap (dashed arrows) for mapping different net objects. The other conditions are ensured by the grammar rules introduced later for modifying diagrams of the language.

The alphabet for P/T net systems is combined with the alphabet for architecture graphs: CombNode combines Node and Net, and CombEdge combines Edge and Morphism. △

A brief example for a sentence over the two-level alphabet is shown in Fig. 9. Please note that in this figure we omit the logical attributes. At the architecture level we have two nodes, namely a producer and a buffer which are connected by an edge. Accordingly there are two P/T nets at the specification level related by a morphism: The P/T net modeling the producer is related to the P/T net modeling the buffer. The morphism consists of one mapping of type PlMap between the buffers of the P/T nets, only.

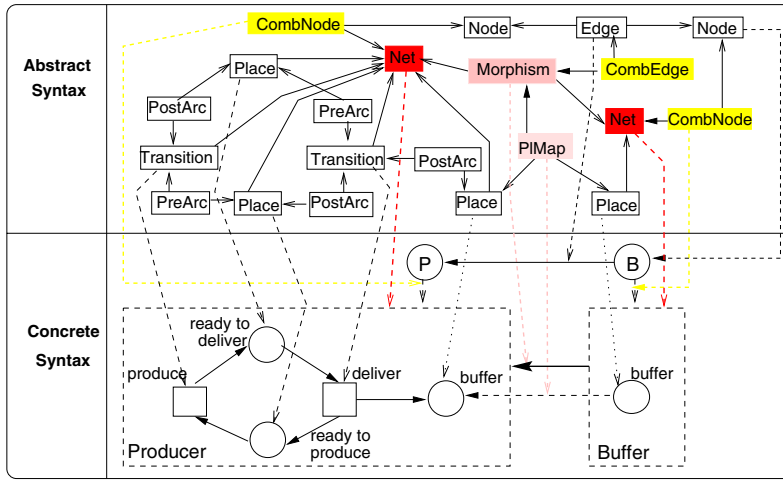


Fig. 9. Visual Sentence over the Two-Level Alphabet

4.2 The Visual Syntax Grammar

The visual alphabet depicted in Fig. 8 is the basis to define the syntax grammar for our two-level language for architecture specifications. The syntax grammar is represented by an attributed graph grammar. It consists of a *start diagram* and a finite set of *rules*. The start diagram and both sides of a rule are diagrams typed over the alphabet, as well as the diagrams which can be derived by applying grammar rules. In the rules we use in addition to the left-hand rule sides so-called negative application conditions (NACs) which restrict the application of

a rule. A NAC is a graph containing a forbidden graph pattern. This means that the rule must not be applied to a sentence if there is a match from the NAC to the sentence, i.e. the forbidden pattern is found in the sentence. Moreover, rule applications can be restricted by boolean conditions over attributes.

In Def. 5 we define the syntax grammar for the two-level language combined by graphs and P/T net systems. This definition has to be fixed once and allows the generation of a syntax-directed editor for manipulating two-level models (specification architectures) in GenGED, i.e., global and local changes are possible. An example for a global rule is the rule adding a new component node which means at the same time to add a new (initially empty) specification net at the specification level. Examples for local rules are rules adding places or transitions to an existing P/T net. In the following, we first present the local rules. They operate on the specification level only and can thus be considered as syntax rules for the VL of P/T nets over the alphabet shown in Fig. 7 (b) extended by the Net symbol introduced by the two-level language alphabet in Definition 3. Please note that we restrict the grammar rules to language-generating rules. Rule morphisms are indicated visually by corresponding node types in the left- and right-hand sides of each rule.

Definition 4 (P/T Net Syntax Grammar Rules).
Fig. 10 illustrates some syntax grammar rules for our P/T net language (at specification level) based on the visual alphabet in Definition 3.

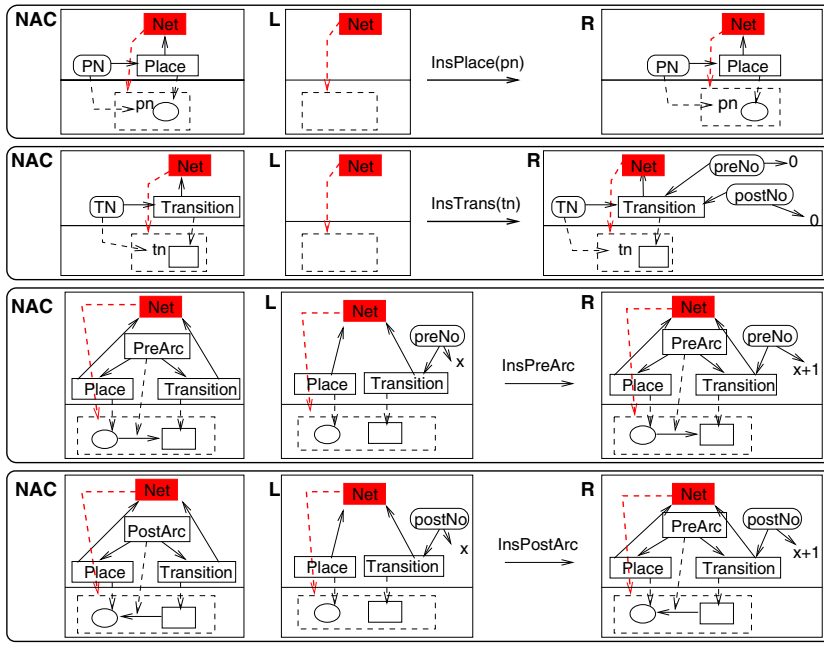


Fig. 10. Visual Syntax Grammar for the P/T Net Language

The first rule InsPlace(pn) supports the insertion of a place together with a place name; the NAC requires that a place with the user-defined name given in the parameter variable pn is not existing so far in the net the rule is applied to. Analogously, the second rule supports the insertion of a transition symbol. Here the integer attributes preNo and postNo are initialized by 0 as the newly inserted transition is not yet connected to any places. The next two rules allow for the insertion of arcs, either running from a place to a transition (insPreArc) or running from a transition to a place (insPostArc). The respective counter is incremented for the transition. The NACs forbid the application if there is already such an arc. In all rules, the graphical constraints – visualized as dotted arcs on the concrete syntax level – ensure that arcs connect places and transitions in a proper way and a name of a net object is placed near the object. △

In general, a VL is generated by applying the syntax grammar rules starting with the the start sentence. Let us now extend our syntax grammar by global rules as well as by rules supporting the visualization of morphisms between P/T net systems in order to generate our two-level language.

Definition 5 (Syntax Grammar for the Two-Level Language).
Fig. 11 shows some global grammar rules for our two-level language. Please note that we used the abbreviations CN and CE for the symbols CombNode and CombEdge. The start graph of the combined syntax grammar is empty reflecting that the editing process starts with an empty editor panel.

The first rule InsComponent inserts a new component to the architecture level combined with the insertion of a new (empty) net specification. The NACs ensure that there exist no node and no net in the specification so far with the same names as the currently inserted ones (uniqueness of names).

The other rules deal with the insertion of mappings and morphisms. Note that we do not provide a rule to insert a morphism. This is done implicitly by the mapping-inserting rules. For the insertion of mappings between places or transitions of different nets we distinguish two cases: Either there is already a morphism between the two corresponding nets (due to previous mappings) or there is no morphism (first insertion of a mapping). In case a mapping is inserted for the first time, the morphism between the two corresponding nets has to be inserted together with an edge connecting two nodes at the architecture level (see e.g. rule InsPlMap1). If a morphism already exists, the mapping simply is added, but the architecture level is not changed (see e.g. rule InsPlMap+). This distinction is realized by the respective NACs and works analogously for all types of mappings.

The rules InsTrMap+ (not illustrated) and InsTrMap1 supporting the insertion of transition mappings are analogical to the insertion of place mappings but contain additionally a rule application condition that ensures that a mapping between transitions is inserted only if the number of places in the pre domain preNo and the number of places in the post domain postNo are the same for both transitions such that the firing behavior of the source net is preserved by the net morphism. Note that arcs can be mapped only if there exist mappings

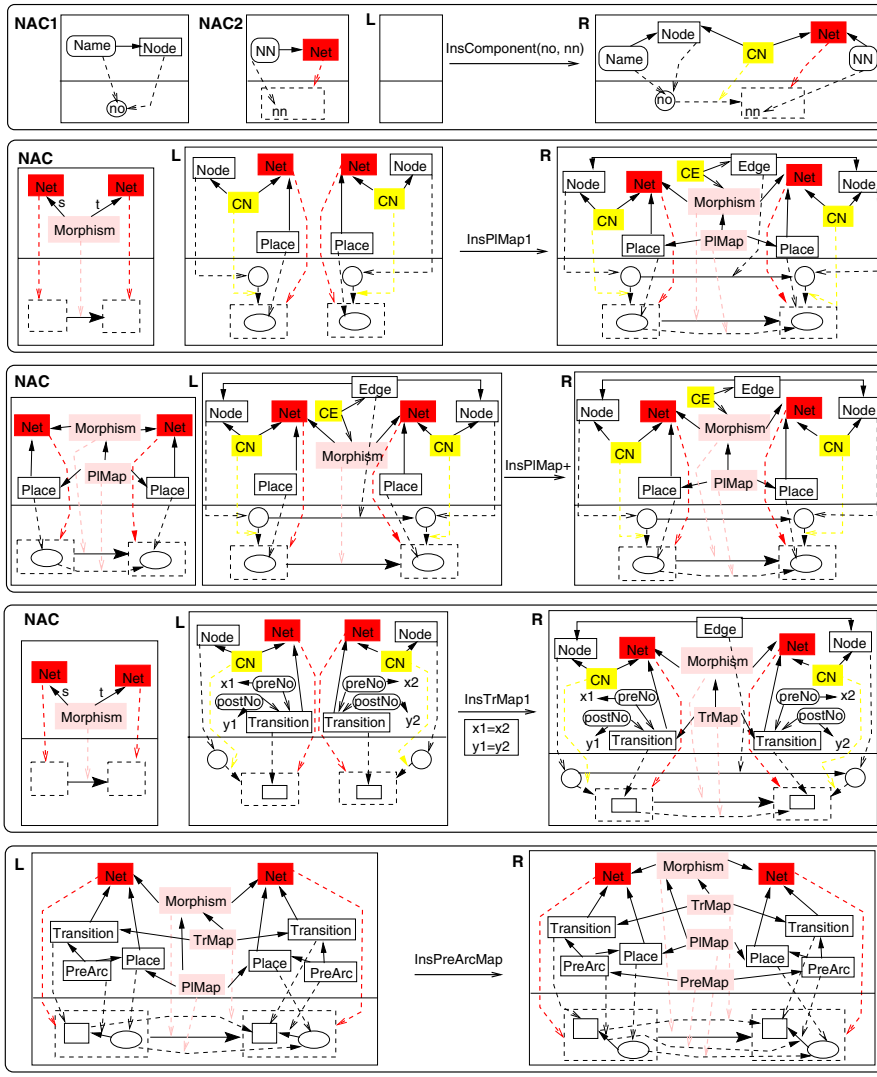


Fig. 11. Global Grammar Rules for the Two-Level Modeling Language

between their start and end nodes (see rule InsPreArcMap). The analogical rule InsPostArcMap is omitted in Fig. 11.

The complete syntax grammar for two-level models now consists of the union of the set of global rules as explained above, the set of local grammar rules shown in Fig. 10, and an empty start sentence. △

The two-level language we can generate by applying the two-level grammar consists of all diagrams over the two-level alphabet shown in Fig. 8. In order to define specification transformation steps in GenGED, the grammar can also be

used to define the left-hand and right-hand sides of the desired transformation rules. Thus a specific model transformation grammar can be visually defined as well, and rules like the transformation rule adding a new consumer (see Fig. 4) may be constructed and applied in the GenGED environment.

5 Conclusion

We have presented an approach for specification architectures that is based on diagram functors. The emphasis of this paper has been on the illustration of the main concepts and the implementation of the example within the GenGED approach. We have given a simple example, namely a producer/consumer system. First the main concepts of our approach have been exemplified, then we have used this example to illustrate the visualization of specification architectures using the GenGED approach.

Our approach for specification architectures is general enough to provide a framework for various specification techniques. It can be employed for textual as well as graphical ones. Examples of specification architectures comprise architectures of COMMUNITY programs [17], distributed graph transformation systems [16] as well as specification architectures of algebraic high-level nets [14]. Hence the question of implementation of changes can be attributed to the question of the specification techniques. Obviously a programming language as COMMUNITY is closer to a real implementation than a P/T net. So it is clearly an important and challenging task to ensure compatibility of model transformation with its implementation. But it is dependent on the underlying specification technique.

Ensuring compatibility between model transformation and its implementation is also the main emphasis in the area of model-based software reengineering. In contrast to other approaches on architecture reconstruction like [3,2], we do not focus on the reengineering task and hence we do not consider code transformation here. Rather we focus on assisting architectural design by offering rule-based transformation steps yielding syntactically correct two-level architecture specifications.

For methodological questions it needs several case studies using various specification techniques to extract a specification independent process model. Hence we have not yet concentrated on this question, but merely have distinguished between local, synchronizing and global transformations. Further research can either consider specific specification techniques or concentrate on the general approach. The first case includes all semantic aspects, like consistent changes of behavior, preservation of properties, and as mentioned above compatibility with realization. The second case focuses on structural questions like compatibility with (categorical) structuring techniques, parallel and sequential independence, and so on.

References

1. R. Bardohl. GenGEd – *Visual Definition of Visual Languages based on Algebraic Graph Transformation*. Verlag Dr. Kovac, 2000. PhD thesis, Technical University of Berlin, Dept. of Computer Science, 1999.
2. J. Bosch. Transformation of Software Architecture. In *Design and Use of Software Architectures: Adopting and Evolving a Product-Line Approach*. Chapter 6, Addison Wesley, 2000.
3. S. Carriere, S. Woods, and R. Kazman. Software Architectural Transformation. In *Proc. 6th Working Conference on Reverse Engineering*, IEEE Computer Society, 1999.
4. P. Donohe, editor. *Software Architecture*. Kluwer Academic Publishers, 1999.
5. C. Ermel, R. Bardohl, and H. Ehrig. Generation of Animation Views for Petri Nets in GENGED. In Ehrig et al (eds.), *Advances in Petri Nets: Petri Net Technologies for Modeling Communication Based Systems*, Springer, LNCS, 2002. To appear.
6. H. Fahmy and R. Holt. Using Graph Rewriting to Specify Software Architectural Transformations. In *Proc. of Automated Software Engineering (ASE 2000)*, 2000.
7. H. Giese, J. Graf, and G. Wirtz. Modeling Distributed Software Systems with Object Coordination Nets. In *Proc. Int. Symposium on software Engineering for Parallel and Distributed Systems (PDSE'98),Kyoto, Japan*, pages 107–116, jul. 1998.
8. Holger Giese and Jörg P. Wadsack. Reengineering for Evolution of Distributed Information Systems. In Scott Tilley, editor, *3rd International Workshop on Net-Centric Computing (NCC 2001), May 14, 2001; Toronto, Canada*, May 2001.
9. M. Goedicke, T. Meyer, and G. Taentzer. ViewPoint-oriented Software Development by Distributed Graph Transformation: Towards a Basis for Living with Inconsistencies. In *Proc. 4th IEEE Int. Symposium on Requirements Engineering (RE'99), June 7-11, 1999, University of Limerick, Ireland*. IEEE Computer Society, 1999. ISBN 0-7695-0188-5.
10. M. Große-Rhode, R. Kutsche, and F. Bübl. Concepts for the Evolution of Component-Based Software Systems. Technical Report TR-2000/11, FB Informatik, TU Berlin, 2000.
11. D. Hirsch, P. Inverardi, and U. Montanari. Graph Grammars and Constraint Solving for Software Architecture Styles. In *Proc. ISAW'98*, 1998.
12. D. Hirsch, P. Inverardi, and U. Montanari. Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. In *Proc. Working IFIP Conference on Software Architecture*, 1999.
13. V. Issarny, L. Bellissard, M. Riveill, and A. Zarras. Component-Based Programming of Distributed Applications. In *Distributed Systems*, pages 327–353. Springer-Verlag, LNCS 1752, 2000.
14. J. Padberg. Formal Foundation for Transformations of Specification Architectures. In *Software Evolution Through Transformations: Towards Uniform Support throughout the Software Life-Cycle (SET'02)*, Satellite Event of ICGT'02, 2002.
15. M. Shaw and D. Garlan. *Software Architecture - Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
16. G. Taentzer. Distributed Graphs and Graph Transformation. *Applied Categorical Structures*, 4(4):431–462, December 1999.
17. M. Wermelinger and J. Fiadeiro. A Graph Transformation Approach to Software Architecture Reconfiguration. In H. Ehrig and G. Taentzer, editors, *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems (GRATRA'00)*. TU Berlin, FB Informatik, TR 2000-2, 2000. Accepted to Journal of Science of Computer Programming.

Decomposing Graphs with Symmetries

Michel Bauderon and Frédérique Carrère

Laboratoire Bordelais de Recherche en Informatique
Université Bordeaux I
33405 Talence Cedex
{bauderon,carrere}@labri.fr

Abstract. While decomposing graphs in simpler items greatly helps to design more efficient algorithms, some classes of graphs can not be handled using the classical techniques.
We show here that a graph having enough symmetries can be factored into simpler blocks through a standard morphism and that the inverse process may be formalized as a pullback rewriting system.

1 Introduction

Several approaches have been used to try and make advantage of the structural properties of graphs, in order to devise more efficient algorithms or to prove properties for specific classes of graphs. To cite but a few, let us merely quote modular decomposition [3], 2-structures [4] or systems of recursive equations.

They normally come with a decomposition procedure/algorithm as well as a way to rebuild the graph from its basic blocks or at least to describe in a convenient way (such as a kind of syntax tree) the initial structure in terms of those blocks.

In an earlier paper [1], we have tried to tackle the case of classes of graphs which are not well receptive to the previously cited methods, such as for instance, square grids. Indeed, we have shown how some classes of graphs endowed with a fairly regular structure were suitable for a new treatment based on a pair of quotient/product operations. The basic idea was to use the regularities of the graph structure to define a pair of "complementary" equivalence relations on the graph yielding two quotient graphs whose categorical product would give back the initial graph.

We present here a different approach, which makes use of some more intrinsic "geometrical" properties (although they do not rely on any embedding in a euclidian space) of graphs to provide a new kind of decomposition, namely the existence of symmetries of the graph, which define a decomposition of the graph as the product of a factor which depends of the graph and of a canonical factor. As a consequence, a graph having enough symmetries or local symmetries (those terms are defined below) may be factored through a standard morphism and the inverse process may be formalized as a (pullback) rewriting system [2].

This paper is organized in three sections. After the basics definitions given in section 2, we study symmetries of graphs (section 3) and described the rewriting

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 45–59, 2002.
© Springer-Verlag Berlin Heidelberg 2002

process in section 4. For lack of space, proofs have been omitted. None of them is really difficult and we preferred to give complete examples. Note that all pullback diagrams have been computed by program since computing pullbacks "by hand" is a bit tricky!

2 Definitions

In this paper, we consider undirected, connected graphs, for which we shall use the following definition.

Definition 1. *A* graph *is a pair of finite sets* $G = \langle V, E \rangle$ *where* V *is the set of* vertices *or* nodes *and* E *is the set of* edges*, together with two mapping* $\sigma, \tau : E \to V$*. If* $(\sigma(e), \tau(e)) = (u, v)$*, we say that* e *is an edge between nodes* u *and* v *and write* $e = [u, v]$ *(the order has no meaning). A node* u *is* reflexive *if there is at least one edge* $e = [u, u]$ *in* E*, called a loop. A vertex with a loop is said to be reflexive. A graph is reflexive if all its vertices are.*

Note that this definition allows multiple edges linking the same nodes and even multiple loops. This possibility will be needed in a technical way to define one of the essential ingredients of our computations, the graph A.

In this paper, we shall nevertheless be mainly interested in *simple graphs* i.e. graphs without multiple edges.

We shall use the word *item* to denote indifferently a node or an edge.

Definition 2. *A* graph morphism $h : G \longrightarrow G'$ *is a pair* $h =< h_V, h_E >$ *with* $h_V : V \longrightarrow V$ *and* $h_E : E \longrightarrow E'$ *such that* $h_E([u, v]) = [h_V(u), h_V(v)]$.

It is well known that the good properties of graph morphisms turn the set of graphs into a category that we shall denote by $\mathcal{G}$. Let us briefly enumerate the main properties of this category that we shall need in the sequel.

Proposition 1. *The category* $\mathcal{G}$ *of graphs has arbitrary products and equalizers. The graph with one vertex and one edge (ie one reflexive vertex) is a terminal object simply denoted by* $\odot$*. It is a neutral element for the product. The category* $\mathcal{G}$ *has arbitrary limits (is complete). In particular,* $\mathcal{G}$ *has pullbacks.*

Since we shall use it intensively, let us recall here that the pullback of two graph morphisms $f_i : G_i \longrightarrow F, i = 1, 2$ is a pair of arrows $h_i : H \longrightarrow G_i, i = 1, 2$ where H is the subgraph of the product consisting of exactly those items (nodes and vertices) on which $f_i \circ h_i$ coincide. Many examples will be provided in the rest of this paper.

We shall need in the sequel two special graphs:

- Ξ is the reflexive graph with:
  - vertices: a, b, c, d, e
  - edges: $e_1 = [a, b], e_2 = [a, e], e_3 = [b, e], e_4 = [c, e], e_5 = [b, c], e_6 = [b, d]$, $e_7 = [a, a], e_8 = [b, b], e_9 = [c, c], e_{10} = [d, d], e_{11} = [e, e], e_{12} = [c, d]$

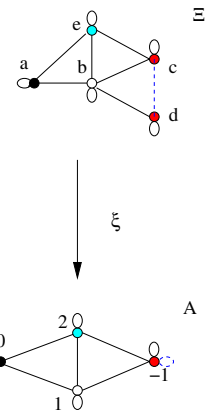


Fig. 1. The graph morphism $\xi : \Xi \longrightarrow A$

– A is the graph with
  - vertices: 0, 1, 2, −1 and
  - edges: $e_1 = [0,1], e_2 = [0,2], e_3 = [1,2], e_4 = [2,-1], e_5 = [1,-1]$, $e_6 = [0,0], e_7 = [1,1], e_8 = [2,2], e_9 = [-1,-1], e_{10} = [-1,-1]$,

and a graph morphism $\xi : \Xi \longrightarrow A$ defined

– on nodes by $\xi(c) = \xi(d) = -1, \xi(a) = 0, \xi(b) = 1, \xi(e) = 2$
– on edges by
  - $\xi(e_1) = e_1$,
  - $\xi(e_2) = e_2$,
  - $\xi(e_3) = e_3$,
  - $\xi(e_4) = e_4$,
  - $\xi(e_5) = \xi(e_6) = e_5$,
  - $\xi(e_9) = \xi(e_{10}) = e_9$,
  - $\xi(e_{12}) = e_{10}$,
  - all loops have obvious projections.

These graphs together with the morphism ξ are depicted on figure 1 and will appear in the right hand column of all the pullback diagrams in the sequel (note that A is not simple: it has two loops on node −1, which allow for a different projection for the edge $[c,d]$ than for the loops on c and d).

As far as possible, we have tried to rely on the reader's intuition when drawing our morphisms: in this case, ξ may be understood as respecting the vertical projection.

3 Graph with Symmetries

3.1 Automorphisms

A graph automorphism is a graph endomorphism which is one to one on both vertices and edges. The set of automorphisms of a graph G is a group under the composition of morphisms. The neutral element is the identity morphism Id.

Let $e \in G$ be any item (vertex or edge) of G. The orbit of e under φ is the (finite) set $o(e) = \{\varphi^n(e),\ n \geq 0\}$. Let us set $e \simeq e' \iff e' \in o(e)$. This defines a congruence on G.

Lemma 1. *$\overline{G} = G/\simeq$ is a graph.*

We let $\overline{\varphi}$ denote the canonical projection $G \longrightarrow \overline{G}$ and we call it a *folding of* G along φ. Examples will be given later.

In the sequel, we shall concentrate on a very simple type of morphisms, and shall in general denote the image $\varphi(u)$ of u by u'. Automorphisms satisfying other types of properties (such as $\varphi^n = Id$ for $n \geq 3$) will be studied in further works.

3.2 Symmetries

Definition 3. *A symmetry is an involution, ie a graph automorphism φ such that $\varphi^2 = Id$.*

Clearly, if φ is a symmetry then for each orbit, $\#o(e) \leq 2$.

Example 1. Let us first consider a square grid G_n of size n. We shall discuss the situation without giving formal definitions of corners, diagonal and median axis or other notions which we hope to be intuitive enough (the same remark holds for the next example). We do not believe either that a picture is necessary.

It is clear from classical geometry (eg by representing the nodes of G_n as the set of points $(i, j), 1 \leq i, j \leq n$ in the euclidian plane) that it has at least six symmetries (the identity, a central symmetry, two symmetries w.r.t. the diagonals of the square and two w.r.t. the median axis of the square).

Now, let a be one of its 4 corners (i.e. its nodes of degree 2), b the opposite corner, c and d the two other corners. Under any automorphism, hence under any symmetry φ, the image of a has to be one of the four corners.

If $a \to b$, c and d may be invariant. Then it is easy to see that φ is the symmetry with axis the diagonal "linking" c to d (a set of vertices which turns out to be a cut set of the grid). Otherwise, ie if $c \to d$, then one can show that φ is the central symmetry (which has an invariant node iff n is odd).

If $a \to c$, then necessarily $b \to d$, and we find the symmetry whose axis is the median axis, "orthogonal" to ac (note that this axis may be a set of vertices or a set of edges, depending on the parity of n). A similar conclusion holds if $a \to d$.

Last, if a is invariant, at least another corner must be, and one may show that b is. This leaves two possibilities: either c and d are invariant and φ is the identity or $c \to d$, and one gets the symmetry with respect to the axis ab.

Since all the possibilities have been considered, this shows that there are, as expected, exactly six symmetries.

In this example, we found that the symmetries of a square grid considered as a graph are exactly those which appear if we consider this graph as embedded in a natural way in the euclidian space. They are of two types: axial symmetries and central symmetries. The first ones have an invariant axis (a set of invariant

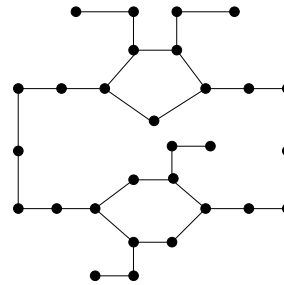


Fig. 2. A graph with a complex symmetry

nodes or edges which cut the graph in two isomorphic parts), the second ones have at most one invariant node which is the center of the symmetry.

This is often the case, but the intuition coming from euclidian geometry is not true in general: symmetries of a graph may be more complex since they can combine both types.

Example 2. This is easily seen on figure 2. The graph has two cycles, each of which has two "antennas". It is easy to check that it has only one symmetry, which has to be "axial" on the upper cycle and "central" on the lower one.

This general situation is described by the next lemma.

A *path* P in G is a sequence $\{u_1, ..., u_n\}$ of nodes such that for each $1 \leq i \leq n$, $[u_i, u_{i+1}]$ is an edge in G. We say that P is *stable* if for each node u of P, u' belongs to P as well.

Let u be a node of G and $u' = \varphi(u)$ be its image under φ and let $P = \{u, u_1, ..., u_n, u'\}$ be a path from u to u' (there exist always one, since the graph is connected).

Lemma 2. *If there exists a node v of P such that either $v = v'$ or $[v, v']$ is an edge of P, then, one can find a path P' from u' to v' such that $P' = \varphi(P)$.*

Otherwise, let $f = u_i$ a node of P such that $f' = u'_i$ belongs to P but u'_{i+1} does not. Then there exists a path P^* from u_i to u'_i such that the cycle obtained from P^* concatenated with the restriction of P to $\{u_i, ..., u'_i\}$ is stable, but has no invariant. Moreover, there exists an other path P^{**} from u'_i to u' such that the concatenation of P^* and P^{**} is exactly the image under φ of the restriction of P to $\{u, ..., u'_i\}$.

In other words, any path linking a node to its image either can be made symmetric or has to cross a stable cycle with no invariant. These two cases are illustrated in example 2 respectively in the upper and lower cycles.

The edges linking P' to f and their images under φ (as described in the lemma) will be said to *cut the cycle* and will be called *cut edges.*

Lemma 3. *Let G be a graph and φ be a symmetry of G. There exists a set C of pair of edges of the form $(e, \varphi(e))$ such that each stable cycle with no invariant is cut exactly once by a pair in C.*

This set is not unique, since the choice of cut edges is to some extent arbitrary.

For the sake of simplicity, a node v such that $v = \varphi(v)$, an edge linking two invariant nodes or an edge of the form $[v, \varphi(v)]$ will all be called *invariant* under φ. We let $Cut(\varphi)$ denote the union of the (uniquely defined) set $Inv(\varphi)$ of invariant items of the graph (containing the subgraph induced by the set of invariant nodes under the action of φ, and all edges of the form $[v, \varphi(v)]$) and of a set C of cut edges as defined in the previous lemma.

Lemma 4. *The subgraph G' of G obtained by removing $Cut(\varphi)$, has two connected components G_1 and G_2, such that $G_2 = \varphi(G_1)$ (and conversely).*

By collapsing each component G_i to a different reflexive node, we get the following:

Proposition 2. *Let G be a graph and φ be a symmetry of G. Then, there exists a morphism $\psi : G \longrightarrow \Xi$ sending G_1 to c, G_2 to d and all cut nodes to b.*

The quotient graph $\overline{G}$ defined in the previous section can be further analyzed to define a new morphism $\widetilde{\varphi}$, which classifies items in G in terms of their behavior under the action of φ, ie which distinguishes "normal" vertices, invariant vertices, vertices linked by an invariant edge and path separated by a cut edge.

Let $\widetilde{\varphi} : \overline{G} \longrightarrow A$ be the unique graph morphism defined by

- $\widetilde{\varphi}(v) = -1$ if $\#o(v) = 2$,
- $\widetilde{\varphi}(v) = 1$ if $\#o(v) = 1$,
- $\widetilde{\varphi}(e) = e_9 = [-1, -1]$ if e is a cut edge,
- $\widetilde{\varphi}(e) = e_{10} = [-1, -1]$ if e is an invariant edge

The main result of this section is that the previous constructions of morphisms define both a way to analyze and synthesize the original graph G.

Theorem 1. *Let G be a graph and φ be a symmetry of G. The pair of arrows $\overline{G} \xleftarrow{\overline{\varphi}} G \xrightarrow{\psi} \Xi$ is the pullback of the pair $\overline{G} \xrightarrow{\widetilde{\varphi}} A \xleftarrow{\xi} \Xi$ in the category of graphs.* □

In other words, as was announced in the introduction, the existence of a symmetry φ on a graph G, gives a decomposition of G into two factors, one depending on G ($\widetilde{\varphi}$) and a generic one (ξ) which depends only on the nature of the transformation (the fact that $\varphi^2 = Id$). Then, G may be computed as a pullback, or even, if we work in the category of graphs over A, as a simple categorical product.

Example 3. Figure 3 shows a case where most difficulties appear, by providing a decomposition of the graph shown in figure 2. The folding of the graphs shows (left hand side column): an invariant edge, collapsed to a loop and an invariant node on the upper cycle, two cut edges linking u to v' and u' to v, which both collapse to the same edge linking the image of u and u' to that of v and v'.

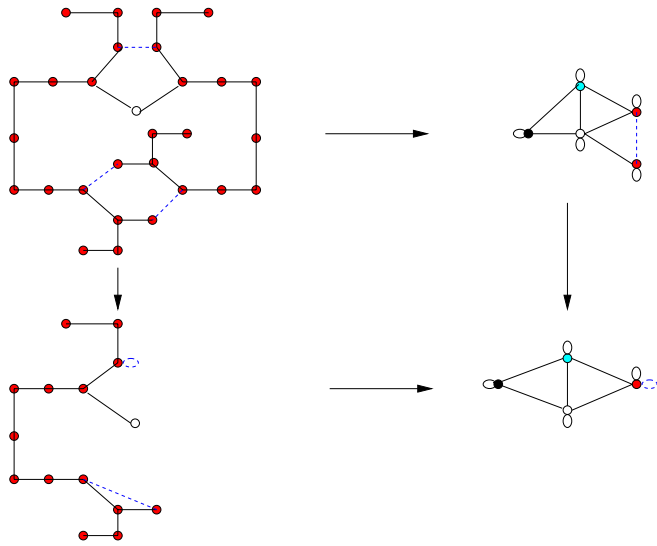


Fig. 3. Decomposition by symmetry

Example 4. An other example may be given with G_n, the square grid of size n. As already stated, such a graph has three kind of symmetries:

- with respect to the diagonals: these two symmetries have a set of invariant nodes which are cut sets of the graph
- with respect to the medians: these two symmetries have either a cut set or a set of invariant edges, depending on whether n is odd or even.
- with respect to the center of the grid, which may be a node.

Figure 4 shows a decomposition w.r.t. to one of the diagonal symmetries, while figure 5 shows a decomposition w.r.t. the central symmetry. The slightly irregular shape of the grid and half grid in figure 4 is intended to insist upon the fact that we do not rely on any euclidian embedding: our grid would not be symmetric!

The reader has probably noticed that nodes a and e of graph Ξ as well as nodes 0 and 2 of graph A did not play any role in the previous decomposition. They will only be useful in the next section, but including them from the beginning allows us to offer a more uniform treatment.

3.3 Local Symmetries

Lemma 5. *L et φ and ψ be two symmetries of a graph G and assume that φ and ψ commute, ie $\varphi \circ \psi = \psi \circ \varphi$. Then, ψ defines a symmetry $\overline{\psi}$ of the quotient graph G/φ.*

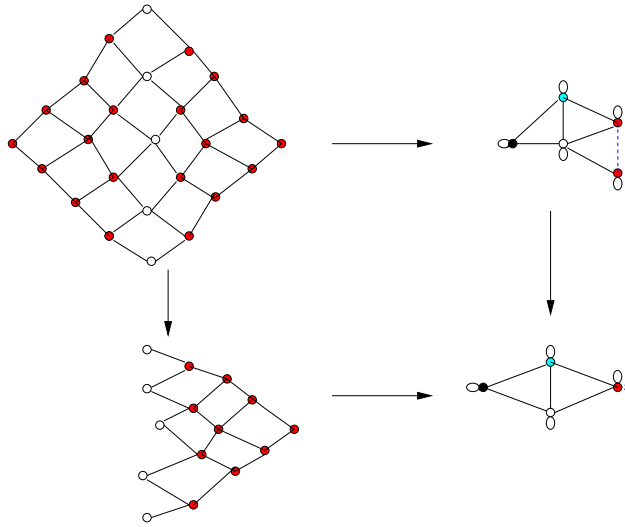


Fig. 4. Decomposing G_5 using a diagonal symmetry

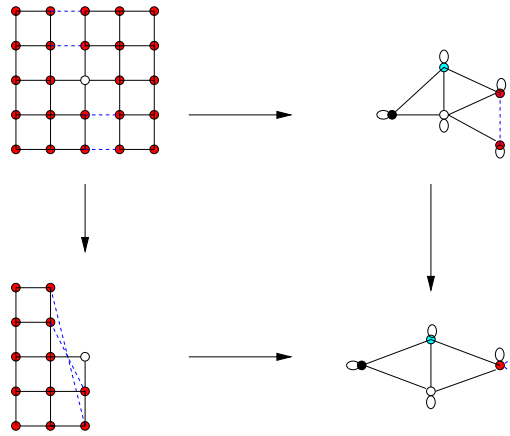


Fig. 5. Decomposing G_5 using the central symmetry

Example 5. Let G_n be the square grid of size n. Then the two diagonals define commuting symmetries φ_1 and φ_2 (the two medians define two other commuting symmetries). It is easily seen that applying first φ_1 to get a quotient graph G_1 and then $\overline{\varphi_2}$ to G_1 defines a two level decomposition of the initial grid.

A set of commuting symmetries of a graph can provide by successive folding a first kind of decomposition of a graph. Unfortunately, as is clearly shown by our running example of square grids, this does not go very far and a new analysis of the reduced graph is necessary if we want to reduce the graph to significantly smaller blocks: in our example of grids, using two commuting symmetries reduces

the grid to something like one fourth of its original size. But then new symmetries appear and must be investigated. In the sequel we will not consider this approach, which would lead to slightly different developments.

A more general approach seems more fruitful, by considering *local symmetries* of graphs, that we shall define now.

Definition 4. *Let G be a graph. We say that an endomorphism φ of G is a local symmetry, if there exits a partial subgraph $C(\varphi)$ separating G into two components G_1 and G_2 (not necessarily connected) such that φ is the identity on $G_2 \cup C(\varphi)$ and $\varphi(G_1)$ is isomorphic to a subgraph of G_2.*

In other words, φ restricted to $G_1 \cup C(\varphi) \cup \varphi(G_1)$ gives rise to a symmetry ϕ which keeps invariant $C(\varphi)$ and exchanges G_1 and $\varphi(G_1)$. We shall say that G_1 is the domain of φ. It is easily seen that inclusion of domains defines a partial order on the set of partial symmetries. Let us say that a local symmetry is maximal if it is a maximal element for that order.

We can then follow a line similar to the previous section on symmetries. The local symmetry φ defines an equivalence relation on G and therefore a canonical projection $\overline{\varphi} : G \longrightarrow \overline{G}$. We can also define a morphism $\psi : G \longrightarrow \Xi$, such that the pair of arrows $\overline{G} \xleftarrow{\overline{\varphi}} G \xrightarrow{\psi} \Xi$ is the pullback of the pair of arrows $\overline{G} \xrightarrow{\widetilde{\varphi}} A \xleftarrow{\xi} \Xi$, yielding a result similar to that of theorem 1. The main difference is that the morphism ψ distinguishes now more components: G_1 and $\varphi(G_1)$ sent respectively to nodes c and d, neighbors of G_1 and $\varphi(G_1)$ sent to node e and other nodes sent to a, making the whole of graphs Ξ and A necessary. This is illustrated in the examples.

Example 6. Let us consider again the case of the square grid G_5 and apply first a symmetry w.r.t. to a diagonal yielding a half-grid (as described in figure 4). Then one may find several local symmetries by considering cuts of $\overline{G_5}$ parallel to the symmetry axis. Let φ be the maximal local symmetry of $\overline{G_5}$. It is shown on figure 6, where the half grid appears on the top left corner. A dotted box indicates the cut used for the local symmetry and a dotted arrow shows the folding which yields the graph in the bottom left corner. It is easily checked that the diagram shown on the figure is actually a pullback.

4 Reconstruction of the Graph

4.1 Pullback Rewriting

Let us recall here the basics of pullback rewriting as defined for instance in [2].

Definition 5. *A* production rule *is a graph morphism $p : R \longrightarrow A$, where A is called the left-hand side of the rule, while R is the right-hand side. An* occurrence *of the left-hand side of p in the graph G is a morphism $x : G \longrightarrow A$. The* rewriting *of G by p at occurrence x is the pullback of x and p in the category of graphs.*

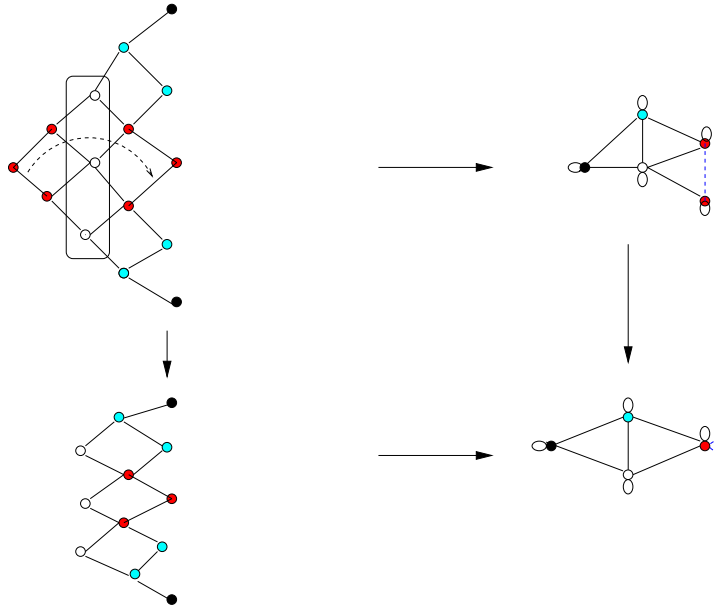


Fig. 6. A local symmetry for the half grid of size 5

The graph A of this definition plays the role of an alphabet in a classical labelling system, to specify the behavior of nodes of the graph G under a rewriting. The main difference is that the labelling mapping is now a graph morphism, which imposes strong constraints on all the possible labellings of a graph.

Actually, the alphabet graph has a very special form, allowing to distinguish between the nodes to be rewritten (called the *unknowns*), those which will not be affected (called *the context*) and those which lie in the zone between unknowns and context (called the *interface nodes* or *neighbors*).

The graph A defined in section 2 is an alphabet with one unknown (node -1), two neighbors (nodes 1 and 2) and a context node (node 0).

It is well known that, in the classical situation of rewriting systems on words, the application of a rule to a word provides an automatic relabelling of the result, defining the occurrences of the unknowns in the resulting term, which allows to proceed to the application of any (applicable) rule to this result. This relabelling comes partly from the rules itself and partly from the initial labelling of the word.

Similarly, for each rewritten graph and each rewriting rule, we must specify the way the result will be relabelled, ie the way to build the new occurrence in the resulting graphs.

For that purpose, a rewriting rule will be defined by two components: the rule itself and a mechanism which specifies the way each node will be labelled in the resulting graph.

In a similar way, an occurrence of a rule will be defined by two components: the occurrence itself and a mechanism which specifies the behavior of the nodes which the occurrence considers as context nodes.

4.2 Rewriting as a Generating System

Results of section 3 may be interpreted in terms of rewriting systems since each decomposition step (as described in theorem 1) was done through a pullback diagram which can be interpreted as well as a reconstruction step.

In fact, only three rules were used in the decomposition, all relying on $\xi : \Xi \longrightarrow A$. They differ only by the relabelling morphism:

1. $R_1 = (\xi, \zeta_1)$ (where $\zeta_1 : \Xi \to \mathcal{A}$ is the morphism defined by $\{a \to 0, b \to 0, c \to 0, d \to 0, e \to 0\}$) is the terminal rule: it handles (local) symmetries by duplicating all nodes labelled by -1 (since Ξ as two such items) and leaving nodes labelled by 1 invariant. It then relabels everything as 0.
2. $R_2 = (\xi, \zeta_2)$ (where $= \zeta_2 : \Xi \to \mathcal{A}$ is the morphism defined by $\{a \to 0, b \to -1, c \to 1, d \to 2, e \to 2\}$) handles (local) symmetries in general.
3. $R_3 = (\xi, = \zeta_3)$ (where $= \zeta_3 : \Xi \to \mathcal{A}$ is the morphism defined by $\{a \to 0, b \to -1, c \to 1, d \to -1, e \to -1\}$) is necessary to handle the last local symmetry before applying a global symmetry R_1.

Note that R_1 and R_2 differ only in that R_1 relabels everything as 0, stopping any possibility of further rewriting. In a way, R_1 is the terminal version of $R2$.

Now if G is a graph, an occurrence on G will be a pair $x = (u, \psi)$ where $u : G \to A$ is the occurrence itself (the morphism which will be used for computing the pullback) and ψ is a family of morphisms $\psi_v : G \to A$, for each $v \in u^{-1}(0)$ which defines the relabelling mechanism on the context part of the graph.

Then the application of rule R_i to G will be simply done by computing the pullback of u and R_i in the category of graphs, i.e. the pair of arrows $g : G' \to G$ and $h : G' \to \Xi$.

The relabelling of the resulting graph G' will be done in the following way:

- the new occurrence G' given by R_i will be $\zeta_i \circ h$ on all nodes,
- except for context nodes whose behavior is defined by $\psi : \psi_v$ is applied if v becomes a neighbor in the rewritten graph.

It is easily checked that:

Lemma 6. *The relabelling (extended in the canonical way to edges) is a well defined graph morphism which may be interpreted as an occurrence on G'.*

And we may state:

Theorem 2. *If G is decomposed via a succession $\varphi_1, \varphi_2, ...\varphi_n$ of a starting (global) symmetry and of $n-1$ local symmetry, it may be reconstructed by applying a sequence of rewriting rules:*

- *for a local symmetry following a global one, apply R_3*
- *for all other local symmetries, apply R_2.*
- *for the starting (local) symmetry from the initial graph, apply R_1 (which is yields a terminal graph),*

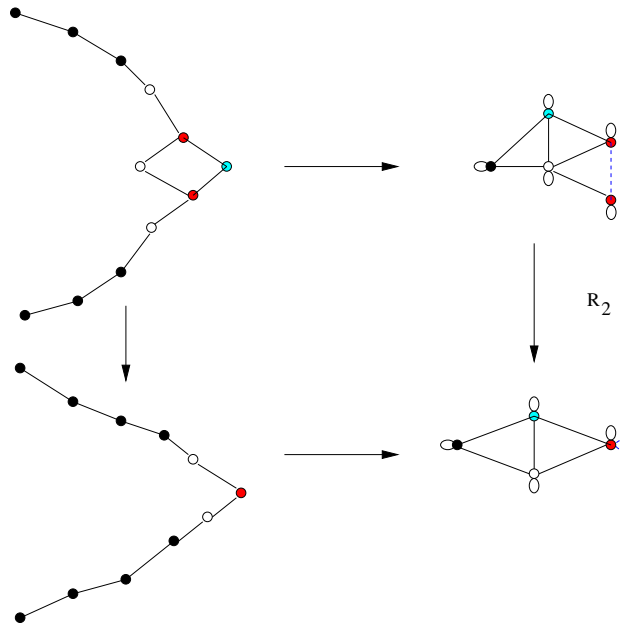


Fig. 7. Step 1

We illustrate this result in the following example.

Example 7. Let G_6 be the square grid of size 5 and P_{11} be the path of length 11, with nodes $v_1, v_2, ..., v_{11}$ in that order. Rather than running backwards the example described in the previous sections (the reader may check that this could as well be done, simply by choosing a different initial occurrence in the first step of the computation), we choose the occurrence of R_2 on P_{11} which is described by the morphism π_{11} in the bottom of figure 7: $v_5, v_7 \to 1, v_6 \to -1$, all other nodes 0. The relabelling morphisms ψ_v states that $\psi_v(v) = 1$ when v becomes a neighbor, all other context nodes remaining untouched.

For simplicity, we shall identify P_{11} and π_{11}. Application of rule R_2 to P_{11} is shown on that same figure. The relabelling computed after applying the rule is shown on the resulting graph.

Two more applications of R_2 are shown on the next two figures 8 and 9, reconstructing stepwise the grid by reverse application of local symmetries.

Two last steps are then necessary. First, an application of R_3 as shown on figure 10 which does the same thing as R_2, but prepares a new relabelling which will allow to correctly apply rule R_1 which has to take care of the final symmetry. The computation is shown on figure 11 This shows that: $G_5 = R_1 \circ R_3 \circ R_2^3(P_{11})$. More generally, one could show that for $n \geq 3$, $G_n = R_1 \circ R_3 \circ R_2^{n-3}(P_{2n-1})$.

As already mentioned, the same computation could have started from a different occurrence on P_{11}. Yet another occurrence, (with a different relabelling mechanism) would lead for instance to the generation of other regular patterns

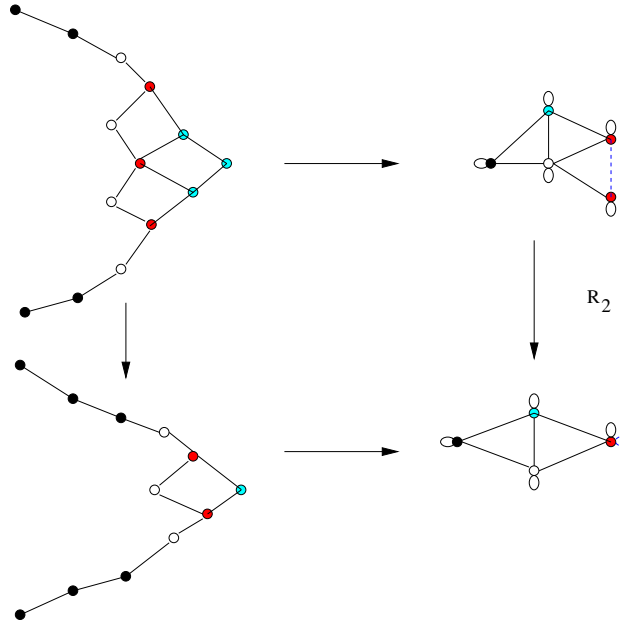


Fig. 8. Step 2

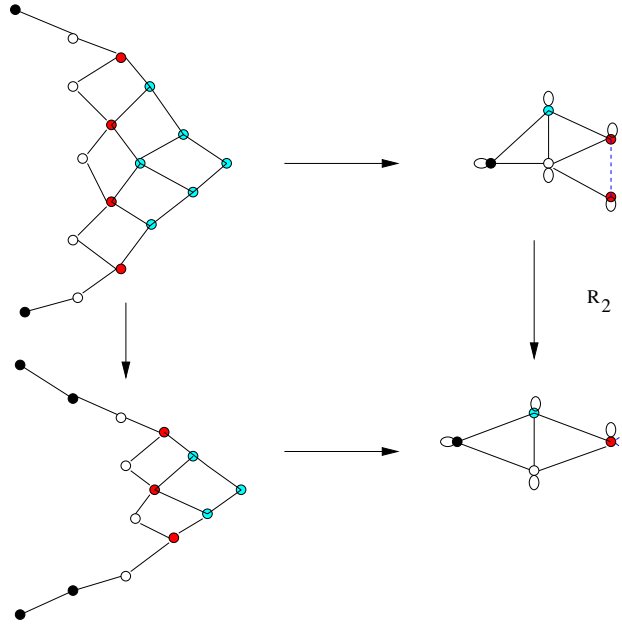


Fig. 9. Step 3

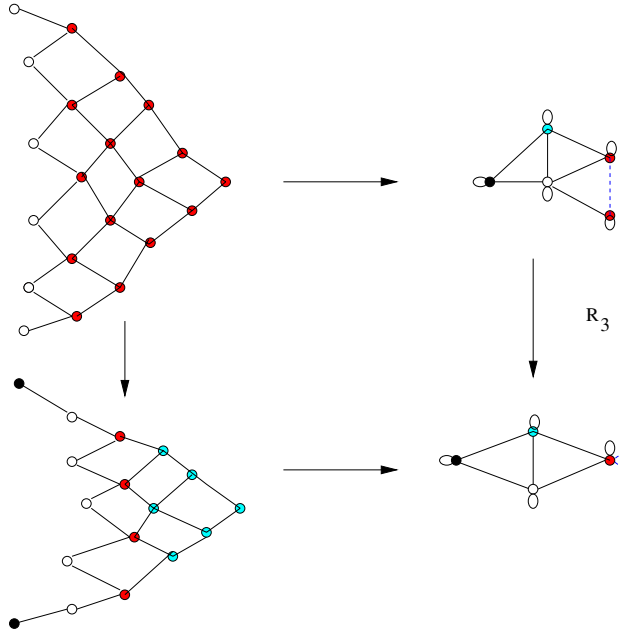


Fig. 10. Step 4

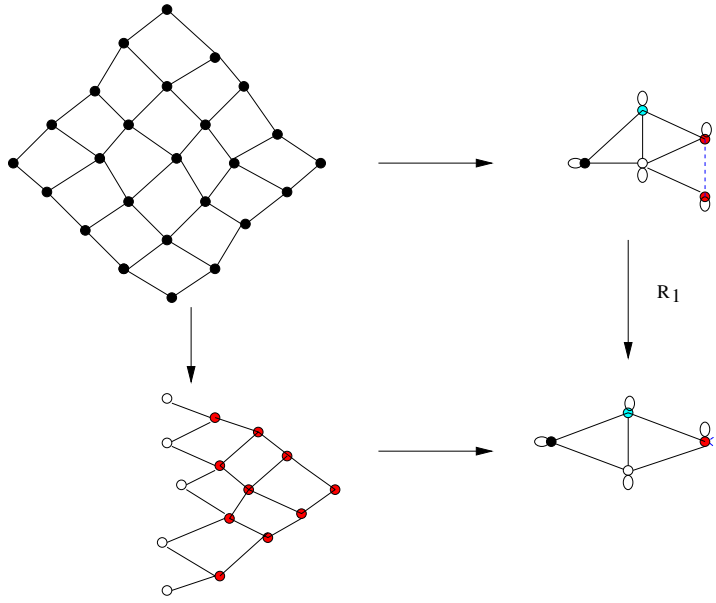


Fig. 11. Step 5

such as hexagonal lattices (we don't have space here to develop that). A classification of all those possibilities remains to be done.

5 Conclusion

In this paper, we have shown how symmetries and local symmetries give rise to a natural factorization of a graph, and provide a way to decompose into simpler items graphs having a sufficiently regular structure. An inverse process is described through a rewriting system to reconstruct the initial graph.

The geometrical properties we have used here are but a very simple type of properties of a graph that may be described through the existence of automorphisms of the graph and which we believe are worth exploring.

References

1. Bauderon M., Carrère F., Orthogonal decomposition of graphs, GRATRA 2000, Joint APPLIGRAPH and GETGRATS Workshop, Technische Universität Berlin, 2000, 2, pp 197-204
2. Bauderon M., Jacquet H., Categorical product as a generic graph rewriting mechanism, Journ. Applied Categorical Structures, 2001, Volume 9, Issue 1, January 2001, pp. 65-82
3. Cournier A., Habib M., A new linear algorithm for modular decomposition, Lect. Notes in Comp. Sci. 787, 1994, 68-84
4. Ehrenfeucht A., Harju T., Rozenberg G., The theory of 2-structures. A framework for decomposition and transformation of graphs. World Scientific Publishing, 1999.

Graph Transformations for the Vehicle Routing and Job Shop Scheduling Problems*

J. Christopher Beck[1], Patrick Prosser[2], and Evgeny Selensky[2]

[1] Cork Constraint Computation Center, University College Cork, Ireland
c.beck@4c.ucc.ie
[2] Department of Computing Science, University of Glasgow, Scotland
{pat,evgeny}@dcs.gla.ac.uk

Abstract. The vehicle routing problem (VRP) and job shop scheduling problem (JSP) are two common combinatorial problems that can be naturally represented as graphs. A core component of solving each problem can be modeled as finding a minimum cost Hamiltonian path in a complete weighted graph. The graphs extracted from VRPs and JSPs have different characteristics however, notably in the ratio of edge weight to node weight. Our long term research question is to determine the extent to which such graph characteristics impact the performance of algorithms commonly applied to VRPs and JSPs. As a preliminary step, in this paper we investigate five transformations for complete weighted graphs that preserve the cost of Hamiltonian paths. These transformations are based on increasing node weights while reducing edge weights or the inverse. We demonstrate how the transformations affect the ratio of edge to node weight and how they change the relative weights of edges at a node. Finally, we conjecture how the different transformations will impact the performance of existing VRP and JSP solving techniques.

1 Introduction

The vehicle routing problem (VRP) and the job shop scheduling problem (JSP) are two common combinatorial problems that have natural graphical representations. In fact, these two problems have a number of similarities both in their graphical representations and on a conceptual level. While the structure of the graphs is largely similar (and is, indeed, identical for core subproblems), the graph characteristics tend to be different. In particular, the difference that forms the focus of the work in this paper is that graphs representing VRP problems tend to have very high edge weights relative to node weights while JSP graphs are the reverse.

It is our conjecture that we can develop a deeper understanding of the models and algorithms for VRPs and JSPs by examining and transforming the characteristics of the underlying graph representations and by experimenting with the transformed problems. In this paper, we present five graph transformations each of which is based on increasing node weights while decreasing edge weights or

* This work was supported by EPSRC research grant GR/M90641 and ILOG SA.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 60–74, 2002.
© Springer-Verlag Berlin Heidelberg 2002

the inverse. From the perspective of VRPs and JSPs, the transformations that increase node weights relative to edge weights will make the graph look more like a JSP problem than a VRP. Similarly, the inverse transformation will give the problem characteristics that are more typical of VRPs. Our long term research direction, therefore, is to identify interesting problem reformulations (perhaps based on graph transformations), apply them to VRP and JSP problems, and empirically analyze the performance of existing VRP and JSP techniques.

Previous work provides a basis for our intuitions and forms a preliminary analysis. In Beck et al. [2] the three order dependent transformations presented below were applied to VRP problems. It was shown that:

- a guided local search based VRP solving technique significantly outperforms a constructive, constraint programming based scheduling technique when applied to VRP problems;
- the VRP solving technique sometimes performs worse when the problems are transformed by the order dependent transformations than it performs on the non-transformed problems;
- the JSP solving technique performs marginally better on the transformed problems than on the non-transformed problems.

While this work is interesting, no systematic analysis of the changes in the graph characteristics induced by the transformation techniques was performed. In addition, neither the direct order independent nor inverse transformations presented below were tested. In this paper, we present the graph transformations in a more general framework and perform a systematic analysis of their impact on complete weighted graphs. Each of these transformations preserves the cost of Hamiltonian paths. We are interested in the extent to which the transformations change the ratio of edge weights to node weights and in the extent to which the relative weights of edges are changed. It is our intuition, based on preliminary empirical evidence, that these two characteristics have an impact on VRP and JSP solving techniques.

In the following section, we define VRPs and JSPs in more depth together with their standard graphical representations. We also discuss some similarities and differences between these problems which provide motivation for our interest in graph transformations. We then present the graph transformations addressed in this paper followed by an empirical study of the impact of the transformations on characteristics of the graphs. We conclude by making some conjectures regarding how the use of the transformations may affect the performance of existing VRP and JSP solving techniques.

2 Vehicle Routing and Job Shop Scheduling

In the *delivery* variant of the *vehicle routing problem* (VRP), m identical vehicles initially located at the depot are to deliver discrete quantities of goods to n customers. The locations of the depot and customers are known. Each customer has a discrete demand for goods and each vehicle has a discrete capacity, i.e., it

can carry quantities less than or equal to its capacity. A vehicle can make only one tour starting at the depot, visiting a subset of customers and returning to the depot. It is also assumed that travel time equals travel distance computed using the Euclidean metric. A solution consists of a set of tours for a subset of vehicles such that (i) all customers are served only once, (ii) the total distance traveled by the fleet is minimal. The problem is NP-hard [7].

An $n \times m$ job shop scheduling problem (JSP) consists of n jobs and m resources. Each job consists of a set of m completely ordered activities, where each activity has a duration for which it must execute and a resource which it must execute on. The complete ordering defines a set of precedence constraints, meaning that no activity can begin execution until the activity that immediately precedes it in the complete ordering has finished execution. Each of the m activities in a single job requires exclusive use of one of the m resources defined in the problem. No activities that require the same resource can overlap in their execution and once an activity is started it must be executed for its entire duration (i.e., no pre-emption is allowed). The job shop scheduling decision problem is to decide if all activities can be scheduled, given for each job a release date of 0 and a due date of the desired makespan, D, while respecting the resource and precedence constraints. The job shop scheduling decision problem is NP-complete [7].

While not part of the basic JSP definition, transition times and resource alternatives have been studied in the scheduling community [6]. In the case of a transition time, there is an additional temporal constraint specifying a minimum time interval that must expire between any pair of activities executed on the same resource. The time interval may differ for different pairs of activities. When alternative resources are represented, rather than having a single resource to execute on, each activity has a set of resources from which the one to execute on must be chosen.

2.1 Graphical Representations

In a graphical representation of a VRP, each node corresponds to a visit. The duration of the visit is the weight of the node. Edges represent the travel between visits, with the weight of the edges representing the time required for the travel. A solution is a set of cycles each of which begins and ends at the distinguished depot node. Aside from the depot node, each node appears in exactly one cycle. An optimal solution is one in which the total weight of the cycles (node weights plus edge weights) is minimized.

A JSP problem can also be represented as a graph. Each activity is represented by a node with the duration of the activity being the node weight. There are two types of edges:

- directed, conjunctive edges represent a precedence constraint specifying that one activity must precede another. Transition time is represented as the weight of the edge.
- undirected, disjunctive edges represent the fact that the activity pair must be ordered but that the specific ordering is not predefined in the problem definition.

A solution is the transformation of each disjunctive edge into a conjunctive edge such that the longest path in the graph is less than or equal to D.

We observe that the core problem of finding a path or cycle in a graph exists in both VRP and JSP. In a VRP we must find a set of cycles while in a JSP we need to find a set of paths, one for each resource, by transforming disjunctive edges into conjunctive edges. In this paper, we will concentrate on the subproblem of finding a single minimum cost Hamiltonian cycle in a complete weighted graph. This is equivalent to a VRP problem with a single vehicle and to a JSP problem with a single resource.

2.2 Motivation: VRP and JSP Similarities

As well as similar graph representations, on a conceptual level, vehicle routing problems and scheduling problems are similar. Consider the following:

- They both involve the execution of tasks (activities in the factory and visits in the VRP).
- A task can only be completed through the use of one or more resources (tools and machines on the shop floor, drivers and vehicles in the VRP).
- Resources are often constrained by capacity that specifies, for example, the number of tasks that can be processed.
- Often there is a set of alternative resources to choose from (similar machines on the shop floor, a fleet of vehicles in the VRP).
- When a task is completed there may be some interval of time that must expire before the resource can be used for another task, and this interval may depend on the pair of consecutive tasks (a transition time or set up cost on a machine in the factory, a travel distance between visits in the VRP).
- There may be further temporal constraints among the tasks, specifying time windows when they can and cannot be executed and/or specifying a necessary relationship between tasks (e.g., task B must not be executed until after task A is finished).
- The problem is solved when the resources have been assigned to tasks and the tasks have been assigned start times or ordered such that all temporal and capacity constraints are respected.
- There is an optimization criterion involving the minimization of various definitions of the length of time necessary to complete all the activities. For example, one may seek to minimize the sum of all transition times.

With these similarities, one might expect that similar technology is used to solve vehicle routing and scheduling problems. But this isn't the case. For example, local search techniques (tabu search, simulated annealing, guided local search) [8,10,17] are more popular for vehicle routing problems whereas complete and quasi-complete search techniques [9] are used more in scheduling. As a single data point, ILOG markets two commercial C++ libraries (ILOG Scheduler and Dispatcher) for solving scheduling and vehicle routing problems respectively. Though both products are built on constraint programming technology, the core

technology in the scheduling product is global constraint propagation [11,12] while the core technology in the vehicle routing product is local search [5].

On the other hand, one can see the following important differences between the two problems:

- Transition time *vs* activity duration - In scheduling problems the duration of an activity is typically much larger than the transition time. This is not the case in the VRP, where, in fact, travel is many times the visit duration. Furthermore, much of the research in scheduling has focused on techniques that reason directly about the durations of activities while putting less emphasis on the transition time. The state of the art in constraint-based scheduling is based to a large extent on global constraint propagation [12,13] and heuristic search techniques [15,1] that do not directly take into account transition time. Therefore, we expect that scheduling problems where the transition time appears to be a key aspect of finding a solution, will be difficult for current scheduling technology [14].
- Alternative resources - VRPs have many more alternative resources than are typically studied in scheduling problems. Although the vehicles might not all be used, a common benchmark set [16] starts with 20 or more vehicles. In contrast, existing scheduling research on resource alternatives typically has many fewer alternatives. For example, Focacci et al. [6] experimented on problems with up to 3 alternatives while Davenport & Beck [4] used problem instances of up to 8 alternatives.
- Complex temporal relationships - Scheduling problems tend to have more complex temporal relationships among activities. As we have seen, precedence constraints among the activities in a job are part of the JSP. More complex metric temporal constraints have been widely explored [3] and strong global constraint propagation techniques that take such relationships into account exist [11]. The standard VRP has no temporal constraints.

What are the characteristics of vehicle routing problems that make them more amenable to local search techniques as opposed to construction methods? What properties of scheduling problems make them more suitable to systematic search and powerful constraint propagation?

In this paper, we focus on the issue of transition time *vs* duration. As noted above, previous work [2] supports the intuition that VRP and JSP techniques perform differently on VRP problems that have been transformed to increase node weight than they do on the original VRP problems. As a first step toward a systematic understanding of the reasons for these performance differences, in this paper we investigate how the transformations affect the graph characteristics.

3 Cost-Preserving Transformations of Complete Undirected Graphs

We are interested in transformations that preserve the cost of any solution on the graph. In the following sections, for simplicity, a solution is assumed to be

a cycle on the graph (i.e., we consider traveling salesman problem graphs; when presenting a transformation we then show how it carries over to the case of Hamiltonian paths). The cost of a solution is assumed to be the sum of the weights of the nodes and edges in the respective cycle.

First, we present transformations that reduce the edge weights on a graph by adding a portion of them to the node weights. We refer to these transformations as *direct*. There also exists an *inverse* transformation which similarly reduces the node weights and increases the edge weights.

3.1 Direct Transformations

Order Dependent Transformation. Consider a traveling salesman problem where nodes have weights as well as edges[1]. In a solution we must visit each node once and once only. The cost of visiting a node is then the weight of the edge entering the node plus the weight of the edge exiting the node, plus the weight of the node itself. We can transform this cost such that the node weight is increased and the weights on the entering and exiting edges are reduced.

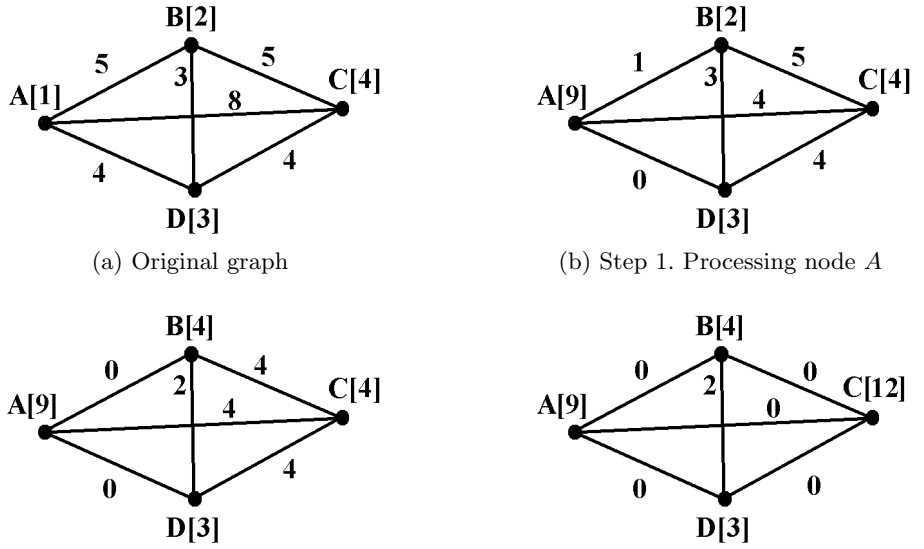
Consider node j, with entering edge (i, j) and exiting edge (j, k), with weights w_j, w_{ij} and w_{jk} respectively. Let $w_{min} = min(w_{ij}, w_{jk})$. We can reduce the weight of both edges by w_{min}, such that at least one of these becomes zero, and add to the node weight $2w_{min}$. This preserves the cost of entering, visiting, and exiting j. More generally, for a TSP we can process each node i as follows:

1. let w_{min} be the weight of the cheapest edge incident on node i;
2. for each edge incident on node i subtract w_{min} from the edge's weight;
3. add $\delta_i = 2w_{min}$ to w_i, the weight of node i;
4. the node now has at least one incident edge of zero weight.

We need only process each node once, i.e., after processing, a node has at least one zero-weight incident edge and re-processing will have no effect. Fig. 1 shows the sequence of transformations of a four-node clique. All nodes start with weights shown in square brackets and are processed in alphabetic order. We start with the initial problem and then process node A. This removes a weight of 4 from A's incident edges and adds a weight of 8 to node A (Fig. 1b). We then process node B. This reduces the weight of incident edges by 1 and adds a weight of 2 to node B (Fig. 1c). Then we move on to node C and reduce the weight of its incident edges by 4 and add a weight of 8 to node C (Fig. 1d). Finally, processing D has no effect because there is a zero-weight incident edge. Note, that if we processed the nodes in a different order we might end up with a different final graph, i.e., the transformation is order dependent.

In a Hamiltonian path we have two distinguished nodes, i.e., the start node s and end node e, and the transformation is then modified as follows. Since s is not entered and e is not exited, we do not add to the weights of those nodes twice the weight of their minimum incident edges. Instead, we add the minimum weight once only, and process all other nodes as above.

[1] For a conventional TSP node weights would then be zero.



(a) Original graph

(b) Step 1. Processing node A

(c) Step 2. Processing node B

(d) Step 3. Processing node C

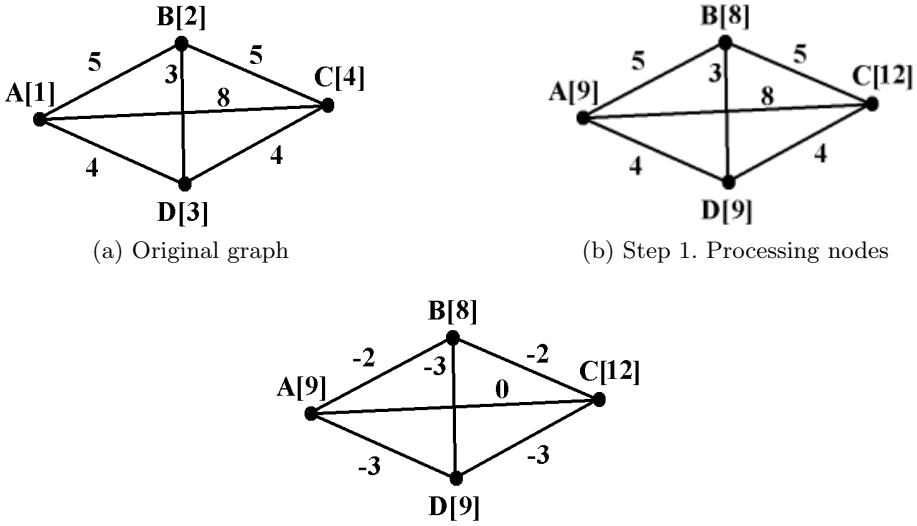
Fig. 1. Transformation of a 4-node clique. Node weights are shown in square brackets.

As noted above, the transformation is order dependent. In this paper we will investigate three such transformations[2]. The first uses a lexicographic ordering of nodes. We will refer to this as a *lex* ordering. The second, we will call *maxMin*; when selecting a node i to process next we choose a node such that its cheapest incident edge is a maximum over all nodes. For example, in Fig. 1 this would initially select node A or node C, as their cheapest incident edges are largest, with a weight of 4. The intuition behind this is that it will attempt to make the biggest reduction in edge weights. The third ordering is *minMin*. This might be thought of as the anti-heuristic, selecting to process a node i with smallest minimum incident edge weight. In Fig. 1 this would initially choose node B or D.

Order Independent Transformation. One can also think of a direct transformation that, unlike the ones presented above, does not depend on the order of processing the nodes. Indeed, instead of taking a node, updating its weight and the weights of the incident edges and then choosing a next node, we can do the following:

1. for each node, add a cost of $\delta_i = 2w_{min}$ to w_i, the weight of node i;
2. after processing all the nodes, for each edge (i, j) subtract a weight of $\frac{\delta_i+\delta_j}{2} = \frac{w'_i-w_i+w'_j-w_j}{2}$ from the cost of edge (i, j), where the prime symbol indicates a transformed value.

[2] In [2] it was shown how to use these transformations in the case of time windows specified on the nodes.



(a) Original graph

(b) Step 1. Processing nodes

(c) Step 2. Processing edges

Fig. 2. Order independent transformation of a 4-node clique. Node weights are shown in square brackets.

This is clearly order independent. For the same example graph as in Fig. 1 this transformation will result in the graph shown in Fig. 2c. It is easy to check that for this graph, the cost of any cycle remains the same after the transformation. The following theorem generalizes this result.

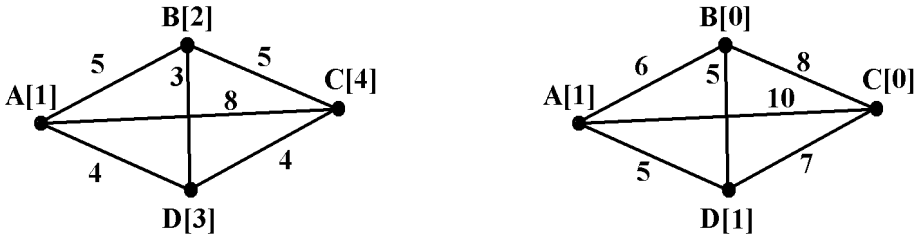
Theorem 1. *The order independent transformation preserves the cost of any solution on an arbitrary complete undirected graph.*

Proof. The proof is by induction. To prove it for $n = 3$ (i.e., for the simplest case of a cycle) we need to express the transformed edge and node weights through the original edge and node weights and substitute these expressions into the cost of the cycle after transformation.

Then assuming $n = 4$ and considering a 4-node tour T we can represent it as two distinct 3-node tours T_1 and T_2 (for which we already know that the transformation preserves the costs) that have a common start (and end) nodes and share an edge. Then we express the cost of T through the costs of T_1 and T_2 bearing in mind that we have to subtract twice the weight of the common edge (because we do not travel along it) as well as the weights of the nodes connected by the common edge (as in T we visit them only once).

Finally, for any $n > 4$ we notice that an n-node tour can be represented as pairs of distinct 3-node tours that, as before, have a common edge and a start node. All the rest can be done as above. *QED.*

If we have a Hamiltonian path, we never enter the start node s. Therefore the amount we add to its weight equals $\delta_s = w_{min}$ instead of $2w_{min}$ as for an arbitrary node. When processing the edges incident on s, we have to subtract a



(a) Original graph (b) Transformed graph

Fig. 3. Inverse transformation of a 4-node clique. Node weights are shown in square brackets

weight of $\delta_s + \frac{\delta_j}{2}$ from the weight of the edge (s, j). The same holds for the end node because we never exit it. All the rest remains as above.

3.2 Inverse Transformation

There also exists an *inverse* transformation that reduces node weights by adding a portion of them to the edge weights.

This transformation is as follows. At each step we take an arbitrary node i. Assume, without loss of generality, that the weight of node i is $w_i = 2k + \lambda$, where λ is 0 if w_i is even and 1 otherwise. Note we can always subtract $2k$ from w_i and add k to the weight of every edge incident upon i. This will preserve the cost of visiting node i.

Observe, first, that this inverse transformation is order independent. This is because each edge will either increase its weight or retain the original weight in the worst case (in fact, the weight of any edge is updated at most twice, i.e., when processing the two nodes connected by the edge). Second, as a result of this transformation we will have 0/1 node weights. A node weight will be 0 if the original node weight is even, and 1 otherwise.

Consider the same example 4-node clique as above (Fig. 3a). The order of processing the nodes is not important so we assume a lexicographic ordering. Processing node A has no effect as there is nothing to subtract from its original weight of 1 ($w_A = 2k+1 = 1$ and we subtract $2k = 0$). Processing node B results in its weight becoming 0 and the weights of edges AB, DB and CB becoming 6, 4 and 6 respectively. After processing node C its weight becomes equal to 0, the weights of edges AC, BC and DC become 10, 8 and 6. Finally, when processing node D we subtract 2 from its weight (this gives us a weight of 1) and add 1 to the weights of nodes AD, BD and CD (they become equal to 5, 5 and 7). Since, as noted above, we end up having 0/1 node weights, processing a node for a second time will have no effect. The transformed graph is shown in Fig. 3b.

By construction, the inverse transformation always preserves the costs of cycles. The transformation carries over to the case of Hamiltonian paths as follows. For an arbitrary node everything is as above. For the start or end node, we subtract the whole node weight and add it to the weight of each incident

edge. As a result, the transformed weights of the start and end node will always be 0 regardless of whether their original weights are odd or even.

4 An Empirical Study

The idea behind our experiments is to study the behavior of the proposed transformations. In [2] it was shown that transformations of the original problem can change the performance of typical JSP and VRP algorithms.

We conjecture that the difference in the performance of these algorithms may be a result of:

1. changes related to the edge weight to node weight ratio (a decrease should favor the JSP techniques);
2. changes in relative edge weights; the relative weights of edges at a node may change and this may serve to disguise real VRP problems and reduce performance in the case of direct transformations or, symmetrically, increase performance in the case of inverse transformations;
3. changes in relative node weights; they might also influence the behavior of different heuristics.

In this section we examine the extent of the changes that the different transformations of sections 3.1 and 3.2 have on randomly generated complete undirected graphs.

The transformations preserve the cost of cycles while changing the weights of edges and nodes. We are interested in the extent to which the weight of a cycle and the weight of the whole graph is transformed to the edges rather than the nodes. To assess this "weight transfer" we have two measurements. The first (expression (1)) measures the change in the proportion of the weight of a cycle that results from the edge weights. We expect the direct transformations to produce negative values as they reduce edge weights in favor of node weights. In contrast, the inverse transformation should produce a positive value. When measuring ρ_c for an instance graph, we generate 1000 random cycles consisting of $|V|$ nodes, where $|V|$ is the number of nodes in the graph, and compute the arithmetic mean $\mu(\rho_c)$.

The second measurement (expression (2)) assesses the relative change in the relative weight contribution of the edges in the overall graph. This again is a measure of the extent to which the transformations increase the edge weights while decreasing the node weights. As with expression (1), we expect the direct transformations to result in negative values and the inverse transformation in positive values.

$$\rho_c = \frac{\sum_{(i,j)\in C} w'_{ij} - \sum_{(i,j)\in C} w_{ij}}{\sum_{(i,j)\in C} w_{ij} + \sum_{i\in C} w_i}, \tag{1}$$

$$\rho = \frac{\frac{\sum_{(i,j)\in E} w'_{ij}}{\sum_{(i,j)\in E} w'_{ij} + \sum_{i\in V} w'_i} - \frac{\sum_{(i,j)\in E} w_{ij}}{\sum_{(i,j)\in E} w_{ij} + \sum_{i\in V} w_i}}{\frac{\sum_{(i,j)\in E} w_{ij}}{\sum_{(i,j)\in E} w_{ij} + \sum_{i\in V} w_i}}. \tag{2}$$

In (1) the summation is for all edges and nodes in a given cycle C. In (2) the summation is for all edges and nodes in a graph; as before, dashed values correspond to the transformed graph.

To estimate the extent of the changes in the relative edge weights we compare all pairs of edges at a node and count the number of times that the weight-based ordering of the pair changes from the original graph to the transformed graph. Similarly to measure the changes in the relative node weights we calculate the number of changes in the relative node weight for all pairs of nodes.

$$\epsilon_{edges} = \frac{\sum \psi'_{ijk}}{N_e}, \tag{3}$$

$$\epsilon_{nodes} = \frac{\sum \xi'_{ij}}{N_n} \tag{4}$$

where $\psi'_{ijk} = 1$ if edges (i, j) and (i, k) at node i change their relative weights as a result of transformation and 0 otherwise; $N_e = |V|\frac{(|V|-1)(|V|-2)}{2}$ is the number of such edge pairs in a graph; $\xi'_{ij} = 1$ if nodes i and j change their relative weight and 0 otherwise; and, finally, $N_n = \frac{|V|(|V|-1)}{2}$ is the overall number of node pairs in the graph. In expression (3) the summation is for all nodes and pairs of their incident edges, in expression (4) for all pairs of nodes.

First, we generate graphs of sizes 10, 20, ..., 100 nodes: 100 instances of each size, 1000 instances in total. Every instance is generated such that the edge weights are uniformly distributed between 1 and 50, the node weights between 0 and 50. To every graph we then apply the five transformations described in sections 3.1 and 3.2.

In Figs. 4 and 5 the arithmetic means $\mu(\rho_c)$ and $\mu(\rho)$ are shown against the graph size. We can see that both $|\mu(\rho_c)|$ and $|\mu(\rho)|$ for the order independent transformation are greater than for any order dependent one. This suggests, therefore, that the order independent transformation may be a better choice when we want to increase an average node weight to edge weight ratio in a graph.

Figs. 6 and 7 depict $\mu(\epsilon_{edges})$ and $\mu(\epsilon_{nodes})$ against the graph size for the direct transformations. Experiments showed that, in contrast to the direct transformations, $\mu(\epsilon_{edges})$ and $\mu(\epsilon_{nodes})$ for the inverse transformation do not depend on the size of graphs. Finally, experimenting with different ranges of edge and node weights in the original generated graphs we obtained results similar to the ones presented above.

Overall, our experiments demonstrate that the order independent direct transformation has a larger impact on the relative edge and node weights than the other direct transformations. That is, the order independent transformation transfers more of the weight of a cycle to the nodes than the other direct transformations. The inverse transformation transfers weights even more strongly as we can see by comparing the absolute values of the inverse transformation to that of the direct transformations.

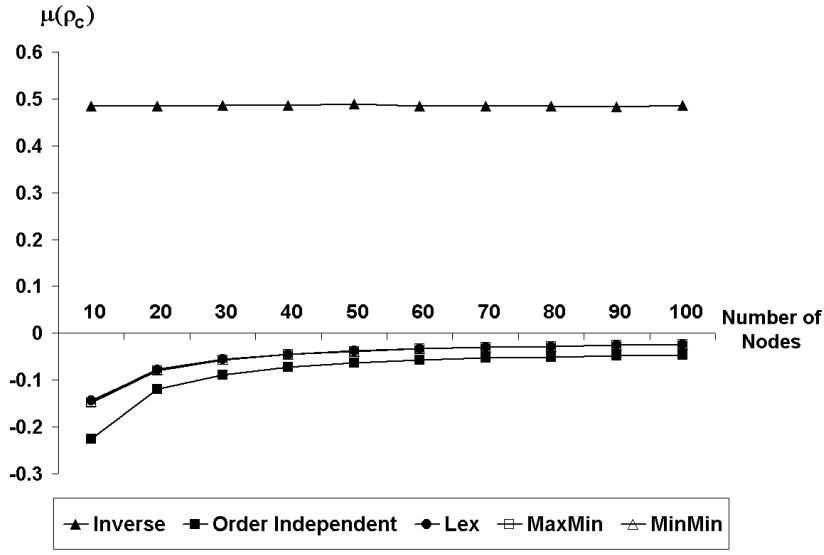


Fig. 4. $\mu(\rho_c)$ *vs* graph size

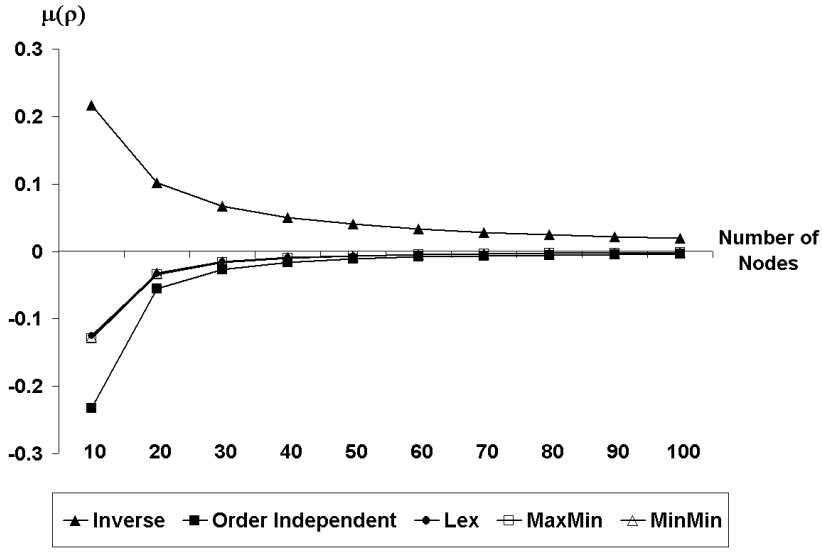


Fig. 5. $\mu(\rho)$ *vs* graph size

5 Conclusion and Future Work

The experiments in this paper contribute to a deeper understanding of the graph transformations and allow us to develop some conjectures with respect

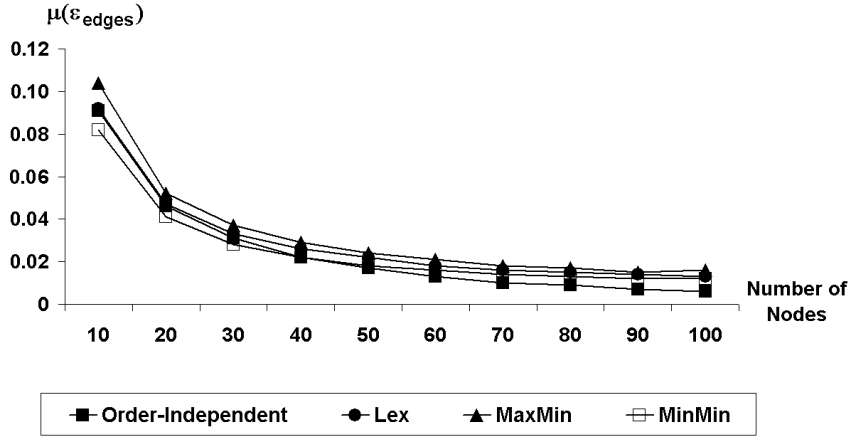


Fig. 6. $\mu(\epsilon_{edges})$ *vs* graph size

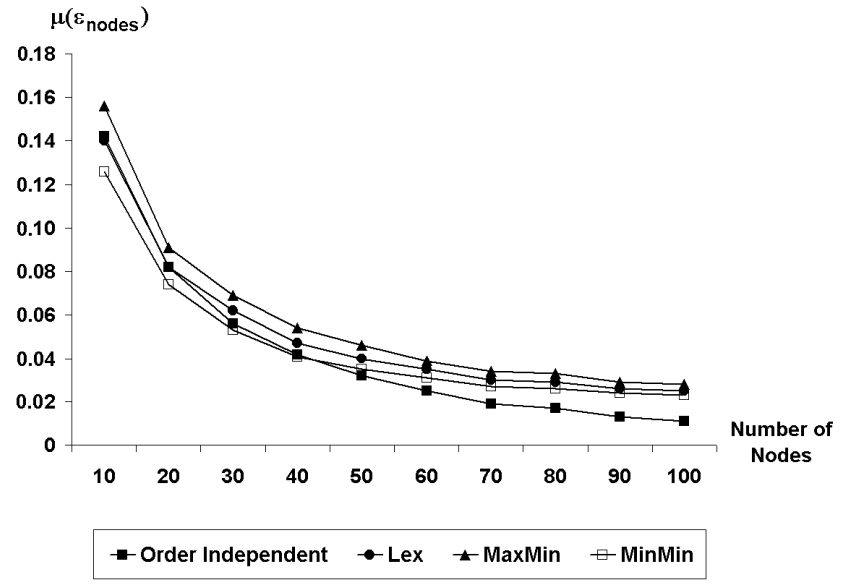


Fig. 7. $\mu(\epsilon_{nodes})$ *vs* graph size

to scheduling and routing performance. We expect for example, that the direct transformations will tend to increase the performance of scheduling algorithms as they reduce the edge weights in favor of node weights. This translates into a reduction in transition time between activities and longer activity durations. As argued above, this results in a problem that is closer to the problems upon which much of the research in scheduling has been done. Within the direct transforma-

tions, we see that the order independent transformation not only transfers more relative weight to the nodes than the other direct transformations but that it does so while performing about the same as the other transformations in terms of relative node weights and relative edge weights. This means that the order independent transformation will not degrade the heuristic ordering of nodes and edges that can be generated from the original problem. As a consequence, we conjecture that the order independent transformation will result in the greatest increase in the performance of scheduling algorithms of all the direct transformations.

In the other direction, the experiments demonstrate that the inverse transformation successfully transforms the graphs to place more weight in the edges of the graphs. We conjecture that this should increase the performance of routing algorithms.

Our future work is to apply these transformations systematically to VRP and JSP problems and to test our conjectures with respect to algorithm performance. In the longer term, we are interested in investigating further transformations in order to isolate the characteristics that contribute to the performance of different search algorithms.

References

1. J. C. Beck and M. S. Fox. Dynamic problem structure analysis as a basis for constraint-directed scheduling heuristics. *Artificial Intelligence*, 117(1):31–81, 2000.
2. J.C. Beck, P. Prosser, and E. Selensky. On the reformulation of vehicle routing probelms and scheduling problems. In *Proceedings of the Symposium on Abstraction, Reformulation and Approximation (SARA)*, 2002.
3. A. Cesta, A. Oddi, and S.F. Smith. A constraint-based method for project scheduling with time windows. *Journal of Heuristics*, 8(1):109–136, Jan 2000.
4. A.J. Davenport and J.C. Beck. An investigation into two approaches for constraint directed resource allocation and scheduling. In *INFORMS*, 1999.
5. B. DeBacker, V. Furnon, P. Shaw, P. Kilby, and P. Prosser. Solving vehicle routing problems using constraint programming and metaheuritics. *Journal of Heuristics*, 6:5001–523, 2000.
6. F. Focacci, P. Laborie, and W. Nuijten. Solving scheduling problems with setup times and alternative resources. In *Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling*, 2000.
7. M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.
8. F. Glover and M. Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
9. W. D. Harvey and M. L. Ginsberg. Limited discrepancy search. In *Proceedings of the Fourteenth International Joint Conference onf Artificial Intelligence (IJCAI-95)*, pages 607–613, 1995.
10. S. Kirkpatrick, C.D. Gelatt, and M.P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
11. P. Laborie. Algorithms for propagating resource constraints in AI planning and scheduling: Existing approaches and new results. In *Proceedings of the 6th European Conference on Planning (ECP01)*, 2001.

12. W. P. M. Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Department of Mathematics and Computing Science, Eindhoven University of Technology, 1994.
13. C. La Pape. Implementation of Resource Constraints in ILOG SCHEDULE: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.
14. E. Selensky. On mutual reformulation of shop scheduling and vehicle routing. In *Proceedings of the 20th UK PLANSIG*, 2001.
15. S. Smith and C. Cheng. Slack based heuristics for constraint satisfaction scheduling. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, pages 139–144, 1993.
16. M. Solomon. Algorithms for the Vehicle Routing and Scheduling Problem with Time Window Constraints. *Operations Research*, 35:254–365, 1987.
17. C. Voudouris and E.P.K. Tsang. Guided Local Search. *European Journal of Operational Research*, 113(2):80–110, 1998.

Call-by-Value λ-Graph Rewriting *Without Rewriting*

Maribel Fernández[1] and Ian Mackie[2]

[1] LIENS (UMR 8548), École Normale Supérieure, 45 Rue d'Ulm, 75005 Paris, France
[2] CNRS-LIX (UMR 7650), École Polytechnique, 91128 Palaiseau Cedex, France

Abstract. Girard's Geometry of Interaction offers a low-level decomposition of the cut-elimination process in linear logic, which can be used as a compilation technique for functional programming languages. It is the basis of the Geometry of Interaction Machine, which performs *call-by-name* computations in graph representations of functional programs *without doing any graph reduction*. Computation is given by a graph traversal algorithm: a simple intuition is that of a single token traveling through a fixed graph (the program to be evaluated), unraveling the evaluation. Here we continue this line of research to derive alternative ways of following this execution path which give *call-by-value* computations.

1 Introduction

Proof nets [10] provide a graphical syntax for proofs in linear logic, the analogous to λ-calculus for Intuitionistic Logic via the Curry-Howard Isomorphism. The Geometry of Interaction [11] provides a semantics of proof nets in terms of a set of *paths*, more precisely, the interpretation of a proof in linear logic is a pair of square matrices $(\Pi^\bullet, \sigma)$, where the matrix $\Pi^\bullet$ contains paths that do not traverse cuts in the proof, and the matrix σ contains information about the cuts in the proof. The Execution Formula allows the cut-elimination process to be modeled in these structures, and using one of the standard translations of the λ-calculus into linear logic proofs, one immediately gets a corresponding interpretation for the λ-calculus. The evaluation of a functional program can then be understood as following a path in a λ-graph, which, for a program, is a deterministic computation. Moreover, in contrast with the standard approach which consists in rewriting the graph, this execution is *reversible*: at any point during the computation, we can decide to go backwards in a deterministic way to undo the computation.

In [16] it was shown how one can obtain, in a very simple and direct way, an abstract machine that implements a call-by-name functional programming language directly from the Geometry of Interaction (the Geometry of Interaction Machine). Briefly, the matrices $(\Pi^\bullet, \sigma)$ can be seen as the code and linking information respectively. Each element of the matrix is transformed into a block of code (which is a direct compilation into machine language) and σ is used to correctly link-up each block of code. The novelty of this approach to implementation is two-fold. First, it offers a direct compilation of the λ-calculus into

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 75–89, 2002.
© Springer-Verlag Berlin Heidelberg 2002

machine language (the paths are built from atomic operations that are in a direct correspondence with machine instructions). Secondly, the run-time system is nothing but a simple token (implemented as global state) which is modified by machine instructions encoding the path. All costs of the computation are accounted for—there is no external copying/discarding machinery at all. However, counterbalanced with these features are a number of negative aspects:

1. The machine does not allow for jumping along the path in the graph to speed up execution—each operation is atomic and the history of the computation is kept sufficiently minimal such that there is just enough information for the token to know which way to go at each step.
2. The evaluation process recomputes functional arguments each time they are required, and thus corresponds to a call-by-name strategy. A call-by-value strategy would avoid recomputation of arguments, but the derivation of a call-by-value machine in the spirit of the Geometry of Interaction Machine has been an open problem up to now.
3. Already in the linear case (λ-terms where no substitution is ever copied or erased) one can obtain exponential length execution traces (with respect to the number of β-reductions required to take the term to normal form). This is essentially a consequence of the fact that η-expansion of a λ-term can cause exponential growth of the size of the term, and the Geometry of Interaction is actually working on η expanded terms.

A number of optimizations for the Geometry of Interaction Machine were proposed to try to overcome some of these deficiencies. The most important one was a notion of question/answer for values of base type (integers, booleans, etc.): while traversing the path, on arriving at a constant (an answer), there is a unique place (the question) where the result is required, and it is possible to jump directly to that point without performing the reversible part of the computation. This partially solves the first problem. This idea was generalized by Danos and Regnier [8] to questions and answers at higher type at the expense of introducing copying machinery. The reversible aspect of the machine is then lost, and the resulting machine (the JAM, Jumping Abstract Machine) corresponds to Krivine's Abstract Machine for the call-by-name λ-calculus.

This paper continues this research line, developing an abstract machine which does not have the deficiencies mentioned above: not only it jumps along the path as the JAM does, but also avoids reevaluation of arguments, implementing a call-by-value strategy. To our knowledge this is the first implementation in the spirit of the Geometry of Interaction Machine that follows a call-by-value strategy.

We use slight variants of the Geometry of Interaction operations, together with a new swap operation. Interestingly, this swapping operation was defined in Girard's original paper [11], but not used in the interpretation of proofs, and does not seem to have found a use in the literature until now. We show that this abstract machine can simulate an environment machine which is closely related with the Categorical Abstract Machine [4] (which also has a swap operation).

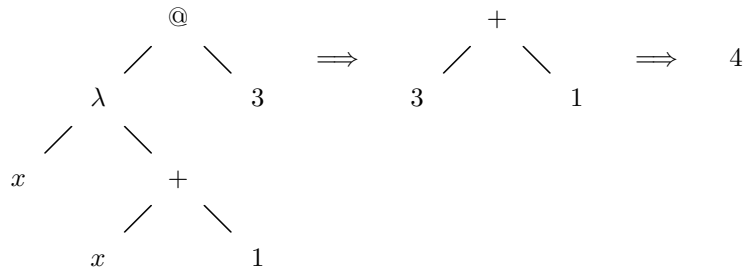
We also show an optimization of the Geometry of Interaction for the linear λ-calculus, both for call-by-name and call-by-value strategies since they are

equivalent in this case. We define an optimized Geometry of Interaction Machine offering a linear time/space algorithm for computing linear programs. We do not pursue this idea further since the generalizations of this machine to the full calculus seem to lead us to the same machine obtained by Danos and Regnier.

The rest of this paper is structured as follows. In the next section we motivate the ideas and recall the necessary background material required for the rest of this paper. In Section 3 we give a call-by-value machine based on the Geometry of Interaction, which is the main contribution of the paper. The correctness of the machine is proved by showing a correspondence with a call-by-value environment machine. In Section 4 we give the optimized linear Geometry of Interaction Machine. We conclude the paper in Section 5.

2 Background and Motivations

We first explain the idea intuitively, with a simple example. Consider the functional program $(\lambda x.x + 1)3$, written in the usual syntax of the λ-calculus [3]. There are a number of standard ways to evaluate this term, all of which would somehow simulate β-reduction either by graph reduction or by compiling into some environment-based machine. The Geometry of Interaction (GOI for short) puts a totally different perspective on this and essentially tries to "read out" the answer *without* actually doing any reduction of the program whatsoever. To compare it with graph-rewriting, below we draw the graph representing this program and the rewrites that would be required to evaluate it (we do not show the steps performing substitution, and the example is too simple to require any notion of sharing, see [17] for a complete guide to λ-graph reduction).



The basic idea of the Geometry of Interaction is that the result of the computation is already in the original graph, but hidden. The scenario for the process of *extracting* the result is easily understood as a simple data flow concept. We start at the root of the graph and try to read out the answer through the redexes in the original graph. We arrive first to an application node (@), so we proceed towards the function, but remembering that we came from the top of the application node. Arriving at the abstraction (λ) we can either continue into the body of the function, or into the variable. We use the history to deduce that we go into the body. We then arrive at the addition node (+) for which we require both arguments. Hence we proceed into the left argument towards the variable x.

Arriving at a variable x requires that we "look-up" the value of x in the graph. For this we exit the λ-node from the variable and continue following the path upwards. Arriving for the second time at the application requires that we look for the argument (since we came from the variable in the λ-node). Now we arrive at the value 3 and return to the part of the graph that requested this value, using the reverse of the path we just followed. Hence we arrive back at the + node, interrogate the second argument and return immediately with the value 1. We can now perform the addition and return back from the addition node following the reverse of the path that we took at the beginning of the computation. So we arrive back at the root of the term with the answer 4.

All this may look rather complicated at first sight, but the general concept is simple; we are doing nothing more than following the path that would have been extracted by reduction.

The easiest way to formalize the notion of path computations is to appeal to the connection with Lévy's labeled λ-calculus (see [14,1]).

Definition 1 (Labels). *Let $\mathcal{L}_0$ be an infinite set of letters (atomic labels). We define a set $\mathcal{L}$ of strings formed on $\mathcal{L}_0$ as follows:*

$$\begin{array}{ll} a \in \mathcal{L}_0 & \Longrightarrow a \in \mathcal{L} \\ \alpha, \beta \in \mathcal{L} & \Longrightarrow \alpha\beta \in \mathcal{L} \\ \alpha \in \mathcal{L} & \Longrightarrow \overline{\alpha} \text{ and } \underline{\alpha} \in \mathcal{L} \end{array}$$

Labeled λ-terms are then usual terms, where each sub-term has a label. We consider terms built out of variables, a generic constant $\star$ (that can be thought of as a natural number or boolean for instance), application and λ-abstraction.

Definition 2 (Labeled reduction). *Labeled β-reduction is defined by the rule: $((\lambda x.M)^\alpha N)^\beta \to \beta\overline{\alpha} \cdot M[\underline{\alpha} \cdot N/x]$, where*

$$\begin{array}{ll} \alpha \cdot \star^\beta & = \star^{\alpha\beta} \\ \alpha \cdot x^\beta & = x^{\alpha\beta} \\ \alpha \cdot (\lambda x.M)^\beta & = (\lambda x.M)^{\alpha\beta} \\ \alpha \cdot (MN)^\beta & = (MN)^{\alpha\beta} \end{array} \qquad \begin{array}{ll} \star^\alpha[N/x] & = \star^\alpha \\ x^\alpha[N/x] & = \alpha \cdot N \\ y^\alpha[N/x] & = y^\alpha \\ (\lambda y.M)^\alpha[N/x] & = (\lambda y.M[N/x])^\alpha \\ (MN)^\alpha[P/x] & = (M[P/x]N[P/x])^\alpha \end{array}$$

When substituting in a λ-abstraction we work modulo α-conversion (renaming of bound variables to avoid variable capture) as usual.

Below is an example term, with the normal form under labeled reduction:

$$\begin{array}{l} (((\lambda x.(x^a x^b)^c)^d((\lambda y.y^e)^f(\lambda z.z^g)^h)^i)^j\star^k)^l \\ \to^* \star^{\overline{lj\overline{d}ca\underline{d}i\overline{f}e\underline{f}hga\underline{d}i\overline{f}e\underline{f}hb\underline{d}i\overline{f}e\underline{f}hgj\overline{d}ca\underline{d}i\overline{f}e\underline{f}hga\underline{d}i\overline{f}e\underline{f}hb\underline{d}i\overline{f}e\underline{f}hk}} \end{array}$$

The label on the resulting term describes a path in the original term represented as a graph, where the underlined subpaths are reversed. The Geometry of Interaction Machine [16] computes this path without reducing the term. A close analysis of this path shows that there are:

- Call-return paths (over/under-lining): Paths are followed forwards and backwards, more precisely, the over-lined paths can be thought of as looking up an argument, and the underlined paths (which are exactly the same paths as the over-lined ones, but reversed) are the return paths. This call/return symmetry, identified by Asperti and Laneve [2], is precisely the key for the optimized GOI-machine (the JAM) presented by Danos and Regnier [8]: they show it is possible to jump directly back to avoid the reverse computation.
- Repeated sub-paths: The path *ifefh* (without over/underlining shown), corresponds to the computation of the argument $(\lambda y.y)(\lambda z.z)$ which is repeated (a call-by-name strategy is in operation).

In this paper we will try, in addition to performing the call/return optimization, also to avoid recomputing repeated sub-paths with a call-by-value strategy. We recall that the labeled λ-calculus is confluent [14]. From this fundamental property we know that no matter which evaluation order we use to reduce a term, the final label (which corresponds to a path) will always be the same. In other words, all strategies compute the same path (different strategies compute it in different ways). The question that we study in this paper then is how we can compute this path to implement a call-by-value strategy in an efficient way using some of the ideas of the GOI.

The Geometry of Interaction. We now recall the basic operations of the GOI. Rather than give all the details of the algebra, we shall just give an interpretation of the operators in the following table, where e and s are lists representing the *environment* and a *stack*, respectively. We use : for the list constructor, and $\cdot$ for an append constructor. This is one of the so-called small models of the algebra. In addition to all the standard operators of the Geometry of Interaction, we consider a new operator s that swaps the environment and the top of the stack. This operator has not been used in any of the previous GOI-machines, but we will see that it is crucial for the call-by-value GOI-machine. It is an interesting observation that all of these operators are linear (nothing is created or destroyed), and therefore can be captured in a monoidal category. The addition of the operator s adds a symmetry operation $A \otimes B \to B \otimes A$, which gives a symmetric monoidal category.

$$
\begin{array}{llll}
\mathsf{p}(e,s) & = (e, p : s) & \mathsf{p}'(e, \sigma : s) & = (e, (p : \sigma) : s) \\
\mathsf{q}(e,s) & = (e, q : s) & \mathsf{q}'(e, \sigma : s) & = (e, (q : \sigma) : s) \\
\mathsf{d}(e,s) & = (e, \Box : s) & \mathsf{b}(e, \sigma : s) & = (\sigma : e, s) \\
\mathsf{t}(e, \sigma_1 : \sigma_2 : s) & = (e, (\sigma_1 \cdot \sigma_2) : s) & \mathsf{s}(e_1, e_2 : s) & = (e_2, e_1 : s)
\end{array}
$$

For each operation *op* there is a reverse one denoted by op^*. We write the composition of operators in the right to left order. The simple example $\mathsf{p}^*\mathsf{p}(e,s) = \mathsf{p}^*(e, p : s) = (e, s)$ shows that this model correctly implements the equation $\mathsf{p}^*\mathsf{p} = 1$ (where 1 is the identity), which is one of the basic equations of the Geometry of Interaction. A more detailed explanation of these operators can be found in the literature, for instance [6,11,15].

3 Call-by-Value GOI-Machine

In this section we give the main contribution of the paper by defining an abstract machine implementing call-by-value in the spirit of the Geometry of Interaction Machine. We begin with an important remark about call-by-value computations. By the very definition of call-by-value, arguments are computed exactly once. We must therefore avoid the recomputation of the parts of the path that correspond to the evaluation of arguments, which, a priori, requires some storing of partial computations: thus there will be no way that one can obtain a completely reversible implementation of call-by-value in the spirit of the GOI-Machine, where the program is a fixed network. We must therefore provide mechanisms for storing these partial results, and be able to jump around the program. This machinery can also be used to avoid the recomputation in the call/return symmetric paths, as done in the call-by-name JAM.

We give now an informal description of the machine. Arguments are evaluated from left-to-right, rather than from right-to-left as in the SECD machine [12] for instance—both ways are possible and lead to variants of the same theme. The program is represented as a graph as usual, with application and abstraction nodes, and variables represented by edges. The machine traverses the graph by starting at the root, and uses the environment and stack as follows:

- To evaluate an application (arriving to an application from the top): the address of the argument together with a copy of the current environment is stored on the stack, and then the path is extended towards the function.
- To evaluate an abstraction (arriving to an abstraction from the top), there are two alternative cases to consider:
  1. The abstraction corresponds to the function in an application. The address of the body of the abstraction together with the current environment is stored on the stack, and the argument (the address of which was on the top of the stack) is evaluated. We thus exchange the current environment with that saved on the stack, and jump to the argument.
  2. The abstraction corresponds to the argument (a value) in an application. The address of the value together with the environment is saved on the top of the environment, and then we jump back to the body of the abstraction, which can be found on the top of the stack. The environments are swapped, that is, the current one is saved, and the machine recovers the correct one needed to evaluate the body of the abstraction.
- Arriving at an abstraction from the variable edge: jump to the value, and restore environment, which is to be found at the top of the environment.
- Leaving the scope of an abstraction: erase the top of the environment.

To define formally the compilation of the λ-calculus we first need to define the object language (the set of instructions of the machine), which is directly derived from the following operations.

Definition 3 (Context Transformers). *A context* (e, s, l) *consists of an environment* e*, a stack* s *and a stack of labels* l*.*

$$
\begin{array}{llll}
\mathsf{p}(e,s,l) & = (e,p:s,l) & \mathsf{push}(l)(e,s,l') & = (e,s,l:l') \\
\mathsf{q}(e,s,l) & = (e,q:s,l) & \mathsf{pack}(e,e':s,l:l') & = (e,(l,e'):s,l') \\
\mathsf{b}(e,e':s,l) & = (e':e,s,l) & \delta(e,s,l) & = (e,e:s,l) \\
\mathsf{s}(e,e':s,l) & = (e',e:s,l) & \epsilon(e,e':s,l) & = (e,s,l) \\
\mathsf{br}(e,s,l:l') & = (e,s,l') & \textit{(and we branch to address } l\textit{)} &
\end{array}
$$

The operations $\mathsf{p},\mathsf{q},\mathsf{b},\mathsf{s}$ are the ones defined in Section 2, extended to work on the context (e,s,l) (they simply ignore the additional l component). To store addresses we introduce two operators: the first, $\mathsf{push}(l)$, which takes a label as an argument, places l on the label stack. The second, pack, pairs the top element of the label stack with the top element of the stack. The δ and ϵ operators are also introduced: δ puts a copy of the environment in the stack, and ϵ erases the top of the stack. We no longer need the $\mathsf{p}',\mathsf{q}',\mathsf{t}$ operators (which are essentially used in the reversible computations): we use labels to find the way around the graph. Finally, br is a branching operation, to implement the jumps.

With the exception of br and ϵ, all these operators *op* have a reverse op^*. It is easy to see that s is self dual, and that pack^* (which can be thought of as unpack) moves the label from the paired top element of the stack to the label stack. From these operators, we define three macros which are used in the compilation.

$$
\begin{array}{ll}
\mathsf{d}(l) & = \mathsf{pack}\ \mathsf{push}(l)\ \delta \\
\mathsf{j}(l) & = \mathsf{br}\ \mathsf{pack}\ \mathsf{push}(l)\ \mathsf{s}\ \mathsf{pack}^* \\
\mathsf{j}' & = \mathsf{br}\ \epsilon\ \mathsf{s}\ \mathsf{pack}^*
\end{array}
$$

Each of these can be seen directly as a context transformer, by simply composing the action of each operator:

$$
\begin{array}{lll}
\mathsf{d}(l)(e,s,l') & = (e,(l,e):s,l') & \\
\mathsf{j}(l)(e,(l',e'):s,l'') & = (e',(l,e):s,l'') & \text{with branch to } l' \\
\mathsf{j}'(e,(l',e'):s,l'') & = (e',s,l'') & \text{with branch to } l'
\end{array}
$$

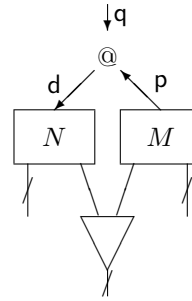
These operators can be mapped directly onto (sequences of) machine instructions, in the same way as the operations given in Section 2 are mapped onto GOI-Machine instructions in [16]. Note however, that it is no longer the case that each one is a constant time operation (for instance δ). To summarize, we have the following machine instructions: $\mathsf{q},\mathsf{p},\mathsf{b},\mathsf{s},\mathsf{d},\mathsf{j},\mathsf{j}',\epsilon,\mathsf{br}$.

Next we show how to compile the λ-calculus into these machine instructions. We begin by giving a graphical representation of λ-terms, where edges of the graph are annotated with instructions. From this representation we will directly generate machine code.

3.1 Encoding of Terms as Graphs

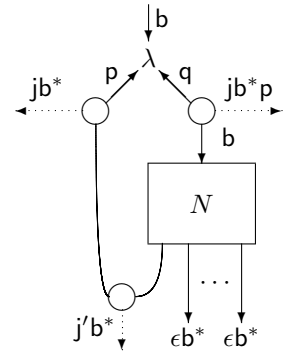
We define graph structures by induction on the structure of λ-terms.

- Variables simply translate into edges of the graph.
- The translation of an application tu is the following graph, where N and M are the graphs for t and u respectively.



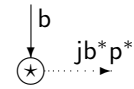
All the free edges at the bottom of the diagram represent the free variables in the term. The common variables occurring in t and u are joined together with the triangular node, which ensures that when we make an abstraction, there will be at most one edge corresponding to the bound variable.

– The translation of an abstraction $\lambda x.t$ is given by the following graph, where N is the graph corresponding to t.



We have introduced three circular nodes in the graph, which we shall call *jumping nodes*. The dashed lines represent connections to other parts of the graph, which will be determined at run-time. We have assumed without loss of generality that the left-most free edge of the net N corresponds to the free variable x of the term t. The b and b^* operators represent a box structure, which is the scope of the abstraction. Note that every way that one exits from the scope of the abstraction there is a b^* operator.

– The translation of the constant $\star$ is a terminal node in the graph, which has the label b, representing a boxed value. Again, we introduce a jumping node which will be determined at run time.

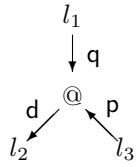


We remark that our graphs are derived from a particular translation of the λ-calculus into linear logic proof structures. We have used the so-called "$D = !(D \multimap D)$" translation, but similar results can be obtained for other translations[1].

3.2 Compilation of Graphs

The annotations in the edges of the graph will be used to generate machine code that computes a path in the graph. Each node of the graph is allocated a fresh address label $(l_1, l_2, \ldots)$ for each incident edge. These labels are used to give the entry points of the code generated when the path enters that point. No code is generated for variables, since this is just linking information for these labels. The notation $\widehat{l}$ is used to refer to the other-end of the edge, i.e. the entry address of the node connected to that edge. We show the compilation of each node.

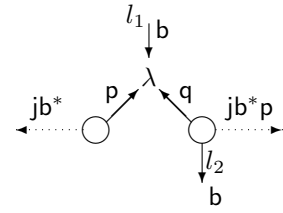
Application. There are three entry points for an application node (see diagram below), but only one will ever be used: the one at the top, l_1. Thus the only code generated is the path arriving from the top, which continues towards the function. For the generated code, we also show the context transformations alongside.



Code		Context
l_1 :		(e, s, l)
	q	$(e, q : s, l)$
	$\mathsf{d}(\widehat{l_3})$	$(e, (\widehat{l_3}, e) : q : s, l)$
	br $\widehat{l_2}$	

We can see how the operators guide the computation: arriving at l_1 we push q onto the stack, and follow the path with the d operator. Note that there is no way that we can follow the computation into the argument, since we cannot apply the p^* operator which would try to pop p from the stack. It is precisely this information that makes this evaluation mechanism deterministic.

Abstraction. We first observe that there are jumping nodes directly below the abstraction, so we shall include those directly in the compilation.



There are two cases for arriving at l_1 depending on whether there is a q (meaning that we are at the function), or a p (meaning that we are at the

[1] The $D = !(D \multimap D)$ translation is also known as the call-by-value translation, but note that it is independent of the evaluation strategy: one could use the call-by-name translation with a call-by-value evaluation strategy for instance.

argument) in the stack. We show the two cases separately, but they are in fact combined with a simple conditional. In each case we show the transformations of the context together with the generated code:

Code	Context	
l_1 :	$(e,(l,e'):q:s,l')$	
b	$((l,e'):e,q:s,l')$	
q^*	$((l,e'):e,s,l')$	
p	$((l,e'):e,p:s,l')$	
b^*	$(e,(l,e'):p:s,l')$	
$\mathsf{j}(\widehat{l_2})$	$(e',(\widehat{l_2},e):p:s,l')$	$(\mathsf{br}\ l)$

Code	Context	
l_1 :	$(e,(l,e'):p:s,l')$	
b	$((l,e'):e,p:s,l')$	
p^*	$((l,e'):e,s,l')$	
b^*	$(e,(l,e'):s,l')$	
$\mathsf{j}(l_1)$	$(e',(l_1,e):s,l')$	$(\mathsf{br}\ l)$

Exiting an abstraction: there are two cases depending on whether we are at the bound variable or a free variable (see the diagram in Section 3.1). If we are on the edge corresponding to the bound variable, then we should jump to the corresponding value, which will be found on the top element of the environment:

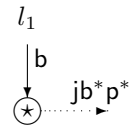
Code	Context	
l_1 :	$((l,e'):e,s,l')$	
b^*	$(e,(l,e'):s,l')$	
j'	(e',s,l')	$(\mathsf{br}\ l)$

In the other case (a variable exiting from the scope of an abstraction), we simply apply the $\epsilon\mathsf{b}^*$ operator, which produces the following transformation:

$$\epsilon\mathsf{b}^*((l,e'):e,s,l') = \epsilon(e,(l,e'):s,l') = (e,s,l').$$

Note that in all the cases where we exit the scope of the abstraction (a box) we apply the b^* operation. When we jump back we apply the b operator since we have returned inside the scope of an abstraction.

Constant. Arriving at a constant $\star$ at the label l_1, we simply return to the caller, using exactly the same sequence $\mathsf{jb}^*\mathsf{p}^*$ of instructions that we used for the argument of an application above.



This completes all the cases for the compilation. The machine is started with the empty state $(\Box,\Box,\Box)$, and stops when no further instruction can be applied, which as we shall see later, corresponds to the value of the term.

3.3 Correctness of the GOI-Machine: An Environment Machine

We can give an alternative presentation of the GOI-Machine in the form of an environment machine. To achieve this, we write terms (rather than the addresses

of the terms) and give rewrite rules on triples (M, e, s) where M is the term (written in de Bruijn notation [9]), e the environment and s a stack, encoding the operations of the Geometry of Interaction. Because we explicitly keep track of which sub-term we are currently evaluating, we no longer need the label stack and branching instructions.

Definition 4 (Call-by-Value Environment Machine).

$$\begin{array}{lll}
\textit{(app)} & (MN, e, s) & \Rightarrow (M, e, (N, e) : q : s) \\
\textit{(fcn)} & (\lambda M, e, (N, e') : q : s) & \Rightarrow (N, e', (M, e) : p : s) \\
\textit{(arg)} & (\lambda M, e, (N, e') : p : s) & \Rightarrow (N, (\lambda M, e) : e', s) \\
\textit{(var1)} & (1, (M, e') : e, s) & \Rightarrow (M, e', s) \\
\textit{(var2)} & (n + 1, u : e, s) & \Rightarrow (n, e, s) \\
(\star) & (\star, e, (N, e') : p : s) & \Rightarrow (N, (\star, e) : e', s)
\end{array}$$

The initial state of the machine for a term M is the configuration $(M, \square, \square)$. The machine stops when no rule can be applied, and the corresponding irreducible configurations are called final *if $s = \square$,* blocking *otherwise.*

This environment machine is closely related to the Categorical Abstract Machine (CAM) [4,5]. We use p, q as markers in the stack, indicating whether we are evaluating the function or its argument. Note also that the exchange of environments in the (*fcn*) and (*arg*) rules, which are implemented with the s operator in the call-by-value GOI-Machine, corresponds to the swap instruction of the CAM. We next show that this environment machine correctly implements the call-by-value strategy, and that the GOI-Machine simulates its execution.

Lemma 1. *The Call-by-Value Environment Machine is deterministic.*

To prove correctness of this machine we use the *calculus of closures* [5]. We recall the system defining the call-by-value strategy, where the values are weak head normal forms in the λ-calculus extended with the constant $\star$.

Definition 5 (Call-by-Value Strategy). *A closure u is a pair $M[\rho]$ where M is a term in de Bruijn notation and ρ a finite list of closures, which might be empty (denoted by $\square$). We can obtain the term represented by a closure $M[\rho]$ by using ρ as a substitution in M, denoted by $Subst(M, \rho)$.*

When $Subst(M, \rho)$ is a closed term we say that the closure $M[\rho]$ is closed. Values of closed closures are abstractions ($\lambda M[\rho]$) or constants ($\star[\rho]$). The relation $M[\rho] \to v$ defines the value v (in a call-by-value strategy) of a closed closure $M[\rho]$, assuming the closures in ρ are values:

$$v \to v \quad \textit{if } v \textit{ is a value}$$

$$n[u_1 \cdots u_m] \to u_n \quad \textit{if } n \leq m$$

$$\frac{M[\rho] \to \lambda P[\rho'] \qquad N[\rho] \to u \qquad P[u{:}\,\rho'] \to v}{(MN)[\rho] \to v}\ (\mathsf{Eval})$$

We say that the value of a closed λ-term M with the call-by-value strategy is V, written $M \to_{CBV} V$, if $M[\Box] \to N[\rho]$ and $Subst(N, \rho) = V$. Values of closed terms are therefore abstractions (λM) or constants (represented by $\star$).

By abuse of notation, we will denote in the same way a list of closures and an environment in the machine.

Theorem 1 (Correctness of the Environment Machine). *Let M be a closed λ-term.*

1. *If $M \to_{CBV} V$ then $(M, \Box, \Box) \Longrightarrow^* (N, \rho, \Box)$ final, and $Subst(N, \rho) = V$.*
2. *If $(M, \Box, \Box) \Longrightarrow^* (N, \rho, s)$, where (N, ρ, s) is irreducible, then either $s = \Box$ and $M \to_{CBV} Subst(N, \rho)$, or $s \not= \Box$ and M does not have a value with the call-by-value strategy.*

To obtain the correctness result for the call-by-value GOI-Machine we will prove that both machines perform the same computation. We show first that the instructions of the GOI-Machine allow us to simulate each of the environment machine rewrite rules (but Geometry of Interaction instructions are more atomic, a rewrite rule corresponds to a sequence of GOI-Machine instructions). Since both machines are deterministic, this is sufficient.

Theorem 2 (Simulation). *Any trace of the environment machine can be simulated by the call-by-value GOI-Machine.*

Theorem 3 (Correctness of the Call-by-Value GOI-Machine). *If t is a closed term having a value u with the call-by-value strategy, then the compilation of t is a GOI program such that when executed by the GOI-Machine, stops at the root of u. If t does not have a value, then the machine blocks or loops forever.*

We end this section with an example showing the trace of the execution of a term in Fig. 1. Rather than show the long trace of the call-by-value Geometry of Interaction machine, we just give the more readable transitions of the environment machine, which are equivalent by Theorem 2.

4 A Linear Geometry of Interaction Machine

The linear λ-calculus is the usual λ-calculus where the possibility for variables to occur several times (or not at all) is disallowed. For this calculus we can define a variant of the GOI-Machine which is more efficient than the JAM and the Call-by-Value Machine. Intuitively, the machine works as follows. As in all the machines based on the GOI, the term to be evaluated is considered as a graph, where we start at the root. We then apply the following rules:

– Arriving at an application: push the address of the argument onto a stack, and proceed towards the function.

term	environment	stack
$(\lambda 11)((\lambda 1)(\lambda 1))\star$	$\square$	$\square$
$(\lambda 11)((\lambda 1)(\lambda 1))$	$\square$	$(\star,\square) : q : \square$
$\lambda 11$	$\square$	$((\lambda 1)(\lambda 1),\square) : q : (\star,\square) : q : \square$
$(\lambda 1)(\lambda 1)$	$\square$	$(11,\square) : p : (\star,\square) : q : \square$
$\lambda 1$	$\square$	$(\lambda 1,\square) : q : (11,\square) : p : (\star,\square) : q : \square$
$\lambda 1$	$\square$	$(1,\square) : p : (11,\square) : p : (\star,\square) : q : \square$
1	$(\lambda 1,\square) : \square$	$(11,\square) : p : (\star,\square) : q : \square$
$\lambda 1$	$\square$	$(11,\square) : p : (\star,\square) : q : \square$
11	$(\lambda 1,\square) : \square$	$(\star,\square) : q : \square$
1	$(\lambda 1,\square) : \square$	$(1,(\lambda 1,\square) : \square) : q : (\star,\square) : q : \square$
$\lambda 1$	$\square$	$(1,(\lambda 1,\square) : \square) : q : (\star,\square) : q : \square$
1	$(\lambda 1,\square) : \square$	$(1,\square) : p : (\star,\square) : q : \square$
$\lambda 1$	$\square$	$(1,\square) : p : (\star,\square) : q : \square$
1	$(\lambda 1,\square) : \square$	$(\star,\square) : q : \square$
$\lambda 1$	$\square$	$(\star,\square) : q : \square$
$\star$	$\square$	$(1,\square) : p : \square$
1	$(\star,\square) : \square$	$\square$
$\star$	$\square$	$\square$

Fig. 1. Example execution trace

- Arriving at an abstraction from the top: pop the stack, and store this address at the λ node in the graph.
- Arriving at an abstraction from the variable: branch to the address stored in the abstraction.

We formalize this as follows. Assume a global stack of addresses, with the usual operations of push, top and pop. Since the number of abstraction nodes in the graph is known at compile time, we can allocate an integer to each one, and use the integer for the representation of variables. We can then have an array of addresses which we call L. Compilation can then be given as:

Definition 6 (Compilation). *If t is a (closed) linear λ-term, then we generate assembly language (where* br *is a branch instruction) as follows:*

$$\mathcal{C}(t) = \begin{array}{r} \mathcal{C}(t)_l \\ end : \mathsf{halt} \end{array}$$

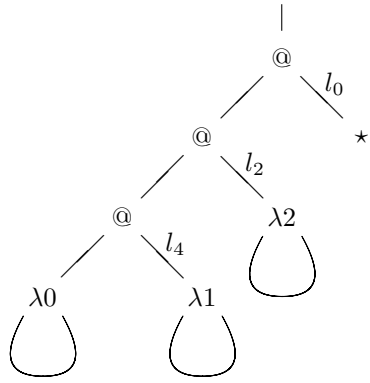
where $\mathcal{C}(\cdot)_l$ is given by (all labels are assumed fresh):

$$\begin{array}{l|l} \mathcal{C}(\star)_l = l : \mathsf{br}\ end & \mathcal{C}(x)_l = \mathsf{br}\ L[x] \\ \hline \mathcal{C}(\lambda x.t)_l = l : L[x] = \mathsf{top} & \mathcal{C}(tu)_l = \mathsf{br}\ l \\ \quad \mathsf{pop} & \qquad \mathcal{C}(u)_{l_u} \\ \quad \mathcal{C}(t)_{l_t} & \qquad l : \mathsf{push}\ l_u \\ \quad \mathcal{C}(t)_{l_t} & \end{array}$$

We give a simple example to show how this machine works. Consider the term $III\star$ where I is $\lambda x.x$. After renaming variables as unique integers we get: $(\lambda 0.0)(\lambda 1.1)(\lambda 2.2)\star$. This kind of term can be used to show that there are exponential length paths in the Geometry of Interaction, but now we can obtain a linear path. The compilation of the entire program is given below. (There is clearly scope for optimization to eliminate simple indirection.)

$$
\begin{array}{lll}
 & \mathsf{br}\ l & \\
l_0 & : \mathsf{br}\ end & \\
l & : \mathsf{push}\ l_0 & \\
 & \mathsf{br}\ l_1 & \\
l_2 & : L[2] = \mathsf{top} & \\
 & \mathsf{pop} & \\
 & \mathsf{br}\ L[2] & \\
l_1 & : \mathsf{push}\ l_2 & \\
 & & \mathsf{br}\ l_3 \\
l_4 & & : L[1] = \mathsf{top} \\
 & & \mathsf{pop} \\
 & & \mathsf{br}\ L[1] \\
l_3 & & : \mathsf{push}\ l_4 \\
l_5 & & : L[0] = \mathsf{top} \\
 & & \mathsf{pop} \\
 & & \mathsf{br}\ L[0] \\
 & & end : \mathsf{halt}
\end{array}
$$

The graph of the term is the following, where we show the trace of the execution working on the stack.



0	1	2	Stack
–	–	–	–
–	–	–	l_0
–	–	–	$l_2 l_0$
–	–	–	$l_4 l_2 l_0$
l_4	–	–	$l_2 l_0$
l_4	l_2	–	l_0
l_4	l_2	l_0	–

Theorem 4 (Correctness). *If t is a closed linear λ-term which evaluates under any evaluation order to a value (either $\star$ or an abstraction), then $\mathcal{C}(t)$ is a program that stops at the root of this value.*

5 Conclusions

The purpose of this paper was to demonstrate the techniques required to compile, in the spirit of the Geometry of Interaction Machine, a call-by-value strategy for execution. We mention two possible directions, which we leave for further study.

– Extensions to include usual data-structures and programming-language constructs (such as if-then-else): These are usually encoded via the additives of linear logic. For the call-by-name GOI-Machine an extension which includes the additives is given in [13].

– Call-by-value strategies for *virtual reduction* [7]: this is a graph rewriting system (thus not a fixed network), but could one impose a strategy for reduction that corresponds to call-by-value?

References

1. A. Asperti and S. Guerrini. *The Optimal Implementation of Functional Programming Languages*, volume 45 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1998.
2. A. Asperti and C. Laneve. Paths, computations and labels in the λ-calculus. In C. Kirchner, editor, *Rewriting Techniques and Applications. 5th International Conference, RTA-93, Montreal, Canada*, volume 690 of *LNCS*, pages 152–167. Springer-Verlag, June 1993.
3. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Company, second, revised edition, 1984.
4. G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.
5. P.-L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82:389–402, 1991.
6. V. Danos. *La Logique Linéaire appliquée à l'étude de divers processus de Normalisation (principalement du λ-calcul)*. PhD thesis, Université Paris VII, 1990.
7. V. Danos and L. Regnier. Local and asynchronous beta-reduction (an analysis of Girard's execution formula). In *Proceedings of the 8th Annual IEEE Symposium on Logic in Computer Science (LICS'93)*, pages 296–306. IEEE Computer Society Press, 1993.
8. V. Danos and L. Regnier. Reversible, irreversible and optimal λ-machines. *Theoretical Computer Science*, 227(1–2):79–97, 1999.
9. N. G. de Bruijn. Lambda calculus notation with nameless dummies. *Indagationes Mathematicae*, 34:381–392, 1972.
10. J.-Y. Girard. Linear Logic. *Theoretical Computer Science*, 50(1):1–102, 1987.
11. J.-Y. Girard. Geometry of interaction 1: Interpretation of System F. In R. Ferro, C. Bonotto, S. Valentini, and A. Zanardo, editors, *Logic Colloquium 88*, volume 127 of *Studies in Logic and the Foundations of Mathematics*, pages 221–260. North Holland Publishing Company, Amsterdam, 1989.
12. P. J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6:308–320, 1964.
13. O. Laurent. A token machine for full geometry of interaction. In S. Abramsky, editor, *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications, (TLCA 2001)*, volume 2044 of *LNCS*, pages 283–297. Springer-Verlag, 2001.
14. J.-J. Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. Thèse d'état, Université Paris VII, January 1978.
15. I. Mackie. *The Geometry of Implementation*. PhD thesis, Department of Computing, Imperial College of Science, Technology and Medicine, September 1994.
16. I. Mackie. The geometry of interaction machine. In *Proceedings of the 22nd ACM Symposium on Principles of Programming Languages (POPL'95)*, pages 198–208. ACM Press, January 1995.
17. S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice Hall International, 1987.

Transformation: The Missing Link of MDA

Anna Gerber, Michael Lawley, Kerry Raymond, Jim Steel, and Andrew Wood

CRC for Enterprise Distributed Systems (DSTC)
{agerber,lawley,kerry,steel,woody}@dstc.edu.au

Abstract. In this paper we explore the issue of transforming models to models, an essential part of the OMG's Model Driven Architecture (MDA) vision. Drawing from the literature and our experiences implementing a number of transformations using different technologies, we explore the strengths and weaknesses of the different technologies and identify requirements for a transformation language for performing the kind of model-to-model transformations required to realise the MDA vision.

1 Introduction

The OMG's Model Driven Architecture (MDA) [16] defines an approach to enterprise distributed system development that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform. The MDA approach envisions mappings from Platform Independent Models (PIMs) to one or more Platform Specific Models (PSMs). The UML Profile for Enterprise Distributed Object Computing (EDOC) [19] represents a first attempt to define a PIM along with several non-normative sketches of mappings to PSMs.

The potential benefits of such an approach are obvious: support for system evolution, high-level models that truly represent and document the implemented system, support for integration and interoperability, and the ability to migrate to new platforms and technologies as they become available.

While technologies such as the Meta Object Facility (MOF) [15] and the UML [17] are well-established foundations on which to build PIMs and PSMs, there is as yet no well-established foundation on which to rely in describing how we take an instance of a PIM and transform it to produce an instance of a PSM.

Our focus is on model-to-model transformations and not with model-to-text transformations. The latter come in to play when taking a final PSM model and using it to produce, for example, Java code or SQL statements. We believe that there are sufficient particular requirements and properties of a model to text transformation, such as templating and boilerplating, that a specialised technology be used. One such technology is Anti-Yacc [7].

Additionally, we will assume that the source and target models are different and leave the problem of update mappings for future work.

In this paper we put MDA to the test, exploring the question of whether we can really describe and perform the required mappings and, if so, then how and with what tools? In Section 2 we begin by surveying some existing related work

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 90–105, 2002.
© Springer-Verlag Berlin Heidelberg 2002

on model transformation, then in Section 3 we describe a number of our own experiments to express mappings from the EDOC Business Process model, our source PIM, to the Breeze Workflow model, our target PSM. In Section 4 we identify a number of key requirements of a transformation language based on these experiences and present a model that captures these requirements. Finally, in Section 5, we conclude with a discussion of what we believe are the next steps to take in determining an appropriate foundation for realising the MDA vision.

2 Existing Approaches

In this section we examine a number of existing approaches to implementing transformations and discuss their suitability to our goals.

2.1 CWM Transformation

Chapter 13 of the OMG's Common Warehouse Metamodel Specification [2] defines a model for describing Transformations. It supports the concepts of both black-box and white-box transformations. Black-box transformations are not of much interest to us because they only associate source and target elements without describing how one is obtained from the other. White-box transformations, however, describe fine-grained links between source and target elements via the *Transformation* element's association to a *ProcedureExpression*. Unfortunately, because it is a generic model and re-uses concepts from UML, a *ProcedureExpression* can be expressed in any language capable of taking the source element and producing the target element. Thus CWM offers no actual mechanism for implementing transformations, merely a model for describing the existence of a mapping.

2.2 Graph Transformation

Varró et al [24,6] describe a system for model transformation based on Graph Transformations [1]. In their approach, a transformation consists of a set of rules combined using a number of operators such as sequence, transitive closure, and repeated application. Each rule identifies before and after sub-graphs, where each sub-graph may refer to source and target model elements and associations between them (introduced by the transformation).

This style of approach to model transformation introduces non-determinism in the rule selection, and in the sub-graph selection when applying a rule.

Also, since rules are applied in a sequence, thus resulting in a series of state changes, one needs to be very careful about repeated rule application to ensure termination, and the order of rule application.

2.3 Generated XSLT

Peltier et al. [21,20] propose that transformation rules are best expressed at the model level and that they should then be translated into a set of rules

```
rule ::= 'rule' [name] [Entity] 'from' Entity [Restriction]? '{' body '}'
body ::= 'parameters' ':-' ...
         'init' ':-' ...
         'inherits' ':-' ...
         'attributes' ':-' ...
         'roles' ':-' ...
```

Fig. 1. Model for an MTRANS transformation rule.

that operate on the concrete representations of model instances. As such, they propose MOF as the common meta-model for representing models, XMI [14] as the concrete expression of model instances, and XSLT as the transformation tool to operate on these concrete instances.

Their model for the transformation rules is show as a grammar in Figure 1. Their rules have a mix of both procedural and declarative styles which is in part due to the fact that a given rule may only define a single target element per source element and that target element construction is explicit.

2.4 Text-Based Tools

Text based tools such as awk and perl are suitable only for the simplest kinds of transformations, largely because they deal with concrete syntax rather than abstract syntax. While arguably more readable and maintainable than XSLT transformations, they require the parsing of input text and serialisation of output text, rather than providing the abstraction of a parse-tree as XSLT does.

3 Our Experiments

In order to identify the requirements for a transformation language suitable for MDA, we primarily attempted to express mappings from the EDOC Business Process model (EDOC-BP) to the Breeze Workflow model using a number of different technologies. Additionally, we attempted a number of other mappings, for example from EDOC-BP to XLANG [23], the underlying model of Microsoft's Biztalk Orchestration, and from Breeze to *dot* [10] allowing for visualisation of the results of mapping from EDOC-BP to Breeze. In the following we describe these attempts and what they taught us.

3.1 XSLT – XMI to XML

Our first attempt at implementing a mapping from EDOC-BP to Breeze used XSLT [25] to map from the XMI representation of the MOF model describing EDOC-BPs to the native XML representation of Breeze workflows [3]. Our motivations for choosing XSLT included the following:

```
<xsl:template match="ECA.BusinessProcessPkg.OutputGroup |
                     ECA.BusinessProcessPkg.ExceptionGroup">
  <xsl:param name="a"/>
  <xsl:variable name="ct" select="concat(@xmi.id,'.condTask')"/>
  <xsl:choose>
    <xsl:when test="self::node()[@isSynchronous = 'true']">
      <xsl:call-template name="condTaskTemplate">
        <xsl:with-param name="ct" select="$ct"/>
        <xsl:with-param name="a" select="$a"/>
      </xsl:call-template>
    </xsl:when>
    <xsl:otherwise>
      <xsl:call-template
          name="asyncCompoundTaskInputGroupOrActivityOutputGroup">
        <xsl:with-param name="a" select="$a"/>
      </xsl:call-template>
    </xsl:otherwise>
  </xsl:choose>
</xsl:template>
```

Fig. 2. Part of the XSLT for mapping EDOC-BP XMI to Breeze XML.

1. both source and target formats were XML,
2. XSLT is based on the concept of matching parts of the source document based on its structure and associating this with the construction of the target document, and
3. a number of XSLT engines are readily available, thus allowing us to focus on describing the mappings rather than having to simultaneously implement a mapping evaluation engine.

Figure 2 shows a fragment of the transformation. This is a relatively simple example that, depending on the attribute `isSynchronous` of the matched OutputGroup or ExceptionGroup either one or another Breeze structure is constructed. Note that ExceptionGroup is a subclass of OutputGroup in the EDOC model, but we cannot exploit this in writing the rule – we must match explicitly on all precise-types.

Not surprisingly, the verbosity of XML as a syntax meant that the mappings became unwieldy and unreadable quite quickly. It was quite difficult to separate the source and target parts of the rules as well as the filtering constraints. In order to write the rules, one needed to be aware of the details of the construction of the syntactic representation of the resulting target model, rather than just its abstract structure. There were also a number of pragmatic problems with the use of XSLT – it is almost impossible to debug, and it is horrendously inefficient since everything is passed by-value which results in a great deal of structure copying.

We also attempted an EDOC-BP to XLANG mapping using XSLT, again because both the source and target instance representations were XML, but this

proved somewhat futile due to the overly restricted nature of XLANG as implemented in the beta version of Biztalk Orchestration available at the time. This meant that the only way to accurately capture the semantics of an EDOC-BP instance using XLANG would have involved embedding the required semantics by generating substantial amounts of code outside of the XLANG model.

3.2 GenGen – a MOF to MOF Transformation Generator

Having learned that describing mappings at a textual level leads to complications due to the continual transitions from concrete to abstract syntax and back again, we decided to implement our own transformation tool that would operate directly on the model instances involved. That is, we would deal simply with abstract syntax to abstract syntax mappings, leaving the rendering of the result to a separate step (and tool).

The GenGen tool uses mapping rules to generate a Java program that applies these rules to model instances stored in MOF repositories. Its structure and operation is shown in Figure 3. Using DSTC's MOF implementation, dMOF [4], we built model repositories for the mapping rules, the EDOC model and the Breeze workflow model based on MOF meta-model descriptions.

The model for describing the mapping rules was based in part on ideas in the Common Warehouse Metadata model [2] which describes correspondences between MOF model instances.

The generated transformation program would apply the rules one after the other to select objects in a source model repository and create and update objects in the target model repository.

Figure 4 shows the model for these transformation rules. It should be noted that this model has the following characteristics:

1. it has a procedural interpretation; a sequence of steps,
2. target model instances are explicitly created,
3. a traceability relation[1] is constructed and maintained as the rules are evaluated,
4. multiple source model instances can be matched,
5. pre-conditions are used to establish/require correlations between tuples of matching source model instances,
6. multiple target model instances can be created by a rule,
7. the expression language for pre-conditions, post-conditions, and actual transformation is either, one of a small set of pre-defined operations or, a string of Java code which is invoked in the context of the matched elements,
8. arbitrary code can be invoked by a rule.

[1] The traceability relation maintains a record of correspondences between source and target model instances. This can then be used for tasks such as debugging, round-tripping, and update propagation.

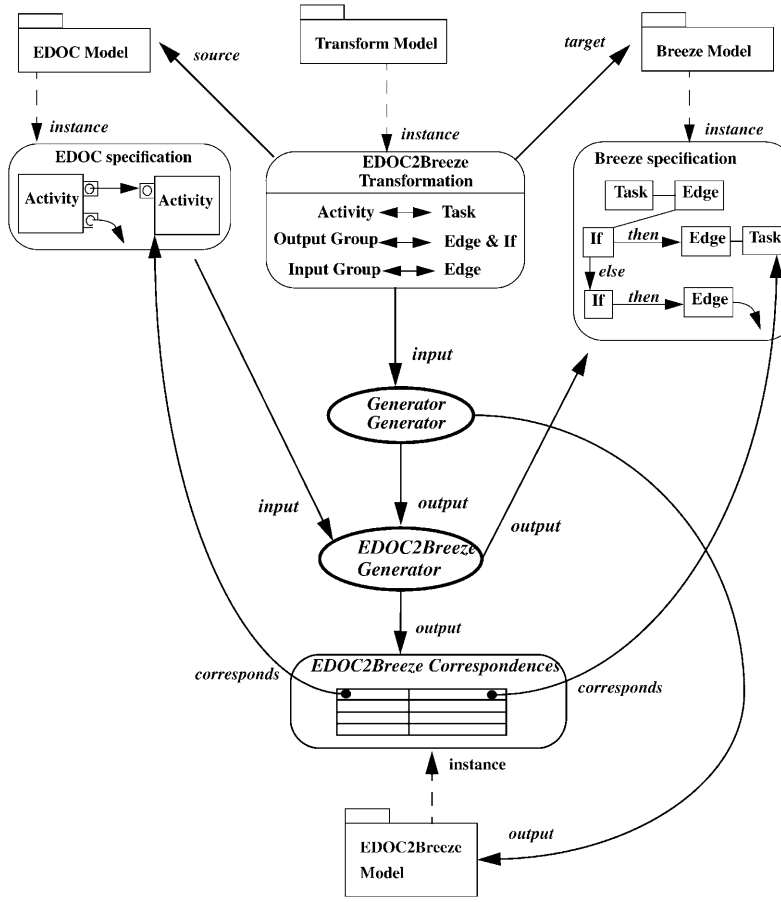


Fig. 3. Applying GenGen to produce an EDOC2Breeze transformer.

Figure 5 shows an incomplete fragment of one of the transformation rules[2] [18] parser. Note that this rule maps from a single source model element set, OutputGroup instances, to two target model class instances and an association. It also defined values for attributes of these instances. It contrasts with the XSLT rule in Figure 2 in that the set of source model instances that match this rule could include ExceptionGroup instances simply by specifying the `use_subtypes` attribute.

The GenGen experiment was quite successful and led to the discovery of the need for non-uniform or asymmetric transformations. These are transformations where an arbitrarily chosen subset (usually just one) of the elements that match

[2] The OMG's Human-Usable Textual Notation specification allows for the configurable definition of textual languages for MOF models. When prototyping tools such as GenGen, we use TokTok [5] to automatically generate a HUTN and associated parser.

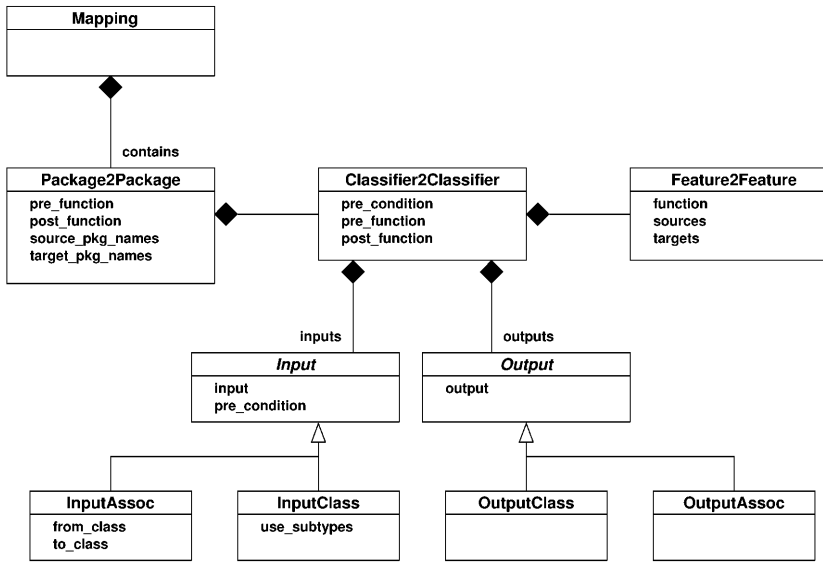


Fig. 4. The GenGen transformation model.

a source pattern is mapped differently to all the others. You can imagine this happening in the generation of something like cascading if-then-else statements. The last match in the transformation either has no *if* condition or no subsequent *else*-clause.

Using Java code fragments for expressing conditions and transformation functions leads to difficulties in expressing the transformations since it requires detailed understanding of how these fragments will be embedded in the generated code. In effect, the GenGen tool embodies a mapping from a hybrid transformation-rule/Java model to the Java language model and knowledge of this mapping is required to express the mapping rules.

After implementing GenGen, we felt that the transformation rules were too procedural and that the declarative nature and unification capabilities of a logic-programming style of transformation rule might be more suitable for describing and implementing mappings.

3.3 Mercury – Declarative Transformation Using Logic Programming

We chose to use the Mercury programming language [22] for a number of reasons. Being a purely declarative logic language, it provided us with unification for pattern matching the source instance-graph. It is also a strongly-typed language with an efficient implementation based on compilation to C and it is advertised as supporting a CORBA IDL mapping [8] which we hoped would allow future connection to our MOF repositories.

```
Classifier2Classifier "FromOutputGroup" {
  InputClass  { input: "OutputGroup" }
  OutputClass { output: "Task" }
  Feature2Feature {
    sources: ("name")
    targets: ("target0_name")
    function: "COPY"
  }
  Feature2Feature {
    targets: ("target0_and_join" "target1_and_join")
    function: "CONSTANT false"
  }
  OutputClass{ output: "ConditionalTask" }
  Feature2Feature {
    targets: ("target1_name")
    function: ‘target1_name=source0_name+"_test";‘
  }
  Feature2Feature {
    sources: ("name")
    targets: ("expression")
    function:
      ‘target1_expression = "flag == " + ’"’ + source0_name + ’"’;‘
  }
  OutputAssoc { output: "conditional_then" }
  Feature2Feature {
    targets: ("cond")
    function:‘target2_cond=target1;‘
  }
}
```

Fig. 5. Incomplete fragment of a GenGen rule for mapping EDOC-BP to Breeze.

The style of transformation we adopted involved writing predicates defining individual pieces of the target model instance in terms of the source model instance. In essence, context-free sub-transformation rules were written, then rules that established context (for example, ensuring an association existed between two source model instances) would invoke the more general sub-transformation rules. This style of writing rules lead to repetitive rule bodies and fragmented transformations; it became difficult to group rules based on either the set of things generated from a given source model element, or the set of source model elements contributing to the generation of a given target model element.

An example of the style of rules is shown in Figure 6. Note that this is a very small fragment of the transformation and that certain unimportant details have been elided.

Since Mercury's type system offers a form of inheritance, *typeclasses*, we attempted to use it to capture the inheritance semantics of our source and target models. However, typeclasses do not provide traditional constructive or aggregation-based inheritance. Instead, they allow common structure in several

```
conditionaltask(Id) :-
        conditionaltask_for_outputgroup_of_activity(Id, _OutputGroup).

conditionaltask_for_outputgroup_of_activity(Id, OG) :-
        outputgroup_of_activity(OG, _Activity),
        mapId(OG^og_id, conditionaltask_for_outputgroup, Id).

outputgroup_of_activity(OutputGroup, Activity) :-
        outputgroup(OutputGroup),
        contains(Activity^a_id, OutputGroup^og_id),
        activity(Activity).
```

Fig. 6. Part of the Mercury rules for mapping EDOC-BP to Breeze.

types to be identified and then used in polymorphic rules. This did not suit our desire to capture the inheritance semantics of the source and target models.

The major lessons learned from the Mercury implementation were that the models, instances, and the meta-model should all be explicitly represented so that the semantics of the models and their instances are available to be used when writing the rules, and that the ability to define multiple targets in a single rule leads to a much more compact and, presumably, more readable set of rules since it allows for greater modularity.

3.4 F-Logic – Declarative Transformation Based on Object-Oriented Logic Programming

F-Logic [9] is one of the most developed and complete formal models for deductive object-oriented languages. Its features include object identity, complex objects, inheritance, polymorphic types, query methods, and encapsulation.

Our primary motivation for turning to F-Logic stemmed from our frustrations with trying to use Mercury's type system to reflect the semantics of the MOF meta-model. However, it also offered a number of other advantages: a very flexible and compact syntax for defining rules that could be interpreted at both the model and instance levels, and the ability to define multiple targets in a single rule. For example, the set of facts:

```
o1 : workflow.
o2 : task.
o3 : task.
o1[name -> "Quokka Example"].
o1[tasks ->> o2].
o1[tasks ->> o3].
o2[name -> "Find quokka"].
o3[name -> "Feed quokka"].
```

can also be written quite compactly as:

```
o1 : workflow [
  name -> "Quokka Example",
  tasks ->> {
    o2 : task [name -> "Find quokka"],
    o3 : task [name -> "Feed quokka"]
  }
].
```

and this compact *molecule* representation can be used in both rule heads and rule bodies allowing rules such as:

```
wf(CT) : workflow [
  tasks ->> t(A) : task[join -> orJoin]
] :-
    CT:compoundTask[
      contains ->> A:activity
    ].
```

which defines two object instances (`wf(CT)` and `t(A)` of classes `workflow` and `task` respectively), an association between the workflow instance and the task instance, and the value (`orJoin`) of an attribute of the task. Without the option of molecular or multiple rule heads, this would have required 4 separate rules with mostly-repeated rule-bodies.

Using this approach we were able to describe an EDOC-BP to Breeze mapping using only 8 transformation rules in addition to the rules that described the MOF-based models. A by-product of this approach is that, where one would normally have grouped rules for readability, the F-Logic syntax allows and encourages rules to be combined. The resulting rules then tend to a form where a set of source model elements are matched based on type, association, and other constraints (attribute values, for example), and a set of resulting target model elements are defined in terms of them.

Interestingly, this grouping of multiple context-related rules into larger rules tends to mirror the kinds of diagrams we would sketch on white-boards to communicate these mappings to one-another. That is, in describing a mapping to a colleague, we would rarely say this element maps to that element, this association to that association and element, etc. Instead, we would say "when you have this arrangement (pattern) of source elements, you get this other arrangement of target elements where this thing corresponds to that thing, etc." (see Figure 7).

The elements of the F-Logic implementation that proved beneficial were:

1. explicit naming of variables,
2. the ability to define multiple targets in a single rule, and
3. the ability to define these targets with differing cardinalities.

For our first attempt at using F-Logic we used the Flora [11] Prolog generating compiler which is an alpha quality implementation. This had two major consequences: 1) it was terribly inefficient, requiring 1.5G of RAM to perform the EDOC-BP-to-Breeze-to-*dot* mapping, and 2) its support for negation was problematic.

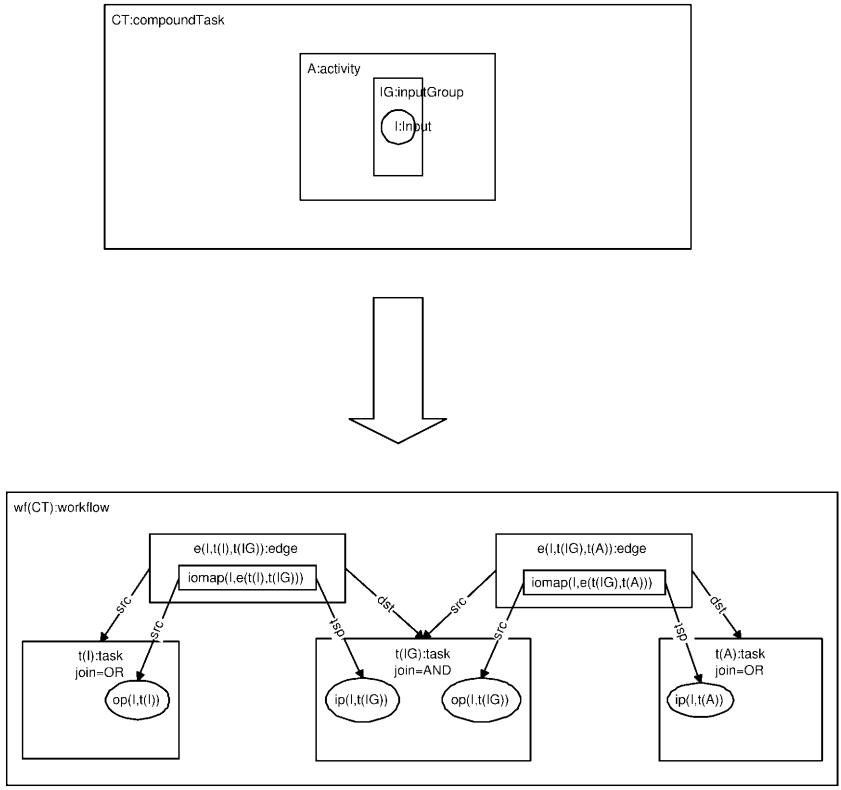


Fig. 7. Pictorial description of a mapping rule, as might be used for communication between people.

4 MDA Mapping Language Requirements

Having experimented with our own transformations and examined the experiments of others, we now have a set of requirements for a transformation language suitable for describing in a precise but readable manner, the kinds of model to model mapping rules required to realise the MDA vision.

4.1 Functional Requirements

The transformation language must be able to:

1. Match sets of source model elements.
2. Match elements by type (include instances of sub-types) and precise-type (exclude instances of sub-types). The mapping for an EDOC ExceptionGroup is different to the mapping for its concrete supertype, OutputGroup, for example.

3. Filter the set of matched tuples based on associations, attribute values, and other criteria. For example, an EDOC Input contained by an InputGroup is mapped differently to an Input contained by an Activity.
4. Establish associations between source and target model elements, possibly implicitly. These associations can then be used for maintaining traceability information.
5. Define different mappings for the first and last elements of the set of matched tuples. For example, when generating cascading if-then-else structures. More generally, matched elements may require a stable total ordering, which could be purely arbitrary, so that mapping rules can identify the successor, predecessor, and index of an element and whether it is the first or last in the set. These abilities are required when, for example, populating a table-like model with matched elements.
6. Handle recursive structure with arbitrary levels of nesting. For example, the uniqueness semantics of the source and target models may differ requiring the construction of *fully-qualified* names with a global scope in the target model from locally-scoped names in the source model.

4.2 Usability Requirements

It is desirable for readability and expressiveness concerns that:

1. multiple target elements are definable in a single rule,
2. rules are able to be grouped naturally for readability and modularity,
3. intermediate transformations should be definable, thus supporting multi-step transformations,
4. embedding of conditions and expressions in the transformation model is explicit and seamless, and
5. optional attributes are easily dealt with.

Reflecting on the transformation model used by GenGen and the implicit model used in the F-Logic based transformation, one can see they are quite similar. The major points of departure being the use of explicitly named variable bindings, varied multiplicities for generated elements, and the embedding of the condition/expression language.

Based on these requirements and a declarative execution model that involves matching a set of tuples or source model elements, filtering the set, and defining a set of resulting target model elements and their associated attributes, we have developed the transformation model shown in Figure 8.

With this model we are anticipating using the model for the Object Constraint Language (OCL) 2.0 [13] as the basis for our expression language.

The model explicitly identifies source elements, target elements, a filter expression, and a mapping expression.

Figure 9 shows an example source and target model, and a mapping rule using possible concrete syntax for the model. Note the use of the nested rule to produce a set of Baz elements per Bar element.

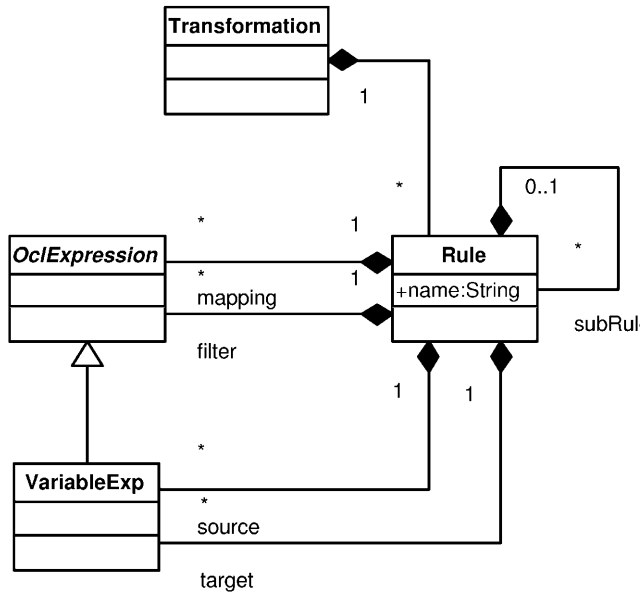
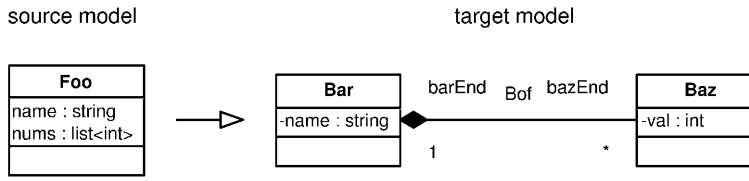


Fig. 8. Core aspects of proposed mapping model.



r1 (x : Foo) → (y : Bar) ::=
y.name = x.name
r2 (a : x.nums) → (z : Baz, a : Bof)
z.val = a, a.barEnd = y, a.bazEnd = z

Fig. 9. Example source and target models and a mapping including a nested rule.

Implementation of a transformation engine supporting the model of Figure 8 remains to be done. We anticipate beginning with an F-Logic or other logic-programming based prototype to sort out the details before adapting the GenGen tool to this model.

5 Conclusion

In carrying out our own transformations and examining the efforts of others it becomes clear that there are two quite different styles of transformation: procedural, with explicit source model traversal and target object creation and update,

and declarative, with implicit source model traversal and implicit target object creation/virtual target objects.

Currently, we tend to a preference for the declarative approach due to the simpler semantic model required to understand the transformation rules – order of rule application and termination semantics are a non-issue with the declarative approach. However, it should also be noted that when considering transformations that update a model, something that is outside the scope of this paper, some form of procedural aspect may be necessary.

In section 4 we identified a number of functional and non-functional requirements for an MDA transformation language. Amongst these, the need to be able to define non-uniform mappings and to define multiple targets with different cardinalities are notable.

We have indicated the desirability to be able to define and perform multi-step transformations, or to define intermediate rules that may define elements that are not part of the target model.

We have not discussed the issue of optimisation of transformations, although our XSLT mapping from EDOC-BP XMI to Breeze XML did include a second set of transformation rules that would map from the output of the first set of simple rules and remove any redundant elements produced by that mapping. Indeed, there is nothing special about an optimisation step other than the source and target models being the same, and the consequent requirement that the transformation rules be able to distinguish between source and target instances.

In defining mappings from model to model, the question of correctness of the mapping arises. There are several notions of correctness that can be considered. The simplest is that of syntactic or structural correctness. That is, given a well-formed instance of the source model, is it always the case that a well-formed instance of the target model is a result of the mapping? The more complex form of correctness is that of semantic correctness; does the result of the transformation *mean* the same thing as the input? One must avoid the implicit assumption, in both of these cases, that only complete or consistent models are of interest. There are cases, such as when transforming a model instance to some visual representation, where this may not be the case.

While our work has not investigated such theoretical issues, we did find it valuable to be able to express consistency criteria specific either to our source and target models, or to the particular transformation that we were implementing when working with both Mercury and the F-Logic implementations. Additionally, limited structural consistency checking can be performed using DTDs for XML-based transformations such as XMI and Breeze XML.

With regards the generic issue of semantic correctness, it is complicated by the fact that the source and target instances are, usually, statements in different domains of discourse and that the complexity of defining correspondences between statements in these two domains is itself tantamount to defining the transformation we wish to check. Thus this is only ever likely to be a feasible task when the source and target models are the same, isomorphic, or only very slightly different.

In conclusion, the MOF 2.0 Query/Views/Transformations RFP, not yet issued, will be the 6th in a series of MOF 2.0 RFPs [12] and has the potential to play a key role in the MDA vision. It is imperative for the success of this vision that an effective means of defining model transformations is developed. Unless such a solution addresses requirements such as those identified in this paper, transformation will remain the missing link of MDA.

Acknowledgments

The work reported in this paper has been funded in part by the Co-operative Centre for Enterprise Distributed Systems Technology (DSTC) through the Australian Federal Government's CRC Programme (Department of Education, Science and Training).

References

1. M. Andries, G. Engels, A. Habel, B. Hoffmann, H.-J. Kreowski, S. Kuske, D. Pump, A. Schürr, and G. Taentzer. Graph transformation for specification and programming. *Science of Computer Programming*, 34(1):1–54, Apr. 1999.
2. CWM Partners. Common Warehouse Metamodel (CWM) Specification. OMG Documents: ad/01-02-{01,02,03}, Feb. 2001.
3. DSTC. Breeze: Workflow with ease, online documentation. `http://www.dstc.edu.au/Research/Projects/Pegamento/Breeze/breeze.html`.
4. DSTC. dMOF: an OMG Meta-Object Facility Implementation. `http://www.dstc.edu.au/Products/CORBA/MOF/index.html`.
5. DSTC. TokTok – The Language Generator. `http://www.dstc.edu.au/Research/Projects/Pegamento/TokTok/index.html`.
6. S. Gyapay and D. Varró. Automatic Algorithm Generation for Visual Control Structures. Technical report, Dept. of Measure,ment and Information Systems, Budapest University of Technology and Economics, Dec. 2000. `http://www.inf.mit.bme.hu/FTSRG/Publications/TR-12-2000.pdf`.
7. D. Hearnden and K. Raymond. Anti-Yacc: MOF-to-text. Submitted to EDOC 2002.
8. D. Jeffery, T. Dowd, and Z. Somogyi. MCORBA: A CORBA Binding for Mercury. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, volume 1551 of *Lecture Notes in Computer Science*, pages 211–227, San Antonio, Texas, Jan. 1999. Springer Verlag.
9. M. Kifer, G. Lausen, and J. Wu. Logical Foundations of Object-Oriented and Frame-Based Languages. *Journal of the ACM*, 42(4):741–843, July 1995.
10. E. Koutsofios and S. North. Drawing graphs with *dot*. `http://www.research.att.com/sw/tools/graphviz/dotguide.pdf`, Feb. 2002.
11. B. Lundäscher, G. Yang, and M. Kifer. FLORA: The Secret of Object-Oriented Logic Programming. Technoical report, SUNY at Stony Brook, 1999.
12. Request for Proposal: MOF 2.0 Core RFP. OMG Document: ad/01-11-05, Nov. 2001.
13. Request for Proposal: UML 2.0 OCL RFP. OMG Document: ad/00-09-03, Sept. 2000.

14. OMG. Interchange Metamodel in XML. OMG Document: formal/01-02-15, Feb. 2001.
15. OMG. Meta Object Facility (MOF) v1.3.1. OMG Document: formal/01-11-02, Nov. 2001.
16. OMG. Model Driven Architecture – A Technical Perspective. OMG Document: ormsc/01-07-01, July 2001.
17. OMG. Unified Modeling Language v1.4. OMG Document: formal/01-09-67, Sept. 2001.
18. OMG. Human-Usable Textual Notation. OMG Document: ad/02-03-02, Apr. 2002.
19. OMG. UML Profile for Enterprise Distributed Object Computing (EDOC). OMG Document: ptc/02-02-05, Feb. 2002.
20. M. Peltier, J. Bézivin, and G. Guillaume. MTRANS: A general framework, based on XSLT, for model transformations. In *WTUML'01, Proceedsings of the Workshop on Transformations in UML*, Genova, Italy, Apr. 2001.
21. M. Peltier, F. Ziserman, and J. Bézivin. On levels of model transformation. In *XML Europe 2000*, pages 1–17, Paris, France, June 2000. Graphic Communications Association.
22. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference*, pages 499–512, Glenelg, Australia, Feb. 1995.
23. S. Thatte. XLANG Web Services for Business Process Design. Microsoft: `http://www.gotdotnet.com/team/xml_wsspecs/xlang-c/default.htm`, 2001.
24. D. Varró, G. Varraó, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. Accepted for Science of Computer Programming.
25. W3C. XSL Transformations (XSLT) v1.0.
W3C Recommendation: `http://www.w3.org/TR/xslt`, Nov. 1999.

Termination Detection of Distributed Algorithms by Graph Relabelling Systems*

Emmanuel Godard, Yves Métivier, Mohamed Mosbah, and Afif Sellami

LaBRI - Université Bordeaux1
351 Cours de la Libération
33405 Talence, France
{godard,metivier,mosbah|sellami}@labri.fr

Abstract. A unified and general scheme for detecting the termination of distributed computations is proposed. This scheme uses the encoding of distributed algorithms in form of graph rewriting systems to transform the problem of adding termination detection to a distributed computation into an operation on graph rewriting systems. Various examples are used to illustrate this approach.

1 Introduction

Distributed termination detection is one of the most important problems in distributed computing due to its numerous practical and theoretical applications. It is closely related to many other problems such as determining a causally consistent global state [4], detecting deadlocks [5], distributed garbage collection [17], and to universal graph reconstruction [14]. Many algorithms have been already designed for particular models and under various assumptions. However, general approaches and unifying methods for dealing with the termination detection problem have been rarely suggested. In this paper, we propose a general framework based on graph relabelling systems to add termination detection to a distributed algorithm. Our work is motivated by the recent result of Métivier and Tel [16] which characterizes the families of networks in which the local detection of the global termination is possible. We give here effective and practical solutions to add termination detection to a distributed algorithm.

Graph Relabelling Systems (GRS) [3] are suitable for encoding and implementing distributed algorithms, for proving their correctness and for understanding their power. These systems do not modify the underlying structure of the graph on which they are applied, but only modify the labels of vertices or edges. Each relabelling step is local, that is on a subgraph of fixed size according to some rules depending on such a subgraph. The power of relabelling systems can be increased using priorities (PGRS) or forbidden contexts (FCGRS). A priority relation (see [3,10,11]) is a partial order on the set of relabelling rules. In PGRS systems, the priority relation determines the order in which rules have to be

* This work is a part of the Visidia project [15], and has been supported by the "Conseil Régional d'Aquitaine".

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 106–119, 2002.
© Springer-Verlag Berlin Heidelberg 2002

applied if they overlap. In FCGRS [12], relabelling rules are equipped with forbidden context, that is graphs which disable the application of the corresponding rule if they are included in the context.

Consider a network represented by a graph whose vertices denote processors, and whose edges denote communication links. The local state of a processor (resp. link) is encoded by the labels attached to the corresponding vertex (resp. edge). Processors communicate only by asynchronous message passing. A relabelling rule is a rewriting rule which has the same underlying fixed graph for its left-hand side and its right-hand side, but with an update of the labels. Relabelling rules can be applied in parallel on the graph unless the occurrences of their left-hand sides share an edge or a vertex. The relabelling is performed until no rule can be applied.

By using various types of relabelling systems, it is possible to capture various paradigms of the distributed computation. These paradigms include sequential or distributed computation, local detection of the global termination, and correctness of computation. For example, there are three relabelling systems for the distributed computation of a spanning tree in an anonymous network [10,2,13,1] depending on whether such a computation is parallel, or whether it detects locally the global termination. We propose a practical and unified method to transform a graph relabelling system into another equivalent one which is equipped with the local detection of the global termination. If $\mathcal{A}$ is a distributed algorithm encoded by a graph relabelling system $\mathcal{R}_{\mathcal{A}}$, and if $\mathcal{T}$ is a termination detection algorithm encoded by a graph relabelling system $\mathcal{R}_{\mathcal{T}}$, then the superposition of $\mathcal{A}$ and $\mathcal{T}$ is obtained by a particular operation of the corresponding relabelling systems $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{R}_{\mathcal{T}}$. That is, we exploit the high level encoding of distributed algorithms by graph relabelling systems to simplify the combination of both algorithms. We illustrate this method through examples including the computation of a spanning tree and the coloration of a ring. For the termination detection algorithms, we use the classical ones, for instance the Dijkstra-Scholten algorithm [7], or the algorithm to detect stability [18].

The paper is organized as follows. In Section 2, we recall formal definitions of graph relabelling systems and present a few examples. Section 3 introduces the product operation of graph relabelling systems. In Section 4, we use a variation of the product to add the termination detection to a distributed algorithm, and we give a few examples. A generalization of our method is given in Section 5.

All the examples discussed in this paper have been implemented and tested on the Visidia tool [15].

2 Graph Relabelling Systems

2.1 Labelled Graphs

Throughout the paper we consider only simple connected graphs where vertices and edges are labelled with labels from a possibly infinite alphabet L. A graph labelled over L will be denoted by (G, λ), where G is a graph and $\lambda\colon V(G) \cup E(G) \to L$ is the function labelling vertices and edges. The graph

G is called the underlying graph, and the mapping λ is a labelling of G. The class of labelled graphs over some fixed alphabet L will be denoted by $\mathcal{G}_L$. Let (G, λ) and (G', λ') be two labelled graphs. Then (G, λ) is a subgraph of (G', λ'), denoted by $(G, \lambda) \subseteq (G', \lambda')$, if G is a subgraph of G' and λ is the restriction of the labelling λ' to $V(G) \cup E(G)$. A mapping $\varphi\colon V(G) \to V(G')$ is an homomorphism from (G, λ) to (G', λ') if φ is a graph homomorphism from G to G' which preserves the labelling, i.e., for all vertices $v, w \in V(G)$ $\lambda'(\varphi(v)) = \lambda(v)$, and $\lambda'(\{\varphi(v), \varphi(w)\}) = \lambda(\{v, w\})$.

The mapping φ is an isomorphism if it is bijective. An *occurrence* of (G, λ) in (G', λ') is an isomorphism φ between (G, λ) and a subgraph (H, η) of (G', λ').

2.2 Graph Relabelling Systems

We describe in this section the formal notion of graph relabelling systems.

Definition 1. *A* (graph) relabelling rule *is a triple* $R = (G_R, \lambda_R, \lambda'_R)$ *such that* (G_R, λ_R) *and* (G_R, λ'_R) *are two labelled graphs. The labelled graph* (G_R, λ_R) *is the* left-hand side *and the labelled graph* (G_R, λ'_R) *is the* right-hand side *of* R.

Definition 2. *A* graph relabelling system *(GRS for short) is a triple* $\mathcal{R} = (L, I, P)$ *where* L *is a set of labels,* I *a subset of* L *called the set of* initial labels *and* P *a finite set of relabelling rules.*

The intuitive notion of computation step will then correspond to the notion of relabelling step:

Definition 3. *A* $\mathcal{R}-$relabelling step *is a 5-tuple* $(G, \lambda, R, \varphi, \lambda')$ *such that* R *is a relabelling rule in* P *and* φ *is both an occurrence of* (G_R, λ_R) *in* (G, λ) *and an occurrence of* (G_R, λ'_R) *in* (G, λ').

Intuitively speaking, the labelling λ' of G is obtained from λ by modifying all the labels of the elements of $\varphi(G_R, \lambda_R)$ according to the labelling λ'_R. Such a relabelling step will be denoted by $(G, \lambda) \xrightarrow[R,\varphi]{} (G, \lambda')$.

The notion of computation then corresponds to the notion of relabelling sequence:

Definition 4. *A* $\mathcal{R}-$relabelling sequence *is a tuple* $(G, \lambda_0, R_0, \varphi_0, \lambda_1, R_1, \varphi_1, \lambda_2, \ldots, \lambda_{n-1}, R_{n-1}, \varphi_{n-1}, \lambda_n)$ *such that for every* i, $0 \leq i < n$, $(G, \lambda_i, R_i, \varphi_i, \lambda_{i+1})$ *is a* $\mathcal{R}-$*relabelling step. The existence of such a relabelling sequence will be denoted by* $(G, \lambda_0) \xrightarrow[\mathcal{R}]{*} (G, \lambda_n)$.

The computation stops when the graph is labelled in such a way that no relabelling rule can be applied:

Definition 5. *A labelled graph* (G, λ) *is said to be* $\mathcal{R}-$irreducible *if there exists no occurrence of* (G_R, λ_R) *in* (G, λ) *for every relabelling rule* R *in* P.

For every labelled graph (G, λ) in $\mathcal{G}_I$ we denote by $Irred_{\mathcal{R}}((G, \lambda))$ the set of all $\mathcal{R}-$irreducible labelled graphs (G, λ) such that $(G, \lambda) \xrightarrow[\mathcal{R}]{*} (G, \lambda')$. Intuitively speaking, the set $Irred_{\mathcal{R}}((G, \lambda))$ contains all the final labellings that can be obtained from a $I-$labelled graph (G, λ) by applying relabelling rules in P and may be viewed as the set of all the possible results of the computations encoded by the system $\mathcal{R}$.

Example 1. We give here an example of a graph relabelling system which computes a spanning tree. The system is given by $\mathcal{R}_1 = (L_1, I_1, P_1)$ defined by $L_1 = \{N, A, 0, 1\}$, $I_1 = \{N, A, 0\}$ and $P_1 = \{R\}$ where R is the following relabelling rule:



We assume that initially a unique vertex has label A, all other vertices having label N and all edges having label 0.

At each step of the computation, an A-labelled vertex u may activate any of its neutral neighbours, say v. In that case, u keeps its label, v becomes A-labelled and the edge $\{u, v\}$ becomes 1-labelled.

Hence, several vertices may be active at the same time. Concurrent steps will be allowed provided that two such steps involve distinct vertices. The computation stops as soon as all the vertices have been activated. The spanning tree is given by the 1-labelled edges. We give an example of the execution of this system in Figure 1.

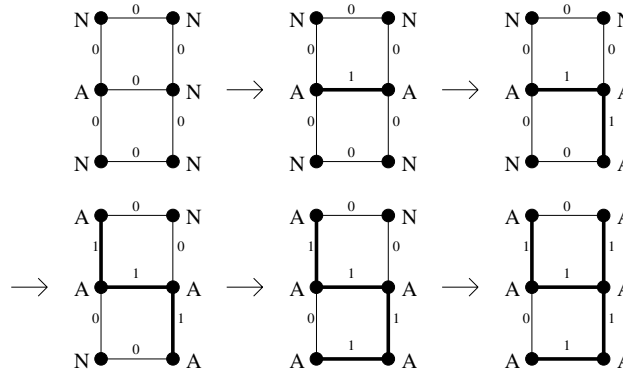
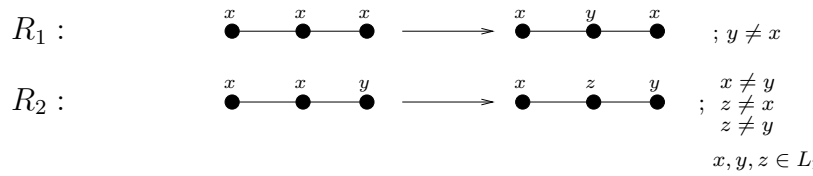


Fig. 1. Distributed computation of a spanning tree

Example 2. It is possible to color a ring with 3 colors using relabelling systems. Let $\{c_1, c_2, c_3\}$ be the set of colors. Let $\mathcal{R}_2 = (L_2, I_2, P_2)$ be the relabelling system defined by $L_2 = \{c_1, c_2, c_3\}$, $I_2 = L_2$ and $P_2 = \{R_1, R_2\}$ where R_1, R_2 are the following relabelling rules:



Initially, vertices are labelled at random.
This relabelling system assigns a correct 3-coloring to the vertices of a ring. Intuitively, the application of a rule will assign a correct and definite color to the middle vertex. Therefore, a finite ring with arbitrary initial coloring can be colored by applying a finite number of times the rules above.
Note that it is an example of a self-stabilizing algorithm [8].

2.3 Graph Relabelling Systems with Forbidden Contexts

The idea we develop here is to prevent the application of a relabelling rule whenever the corresponding occurrence is "included" in some special configuration, called a context. More formally, we have:

Definition 6. *Let (G, λ) be a labelled graph. A* context *of (G, λ) is a triple (H, μ, ψ) such that (H, μ) is a labelled graph and ψ an occurrence of (G, λ) in (H, μ).*

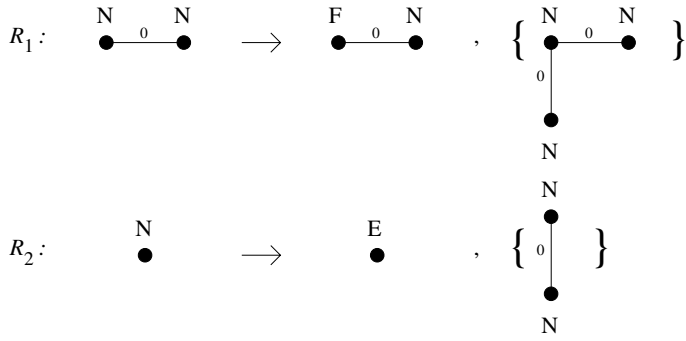
Definition 7. *A* relabelling rule with forbidden contexts *is a 4-tuple $R = (G_R, \lambda_R, \lambda'_R, F_R)$ such that $(G_R, \lambda_R, \lambda'_R)$ is a relabelling rule and F_R is a finite set of contexts of (G_R, λ_R).*

Definition 8. *A* graph relabelling system with forbidden contexts *(FCGRS for short) is a triple $\mathcal{R} = (L, I, P)$ defined as a GRS except that the set P is a set of relabelling rules with forbidden contexts.*

A relabelling rule with forbidden contexts may be applied on some occurrence if and only if this occurrence is not "included" in an occurrence of any of its forbidden contexts. More formally:

Definition 9. *A* $\mathcal{R}-$relabelling step *is a 5-tuple $(G, \lambda, R, \varphi, \lambda')$ such that R is a relabelling rule with forbidden contexts in P, φ is both an occurrence of (G_R, λ_R) in (G, λ) and an occurrence of (G_R, λ'_R) in (G, λ'), and for every context (H_i, μ_i, ψ_i) of (G_R, λ_R), there is no occurrence φ_i of (H_i, μ_i) in (G, λ) such that $\varphi_i(\psi_i(G_R, \lambda_R)) = \varphi(G_R, \lambda_R)$.*

Example 3. In this example, we give a relabelling system encoding a tree election algorithm. Let $\mathcal{R}_3 = (L_3, I_3, P_3)$ be the FCGRS defined by $L_3 = \{N, F, E, 0\}$, $I_3 = \{N, 0\}$ and $P_3 = \{R_1, R_2\}$ where R_1, R_2 are the following relabelling rules with forbidden contexts :



Let us call a *pendant* vertex any $N-$labelled vertex having exactly one $N-$labelled neighbour. The rule R_1 then consists in "cutting" a pendant vertex in the tree (since the forbidden context ensures that this vertex has no other $N-$labelled neighbour) by giving it a $F-$label. Thus, if (G, λ) is a labelled tree where all vertices are $N-$labelled and all edges are $0-$labelled then this cutting procedure leads to a unique $N-$labelled vertex which becomes elected thanks to the rule R_2. It is not difficult to observe that every vertex in the tree may be elected by this algorithm.

3 Products of Relabelling Systems

In this section, we will introduce the product of two relabelling systems. This operation will be used to add termination detection to a distributed algorithm. The main idea is to apply this product to combine two relabelling systems (one encoding the distributed algorithm, and another encoding a termination detection algorithm).

Definition 10. *Let $G = (V, E)$ be a graph. Let L_1 and L_2 be two finite alphabets and let λ_1 and λ_2 be two labelling functions. The result of the product of the labellings λ_1 and λ_2, denoted $\lambda_1 \times \lambda_2$, is defined as follows :*
$\lambda_1 \times \lambda_2 : V \cup E \longrightarrow L_1 \times L_2$.
$\lambda_1 \times \lambda_2(v) = (\lambda_1(v), \lambda_2(v))$.

Let $\lambda = \lambda_1 \times \lambda_2$. Without loss of generality, we will extend the labelling function λ_1 (resp. λ_2) to λ as follows. We will write $\lambda(1)$ (resp. $\lambda(2)$) for the projection of the ith component of the codomain of λ, i.e. $\lambda(i)\colon V \cup E \to L_i$ with $\lambda(i)(x) = y_i$ for $\lambda(x) = (y_1, y_2)$. In fact, sometimes we need to handle rather the labelling of only one system within the product.

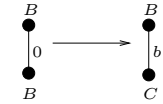
Definition 11. *Let $\mathcal{R}_1 = (L_1, I_1, P_1)$ and $\mathcal{R}_2 = (L_2, I_2, P_2)$ be two relabelling systems. The product of $\mathcal{R}_1$ and $\mathcal{R}_2$ is the relabelling system $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_2$ defined by (L, I, P) where $L = L_1 \times L_2$, $I = I_1 \times I_2$ and $P = P_1 \cup P_2$.*

The set of relabelling rules of $\mathcal{R}$ is the union of the relabelling rules of $\mathcal{R}_1$ and $\mathcal{R}_2$ by extending the labelling. That is, for every rule $R_1 = (G_{\mathcal{R}_1}, \lambda_{\mathcal{R}_1}, \lambda'_{\mathcal{R}_1})$ in $\mathcal{R}_1$, the relabelling system $\mathcal{R}$ contains all rules $R = (G_{\mathcal{R}}, \lambda_{\mathcal{R}}, \lambda'_{\mathcal{R}})$ with $G_{\mathcal{R}} = G_{\mathcal{R}_1}$,

$\lambda_{\mathcal{R}}(1) = \lambda_{\mathcal{R}_1}$ and $\lambda'_{\mathcal{R}}(1) = \lambda'_{\mathcal{R}_1}$. Similarly, the rules of $\mathcal{R}_2$ are extended to be added to $\mathcal{R}$.

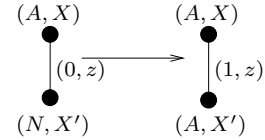
Example 4. Let us illustrate the previous definition by an example. Consider the following relabelling system $\mathcal{S} = (L_4, I_4, P_4)$ where $L_4 = \{B, C, 0, b\}$, $I_4 = \{B, 0\}$ and P_4 contains the following relabelling rule:

R :



Now, the product $\mathcal{R}_1 \times \mathcal{S}$, where $\mathcal{R}_1$ is the relabelling system given in Example 1, is the relabelling system (L, I, P) where $L = L_1 \times L_4$, $I = I_1 \times I_4$ and P contains the following relabelling rules:

R_1 :



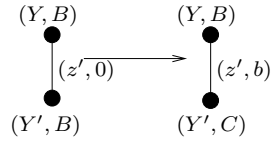
Where $X, X' \in \{B, C\}$,

$Y, Y' \in \{A, N\}$,

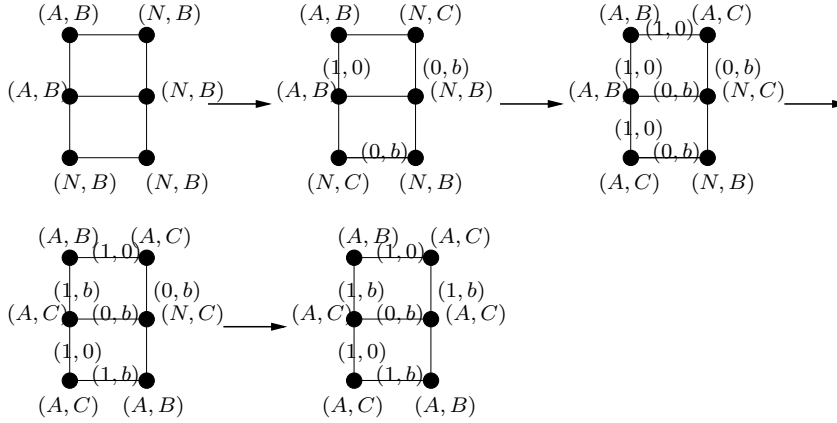
$z \in \{0, b\}$,

$z' \in \{0, 1\}$.

R_2 :



An example of execution of this system is given by the following figure.



Lemma 1. *Let $\mathcal{R}_1 = (L_1, I_1, P_1)$ and $\mathcal{R}_2 = (L_2, I_2, P_2)$ be two relabelling systems and let $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_2$ be the product of $\mathcal{R}_1$ and $\mathcal{R}_2$. Whenever $\mathcal{R}_1$ and $\mathcal{R}_2$ terminate, so does $\mathcal{R}$.*

Lemma 2. *Let $\mathcal{R}_1 = (L_1, I_1, P_1)$ and $\mathcal{R}_2 = (L_2, I_2, P_2)$ be two relabelling systems and let $\mathcal{R} = \mathcal{R}_1 \times \mathcal{R}_2$ be the product of $\mathcal{R}_1$ and $\mathcal{R}_2$. The set of invariants of $\mathcal{R}$ is the union of the sets of invariants of the two relabelling systems $\mathcal{R}_1$ and $\mathcal{R}_2$.*

4 Termination Detection of a Graph Relabelling System

4.1 The Termination Detection Problem

Consider a distributed system where processors communicate only by message passing. In such a system, a processor exchanges messages exclusively with its neighbours. Therefore, it is not trivial for a processor to have an up-to-date knowledge on the global state of the system, and to decide whether a distributed computation has finished. There are two types of termination. In an implicitly terminating algorithm, each execution is finite and in the last state of the execution each node has the correct result. However, the nodes are not aware that their state is the last one in the execution. Termination is said to be explicit in a process if that process is in a terminal configuration and its state is terminal. There were many proposals for *termination detection* algorithms: such algorithms transform implicitly into explicitly terminating algorithms. Several conditions were found to allow such algorithms and for each of these conditions a specific algorithm was given (see [16]). In this paper, by termination we mean explicit termination.

In the next section, we give a solution to make a graph relabelling system detect locally the global termination. We use termination detection techniques such as the Dijkstra-Scholten algorithm [7], or the algorithm to detect stable property [18]. In fact, we reduce the problem of termination detection to a product of two relabelling systems. We start by a description of some termination detection algorithms expressed as relabelling systems. Then we will show how to use them for detecting termination of other distributed algorithms.

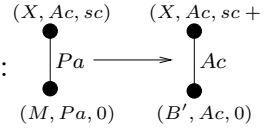
4.2 Examples of Termination Detection Algorithms

In this section, we present some distributed algorithms which detect the global termination of a graph relabelling system.

The Dijkstra-Scholten Algorithm. Consider a network of processors modeled by a graph. Each processor, represented by a vertex, is always in one of two states *active* or *passive*. The processors communicate solely by messages. Some of these messages are referred to as activation messages; they may set the receiver into an active state. An active processor maintains a count of the number of activations it has sent which have not yet been acknowledged. Initially, exactly one processor is active; then it activates its neighbours. The set of active nodes forms a dynamic tree where the root is the initial active node, and a node is the father of the nodes it activates. A leaf node is deleted from the tree if it has no active descendants; i.e., it has received acknowledgments for the activation messages it has sent. In this case, it sends a signal to acknowledge the activation to its father who will decrease its own count of active sons. We will call this tree the *control tree*. The computation terminates when the root has no active descendants.

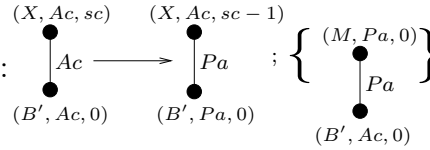
We give now a relabelling system to encode this algorithm. To do so, we equip each vertex of the graph with a counter (say sc) and with a flag describing its activity status (say a value in $\{Ac, Pa\}$). Hence, a vertex will be labelled by a triple (X, Y, sc), where X is its label M, B, B'; Y is its activity status, and sc the counter described above. We will use the label M for vertices which have never been activated, B for the initial vertex and B' for vertices beeing active at least once. An edge will be labelled by a label in $\{Ac, Pa\}$ indicating whether or not it belongs to the control tree. Initially, each edge is labelled Pa and each node labelled $(M, Pa, 0)$; a distinguished node is labelled $(B, Ac, 0)$ will be the root of the control tree. This node initializes the algorithm. We assume that the initial graph is finite. In this case, the value of the counter sc for each vertex is finite. Let $\mathcal{R}_5 = (L_5, I_5, P_5)$ be the relabelling system defined by $L_5 = \{\{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{Pa, Ac\}$, $I_5 = \{(B, Ac, 0), (M, Pa, 0), Pa\}$, and P_5 consisting of the following relabelling rules:

R_1 :



; $X \in \{B, B'\}$

R_2 :



To prove the correctness of this algorithm, we give the following invariants. Let $G(V, E, \lambda)$ be a connected labelled graph such that a distinguished vertex is labelled $(B, Ac, 0)$, all other vertices are labelled $(M, Pa, 0)$ and every edge is labelled Pa. Let $G'(V, E, \lambda')$ be a graph such that: $G(V, E, \lambda) \xrightarrow[\mathcal{R}]{*} G'(V, E, \lambda')$. The graph $G'(V, E, \lambda')$ satisfies:

1. All edges incident to a $(M, Pa, 0)$-labelled vertex are labelled Pa.
2. A (B, Ac, sc)-labelled vertex has at least Ac-labelled incident edge, $(sc \not= 0)$.
3. The subgraph induced by the edges labelled Ac has no cycle.
4. The (B, Ac, sc) and (B', Ac, sc') labelled vertices are connected by Ac-labelled edges.
5. A $(X, Pa, 0)$-labelled vertex has no $(M, Pa, 0)$-labelled neighbours, where $X \in \{B, B'\}$.
6. If G' be an irreducible graph obtained from G then all vertices are $(B', Pa, 0)$-labelled except one are $(B, Pa, 0)$-labelled.

The detection of termination of the algorithm occurs as soon as the counter (sc) of the root becomes zero (0).

Other examples of termination detection algorithms can be described by graph relabelling systems [9]. These include in particular the Dijkstra-Feijen-Van Gasteren algorithm [6].

SSP. We describe the algorithm by Szymanski, Shi and Prywes (the SSP algorithm for short) [18].

We consider a distributed algorithm which terminates when all processes reach their local termination conditions. Each process is able to determine only its own termination condition. The SSP algorithm detects an instant in which the entire computation is achieved.

Let G be a graph. To each node v are associated a predicate $P(v)$ and an integer $a(v)$. Initially $P(v)$ is false and $a(v)$ is equal to -1. Transformations of the value of $a(v)$ are defined by the following rules.

Each local computation acts on the integer $a(v_0)$ associated to the vertex v_0; the new value of $a(v_0)$ depends on values associated with the neighbors of v. More precisely, let v_0 be a vertex and let $\{v_1, ..., v_d\}$ the set of vertices adjacent to v_0.

We consider in this section the following assumption. For each node v, the value $P(v)$ eventually becomes true and remains true forever.

- If $P(v_0) = false$ then $a(v_0) = -1$;
- if $P(v_0) = true$ then $a(v_0) = 1 + Min\{a(v_k) \mid 0 \leq k \leq d\}$.

4.3 Composing Graph Relabelling Systems to Detect the Global Termination

Let $\mathcal{R}_{\mathcal{A}} = (L_{\mathcal{A}}, I_{\mathcal{A}}, P_{\mathcal{A}})$ be a graph relabelling system encoding a distributed algorithm for which termination detection will be added. We assume that within a finite number of steps, each vertex locally terminates; that is, no rule of $P_{\mathcal{A}}$ can be applied. Note that such a configuration can be easily tested by using forbidden contexts consisting of the set of rules of $P_{\mathcal{A}}$. For the relabelling system in Example 1 of Section 2, the local termination for a vertex is reached if it is labelled A and its neighbours are labelled A. Our aim is to apply the termination detection algorithm on the vertices being in a terminal state to build a global termination detection. More precisely, let $\mathcal{R}_{\mathcal{T}} = (L_{\mathcal{T}}, I_{\mathcal{T}}, P_{\mathcal{T}})$ be a relabelling system encoding a termination detection algorithm. Assume that the rule of $P_{\mathcal{T}}$ changing the state of a processor from an active to a passive state (or from *False* state to *True* State like in the SSP algorithm) is R_{Pa}. In order to detect the global termination of $\mathcal{R}_{\mathcal{A}}$ by using $\mathcal{R}_{\mathcal{T}}$, we build the product $\mathcal{R}_{\mathcal{A}} \times \mathcal{R}_{\mathcal{T}}$ and add to the rule R_{Pa} a set of forbidden contexts indicating the local termination of the algorithm for the current vertex (no rule of the algorithm involving the current vertex will be applied). We denote the resulting relabelling system by $\mathcal{R}_{\mathcal{A}} \otimes \mathcal{R}_{\mathcal{T}}$.

Theorem 1. *Let $\mathcal{R}_{\mathcal{A}} = (L_{\mathcal{A}}, I_{\mathcal{A}}, P_{\mathcal{A}})$ be a graph relabelling system encoding a distributed algorithm $\mathcal{A}$. Let $\mathcal{R}_{\mathcal{T}} = (L_{\mathcal{T}}, I_{\mathcal{T}}, P_{\mathcal{T}})$ be a relabelling system encoding a termination detection algorithm T. The relabelling system $\mathcal{R}_{\mathcal{TA}} = \mathcal{R}_{\mathcal{A}} \otimes \mathcal{R}_{\mathcal{T}}$ encodes the algorithm $\mathcal{A}$ with the termination detection property.*

Proof: We sketch the proof and use the following examples to illustrate this method. Termination and correctness of $\mathcal{R}_{\mathcal{TA}}$ are obvious, since they can be

obtained from those of $\mathcal{R}_{\mathcal{A}}$ and $\mathcal{R}_{\mathcal{T}}$. Now, to prove the termination detection property, it suffices to observe that an "active" processor becomes "passive" only if it terminates locally the execution of $\mathcal{A}$ (no rule of $\mathcal{R}_{\mathcal{A}}$ is applied). In such a case, the termination detection algorithm is executed (in fact, the rule R_{Pa} is applied). Since $\mathcal{R}_{\mathcal{T}}$ is expected to have the termination detection property, so is $\mathcal{R}_{\mathcal{T}\mathcal{A}}$.

Computation of a Spanning Tree with the Dijkstra-Scholten Termination Detection. Now, given a relabelling system encoding a distributed algorithm which is initialized by exactly one vertex, to detect termination by the previous technique, one has to embed the relabelling computation into the control tree. A node will be passive if it has locally terminated (no rule can be applied on its context). The rules of the relabelling system are mapped using the new triple labelling. In order to keep up with the control tree of the termination algorithm, a relabelling rule encoding local termination and acknowledgment must be added. The relabelling system in Example 1 of Section 2.2 combined with Dijkstra termination techniques becomes the relabelling system with forbidden contexts $\mathcal{R}_6 = (L_6, I_6, P_6)$, defined by $L_6 = \{\{A, A', N\} \times \{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{\{0, 1\} \times \{Ac, Pa\}\}$, $I_6 = \{(A, M, Ac, 0), (N, B, Ac, 0), (N, M, Pa, 0), (0, Pa)\}$, and P_6 consisting of the following relabelling rules:

$R_1 : \begin{array}{c} (A, X, AC, sc) \\ PA \mid 0 \\ (N, X', AC', sc') \end{array} \longrightarrow \begin{array}{c} (A, X, AC, sc) \\ PA \mid 1 \\ (A, X', AC', sc') \end{array}$

$R_2 : \begin{array}{c} (Y, Z, Ac, sc) \\ Pa \mid S \\ (Y, M, Pa, 0) \end{array} \longrightarrow \begin{array}{c} (Y, Z, Ac, sc+1) \\ Ac \mid S \\ (Y, B', Ac, 0) \end{array}$

$R_3 : \begin{array}{c} (Y, Z, Ac, sc) \\ Ac \mid S \\ (A, B', Ac, 0) \end{array} \longrightarrow \begin{array}{c} (Y, Z, Ac, sc-1) \\ Pa \mid S \\ (A, B', Pa, 0) \end{array}$

Where $X, X' \in \{B, B', M\}$,
$Y, Y' \in \{A, N\}$,
$Z \in \{B, B'\}$,
$AC, AC' \in \{Ac, Pa\}$,
$PA \in \{Ac, Pa\}$,
$S, S' \in \{0, 1\}$.

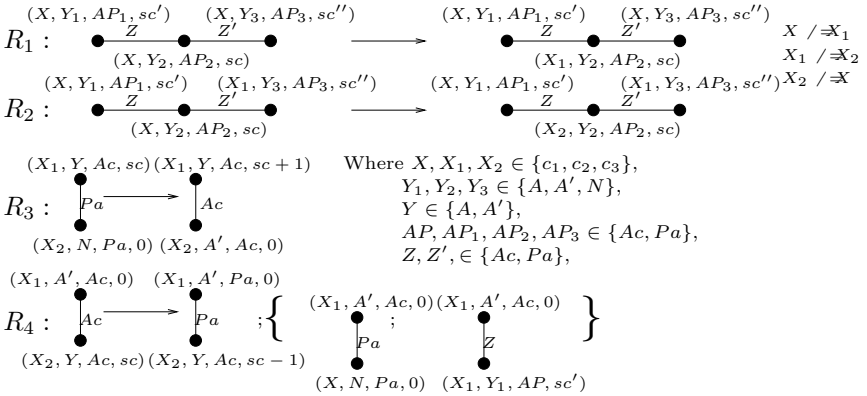
; $\left\{ \begin{array}{c} (Y, M, Pa, 0) \\ Pa \mid S' \\ (A, B', Ac, 0) \end{array} , \begin{array}{c} (N, Z, AC, sc') \\ PA \mid 0 \\ (A, B', Ac, 0) \end{array} \right\}$

Note that we give here a simplified system of the combination of the relabelling system of Section 2 with that of the Dijkstra-Scholten relabelling system. It is a relabelling system with forbidden contexts. The rules R_1 and R_2 are similar to the original ones, however, the rule R_3 has another forbidden context as explained previously. A node locally terminates if it has no neighbour labelled N which has not been yet active. Such a node will be set in passive status, and will send an acknowledgment to its father if its counter is null. In the relabelling rule, we just decrement the counter sc of the father. The whole computation terminates if the root becomes passive and has no active descendants.

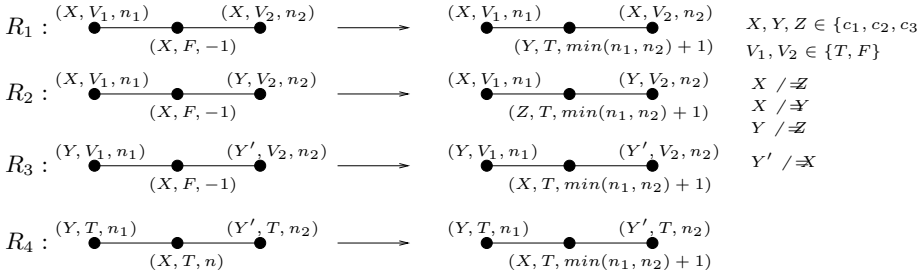
Theorem 2. *The relabelling system $\mathcal{R}_6$ computes a spanning tree. This algorithm has the property of detection of the global termination.*

The proof of this theorem is based on the invariants given for the Dijkstra-Scholten algorithm and those of Example 1.

3-Coloration of a Ring with the Dijkstra-Scholten termination Detection. Consider the relabelling system of Section 2. It colors a ring with 3 colors, but it does not detect termination. Now, we show by the method described before how to add the termination detection. The local termination detection property is that a node with the label X has no neighbour labelled X, where $X \in \{c_1, c_2, c_3\}$. We will add this property to the second rule of the Dijkstra-Scholten Algorithm. The obtained algorithm is the following. Let $\mathcal{R}_7 = (L_7, I_7, P_7)$, defined by $L_7 = \{\{c_1, c_2, c_3\} \times \{B, B', M\} \times \{Ac, Pa\} \times [0..n]\} \cup \{Ac, Pa\}$, $I_7 = \{(X, B, Ac, 0), (X', M, Pa, 0), Pa\}$, where X and $X' \in \{c_1, c_2, c_3\}$, and P_7 consisting of the following relabelling rules:



3-Coloration of a Ring with the Detection of Stable Properties. We consider the same example as before but produced with the SSP algorithm [18] and moreover each vertex knows the size of the graph. Let $\mathcal{R}_8 = (L_8, I_8, P_8)$, defined by $L_8 = \{\{c_1, c_2, c_3\} \times \{T, F\} \times [-1..n]\}$, $I_8 = \{(X, F, -1)\}$, where $X \in \{c_1, c_2, c_3\}$, and P_8 consisting of the following relabelling rules:



The algorithm terminates if each node has the n-label greater or equal to the diameter of the graph, i.e. $n \geq S$, where S is the diameter of the graph.

5 Generalization of Termination Detection

In this section, we give general results obtained using the SSP algorithm to detect locally the global termination. First, we consider the case where each vertex is able to detect locally its own local termination, and where the diameter or the size of the graph is known. Then, we consider a variant of the SSP algorithm and give a more general result.

5.1 From Local Termination Detection to Global Termination Detection

For the last example (the 3-Coloration), it is clear that when a vertex is correctly colored, it sets the predicate P to the value *true*. In such a case, it detects locally the local termination. The predicate P remains true until the local detection of the global termination. More generally, the SSP's algorithm can transform a relabelling system for which each vertex is able to detect its local termination into another relabelling system with the local detection of the global termination.

Theorem 3. *Let $\mathcal{R}$ be a graph relabelling system. Suppose that for an initial graph with known size, each vertex is able to detect its local termination. The system $\mathcal{R}$ can be transformed to a graph relabelling system with the local detection of the global termination.*

The proof is based on the SSP algorithm, and is similar to the 3-Coloration example. By assigning true predicates to vertices which detect the local termination, and by spreading the values of the integers $a(v)$, it is possible to detect the global termination.

5.2 Termination Detection by a Variant of the SSP Algorithm

For the previous Theorem, we assumed that each vertex is able to detect its own local termination. We can weaken this assumption by dealing only with the applicability or not of relabelling rules. That is, during the relabelling process, if for a vertex v none of the rules is applicable to v (and its context), then the value of the predicate P for v is set temporarily to *true*. The computation process continues like for the previous case. Now, if later on, the vertex v is relabelled (for instance by one of its neighbours), then the value of the predicate P is reinitialized to *false* for v and $a(v)$ is set to -1. Again, if P becomes true, then $a(v)$ is set to 0 and so on. Therefore, we use a generalization of the SSP algorithm by dropping the last assumption. We can prove that it is possible to detect locally the global termination.

Theorem 4. *Let $\mathcal{R}$ be a graph relabelling system. Suppose that the diameter of the initial graph is known. The system $\mathcal{R}$ can be transformed to a graph relabelling system with the local detection of the global termination.*

References

1. M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, and A. Sellami. Visualization of distributed algorithms based on labeled rewriting systems. In *Second International Workshop on Graph Transformation and Visual Modeling Techniques, Crete, Greece*, July 12-13, 2001.
2. M. Bauderon, Y. Métivier, M. Mosbah, and A. Sellami. From local computations to asynchronous message passing systems. Technical Report RR-1271-02, LaBRI, 2002.
3. M. Billaud, P. Lafon, Y. Métivier, and E. Sopena. Graph rewriting systems with priorities. *Lecture notes in computer science*, 411:94–106, 1989.
4. K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transations on Computer Systems*, 3(1):63–75, 1985.
5. K. Mani Chandy, Laura M. Haas, and Jayadev Misra. Distributed deadlock detection. *ACM Transactions on Computer Systems*, 1(2):144–156, May 1983.
6. Edsger W. Dijkstra, W. H. J. Feijen, and A. J. M. van Gasteren. Derivation of a termination detection algorithm for distributed computations. *Information Processing Letters*, 16(5):217–219, 1983.
7. E.W. Dijkstra and C.S. Sholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(1):1–4, 1980.
8. S. Dolev. *Self-stabilization*. The MIT Press, 2000.
9. E. Godard, Y. Métivier, M. Mosbah, and A. Sellami. Termination detection of distributed algorithms by graph relabelling systems. Technical report, LaBRI-University of Bordeaux 1, 2002.
10. I. Litovsky and Y. Métivier. Computing trees with graph rewriting systems with priorities. *Tree automata and languages*, pages 115–139, 1992.
11. I. Litovsky and Y. Métivier. Computing with graph rewriting systems with priorities. *Theoret. Comput. Sci.*, 115:191–224, 1993.
12. I. Litovsky, Y. Métivier, and E. Sopena. Different local controls for graph relabelling systems. *Math. Syst. Theory*, 28:41–65, 1995.
13. Y. Métivier, M. Mosbah, and A. Sellami. Proving distributed algorithms by graph relabeling systems: Examples of trees in networks with processor identities. In *Applied Graph Transformations*, Grenoble, April, 2002.
14. Y. Métivier and G. Tel. Termination detection and universal graph reconstruction. In *International Colloquium on structural information and communication complexity*, pages 237–251. Carleton scientific press, 2000.
15. M. Mosbah and A. Sellami. Visidia: A tool for the visualization and similation of distributed algorithms. *http://www.labri.fr/visidia/*.
16. G. Tel. *Introduction to distributed algorithms*. Cambridge University Press, 2000.
17. Gerard Tel and Friedemann Mattern. The derivation of distributed termination detection algorithms from garbage collection schemes. *ACM Transactions on Programming Languages and Systems*, 15(1):1–35, January 1993.
18. B. Szymanski Y. Shi and N. Prywes. Terminating iterative solutions of simultaneous equations in distributed message passing systems. In *4th International Conference on Distributed Computing Systems*, pages 287–292, 1985.

Graph Transformation with Time: Causality and Logical Clocks

Szilvia Gyapay[1], Reiko Heckel[2], and Dániel Varró[1]

[1] Dept. of Measurement and Information Systems,
Budapest University of Technology and Economics
H-1521 Budapest, Hungary
{gyapay,varro}@mit.bme.hu
[2] Dept. of Math. and Comp. Science, University of Paderborn
D-33095 Paderborn, Germany
reiko@upb.de

Abstract. Following TER nets, an approach to the modelling of time in high-level Petri nets, we propose a model of time within (attributed) graph transformation systems where logical clocks are represented as distinguished node attributes. Corresponding axioms for the time model in TER nets are generalised to graph transformation systems and semantic variations are discussed. They are summarised by a general theorem ensuring the consistency of temporal order and casual dependencies.
The resulting notions of *typed graph transformation with time* specialise the algebraic double-pushout (DPO) approach to typed graph transformation. In particular, the concurrency theory of the DPO approach can be used in the transfer of the basic theory of TER nets.

1 Introduction

Recently, a number of authors have advocated the use of graph transformation as a semantic framework for visual modelling techniques both in computer science and engineering (see, e.g., the contributions in [4, 3]). In many such techniques, the modelling of time plays a relevant role. In particular, techniques for embedded and safety critical systems make heavy use of concepts like timeouts, timing constraints, delays, etc., and correctness with respect to these issues is critical to the successful operation of these systems. At the same time, those are exactly the systems where, due to the high penalty of failures, formally based modelling and verification techniques are most successful. Therefore, neglecting the time aspect in the semantics of visual modelling techniques, we disregard one of the crucial aspects of modelling.

So far, the theory of graph transformation provides no support for the modelling of time in a way which would allow for *quantified* statements like "this action takes 200ms of time" or "this message will only be accepted within the next three seconds", etc. However, from a more abstract, *qualitative* point of view we can speak of temporal and causal ordering of actions thus abstracting from actual clock and timeout values. Particularly relevant in this context is the

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 120–134, 2002.
© Springer-Verlag Berlin Heidelberg 2002

theory of concurrency of graph transformation, see [14, 6, 1] or [2] for a recent survey.

It is the objective of this paper to propose a quantitative model of time within graph transformation which builds on this more abstract qualitative model. Therefore, we will not add time concepts on top of an existing graph transformation approach, but we show how, in particular, typed graph transformation systems in the double-pushout (DPO) approach [6] can be extended *from within* with a notion of time. This allows both the straightforward transfer of theoretical results and the reuse of existing tools.

In a recent preliminary paper [10], we have already outlined our approach, proposing several alternative definitions and discussing their consequences with respect to the existence of a globally time-ordered sequence of transformations. Such property is desirable because it witnesses the consistency of time values attached to vertices with the causal dependencies between steps. In this paper, we refine the alternatives discussed in [10] and prove a general condition on graph transformation systems with time ensuring the desired consistency property, thus effectively solving the problem posed in [10] in the most general case.

The following section outlines our general approach of the problem, which is motivated by a corresponding development in Petri nets, briefly to be reviewed in Section 3. Graph transformation with time is introduced and investigated in Section 4 while Section 5 concludes the paper.

2 From Nets to Graph Transformation, with Time

When trying to incorporate time concepts into graph transformation, it is inspiring to study the representation of time in Petri nets. Nets are formally and conceptually close to graph transformation systems which allows for the transfer of concepts and solutions. This has already happened for relevant parts of the concurrency theory of nets which, as mentioned above, provides a qualitative model of time based on the causal ordering of actions.

In particular, we will follow the approach of time ER nets [11]. These are simple high-level nets which introduce time as a distinguished data type. Then, time values can be associated with individual tokens, read and manipulated like other token attributes when firing transitions. In order to ensure meaningful behaviour (like preventing time from going backwards) constraints are imposed which can be checked for a given net. The advantage of this approach with respect to our aims is the fact that time is modelled within the formalism rather than adding it on top as a new formal concept.

Based on the correspondence of Petri nets and (typed) graph transformation, which regards Petri nets as rewriting systems on multi-sets of vertices [5], we can derive a model of time within typed graph transformation systems with attributes. The correspondence is visualised in Table 1. Besides (low-level) place-transition nets and typed graph transformation systems, it relates (high-level) environment-relationship nets to typed graph transformation with attributes. This relationship, which has first been observed in the case of algebraic high-

Table 1. Corresponding Petri net and graph transformation variants

	Petri nets	graph transformation systems
low-level	PT nets	typed graph transformation (TGT)
high-level	ER nets	typed graph transformation with attributes (TGTA)
with time	TER nets	typed graph transformation with time (TGTT)

level nets [8] and attributed graph transformation [17] in [18], shall enable us to transfer the modelling of time in time ER nets to typed graph transformation with attributes.

Next, we review time environment-relationship (TER) nets [11] in order to prepare for the transfer to typed graph transformation systems in Section 4.

3 Modelling Time in Petri Nets

There are many proposals for adding time to Petri nets. In this paper we concentrate on one of them, time ER nets [11], which is chosen for its general approach of considering time as a token attribute with particular behaviour, rather than as an entirely new concept. As a consequence, time ER nets are a special case of ER nets.

3.1 ER Nets

ER (environment-relationship) nets are high-level Petri nets (with the usual net topology) where tokens are environments, i.e., partial functions $e : ID \to V$ associating attribute values from a given set V to attribute identifiers from a given set ID. A marking m is a multi-set of environments (tokens).

To each transition t of the net with pre-domain $p_1 \dots p_n$ and post-domain $p'_1 \dots p'_m$, an action $\alpha(t) \in Env^n \times Env^m$ is associated. The projection of $\alpha(t)$ to the pre-domain represents the firing condition, i.e., a predicate on the tokens in the given marking which controls the enabledness of the transition. If the transition is enabled, i.e., in the given marking m there exist tokens satisfying the predicate, the action relation determines possible successor markings.

Formally, a transition t is enabled in a marking m if there exists a tuple $\langle pre, post\rangle \in \alpha(t)$ such that $pre \leq m$ (in the sense of multiset inclusion). Fixing this tuple, the successor marking m' is computed, as usual, by $m' = (m - pre) + post$, and this firing step is denoted by $m[t(pre, post)\rangle m'$. A firing sequence of $s = m_0[t_1(pre_1, post_1)\rangle \dots [t_{k-1}(pre_{k-1}, post_{k-1})\rangle m_k$ is just a sequence of firing steps adjacent to each other.

3.2 Time ER Nets

Time is integrated into ER nets by means of a special attribute, called *chronos*, representing the time of creation of the token as a time stamp. Constraints on the time stamps of both (i) given tokens and (ii) tokens that are produced can be

specified by the action relation associated to transitions. To provide a meaningful model of time, action relations have to satisfy the following axioms with respect to *chronos* values [11].

Axiom 1: Local monotonicity For any firing, the time stamps of tokens produced by the firing can not be smaller than time stamps of tokens removed by the firing.

Axiom 2: Uniform time stamps For any firing $m[t(pre, post)\rangle m'$ all time stamps of tokens in *post* have the same value, called the *time of the firing*.

Axiom 3: Firing sequence monotonicity For any firing sequence s, firing times should be monotonically nondecreasing with respect to their occurrence in s.

The first two axioms can be checked locally based on the action relationships of transitions. For the third axiom, it is shown in [11] that every sequence s where all steps satisfy Axioms 1 and 2 is *permutation equivalent* to a sequence s' where also Axiom 3 is valid. Here, permutation equivalence is the equivalence on firing sequences induced by swapping independent steps. Thus, any firing sequence can be viewed as denoting a representative, which satisfies Axiom 3.

It shall be observed that TER nets are a proper subset of ER nets, i.e., the formalism is not extended but specialised. Next, we use the correspondence between graph transformation and Petri nets to transfer this approach of adding time to typed graph transformation systems.

4 Modelling Time in Graph Transformation Systems

Typed graph transformation systems provide a rich theory of concurrency generalising that of Petri nets [2]. In order to represent time as an attribute value, a notion of typed graph transformation with attributes is required. We propose an integration of the two concepts (types and attributes) which presents attribute values as vertices and attributes as edges, thus formalising typed graph transformation with attributes as a special case of typed graph transformation.

Next, we give a light-weight (set-theoretic) presentation of the categorical DPO approach [9] to the transformation of typed graphs [6].

4.1 Typed Graph Transformation

In typed graph transformation, graphs occur at two levels: the type level and the instance level [6]. A fixed *type graph* TG (which may be thought of as an abstract representation of a class diagram) determines a set of *instance graphs* $\langle G, g : G \to TG\rangle$ which are equipped with a structure-preserving mapping g to the type graph (formally expressed as a *graph homomorphism*).

A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges which appear in both L and R are connected to the same vertices in

both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

A *graph transformation* from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

A transformation sequence $G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n$ is a sequence of consecutive transformation steps.

On transformation sequences, a notion of equivalence is defined which generalises the permutation equivalence on firing sequences: two sequences are equivalent if they can be obtained from each other by repeatedly swapping independent transformation steps. This equivalence has been formalised by the notion of *shift-equivalence* [14] which is based on the following notion of independence of graph transformations. Two transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1 \overset{p_2(o_2)}{\Longrightarrow} X$ are *independent* if the occurrences $o_1(R_1)$ of the right-hand side of p_1 and $o_2(L_2)$ of the left-hand side of p_2 do only overlap in objects that are preserved by both steps, formally $o_1(R_1) \cap o_2(L_2) \subseteq o_1(L_1 \cap R_1) \cap o_2(L_2 \cap R_2)$. This is more sophisticated than the notion of independent firings of transitions which are required to use entirely disjoint resources.

4.2 Typed Graph Transformation with Attributes

Assuming a set of data type symbols S, a *type graph with attribute declarations* (based on S) is a graph TG whose set of vertices TG_V contains S. Therefore, data type symbols are vertex types so that edges, representing attribute declarations, may be drawn towards them from ordinary vertices. This is compatible with notions of attributed graphs, like [17], where attribute carriers are used to relate graph elements and data values. Notice, however, that we limit ourselves to attributed vertices, and that we do not extend but refine the notion of graph.

Given a data domain D_s for every data type symbol s, an *instance graph with attributes* over the type graph TG is an instance graph $\langle G, g : G \to TG \rangle$ over TG (in the above sense) such that $g^{-1}(s) = D_s \subseteq G_V$. Therefore, all vertices $x \in G_V$ with $g(x) \in S$ represent attribute values which may or may not be referenced by an edge from another vertex. As a consequence, instance graphs will be usually infinite. E.g., if the data type $\mathbb{N}$ of natural numbers is present, each $n \in \mathbb{N}$ will be a separate vertex.

Morphisms between instance graphs with attributes $\langle G, g : G \to TG \rangle$ and $\langle H, h : H \to TG \rangle$ are typed graph morphisms $f : G \to H$, i.e., graph morphisms compatible with the typing ($h \circ f = g$) and preserving the data domains; formally

$f|_S = id_{G|_S}$, if we denote by $f|_S : G|_S \to H|_S$ the restriction of f_V to vertices $x \in V$ of type $g(x) \in S$.

From this point on, all other notions, like rule, occurrence, transformation, transformation sequence, etc. are defined as in the previous subsection. Also, relevant results like the Local Church-Rosser Theorem, the Parallelism theorem, and the corresponding equivalence on transformation sequences based on shifting or swapping of independent transformations are easily transferred.

It is worth noticing that, in contrast with ER nets, attributes in our model are typed, that is, different types of nodes may have different selections of attributes. However, like in ER nets, our data types have no syntax: We only consider sets of values without explicit algebraic structure given by operations. As a consequence, we do not explicitly represent variables within rules and variable assignments as part of occurrences: A rule containing variables for attribute calculation and constraints is considered as a syntactic representation of the (possibly infinite) set of its instances where the variables and expressions are replaced by concrete values.

4.3 Typed Graph Transformation with Time

To incorporate time into typed graph transformation with attributes, we follow the approach of TER nets as discussed in Section 3. Therefore, a time data type is required as domain for time-valued attributes.

A *time data type* $T = \langle D_{time}, +, 0, \geq \rangle$ is a structure where $\geq$ is a partial order with 0 as its least element. Moreover, $\langle +, 0 \rangle$ form a monoid (that is, $+$ is associative with neutral element 0) and $+$ is monotone wrt. $\geq$. Obvious examples include natural or real numbers with the usual interpretation of the operations, but not dates in the YY:MM:DD format (e.g., due to the Y2K problem).

A *type graph with time* TG is a type graph with attribute declarations based on a set of data type symbols S that contains a special symbol *time*. An instance graph with time over TG for a given time data type $T = \langle D_{time}, +, 0, \geq \rangle$ is an instance graph $\langle G, g : G \to TG \rangle$ over TG such that the data type sort *time* is interpreted by D_{time}, that is, $D_{time} = \{x \in G_V | g(x) = time\}$. Graph morphisms are defined as before.

Given a graph transformation rule $p : L \to R$ over a type graph with time, we say that

- *p reads the chronos value c of v* if $v \in L$ has a chronos attribute of value c, that is, there exists an edge $e \in L$ with $src(e) = v$ and $tar(e) = c \in D_{time}$.
- *p writes the chronos value c of v* if $v \in R$ has a chronos attribute of value c which is not present in L, i.e., there exists an edge $e \in L$ with $src(e) = v$ and $tar(e) = c \in D_{time}$ and $e \not\in L$.

Given a transformation $G \stackrel{p(o)}{\Longrightarrow} H$ we say that $p(o)$ reads/writes the chronos value of w if there exists $v \in L \cup R$ such that $o(v) = w$ and p reads/writes the chronos value of v.

It is important to note that, writing an attribute value of a vertex v which is preserved by the rule (i.e., it belongs both to L and R) means deleting the edge

from v to the old value and creating a new link to another value. Therefore, in this case, writing implies reading the value.

The definition of graph transformation rules with time has to take into account the particular properties of time as expressed, for example, by the axioms in Section 3. The direct transfer of axioms 1 and 2 leads to the following well-formedness conditions.

A *graph transformation rule with time* is a graph transformation rule over a type graph with time satisfying the following conditions.

1. **Local monotonicity:** All chronos values written by p are higher than any of the chronos values read by p.
2. **Uniform duration:** All chronos values written by p are equal.

A graph transformation system with time is an attributed graph transformation system over a type graph with time whose rules satisfy the conditions above.

This ensure a behaviour of time which can be described informally as follows. According to condition 1 an operation or transaction specified by a rule cannot take negative time, i.e., it cannot decrease the clock values of the nodes it is applied to. Condition 2 states an assumption about atomicity of rule application, that is, all effects specified in the right-hand side are observed at the same time. Given a transformation $G \overset{p(o)}{\Longrightarrow} H$ using rule p, this time *firing time*, is denoted by $time(p(o))$.

Notice that, due to the more general nature of typed graph transformation in comparison with ER nets, there exist some additional degrees of freedom.

Existence of time-less vertex types: ER nets are untyped (that is, all tokens have (potentially) the same attributes) while in typed graph transformation we can declare dedicated attributes for every vertex type. Therefore, we do not have to assume an attribute *chronos* for all vertex types, but could leave the decision about how to distribute chronos attributes to the designer. As we consider time as a distinguished semantic concept, which should not be confused with time-valued data, we do not allow more than one chronos attribute per vertex. This does not forbid us to model additional time-valued data by ordinary attributes.

Update of chronos values for preserved vertices: The second degree of freedom comes from the (well-known) fact that graph transformations generalise Petri nets by allowing contextual rewriting: All tokens in the post-domain of a transition are newly created while in the right-hand side of a graph transformation rule there may be vertices that are preserved. This allows to leave the chronos values of vertices in $L \cap R$ unchanged, creating new timestamps only for the newly generated items.

If we take in both cases the most restrictive choice, i.e., *chronos values for all types* and *update of chronos values for all vertices in* R, we can show, in analogy with TER nets, that for each transformation sequence s using only rules that satisfy the above two conditions, there exists an equivalent sequence s' such that

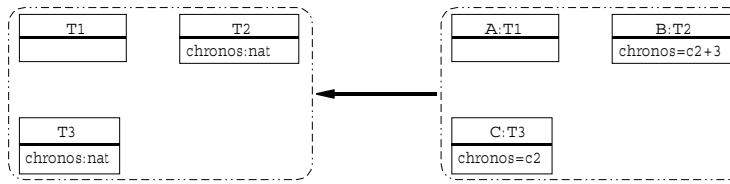


Fig. 1. Type graph (left) and instance graph (right)

s' is time-ordered, that is, time is monotonically non-decreasing as the sequence advances.

Theorem 1 (global monotonicity). *Given a graph transformation system with time $\mathcal{G}$ such that*

- *its type graph declares a chronos attribute for every vertex type*
- *its rules write the chronos values of all vertices in their right-hand sides.*

In this case, for every transformation sequence s in $\mathcal{G}$ there exists an equivalent sequence $s' = G_0 \overset{p_1(o_1)}{\Longrightarrow} \dots \overset{p_n(o_n)}{\Longrightarrow} G_n$ in $\mathcal{G}$ such that s' is time-ordered, that is, $time(p_i(o_i)) \leq time(p_{i+1}(o_{i-1}))$ for all $i \in \{0, \dots, n\}$.

Proof. As a consequence of Theorem 2 below.

This is no longer true if we use the more liberal interpretation in any of the above choices, as shown by the following example.

Example. Figure 1 shows a type graph TG and a (generic) instance graph IG of TG, respectively. The type graph defines three vertex types: $T1, T2$ and $T3$. For $T2$ and $T3$ *chronos* attributes are declared, while $T1$ has no attribute. (Our example does not need edges.)

The instance graph contains nodes $A : T1$ (i.e., a $T1$-typed node named A), $B : T2$, and $C : T3$, where C has the chronos value $c2$ and the chronos value of B is $c2 + 3$. Two rules, p_1 and p_2, are defined in Figure 2. By applying p_1, nodes $a : T1$ and $b : T2$ are matched and the chronos value of b is increased by 4 time units.

Rule p_2 requires nodes $a : T1$ and $b : T3$. The former is deleted and the time of the latter is increased by 2 units. (Note, that the use of similar names does not imply any connection between the elements of different rules.)

An application of these rules to the instance graph in Figure 1 is shown in Figure 3. Both p_1 and p_2 are applicable to the graph. Applying first p_1 and then p_2 leads to the graph in the lower right. In this sequence, first the chronos value of B is increased, and then A is deleted and the chronos value of C is increased. The occurrences and the firing times of the steps are denoted next to the arrows.

At this point, two observations are crucial. First, the two steps are not independent, that is there exists no equivalent sequence where p_1 and p_2 are applied in the reverse order. This is because $A \in (o_1(L_1 \cap R_1)) \cap o_2(L_2 \setminus R_2))$, i.e., A is

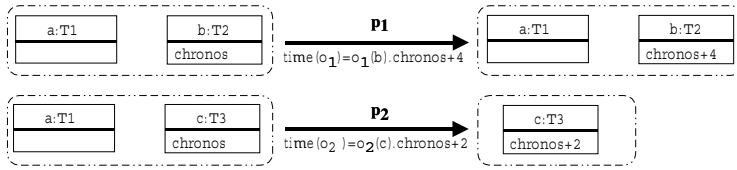


Fig. 2. Rules p_1 and p_2

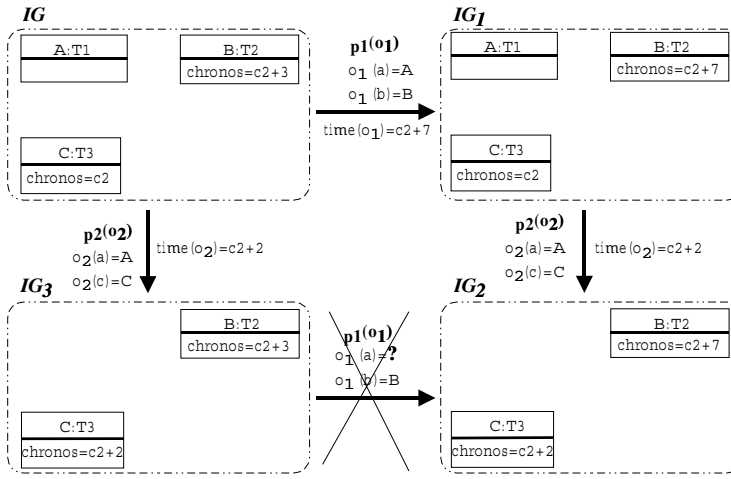


Fig. 3. Rule sequence

deleted by p_2 but required by p_1. Second, the sequence $IG \overset{p_1(o_1)}{\Longrightarrow} IG_1 \overset{p_2(o_2)}{\Longrightarrow} IG_2$ is not time-ordered because the firing time of the latter is smaller than the firing time of the first.

Note that, if A would have a chronos attribute *and* all chronos values would be updated as required by Condition 2, A should get $time(p_1(o_1))$ thus disabling its deletion at a lower time.

More generally, the problem is to ensure the consistency of causality and time in the sense that, whenever two steps are causally dependent, they must communicate their clock values. Indeed, this idea is crucial to many algorithms for establishing consistent global time in distributed systems, based on logical clocks. The next theorem formalises this statement.

Theorem 2 (global monotonicity). *Given a graph transformation system with time $\mathcal{G}$ such that for all transformations $G \overset{p_1(o_1)}{\Longrightarrow} X \overset{p_2(o_2)}{\Longrightarrow} H$ in $\mathcal{G}$ that are* not *sequentially independent, there exists a vertex $v \in o_1(R_1) \cap o_2(L_2)$ whose chronos value is written by p_1 and read by p_2. In this case, for every transformation sequence s in $\mathcal{G}$ there exists an equivalent sequence $s' = G_0 \overset{p_1(o_1)}{\Longrightarrow} \ldots \overset{p_n(o_n)}{\Longrightarrow} G_n$ in $\mathcal{G}$ such that s' is time-ordered.*

Proof. The main line of the proof is as follows.

1. Our first observation is that the fact that two transformations $G \overset{p_1(o_1)}{\Longrightarrow} X \overset{p_2(o_2)}{\Longrightarrow} H$ are *not sequentially independent* implies that they are *time ordered*, i.e., $time(p_1(o_1)) \leq time(p_2(o_2))$. This is guaranteed by the existence of a common vertex $v \in o_1(R_1) \cap o_2(l_2)$ with a chronos value written by p_1 and read by p_2, which is
   (a) *exactly* the time of transformation $p_1(o_1)$ (due to the *"uniform duration"* condition),
   (b) *at most* the time of transformation $p_2(o_2)$ (as a consequence of the *"local monotonicity"* condition).
2. Then if two transformations are *not time ordered* and they are *sequentially independent*, we swap them in the rule application sequence[1] . We continue the swap operation until no such transformation pairs can be found.
3. We state that after the *termination* of this swapping algorithm, a time ordered transformation sequence is obtained.
   (a) Let us suppose indirectly that there exist two transformations $G \overset{p_a(o_a)}{\Longrightarrow} X \overset{p_b(o_b)}{\Longrightarrow} H$ that violate the condition of time ordered sequences, i.e. $time(p_a(o_a)) > time(p_b(o_b))$.
   (b) However, if these transformations are *sequentially independent* then the algorithm in Item 2 can still be applied to them, which contradicts the assumption of termination.
   (c) On the other hand, if transformations $p_a(o_a)$ and $p_b(o_b)$ are *not sequentially independent* (but they are not time ordered by the indirect assumption), then we have a contradiction with our first observation, which established that two sequentially dependent transformations with a common vertex are always time ordered. □

Notice that the condition above can be effectively verified by checking all non-independent two-step sequences in $\mathcal{G}$ where $x = o_1(R_1) \cup o_2(L_2)$.

4.4 An Example for Time Ordered Sequences

In the sequel, our main theorem (Theorem 2) is discussed from a practical point of view on a small example of a communication system, which models *processes* sending *messages* to each other via *channels*. A message is sent via an *output* channel of a process, which *stores* the message until received via the *input* channel of the other process. The structure of our communication system is captured on the type graph of Fig. 4, while a sample system containing only two processes P_1 and P_2 with a single channel C between them is also depicted there.

We introduce the following two basic operations (see Fig. 5) that can be performed in our system.

[1] This algorithm is, in fact, conceptually similarly to the trick applied in the construction of a shift equivalent transformation sequence.

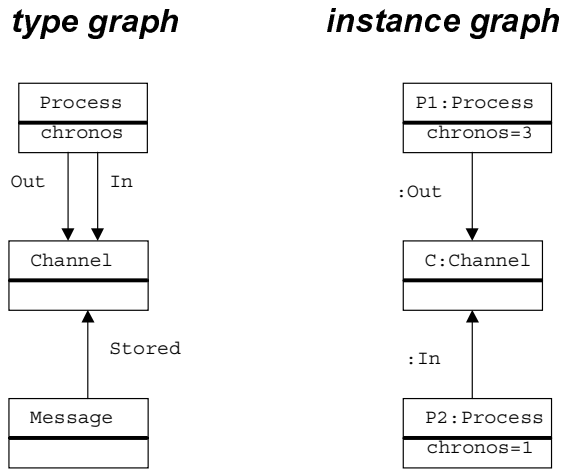


Fig. 4. A communication system: type and instance graphs

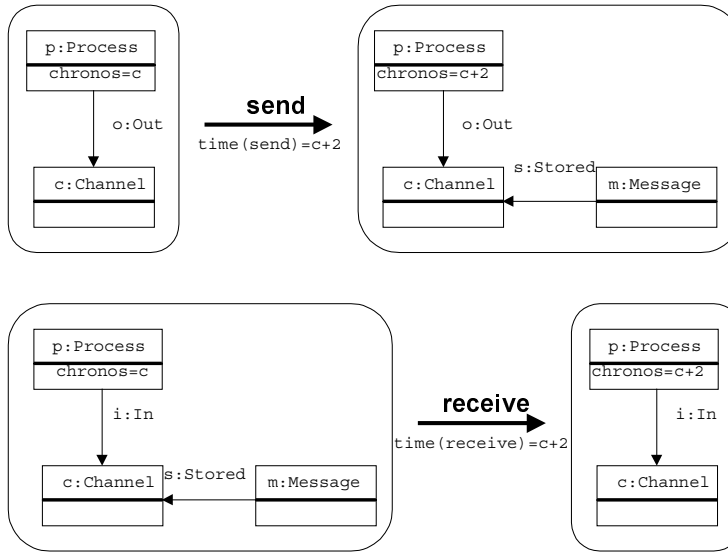


Fig. 5. Sending and receiving messages

- **Sending messages:** When process p_1 aims at sending a message, a message object m is generated and placed into the output channel c. The application of the *send* rule takes 2 time units.
- **Receiving messages:** When a message m arrives at the input port of a process p, then the process receives the message by removing the message object from the channel and destroying it afterwards. The application of *receive* rule takes 2 time units as well.

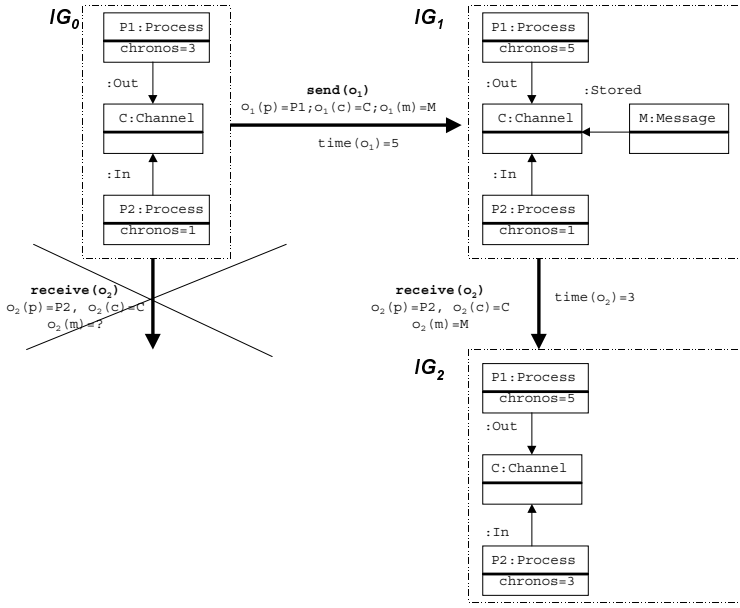


Fig. 6. A transformation violating time orderedness

One can easily check that both rules satisfy the well-formedness conditions for graph transformation rules with time. However, the *send* rule computes its time from the chronos value of the sender process P_1, while the *receive* rule takes its time from (the chronos value of) the receiver process P_2, but no timestamps are attached to messages.

This turns out to be insufficient to guarantee the existence of time ordered transformation sequences, when considering, for instance, the transformation depicted in Fig. 6. In this case, the clock of the sender process P_1 is ahead of the chronos value of the receiver process ($chronos(P_1) = 3$ vs. $chronos(P_2) = 1$). Since no timestamps are attached when a message is sent, the receiver cannot synchronise its clock to the sender when the message is processed yielding a transformation sequence that is not time ordered ($time(send) = 5$ but $time(receive) = 3$ as each operation takes 2 time units). However, we cannot swap the two transformations *send* and *receive* since they are not sequentially independent.

It can be observed how the condition of Theorem 2 is violated. While (the application of) *send* is not sequentially independent from *receive* (a message object is created by *send* and required as a precondition by *receive*), there are no graph objects with chronos values written by *send* (as *send* rule only writes the chronos of process P_1) and read by *receive* (which only reads the chronos of process P_2).

The solution to the problem is well-known: a timestamp is needed to be attached to each message (see the corresponding rule in Fig 7). In terms of

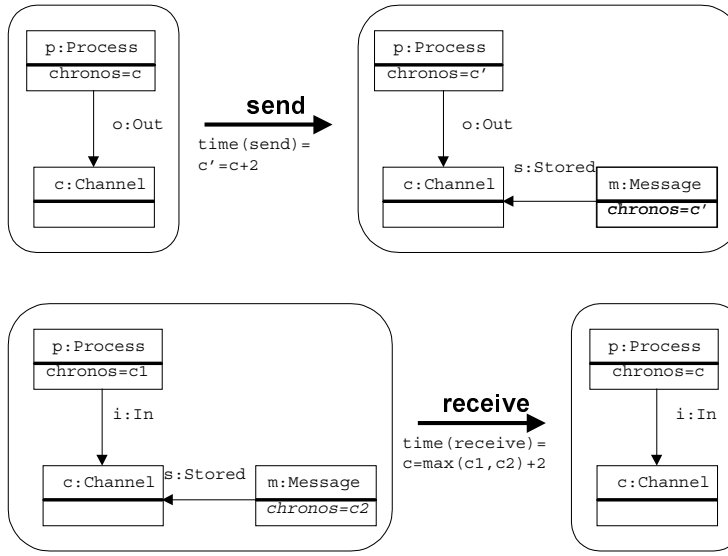


Fig. 7. Synchronisation by timestamps

graph transformation systems with time, the *send* rule needs to write the chronos attribute of the message, while *receive* rule is required to read the chronos value of the message in order to synchronise its own clock.

This time, our global monotonicity theorem trivially holds, since the chronos value of each *message* object is written by the *send* rule and read by the *receive* rule. Thus in a transformation sequence where a certain application of *send* precedes the application of *receive*, the time of *receive* cannot be less then the time of *send* due to the well-formedness conditions 1 and 2.

5 Conclusion

We have transferred the model of time within ER nets, a kind of high-level Petri nets, to graph transformation systems. The resulting notion is a special case of typed graph transformation, where certain vertices are interpreted as time values and edges towards these vertices are time-valued attributes.

We have discussed some choices and their semantic consequences leading to the establishment of a *global monotonicity theorem*, which provides a sufficient condition for the existence of time ordered transformation sequences. This theorem generalises the idea behind familiar algorithms for establishing consistent logical clocks via time stamps in distributed systems [16], which are based on more specific computational models similar to the example in the last subsection.

It requires a deeper analysis of potential applications, in particular, the use of time in diagrammatic techniques like statecharts or sequence diagrams and their existing formalisations within graph transformation [7, 12, 13, 15], to understand if our choices are the right ones.

Acknowledgement

We wish to thank Luciano Baresi for his introduction to Petri nets with time.

References

1. P. Baldan. *Modelling Concurrent Computations: from Contextual Petri Nets to Graph Grammars*. PhD thesis, Dipartimento di Informatica, Università di Pisa, 2000.
2. P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*, pages 107 – 188. World Scientific, 1999.
3. L. Baresi, M. Pezzé, and G. Taentzer, editors. *Proc. ICALP 2001 Workshop on Graph Transformation and Visual Modeling Techniques, Heraklion, Greece*, Electronic Notes in TCS. Elsevier Science, July 2001.
4. A. Corradini and R. Heckel, editors. *Proc. ICALP 2000 Workshop on Graph Transformation and Visual Modeling Techniques*, Geneva, Switzerland, July 2000. Carleton Scientific. `http://www.di.unipi.it/GT-VMT/`.
5. A. Corradini and U. Montanari. Specification of Concurrent Systems: from Petri Nets to Graph Grammars. In *Quality of Communication-Based Systems*, pages 35–52. Kluwer Academic Publishers, 1995.
6. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
7. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*, pages 323–337. Springer-Verlag, 2000.
8. H. Ehrig, J. Padberg, and L. Ribeiro. Algebraic high-level nets: Petri nets revisited. In *Recent Trends in Data Type Specification*, pages 188–206, Caldes de Malavella, Spain, 1994. Springer Verlag. Lecture Notes in Computer Science 785.
9. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
10. S. Gyapay and R. Heckel. Towards graph transformation with time. In *Proc. ETAPS'02 Workshop on Application of Graph Transformation, Grenoble, France*, 2002.
11. C. Ghezzi, D. Mandrioli, S. Morasca, and Pezzè. A unified high-level petri net formalism for time-critical systems. *IEEE Transactions on Software Engineering*, 17(2):160–172, 1991.
12. J.H. Hausmann, R. Heckel, and S. Sauer. Towards dynamic meta modeling of UML extensions: An extensible semantics for UML sequence diagrams. In M. Minas and A. Schürr, editors, *Symposium on Visual Languages and Formal Methods, IEEE Symposia on Human Computer Interaction (HCI 2001), Stresa, Italy*, Los Alamitos, CA, September 2001. IEEE Computer Society Press.
13. C. E. Hrischuk. A Model Making Automation Process (MMAP) using a graph grammar formalism. In *Theory and Application of Graph Transformations, 6th International Workshop, TAGT'98, Paderborn, Germany, Selected Papers*, volume 1764 of *LNCS*, pages 442–454. Springer-Verlag, 2000.

14. H.-J. Kreowski. *Manipulation von Graphmanipulationen*. PhD thesis, Technical University of Berlin, Dep. of Comp. Sci., 1977.
15. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001, Toronto, Kanada*, volume 2185 of *LNCS*, pages 241–256. Springer-Verlag, 2001.
16. L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.
17. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.
18. L. Ribeiro. *Parallel Composition and Unfolding Semantics of Graph Grammars*. PhD thesis, TU Berlin, 1996.

Relabelling in Graph Transformation

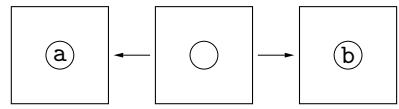
Annegret Habel[1] and Detlef Plump[2]

[1] Fachbereich Informatik, Universität Oldenburg
Postfach 2503, D-26111 Oldenburg, Germany
habel@informatik.uni-oldenburg.de
[2] Department of Computer Science, The University of York
York YO10 5DD, United Kingdom
det@cs.york.ac.uk

Abstract. The traditional double-pushout approach to graph transformation does not allow to change node labels in an arbitrary context. We propose a simple solution to this problem, namely to use rules with partially labelled interface graphs and to match rules injectively. In [8] we have shown that injective matching makes the double-pushout approach more expressive, and here we further generalise that approach. Besides solving the relabelling problem, our framework allows to write rules with partially labelled left-hand sides which are equivalent to (possibly infinite) sets of rules in the traditional setting. Unlike previous work on rules with partially labelled graphs, we do not need any labelling condition on matching morphisms, nor do we exclude node merging rules.

1 Introduction

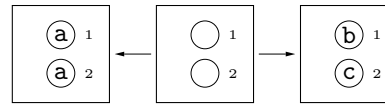
The double-pushout approach to graph transformation has been studied for almost 30 years and has been applied in several areas of computer science, see the recent handbook volumes [12,4,7]. However, the traditional formulation of the double-pushout approach has the drawback that it is impossible to write a rule that changes the label of a node in an arbitrary context, that is, regardless of the number of edges incident to a node. For example, to change a label `a` into `b` one would like to write a rule like the following:



The node in the interface graph is unlabelled, but the traditional double-pushout approach is based on totally labelled graphs and label-preserving graph morphisms which exclude such a rule. (See [3,2] for introductions to the double-pushout approach.) A problem with partially labelled graphs is that the so-called gluing condition does no longer guarantee the applicability of a rule. This condition forbids to remove a node that is incident to a context edge or is the image of two distinct nodes in the rule. For example, consider the rule[1]

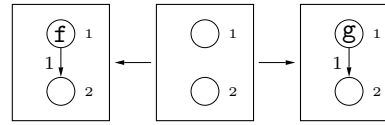
[1] Nodes are numbered to indicate the graph morphisms.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 135–147, 2002.
© Springer-Verlag Berlin Heidelberg 2002



and a graph G consisting of a single node labelled with `a`. There is a non-injective graph morphism from the left-hand side of the rule to G, and this morphism satisfies the gluing condition since the rule preserves both nodes. But the rule is not applicable to G because, intuitively, it requires the label `a` of G's node to be changed to both `b` and `c`. This problem vanishes into thin air if the left-hand sides of rules are matched injectively.

If unlabelled nodes are used not only in interfaces of rules but also in left- and right-hand sides, single rules can specify transformations that otherwise require possibly infinite sets of rules. For example, the rule



implements the term rewriting rule $\mathtt{f(x)} \to \mathtt{g(x)}$ on node-labelled term graphs (see also Example 1 below). If we allowed only interface graphs to be partially labelled, this rule had to be replaced by a set of rules in which node 2 is labelled with all given function symbols and variables — of which there may be infinitely many.

In this paper we show that in the double-pushout approach on partially labelled graphs, a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ is applicable to a graph G via an injective morphism $g\colon L \to G$ if and only if g satisfies the dangling condition. This condition requires that nodes to be removed must not be incident to context edges. Moreover, we show that the graph resulting from the rule application is unique up to isomorphism, and is totally labelled if G is totally labelled. For these results we require a mild condition on undefined labels in rules and that pushouts are "natural" in the sense that they are simultaneously pullbacks. Both requirements are always satisfied in the double-pushout approach with injective matching on totally labelled graphs, so our framework is a conservative extension of that approach.

The rest of this paper is organised as follows. The next section introduces partially labelled graphs, transformation rules and direct derivations in form of two natural pushouts. In Section 3 we show that the category of partially labelled graphs possesses pullbacks and give sufficient conditions for the existence of pushouts. The existence and uniqueness of natural pushout complements and direct derivations is proved in Section 4. We conclude in Section 5 by comparing our approach with previous work.

2 Graphs, Rules and Derivations

A *partially labelled graph* is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{l}_{G,\mathrm{V}}, \mathrm{l}_{G,\mathrm{E}})$ consisting of two finite sets V_G and E_G of *nodes* and *edges*, two source and target func-

tions $s_G, t_G: E_G \to V_G$, and two partial labelling functions $l_{G,V}: V_G \to \mathcal{C}_V$ and $l_{G,E}: E_G \to \mathcal{C}_E$ [2], where $\mathcal{C}_V$ and $\mathcal{C}_E$ are fixed sets of node and edge labels. For simplicity, partially labelled graphs are called *graphs* in the following. A graph G is *totally labelled* if $l_{G,V}$ and $l_{G,E}$ are total functions.

A *graph morphism* $g: G \to H$ between two graphs G and H consists of two functions $g_V: V_G \to V_H$ and $g_E: E_G \to E_H$ that preserve sources, targets and labels, that is, $s_H \circ g_E = g_V \circ s_G$, $t_H \circ g_E = g_V \circ t_G$, and $l_H(g(x)) = l_G(x)$ for all x in $\mathrm{Dom}(l_G)$ [3]. The morphism g *preserves undefinedness* if $l_H(g(x)) = \perp$ for all x in $G - \mathrm{Dom}(l_G)$, and it *reflects undefinedness* if $g^{-1}(x) \neq \emptyset$ for all x in $H - \mathrm{Dom}(l_H)$. A morphism g is *injective* (*surjective*) if g_V and g_E are injective (surjective), and an *isomorphism* if it is injective, surjective and preserves undefinedness. In the latter case G and H are *isomorphic*, which is denoted by $G \cong H$. Furthermore, we call g an *inclusion* if $g(x) = x$ for all x in G. (Note that inclusions need not preserve undefinedness.) Partially labelled graphs and graph morphisms constitute a category, where composition of morphisms is defined componentwise as function composition.

Definition 1 (Rule). A *rule* $r = \langle L \leftarrow K \rightarrow R \rangle$ consists of two graph morphisms $K \to L$ and $b: K \to R$ such that $K \to L$ is an inclusion and

(1) for all $x \in L$, $l_L(x) = \perp$ implies $x \in K$ and $l_R(b(x)) = \perp$,
(2) for all $x \in R$, $l_R(x) = \perp$ implies $l_L(x') = \perp$ for exactly one $x' \in b^{-1}(x)$.

We call L the *left-hand side*, R the *right-hand side* and K the *interface* of r. The rule r is *injective* if $b: K \to R$ is injective. Note that conditions (1) and (2) are trivially satisfied if L and R are totally labelled[4].

Example 1. In *term graph rewriting* one implements term rewriting systems by graph transformation to improve the efficiency of computations (see [10] for a survey). Our framework allows to translate term rewriting rules into graph transformation rules operating on node-labelled term graphs. A node with a function symbol `f` of arity n has n outgoing edges, labelled with 1 to n, whose destinations represent the arguments of `f`. As an example for the translation of term rewriting rules into graph transformation rules, Figure 1 shows the rule corresponding to the term rewriting rule $\mathtt{f(x)} \to \mathtt{x}$. Node 2, representing the variable `x`, has to be unlabelled to make the rule applicable independently of `f`'s argument. Upon application of the rule, nodes 1 and 2 are merged into a single node labelled with the function symbol of `f`'s argument.

[2] Given sets A and B, a partial function $f: A \to B$ is a function from some subset A' of A to B. The set A' is the *domain* of f and is denoted by $\mathrm{Dom}(f)$. We say that $f(x)$ is *undefined*, and write $f(x) = \perp$, if x is in $A - \mathrm{Dom}(f)$.

[3] We often do not distinguish between nodes and edges in statements that hold analogously for both sets.

[4] We also remark that in (2) the more liberal "for at most one $x' \in b^{-1}(x)$" could be used, but then Theorem 2 had to require the present condition.

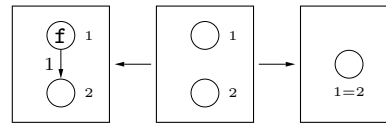


Fig. 1. A rule implementing the term rewriting rule $\mathtt{f(x)} \to \mathtt{x}$

A diagram of graph morphisms

$$\begin{array}{ccc} K & \longrightarrow & R \\ \downarrow & & \downarrow \\ D & \longrightarrow & H \end{array}$$

is a *pushout* if (i) $K \to R \to H = K \to D \to H$ and (ii) for every pair of graph morphisms $\langle R \to H', D \to H' \rangle$ with $K \to R \to H' = K \to D \to H'$, there is a unique morphism $H \to H'$ such that $R \to H' = R \to H \to H'$ and $D \to H' = D \to H \to H'$. The above diagram is a *pullback* if property (i) holds and if for every pair of graph morphisms $\langle K' \to R, K' \to D \rangle$ with $K' \to R \to H = K' \to D \to H$, there is a unique morphism $K' \to K$ such that $K' \to R = K' \to K \to R$ and $K' \to D = K' \to K \to D$. A pushout is *natural* if it is simultaneously a pullback.

Definition 2 (Direct derivation). A *direct derivation* from a graph G to a graph H via a rule $r = \langle L \leftarrow K \to R \rangle$ consists of two natural pushouts as in Figure 2, where $g\colon L \to G$ is injective. We write $G \Rightarrow_{r,g} H$ if there exists such a direct derivation.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ \downarrow & & (1) & & (2) \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Fig. 2. A direct derivation

Remark 1. Given a rule $r = \langle L \leftarrow K \to R \rangle$ and an injective morphism $L \to G$, the pushouts in Figure 2 are natural if L, K and R are totally labelled. But in general a rule may admit both a natural and a non-natural double-pushout. For example, in Figure 3 only the left double-pushout is natural. We characterise the naturalness of pushouts in Lemma 3.

Remark 2. Employing injective matching in the double-pushout approach is not a restriction since every derivation with a set of rules $\mathcal{R}$ in the traditional approach can be simulated, using injective matching, by a derivation with the set

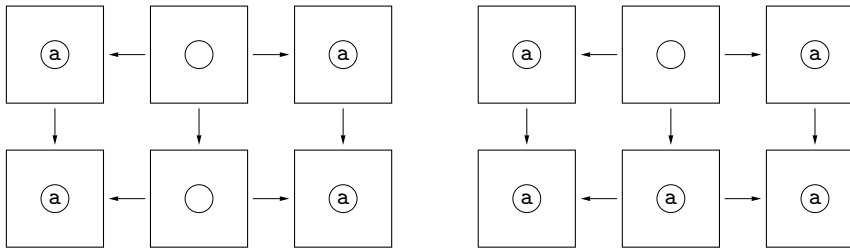


Fig. 3. A natural and a non-natural double-pushout

$\mathrm{Q}(\mathcal{R})$ of quotient rules of $\mathcal{R}$. The quotient rules of a rule are obtained, roughly speaking, by considering all possible identifications among the nodes and edges of the rule. Actually, injective matching makes the double-pushout approach more expressive because, by omitting some quotients, one gets a finer control on transformations than in the traditional framework. We refer to [8] for precise results on the expressiveness gained in this way.

3 Existence of Pullbacks and Pushouts

In this section we prove two lemmata about the existence of pullbacks and pushouts in the category of partially labelled graphs. We also characterise the naturalness of pushouts.

Lemma 1 (Existence of pullbacks). *Given graph morphisms $g\colon L \to G$ and $c\colon D \to G$, there exist a graph K and graph morphisms $b\colon K \to L$ and $d\colon K \to D$ such that diagram (1) in Figure 4 is a pullback.*

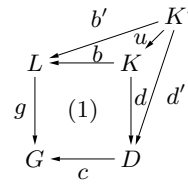


Fig. 4. A pullback

Proof. The node and edge sets of K are constructed as $\{\langle x, y\rangle \in L \times D \mid g(x) = c(y)\}$, the source mapping is defined by $\mathrm{s}_K(\langle x, y\rangle) = \langle \mathrm{s}_L(x), \mathrm{s}_D(y)\rangle$, the target mapping is defined analogously, and the labelling is given by

$$\mathrm{l}_K(\langle x, y\rangle) = \underline{\text{if}}\ \mathrm{l}_L(x) = \mathrm{l}_D(y) \neq \perp\ \underline{\text{then}}\ \mathrm{l}_L(x)\ \underline{\text{else}}\ \perp.$$

Let $b\colon K \to L$ and $d\colon K \to D$ be the projections from $L \times D$ to L and D, that is, $b(\langle x, y\rangle) = x$ and $d(\langle x, y\rangle) = y$, separately for nodes and edges. As in the case of

totally labelled graphs, it is easy to check that b and d preserve the sources and targets of edges. To see that b and d preserve labels, consider $\langle x, y\rangle \in \mathrm{Dom}(\mathrm{l}_K)$. Then $\mathrm{l}_L(b(\langle x, y\rangle)) = \mathrm{l}_L(x) = \mathrm{l}_K(\langle x, y\rangle)$ and $\mathrm{l}_D(d(\langle x, y\rangle)) = \mathrm{l}_D(y) = \mathrm{l}_L(x) = \mathrm{l}_K(\langle x, y\rangle)$ by definition of b, d and l_K. Hence b and d are graph morphisms.

Next we show that square (1) in Figure 4 is a pullback. By definition of b and d, (1) commutes. To see that (1) satisfies the universal property, let $b'\colon K' \to L$ and $d'\colon K' \to D$ be graph morphisms with $g \circ b' = c \circ d'$. There is only one choice (implying uniqueness) for a morphism $u\colon K' \to K$ such that $b \circ u = b'$ and $d \circ u = d'$: define $u(z) = \langle b'(z), d'(z)\rangle$, for all $z \in K'$. It remains to be shown that u is a graph morphism. The proof that u preserves sources and targets of edges is the same as in the case of totally labelled graphs, see [3]. To show that u is label-preserving, let $z \in \mathrm{Dom}(\mathrm{l}_{K'})$. If $\mathrm{l}_L(b'(z)) = \mathrm{l}_D(d'(z)) \not= \bot$, we have $\mathrm{l}_K(u(z)) = \mathrm{l}_K(\langle b'(z), d'(z)\rangle) = \mathrm{l}_L(b'(z)) = \mathrm{l}_{K'}(z)$ by definition of u and l_K, and since b' is label-preserving. If $\mathrm{l}_L(b'(z)) = \bot$ or $\mathrm{l}_D(d'(z)) = \bot$, then $\mathrm{l}_K(u(z)) = \mathrm{l}_K(\langle b'(z), d'(z)\rangle) = \bot = \mathrm{l}_{K'}(z)$ by definition of u and l_K, and since b' and d' are label-preserving. This concludes the proof that diagram (1) is a pullback. □

Lemma 2 (Existence of pushouts). *Let $b\colon K \to R$ and $d\colon K \to D$ be graph morphisms such that d is injective and for all x in R, $\{\mathrm{l}_R(x)\} \cup \mathrm{l}_D(d(b^{-1}(x)))$ contains at most one element. Then there exist a graph H and graph morphisms $R \to H$ and $D \to H$ such that diagram (2) in Figure 5 is a pushout.*

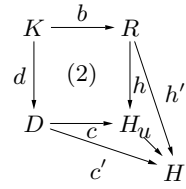


Fig. 5. A pushout

Proof. Construct the node and edge sets of H as $(D - d(K)) + R$, and define $\mathrm{s}_H(e) =$ if $e \in E_R$ then $\mathrm{s}_R(e)$ else $\mathrm{s}_D(e)$, $\mathrm{t}_H(e)$ analogously to $\mathrm{s}_H(e)$, and

$$\mathrm{l}_H(x) = \begin{cases} \mathrm{l}_R(x) & \text{if } x \in R \text{ and } \mathrm{l}_R(x) \not= \bot, \\ \mathrm{l}_D(d(x')) & \text{if } x \in R,\ \mathrm{l}_R(x) = \bot \text{ and } \mathrm{l}_D(d(x')) \not= \bot \\ & \text{for some } x' \in b^{-1}(x), \\ \bot & \text{if } x \in R,\ \mathrm{l}_R(x) = \bot \text{ and } \mathrm{l}_D(d(x')) = \bot \\ & \text{for all } x' \in b^{-1}(x), \\ \mathrm{l}_D(x) & \text{if } x \in (D - d(K)). \end{cases}$$

Note that l_H is well-defined because for all $x \in R$, the set $\{\mathrm{l}_R(x)\} \cup \mathrm{l}_D(d(b^{-1}(x)))$ contains at most one element. Now define $h\colon R \to H$ and $c\colon D \to H$ by $h(x) = x$

and $c(x) = \underline{\text{if}}\ x = d(x')$ for some $x' \in K\ \ \underline{\text{then}}\ b(x')\ \underline{\text{else}}\ x$, separately for nodes and edges. The proof that h and c preserve sources and targets is the same as in the case of totally labelled graphs, see [3]. To show that h and c preserve labels, let $x \in \mathrm{Dom}(\mathrm{l}_R)$. Then $\mathrm{l}_H(h(x)) = \mathrm{l}_R(x)$ by definition of h and l_H, so h is label-preserving. Consider now $x \in \mathrm{Dom}(\mathrm{l}_D)$. If $x \in (D - d(K))$, then $\mathrm{l}_H(c(x)) = \mathrm{l}_D(x)$ by definition of c and l_H. If $x \in d(K)$, then $x = d(x')$ for some $x' \in K$. There are two cases:

- $\mathrm{l}_R(b(x')) \not= \perp$. Then $\mathrm{l}_H(c(x)) = \mathrm{l}_R(b(x')) = \mathrm{l}_D(d(x')) = \mathrm{l}_D(x)$ by definition of l_H, the assumption that for all $x \in R$, $\{\mathrm{l}_R(x)\} \cup \mathrm{l}_D(d(b^{-1}(x)))$ contains at most one element, the fact that $x \in \mathrm{Dom}(\mathrm{l}_D)$, and the fact that d is label-preserving.
- $\mathrm{l}_R(b(x')) = \perp$. Then $\mathrm{l}_H(c(x)) = \mathrm{l}_D(d(x')) = \mathrm{l}_D(x)$ by the commutativity of the diagram, the definition of l_H and the fact that d is label-preserving.

Thus h and c are graph morphisms.

Next we show that diagram (2) in Figure 5 is a pushout. By definition of h and c, we have $h \circ b = c \circ d$. To show the universal property, let $h' \colon R \to H'$ and $c' \colon D \to H'$ be graph morphisms with $h' \circ b = c' \circ d$. There is only one choice (implying uniqueness) to define $u \colon H \to H'$ such that $u \circ h = h'$ and $u \circ c = c'$: let $u(x) = \underline{\text{if}}\ x \in R\ \underline{\text{then}}\ h'(x)\ \underline{\text{else}}\ c'(x)$, separately for nodes and edges. It remains to be shown that u is a graph morphism. That u is structure-preserving follows as in the proof of Lemma 2.8 in [3]. To show that u preserves labels, let $x \in \mathrm{Dom}(\mathrm{l}_H)$. We have three cases:

- $x \in R$ and $\mathrm{l}_R(x) \not= \perp$. Then $\mathrm{l}_{H'}(u(x)) = \mathrm{l}_{H'}(h'(x)) = \mathrm{l}_R(x) = \mathrm{l}_H(x)$ by the definition of u, the fact that $h' \colon R \to H'$ preserves labels, and the definition of l_H.
- $x \in R$, $\mathrm{l}_R(x) = \perp$ and $\mathrm{l}_D(d(x')) \not= \perp$ for some $x' \in b^{-1}(x)$. Then $\mathrm{l}_{H'}(u(x)) = \mathrm{l}_{H'}(h'(b(x'))) = \mathrm{l}_{H'}(c'(d(x'))) = \mathrm{l}_D(x) = \mathrm{l}_H(x)$ by the definition of u, the fact that $h' \circ b = c' \circ d$, the fact that $c' \colon D \to H'$ preserves labels, and the definition of l_H.
- $x \in (D - d(K))$. Then $\mathrm{l}_{H'}(u(x)) = \mathrm{l}_{H'}(c'(x)) = \mathrm{l}_D(x) = \mathrm{l}_H(x)$ by definition of u, the fact that $c' \colon D \to H'$ preserves labels, and the definition of h.

This concludes the proof that square (2) in Figure 5 is a pushout. □

Next we characterise the naturalness of pushouts.

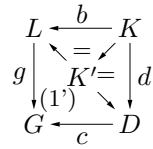
Lemma 3 (Characterisation of natural pushouts). *Given two graph morphisms $b \colon K \to L$ and $d \colon K \to D$ such that b is injective, pushout (1) in Figure 2 is natural if and only if*

$$(*)\ \textit{for all } z \in K,\ \mathrm{l}_K(z) = \perp \textit{ implies } \mathrm{l}_L(b(z)) = \perp \textit{ or } \mathrm{l}_D(d(z)) = \perp.$$

Proof. Let square (1) in Figure 2 be a natural pushout with graph morphisms $g \colon L \to G$ and $c \colon D \to G$. Since (1) is a pullback, an explicit construction of K up to isomorphism is given by $\{\langle x, y\rangle \in L \times D \mid g(x) = c(y)\}$, separately for nodes

and edges, where $l_K(\langle x, y\rangle) = l_L(x)$ if and only if $l_L(x) = l_D(y) \neq \perp$. It follows that $l_K(\langle x, y\rangle) = \perp$ if and only if $l_L(x) = \perp$ or $l_D(y) = \perp$. Hence condition $(*)$ is satisfied.

Conversely, let (1) be a pushout satisfying condition $(*)$. In the diagram below, let (1') be a pullback of $g\colon L \to G$ and $d\colon D \to G$. Let b' and d' be the morphisms from K' to L and D, respectively. By the universal property of (1'), there exists a unique morphism $u\colon K \to K'$ such that $b' \circ u = b$ and $d' \circ u = d$.



We show that u is an isomorphism. Injectivity of u follows from injectivity of $b = b' \circ u$. To see that u is surjective, consider some $z' \in K'$. Then $g(b'(z')) = c(d'(z'))$ by commutativity of diagram (1'). Hence, by injectivity of b and the pushout characterisation of [5] (Theorem 1.2) applied to diagram (1), there is some $z \in K$ such that $b(z) = b'(z')$ and $d(z) = d'(z')$. Applying the pullback characterisation of [5] (Theorem 1.7) to (1') gives $u(z) = z'$.

Finally, u preserves undefinedness by condition $(*)$. Thus u is an isomorphism, implying that diagram (1) is a pullback and hence a natural pushout. □

4 Existence and Uniqueness of Direct Derivations

Our main result (Theorem 1) will show that, given a rule $r = \langle L \leftarrow K \rightarrow R\rangle$ and an injective graph morphism $g\colon L \to G$, there exists a direct derivation as in Figure 2 if and only if g satisfies the dangling condition. Moreover, in this case r and g determine D and H uniquely up to isomorphism. The proof of this result is based on the following lemma which provides a condition for the existence and uniqueness of "natural pushout complements".

A graph morphism $g\colon L \to G$ satisfies the *dangling condition* with respect to an inclusion $K \to L$, if no node in $g(L) - g(K)$ is incident to an edge in $G - g(L)$.

Lemma 4 (Existence and uniqueness of natural pushout complements). *Let $g\colon L \to G$ be an injective graph morphism and $K \to L$ an inclusion that reflects undefinedness. Then there exist a graph D and graph morphisms $K \to D$ and $D \to G$ such that diagram (1) in Figure 2 is a natural pushout if and only if g satisfies the dangling condition. Moreover, in this case D is unique up to isomorphism.*

Proof. "Only if": Let diagram (1) in Figure 2 be a pushout. By forgetting all labels, this diagram becomes a pushout in the category of unlabelled graphs. By the pushout characterisation of [5] (Theorem 1.2), g satisfies the dangling condition in that situation. Since the dangling condition is not concerned with labels, g in diagram (1) satisfies the dangling condition as well.

"If": Let g be an injective graph morphism satisfying the dangling condition. Construct the graph D as follows: D has node and edge sets $(G - g(L)) + g(K)$, and s_D and t_D are the restrictions of s_G and t_G to these sets. Note that s_D and t_D are well-defined because g satisfies the dangling condition. We define D's labelling by

$$l_D(x) = \underline{\text{if}}\ x = g(x') \text{ and } l_L(x') \not= \bot \text{ for some } x' \text{ in } K \ \underline{\text{then}}\ l_K(x') \ \underline{\text{else}}\ l_G(x),$$

separately for nodes and edges. These mappings are well-defined because $K \to L$ and $L \to G$ are injective. Next define the graph morphism $c\colon D \to G$ as the inclusion of D in G and the morphism $d\colon K \to D$ as the restriction of $K \to L \to G$ to D. In [11] (Lemma 4.4) it is shown that c and d defined in this way preserve sources and targets. We show that they preserve labels in our case. For $x \in \mathrm{Dom}(l_K)$, we have $l_D(d(x)) = l_K(x)$ by definition of d and l_D. So d is label-preserving. To see that c is label-preserving as well, let $x \in \mathrm{Dom}(l_D)$. There are two cases to consider.

- $x \in G - g(L)$. Then $l_G(c(x)) = l_G(x) = l_D(x)$ by the definition of c and l_D.
- $x \in g(K)$. Let $x' \in K$ with $g(x') = x$. If $l_L(x') \not= \bot$, then $l_G(c(x)) = l_G(x) = l_G(g(x')) = l_L(x') = l_K(x') = l_D(x)$ by definition of c, the fact that $K \to L$ and $L \to G$ are label-preserving, and the definition of l_D. On the other hand, if $l_L(x') = \bot$, then $l_G(c(x)) = l_G(x) = l_D(x)$ by definition of l_D.

So c is label-preserving and hence c and d are graph morphisms in the category of partially labelled graphs.

By forgetting all labels, diagram (1) becomes a diagram in the category of unlabelled graphs. In [11] (Lemma 4.4) it is shown that (1) is a pushout in that case. Denote the inclusion $K \to L$ by b. The pushout property means that $g \circ b = c \circ d$, and that for every unlabelled G' and graph morphisms $g'\colon L \to G'$ and $c'\colon D \to G'$ with $g' \circ b = c' \circ d$, there is a unique morphism $u\colon G \to G'$ such that $u \circ g = g'$ and $u \circ c = c'$. To prove that (1) remains a pushout if the graphs have labels, we have to show that u preserves labels. Let $x \in \mathrm{Dom}(l_G)$. Since $G = (D - d(K)) \cup g(L)$, separately for nodes and edges, we consider the following cases:

- $x \in g(L)$ and $l_L(\bar{x}) \not= \bot$ for the unique $\bar{x} \in L$ with $g(\bar{x}) = x$. Then $l_{G'}(u(x)) = l_{G'}(u(g(\bar{x}))) = l_{G'}(g'(\bar{x})) = l_L(\bar{x}) = l_G(g(\bar{x})) = l_G(x)$ since g' and g are label-preserving.
- $x \in g(L)$ and $l_L(\bar{x}) = \bot$ for the unique $\bar{x} \in L$ with $g(\bar{x}) = x$. Since the inclusion $K \to L$ reflects undefinedness, we have $\bar{x} \in K$. By definition of l_D, $l_D(d(\bar{x})) = l_G(d(\bar{x})) = l_G(g(\bar{x})) = l_G(x)) \not= \bot$. Thus $l_{G'}(u(x)) = l_{G'}(u(g(\bar{x}))) = l_{G'}(u(d(\bar{x}))) = l_{G'}(c'(d(\bar{x}))) = l_D(d(\bar{x})) = l_G(x)$, since c' preserves labels.
- $x \in (D - d(K))$. Then $l_{G'}(u(x)) = l_{G'}(c'(x)) = l_D(x) = l_G(x)$ by definition of l_D and since c' is label-preserving.

So u preserves labels and hence diagram (1) is a pushout in the category of partially labelled graphs. Moreover, by definition of l_D, property $(*)$ of Lemma

3 holds: for all $x \in K$, $l_K(x) = \bot$ implies $l_L(x) = \bot$ or $l_D(d(x)) = \bot$. Thus, by Lemma 3, diagram (1) is a natural pushout.

It remains to be shown that D is unique up to isomorphism. Again we first forget the labels of all graphs in diagram (1) to obtain a pushout in the category of unlabelled graphs. Then by Lemma 4.2 in [11], D is uniquely determined up to isomorphism. To lift this result to our setting, we have to show that D's labelling is uniquely determined. Consider any $x \in D$.

Case 1: $x \in (G - g(L))$. It can be shown that for every pushout of form (1), for each $x \in G$ there is $x' \in L$ with $g(x') = x$ and $l_L(x') = l_G(x)$ or there is $x' \in D$ with $c(x') = x$ and $l_D(x') = l_G(x)$. Since in the present case $x \not\in g(L)$ and c is an inclusion, $l_D(x)$ must be equal to $l_G(x)$.

Case 2: $x \in g(K)$. Let x' be the unique element in K with $g(x') = x$.

Case 2.1: $l_L(x') = \bot$. Then by the property of pushouts mentioned in Case 1 and since c preserves labels, again $l_D(x)$ must equal $l_G(x)$.

Case 2.2: $l_L(x') \not= \bot$. If $l_K(x') \not= \bot$, then $l_D(x) = l_D(d(x')) = l_K(x')$ because d preserves labels. If $l_K(x') = \bot$, then by the characterisation of natural pushouts in Lemma 3, $l_D(x) = l_D(d(x')) = \bot$. □

The following theorem is our main result.

Theorem 1 (Existence and uniqueness of direct derivations). *Given a rule $r = \langle L \leftarrow K \rightarrow R \rangle$ and an injective graph morphism $g\colon L \to G$, there exists a direct derivation as in Figure 2 if and only if g satisfies the dangling condition. Moreover, in this case D and H are unique up to isomorphism.*

Proof. "Only if": For a direct derivation as in Figure 2, g satisfies the dangling condition by Lemma 4.

"If": If g satisfies the dangling condition, then by Lemma 4 there exist a graph D and graph morphisms $d\colon K \to D$ and $D \to G$ such that diagram (1) in Figure 2 is a natural pushout. In order to obtain pushout (2) of Figure 2 by Lemma 2, we have to show that for all $x \in R$, the set $\{l_R(x)\} \cup l_D(d(b^{-1}(x)))$ contains at most one element.

Case 1: $l_R(x) = \bot$. Then $l_K(x') = \bot$ for all $x' \in b^{-1}(x)$ because b preserves labels. By the definition of a rule, there is exactly one $x' \in b^{-1}(x)$ with $l_L(x') = \bot$. Hence for each other element $\bar{x} \in b^{-1}(x)$, $l_L(\bar{x}) \not= \bot$ and therefore $l_D(d(\bar{x})) = \bot$ by the characterisation of natural pushouts in Lemma 3. Thus $\{l_R(x)\} \cup l_D(d(b^{-1}(x)))$ contains at most one label.

Case 2: $l_R(x) \not= \bot$. Then, by the definition of a rule, $l_L(x') \not= \bot$ for all $x' \in b^{-1}(x)$. Consider any $x' \in b^{-1}(x)$.

Case 2.1: $l_K(x') \not= \bot$. Then $l_D(d(x')) = l_K(x') = l_R(b(x')) = l_R(x)$ because d and b preserve labels.

Case 2.1: $l_K(x') = \bot$. Then $l_D(d(x')) = \bot$ by the characterisation of natural pushouts in Lemma 3.

Hence in Case 2, elements in $b^{-1}(x)$ are either labelled with $l_R(x)$ or are unlabelled. Thus again $\{l_R(x)\} \cup l_D(d(b^{-1}(x)))$ contains at most one label.

Now Lemma 2 shows that pushout (2) in Figure 2 exists. We also have to show that (2) is natural. Consider any $x \in K$ with $l_K(x) = \bot$ and $l_R(b(x)) \not= \bot$. Then

$l_L(x) \not= \perp$ by the definition of a rule. Hence $l_D(d(x)) = \perp$ by the naturalness of pushout (1) and Lemma 3. Thus, by Lemma 3, diagram (2) is a natural pushout.

Finally, by Lemmas 4 and 2, the graphs D and H are uniquely determined up to isomorphism. This concludes the proof of Theorem 1. □

Theorem 2 (Totality of labelling). *For every direct derivation $G \Rightarrow H$, H is totally labelled if and only if G is totally labelled.*

Proof. "If": Let G be totally labelled. By the uniqueness of H up to isomorphism (Theorem 1) and the proof of Lemma 2, we can assume that H has node and edge sets $(D - d(K)) + R$, where D is constructed as in the proof of Lemma 4. Let $x \in H$. We consider three cases. If $x \in (D - d(K))$, then $l_H(x) = l_D(x) = l_G(x) \not= \perp$ by definition of l_H and l_D, and since G is totally labelled. If $x \in R$ with $l_R(x) \not= \perp$, then $l_H(x) = l_R(x) \not= \perp$. Finally, if $x \in R$ with $l_R(x) = \perp$, then $l_L(x') = \perp$ for exactly one $x' \in b^{-1}(x)$ by the definition of a rule. Hence $l_H(x) = l_D(d(x')) = l_G(g(x')) \not= \perp$ by definition of l_H and l_D, and because G is totally labelled.

"Only if": Let $x \in G$ with $l_G(x) = \perp$. We consider two cases, using the constructions of D and H referred to above. If $x \notin g(L)$, then $l_H(x) = l_D(x) = l_G(x) = \perp$ by definition of l_H and l_D. If $x \in g(L)$, then by the definition of a rule there is some $x' \in K$ such that $g(x') = x$ and $l_R(b(x')) = \perp$. Since also $l_D(d(x')) = l_D(x) = \perp$, we have $l_H(h(x')) = \perp$ by definition of l_H in Lemma 2. □

5 Related Work

In this section we compare our approach with previous work on partially labelled graphs in the double-pushout approach.

Rosen [11] allows partially labelled graphs in rules (but requires that transformed graphs are totally labelled) and considers possibly non-injective matching morphisms. To ensure the existence of direct derivations, matching morphisms must not only satisfy the gluing condition but also an additional labelling condition. Moreover, non-injective rules as the rule in Figure 1 cannot be handled. Probably the most serious drawback of [11] is that direct derivations are not double-pushouts in the usual sense but consist of two pushouts which need not share a common morphism. That is, the morphism $K \to D$ in Figure 2 is replaced by two morphisms $K \to D_1$ and $K \to D_2$ where the transition from D_1 to D_2 corresponds to a relabelling. This feature makes the approach of [11] complicated, especially for proving properties of derivations.

In [6], Ehrig et al. extend the double-pushout approach from graphs to relational structures. Their results can be specialised to partially labelled graphs. Corollary 5.9 in [6] states under which conditions certain special derivations starting from a totally labelled graph yield a totally labelled graph. The approach of [6] (specialised to graphs) is more restrictive than the present approach in that only injective rules are allowed. Moreover, since matching morphisms are

allowed to be non-injective, a labelling condition has to be satisfied in addition to the gluing condition.

Parisi-Presicce, Ehrig and Montanari [9] extend the traditional double-pushout approach by considering graphs with labels from an alphabet equipped with a partial order. Since every partially labelled graph can be seen as a totally labelled graph over a label alphabet containing a special label $\perp$ and being equipped with the flat order $\{\langle \perp, x\rangle \mid \perp \not= x\}$, their approach can be specialised to the case of partially labelled graphs. Again [9] is more restrictive than the present approach in that only injective rules are allowed. For example, our rule in Figure 1 cannot be used. Moreover, an injective rule like the second rule in the introduction does not satisfy the consistency condition of [9]. Apart from these restrictions, Proposition 3.13 in [9] states that a direct derivation exists if and only if the gluing condition is satisfied. However, a counterexample is given by the rule having a single node labelled with $\perp$ as left-hand side, an empty interface and an empty right-hand side (which is allowed in [9]). The unique graph morphism from the left-hand side to a graph consisting of a single node not labelled with $\perp$ satisfies the gluing condition, but there does not exist a pushout complement in this situation.

Finally, Berthold, Fischer and Koch [1] recently considered the double-pushout approach over partially attributed graphs. They employ injective matching morphisms but restrict themselves to injective rules. The framework is liberal with respect to attributes in rules but requires extra conditions besides the usual gluing condition for matching morphisms. [1] does not contain correctness proofs for the indicated constructions of pushouts and pushout complements.

References

1. Michael R. Berthold, Ingrid Fischer, and Manuel Koch. Attributed graph transformation with partial attribution. In *Proc. Joint APPLIGRAPH/GETGRATS Worshop on Graph Transformation Systems (GRATRA 2000)*, pages 171–178, 2000.
2. Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation — Part I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 1, chapter 3, pages 163–245. World Scientific, 1997.
3. Hartmut Ehrig. Introduction to the algebraic theory of graph grammars. In *Proc. Graph-Grammars and Their Application to Computer Science and Biology*, volume 73 of *Lecture Notes in Computer Science*, pages 1–69. Springer-Verlag, 1979.
4. Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools.* World Scientific, 1999.
5. Hartmut Ehrig and Hans-Jörg Kreowski. Pushout-properties: An analysis of gluing constructions for graphs. *Mathematische Nachrichten*, 91:135–149, 1979.
6. 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.

7. Hartmut Ehrig, Hans-Jörg Kreowski, Ugo Montanari, and Grzegorz Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency, Parallelism, and Distribution*. World Scientific, 1999.
8. Annegret Habel, Jürgen Müller, and Detlef Plump. Double-pushout graph transformation revisited. *Mathematical Structures in Computer Science*, 11(5):637–688, 2001.
9. Francesco Parisi-Presicce, Hartmut Ehrig, and Ugo Montanari. Graph rewriting with unification and composition. In *Proc. Graph-Grammars and Their Application to Computer Science*, volume 291 of *Lecture Notes in Computer Science*, pages 496–514. Springer-Verlag, 1987.
10. Detlef Plump. Term graph rewriting. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation*, volume 2, chapter 1, pages 3–61. World Scientific, 1999.
11. Barry K. Rosen. Deriving graphs from graphs by applying a production. *Acta Informatica*, 4:337–357, 1975.
12. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.

Euler Graphs, Triangle-Free Graphs and Bipartite Graphs in Switching Classes

Jurriaan Hage[1], Tero Harju[2], and Emo Welzl[3]

[1] Inst. of Information and Computing Sci., Univ. Utrecht, P.O.Box 80.089, 3508 TB Utrecht, Netherlands
jur@cs.uu.nl
[2] Dept. of Mathematics, University of Turku, FIN-20014 Turku, Finland
[3] Institut für Theoretische Informatik, ETH Zentrum, IFW B49.2, Haldeneggsteig 4, CH-8092 Zürich, Switzerland

Abstract. In the context of graph transformations we look at the operation of switching, which can be viewed as an elegant method for realizing global transformations of graphs through local transformations of the vertices. A switching class is then a set of graphs obtainable from a given start graph by applying the switching operation.
Continuing the line of research in Ehrenfeucht et al. we consider the problem of detecting three kinds of graphs in switching classes. For all three we find algorithms running in time polynomial in the number of vertices in the graphs, although switching classes contain exponentially many graphs.

1 Introduction

The operation of switching is an elegant example of a graph transformation, where the global transformation of a graph is achieved by applying local transformations to the vertices. The elegance stems from the fact that the local transformations are group actions, and hence basic techniques of group theory can be applied in developing the theory of switching.

The use of switching presented in this paper is motivated by operations for formalizing specific types of networks of processors, as introduced in [6] by Ehrenfeucht and Rozenberg.

For a finite undirected graph $G = (V, E)$ and a subset $\sigma \subseteq V$, the *switch* of G by σ is defined as the graph $G^\sigma = (V, E')$, which is obtained from G by removing all edges between σ and its complement $V - \sigma$ and adding as edges all nonedges between σ and $V - \sigma$. The switching class $[G]$ determined (generated) by G consists of all switches G^σ for subsets $\sigma \subseteq V$.

A switching class is an equivalence class of graphs under switching, see the survey papers by Seidel [13] and Seidel and Taylor [14]. Generalizations of this approach – where the graphs are labelled with elements of a group other than $\mathbf{Z}_2$ – can be found in Gross and Tucker [7], Zaslavsky [15], the book of Ehrenfeucht, Harju and Rozenberg [5] and the thesis of Hage [8].

A property $\mathcal{P}$ of graphs can be transformed into an existential property of switching classes as follows:

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 148–160, 2002.
© Springer-Verlag Berlin Heidelberg 2002

$\mathcal{P}_\exists(G)$ if and only if there is a graph $H \in [G]$ such that $\mathcal{P}(H)$.

We will also refer to $\mathcal{P}_\exists$ as "the problem $\mathcal{P}$ for switching classes".

Let G be a graph on n vertices. It is easy to show that there are 2^{n-1} graphs in $[G]$, and so checking whether there exists a graph $H \in [G]$ satisfying a given property $\mathcal{P}$ requires exponential time, if each graph is to be checked separately.

However, we prove that deciding that a switching class is bipartite (i.e., it contains at least one bipartite graph) can be done in polynomial time. The same result holds for the predicates "triangle-free" and "eulerian". The solution to the latter problem can also be found in the thesis of Hage [8], but has never been published in a paper.

A little now about the motivation for our research. In [9], we considered the problem of characterizing the switching classes which contain only cyclic graphs. We did this by means of a list of forbidden subgraphs. In the process of this research we made extensive use of computer programs. However, as stated earlier, switching classes grow unwieldy quite soon and having an efficient algorithm to check whether a switching class contains any acyclic graphs at all, would be of great help. For the case of acyclic graphs such an algorithm exists (Theorem 1 of Ehrenfeucht, Hage, Harju and Rozenberg [4] or Theorem 3.21 of Hage [8]).

After considering this problem for acyclic graphs we turned to generalizations of the acyclic graphs: the bipartite and the triangle-free graphs. The bipartite graphs and the triangle-free graphs are natural classes of graphs in the context of switching, see Seidel [13] and Seidel and Taylor [14]. In the latter case, it seems that there is not a reasonably elegant result using a list of forbidden subgraphs: this list soon grows to enormous size. For the bipartite case however the computational results so far are promising.

The computational complexity of various problems for switching classes were considered by Ehrenfeucht, Hage, Harju and Rozenberg [4]. For instance, there it was proved that for a large number of predicates $\mathcal{P}$ decidable in polynomial time there exists a polynomial algorithm to decide $\mathcal{P}_\exists(G)$. These predicates have the property, that for every graph for which the predicate holds, there is a vertex with degree at most k (or degree at least $|V(G)| - k$) for some constant k.

Although these problems can now be considered solved, there remain many problems for which a constant bound on the degrees cannot be given. In the present paper we show that the three problems to decide whether a switching class contains a bipartite respectively eulerian respectively triangle-free graph are all polynomial time (in the number of vertices of the graphs). Although we consider graphs up to equality (i.e. the names of the vertices do matter), most results carry over to the situation where the graphs are considered up to isomorphism.

Surprisingly, in view of the fact that switching classes grow exponentially in the size of the vertex set of the graph, problems that are polynomial for graphs, hardly ever are exponential for switching classes. In fact, we only know of one such problem, the problem of deciding whether a switching class contains a k-regular graph for some k [10].

2 Preliminaries

For a (finite) set V, let $|V|$ be the *cardinality* of V. We shall often identify a subset $A \subseteq V$ with its characteristic function $A : V \to \mathbf{Z}_2$, where $\mathbf{Z}_2 = \{0, 1\}$ is the cyclic group of order two. We use the convention that for $x \in V$, $A(x) = 1$ if and only if $x \in A$. The restriction of a function $f : V \to W$ to a subset $A \subseteq V$ is denoted by $f|_A$. We denote set difference by $A - B$. It contains the elements in A which are not in B. If B is a singleton $\{b\}$, then we may write $A - b$ for brevity.

The set $E(V) = \{\{x, y\} \mid x, y \in V,\ x \neq y\}$ denotes the set of all unordered pairs of distinct elements of V. We write xy or yx for the undirected pair $\{x, y\}$. The graphs of this paper will be finite, undirected and simple, i.e., they contain no loops or multiple edges. We use $E(G)$ and $V(G)$ to denote the set of edges E and the set of vertices V, respectively, and $|V|$ and $|E|$ are called the *order*, respectively, *size* of G. Analogously to sets, a graph $G = (V, E)$ will be identified with the characteristic function $G : E(V) \to \mathbf{Z}_2$ of its set of edges so that $G(xy) = 1$ for $xy \in E$, and $G(xy) = 0$ for $xy \notin E$. Later we shall use both notations, $G = (V, E)$ and $G : E(V) \to \mathbf{Z}_2$, for graphs.

Let $G = (V, E)$ be a graph. A vertex $x \in V$ is *adjacent* to $y \in V$ if $xy \in E$. The *degree* of x in G is the number of vertices it is adjacent to. The *neighbours* of u in G, denoted $N_G(u)$, or $N(u)$ if G is clear from the context, is the set of vertices adjacent to u in G. A vertex which is not adjacent to any other vertex in a graph is called *isolated*.

For a graph $G = (V, E)$ and $X \subseteq V$, let $G|_X$ denote the *subgraph* of G *induced* by X. Hence, $G|_X : E(X) \to \mathbf{Z}_2$. For two graphs G and H on V we define $G + H$ to be the graph such that $(G + H)(xy) = G(xy) + H(xy)$ for all $xy \in E(V)$.

Some graphs we will encounter in the sequel are K_V, the clique on the set of vertices V, and $\overline{K}_V$, the complement of K_V which is the discrete graph on V; the complete bipartite graph on A and $V - A$ is denoted by $K_{A,V-A}$. If the choice of vertices is unimportant we can write K_n, $\overline{K}_n$ and $K_{m,n-m}$ for $n = |V|$ and $m = |A|$.

An euler graph is a connected graph consisting of a single closed walk such that each edge is traversed once (but a vertex can be visited several times). It is well-known that a graph is eulerian if and only if all the degrees of its vertices are even. We can extend the definition of euler graphs to disconnected graphs by demanding that all components are eulerian. These graphs are here called *even* graphs. In a similar fashion we call graphs in which every vertex has odd degree an *odd* graph.

A graph $G = (V, E)$ is *bipartite* if V can be partitioned into two sets A and $V - A$, such that all edges in E are between A and $V - A$. An equivalent characterization is to say that G is bipartite if and only if it contains no cycles of odd length. Deciding whether a graph is bipartite is easily done by visiting each vertex at most once. So it can be checked for graphs very efficiently. The bipartite graphs include among them all acyclic graphs, because these do not contain cycles of any kind.

The triangle-free graphs generalize bipartite graphs by only disallowing triangles: a graph $G = (V, E)$ is *triangle-free* if G does not contain an induced K_3. Note that if G is triangle-free, then its largest clique (i.e., complete subgraph) has at most two vertices. Deciding whether a graph is triangle-free is easily done by examing every subset of three vertices to see if all three edges in the triangle are in the graph.

Triangle-freeness can be decided in time cubic in the number of vertices. Whether a graph is eulerian and bipartite can both be decided in time quadratic in the number of vertices of the graph. In the following we shall consider all three problems for switching classes and show that the problems remain polynomial.

We continue now with definitions for the switching of graphs.

A *selector* for G is a subset $\sigma \subseteq V(G)$, or alternatively a function $\sigma\colon V(G) \to \mathbf{Z}_2$. We reserve lower case σ for selectors (subsets). A *switch* of a graph G by σ is the graph G^σ such that for all $xy \in E(V)$,

$$G^\sigma(xy) = \sigma(x) + G(xy) + \sigma(y) \ .$$

Clearly, this definition of switching is equivalent to the one given at the beginning of the introduction. The set $[G] = \{G^\sigma \mid \sigma \subseteq V\}$ is called the *switching class* of G. The set of graphs $[G]$ is called a switching class, because switching is a reflexive, symmetric and transitive operation: composition of two selectors amounts to taking the symmetric difference. This result can be used to prove the following.

Lemma 1. *It holds that $G^{\sigma_1} = G^{\sigma_2}$ if and only if $\sigma_1 = \sigma_2$ or $\sigma_1 = V(G) - \sigma_2$.*

A selector σ is *constant* on $X \subseteq V$ if $X \subseteq \sigma$, or $X \cap \sigma = \emptyset$. The name arises from the fact that, in these cases, $G|_X = G^\sigma|_X$.

We now give a few results from the literature that will be used throughout this paper, see Seidel [13], Ehrenfeucht, Harju and Rozenberg [5] and Hage [8].

Lemma 2. *The switching class $[\overline{K}_V]$ equals the set of all complete bipartite graphs on V.*

Proof. Given any complete bipartite graph $K_{\{\sigma, V-\sigma\}}$, we can obtain it from $\overline{K}_V$ by switching with respect to σ. Also, every switch of $\overline{K}_V$ can divide its partition at most in two, so it must be a bipartite graph. It is obviously complete bipartite, because $\overline{K}_V$ has no edges. □

From the observation that computing G^σ amounts to computing $G + K_{\sigma, V(G)-\sigma}$ we obtain the following result.

Lemma 3. *It holds that $G \in [H]$ if and only if $G + H \in [\overline{K}_V]$.*

Lemma 4. *Let $G = (V, E)$ be a graph, $u \in V$ and $A \subseteq V - \{u\}$. There exists a unique graph $H \in [G]$ such that the neighbours of u in H are the vertices in A.*

Proof. The vertex u has no adjacent vertices in $G_u = G^{N(u)}$, where $N(u)$ is the set of neighbours of u in G. Switching G_u with respect to A connects u to every vertex in A (and no others) yielding H.

To show that H is unique: let H' be such that $N(u) = A$ in H'. Since H and H' are in the same switching class $H + H'$ is a complete bipartite graph (Lemma 3), say $G_{B,V-B}$. Because u has the same neighbours in both, u is isolated in $G_{B,V-B}$. Hence, $G_{B,V-B}$ is a discrete graph and, consequently, $H = H'$. □

As a corollary we easily find that for every vertex $x \in V(G)$, there is a unique graph in $[G]$ where x is isolated.

3 Euler Switching Classes

Let G be a graph. Recall that G is called an *even* (*odd*) graph if all vertices are of even (odd) degree.

Seidel [12] proved that if G is of odd order, then the switching class $[G]$ contains a unique even graph. Such a result is interesting because it tells us that every graph having an odd number of vertices can be constructed from an even graph using the rather simple transformation of switching. (See also Mallows and Sloane [11] and Cameron [2] for the connection of eulerian graphs to switching.) For a proof we refer the reader to the thesis of Hage [8].

Theorem 1 (Seidel [12]). *Let G be a graph of odd order. Then $[G]$ contains a unique even graph.*

A general uniqueness result such as Theorem 1 is not possible without restrictions on the vertex set. Indeed, if $\mathcal{P}$ is any graph property, which is preserved under isomorphisms, then there exists a switching class that has no graphs with $\mathcal{P}$ or that has at least two graphs with $\mathcal{P}$. This statement follows from a result on automorphisms of switching classes, see Cameron [2]: there exist switching classes where the first cohomological invariant is different from zero, and therefore there exist switching classes for which the group of automorphisms is strictly larger than the group of automorphisms of its graphs.

For graphs of even order, a switching class $[G]$ can contain only noneulerian graphs, e.g. take the switching class $[P_4]$, where P_4 is the path on 4 vertices. However, it holds that either $[G]$ has no even and no odd graphs, or exactly half of its graphs are even while the other half are odd, as first proved in Ehrenfeucht, Hage, Harju and Rozenberg [4]. (This is a result that is not likely to hold when graphs are considered up to isomorphism.)

To see this, define $u \sim_G v$, if $d_G(u) \equiv d_G(v) \pmod 2$, that is, if the degrees of u and v have the same parity. This relation is an equivalence relation on $V(G)$.

Assume then that the order n of G is even. If we consider singleton selectors σ only (hence switching with respect to one vertex), then it is easy to see that $\sim_G$ and $\sim_{G^\sigma}$ coincide for all selectors σ. In other words, if G has even order, then the relation $\sim_G$ is an invariant of the switching class $[G]$. This means that if $[G]$ contains an even graph, then all graphs in $[G]$ are either even or odd. Moreover, if G is even, and $\sigma : V(G) \to \mathbf{Z}_2$ is a singleton selector, then for each $v \in V(G)$, $d_G(v)$ and $d_{G^\sigma}(v)$ have different parity.

Theorem 2. *Let G be a graph of even order and such that $[G]$ contains an even graph. Then $[G]$ contains an eulerian graph if and only if $[G]$ does not contain a complete graph.*

Proof. We may assume that G itself is even, otherwise switch at an arbitrary vertex. However, this graph may not be connected and we need to find a connected even switch of G. This can be done as follows: let u be any vertex of G. Let $V = V(G)$ and let $O = V - (\{u\} \cup N_G(u))$, the vertices that are not neighbours of u in G. Note that the set O has odd cardinality. Switch G with respect to $\sigma = O - \{v\}$ for an arbitrary $v \in O$. The resulting graph G^σ is again even, but it may still not be connected, but at least $G^\sigma|_{V-v}$ is. However, if $G^\sigma|_{V-v}$ is not the complete graph, we can choose a vertex $w \in V - \{u, v\}$ that is not connected to all vertices in $V - v$. Switching with respect to $\{v, w\}$ now gives us a connected even graph. Note that in fact $[K_n]$ for even n does have even graphs, but no eulerian graphs, because K_n is the only connected graph and it is not even, but odd. □

From the above it follows that the existence of an even graph in a switching class can be determined in time linear in the size of the graph: in the odd order case, the answer is always *yes*, while in the even order case, we need only verify whether G is an even or odd graph. If either of these is the case, then the answer is *yes*, otherwise it is *no*.

We can extend this algorithm to check for eulerian graphs: for graphs G of odd order it is simply a question of finding the unique even graph as in the proof Theorem 1 and verifying whether or not it is connected. For the even case we can use the algorithm implied in the proof of Theorem 2: only if $[G]$ contains an even graph, can it possibly contain an eulerian graph. And it will do so, except for the case when G can be switched into a complete graph.

4 Triangle-Free Switching Classes

The smallest graph that generates a switching class without bipartite graphs, also has no triangle-free graphs. It is the switching class $[K_5]$, which contains in addition to K_5 only $K_4 \oplus K_1$ and $K_3 \oplus K_2$, where $\oplus$ denotes the operation of disjoint union of two graphs. In this section we investigate the problem of deciding whether a switching class contains any triangle-free graphs.

The algorithm is specified by means of a reduction to the 2SAT problem, a conjunction of disjunctions such that each disjunction contains at most two literals, where in our case each literal stands for a vertex in the graph. If the literal occurs positively this will mean that the vertex is in the selector corresponding to the instance of 2SAT. If it occurs negatively, it will not occur in this selector.

An example instance of this problem is $\overline{u} \wedge (u \vee v)$. A selector that satisfies this proposition does not select u, because of the first clause and hence *must* select v to make the second clause true.

In what follows we shall omit $\wedge$ from the propositions by writing, e.g., $\overline{u}(u \vee v)$. We often use $\otimes$ for the xor connective. Recall that $x \otimes y$ is equivalent to $(x \vee y)(\overline{x} \vee \overline{y})$.

Theorem 3. *Deciding whether or not a switching class contains a triangle-free graph can be decided time polynomial in the order of the graph.*

Proof. We may assume, by Lemma 4, that G contains an isolated vertex v and we set $\sigma(v) = 1$. Hence, our 2SAT formula contains the clause v. Note that our choice to select v is no restriction, because a selector is equivalent to its complement.

If G contains K_4, then we are done, because then we can conclude that G does not have a triangle-free switch (take the K_4 with isolated vertex v). Second, if the graph G is triangle-free, then we can also stop. In the other case, G contains a triangle, say T. Assume that σ is the selector that maps G into a triangle-free graph. Obviously, σ is not constant on T. In this case only one degree of freedom remains: which single vertex $u \in T$ has $\sigma(u) = 0$? If we had two vertices in T with $\sigma(u_1) = \sigma(u_2) = 0$, then $\{u_1, u_2, v\}$ would be a triangle in G^σ.

We now choose an arbitrary vertex $u \in T$ and set $\sigma(u) = 0$ and $\sigma(x) = 1$ for $x \in T - \{u\}$. All other neighbours of u in G must also be selected: if one of them, say z, was not, then $\{z, u, v\}$ would be a triangle in G^σ.

The foregoing can be summarized as follows: the selector σ satisfies the formula

$$\alpha_G(u) = v\overline{u} \bigwedge_{x \in N(u)} x$$

Note that $G|_{N(u)}$ can have no induced triangles, because then G would contain an induced K_4, a case we have already taken care of.

The only remaining vertices are those in $M(u) = V(G) - (N(u) \cup \{u, v\})$. Consider an edge $xy \in E(G|_{M(u)})$. If both of them or neither of them are selected by σ, then G^σ contains either a triangle $\{x, y, u\}$ or $\{x, y, v\}$, depending on whether $\sigma(x) = 1$ or $\sigma(x) = 0$ respectively.

In other words we obtain

$$\beta_G(u) = \alpha_G(u) \bigwedge_{xy \in E, x, y \in M(u)} (x \otimes y).$$

Note that this formula can only be satisfied if $G|_{M(u)}$ is bipartite.

Let τ be any selector that satisfies $\beta_G(u)$. Then G^τ has the following properties:

i. v is not in any triangle in G^τ,
ii. u is not in any triangle in G^τ,
iii. $G^\tau|_{N(u)\cup\{u,v\}}$ is triangle-free, and
iv. $G^\tau|_{M(u)}$ is bipartite and hence triangle-free.

We have now come this far: if σ satisfies $\beta_G(u)$, then the only triangles that can remain in G^σ involve either

– two vertices from $N(u)$ and one from $M(u)$, or
– two vertices from $M(u)$ and one from $N(u)$.

We continue now with the prevention of triangles $\{x, y, z\}$ with $x \in M(u)$ and $y, z \in N(u)$. We need to consider only triangles $\{x, y, z\}$ and the case that $G(yz) = 1, G(xy) = 0, G(xz) = 0$. This follows from the fact that $G|_{\{x,y,z\}}$ must be a switch of K_3, so it is either K_3 or an edge with an isolated vertex. In the latter case, there are three possibilities: the edge in question is xy, xz or yz. In the former two cases, yz will not be an edge in a switch G^σ of G that satisfies $\beta_G(u)$, because σ is constant on $N(u)$. So in the end only the third case, $yz \in E(G)$ needs to be considered.

We define

$$X_0 = \{x \in M(u) \mid \exists y, z \in N(u) : G(yz) = G(xy) = G(xz) = 1\},$$

$$X_1 = \{x \in M(u) \mid \exists y, z \in N(u) : G(yz) = 1, G(xy) = 0, G(xz) = 0\}$$

and take as our improved formula

$$\gamma_G(u) = \beta_G(u) \bigwedge_{x \in X_1} x \bigwedge_{x \in X_0} \overline{x}.$$

The conjunction over X_0 forces all triangles involving one vertex from $M(u)$ and two from $N(u)$ to be removed from G. On the other hand, the conjunction over X_1 prevents the introduction of triangles of this kind after switching. It is possible for $X_0 \cap X_1$ to be non-empty, but in that case the formula will not be satisfied by any selector.

For the final case, let $x, y \in M(u)$ and $z \in N(u)$. As we did earlier, we need only consider the case that the subgraph $G|_{\{x,y,z\}}$ is a switch of K_3. Every edge in $G|_{M(u)}$ will be destroyed by any selector that satisfies $\gamma_G(u)$. This immediately takes care of the case that $G(xy) = 1$ (this includes the case that $\{x, y, z\}$ is a triangle in G).

The second case is when $G(xy) = 0$. In that case either $G(xz) = 0$ and $G(yz) = 1$ or vice versa, $G(xz) = 1$ and $G(yz) = 0$. We assume the former. In that case, the only way in which we can possibly introduce a triangle is by $\sigma(x) = 0$ and $\sigma(y) = 1$. For this reason we add for all such pairs x and y the clause $x \vee \overline{y}$ to the formula $\gamma_G(u)$, obtaining in this way our final formula $\delta_G(u)$.

If there is no selector that satisfies $\delta_G(u)$, then we can construct the formula $\delta_G(u')$ for the other two vertices $u' \in T - \{u\}$. If neither of these yield a formula that can be satisfied by a selector, then the switching class contains no triangle-free graphs: the choice of T is an arbitrary one, because a selector must dissolve *all* existing triangles in G. The completeness of the algorithm then follows from the fact that the 2SAT formulas by construction only forbid selectors that are guaranteed to map into graphs with triangles.

Note that if $\delta_G(u)$ is satisfiable for some u, then we can easily construct the triangle-free graph by constructing the selector that satisfies the formula and applying it to G. We have shown that there are only three choices for u, and in all three cases, the formula has size at most quadratic in the number of vertices. 2SAT itself can be solved in time linear in the size of the formula [1] so we obtain a polynomial time complexity for our problem. □

Example 1. We shall now give a small example, one which also indicates the need for trying all three vertices of a chosen triangle.

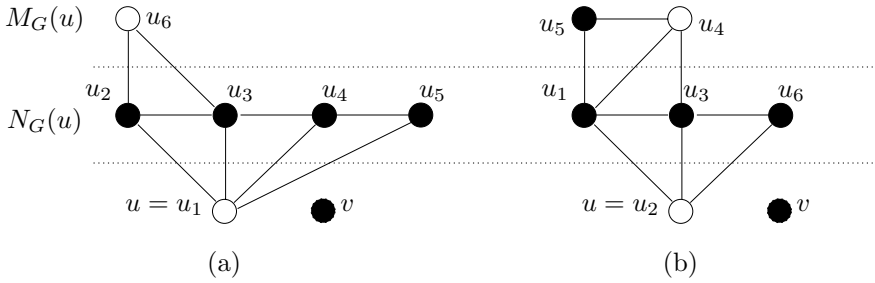


Fig. 1. Choosing $u = u_1$ does not give us a switch, choosing $u = u_2$ does.

In Figure 1 we have depicted the same graph twice, with different choices for u. Choosing u_1 for u does not give us a switch into a triangle-free graph (see Figure 1(a)). The reason is that the triangle $\{u_2, u_3, u_6\}$ must be removed (so u_6 must not be switched), but this induces a triangle $\{u_4, u_5, u_6\}$. We get a similar situation for $u = u_3$. However, if we choose $u = u_2$, then the situation of Figure 1(b) arises, for which we can construct the formula

$$\delta_G(u_2) = v\overline{u}_2u_1u_3u_6(u_4 \vee u_5)(\overline{u}_4 \vee \overline{u}_5)\overline{u}_4u_5.$$

In this case there is one selector that satisfies the formula, the one which selects u_5, but does not select u_4 (the selector also selects all other vertices except u_2). Applying this selector to the graph results in the simple cycle of length 7.

5 Bipartite Switching Classes

In this section we consider the problem of determining whether a switching class contains a bipartite graph.

Starting from a bipartite graph and applying an arbitrary switch, the resulting graph is always 4-colourable (Lemma 3.30 in [8]). Hence if we start from a bipartite graph, say $G = (V, E)$ with bipartition $A, V - A$, and we switch with respect to a selector σ then we obtain the graph G^σ, in which

- the sets $\sigma \cap A$, $\sigma \cap (V - A)$, $(V - \sigma) \cap A$ and $(V - \sigma) \cap (V - A)$ are independent,
- the sets $\sigma \cap A$ and $(V - \sigma) \cap A$ are completely connected,
- as are the sets $(V - \sigma) \cap (V - A)$ and $V \cap (V - A)$.

Note that up to three of the four sets may be empty. An example where $A \cap \sigma$ is empty is given in Figure 2.

In the following we shall derive a 2SAT formula which specifies for each vertex, whether it must be in either the top or lower complete bipartite graph, or it may be in either one. We begin however with a small but essential lemma.

Lemma 5. *A graph G is complete bipartite if and only if G does not have an induced K_3 nor $K_2 \oplus K_1$.*

Proof. By Lemma 2, all complete bipartite graphs can be switched into the discrete graph, and these are the only graphs for which this holds.

The result on triangles by Seidel [13] states that the parities of edges in any induced subgraph on three vertices is invariant under switching and, the other way around, if two graphs (on the same vertex set) have the same parities for corresponding triangles, then they can be switched into each other.

The invariance implies that for all complete bipartite graphs, the parity of edges of T is zero, as it is for the discrete graph. But the parity of edges in both K_3 and $K_2 \oplus K_1$ is one, and they are the only such graphs on three vertices. □

Theorem 4. *Deciding that a switching class contains a bipartite graph or not can be done in time cubic in the number of vertices in the graph.*

Proof. Let $G = (V, E)$ be a graph in the switching class. We start by fixing a single vertex $u \in V$ to go "up". "Up" refers to one of the sets in the partition of the bipartite graph whose existence we are trying to prove (the set A in Figure 2). Up will be represented as 0 (or, equivalently, false) in our formulas, down will be represented by 1 (true).

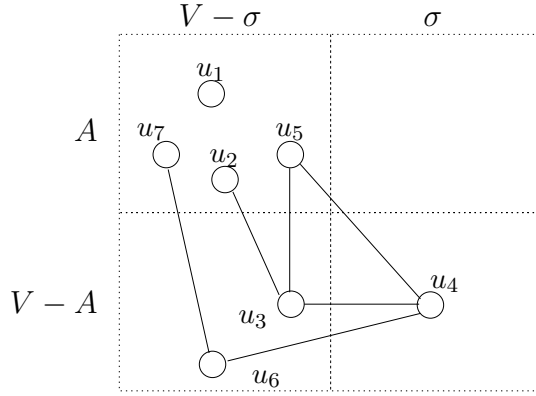


Fig. 2.

By Lemma 5, the only thing to take care of, is to forbid subgraphs K_3 and $K_2 \oplus K_1$. The formula that forbids K_3 and $K_2 \oplus K_1$ in the top complete bipartite graph (assuming that u is part of it) is as follows (here $V' = V - \{u\}$):

$$up_G(u) = \bigwedge\{x \vee y \mid G(xu) = G(yu), G(xy) = 1, x, y \in V'\} \wedge \bigwedge\{x \vee y \mid G(xu) \neq G(yu), G(xy) = 0, x, y \in V'\}$$

The first part of the formula takes care of the case that xy is an edge in G. We then have to force one of x and y down to the lower part (i.e., one of them

must be true). This includes both the case of K_3 and one of the three cases of an induced $K_2 \oplus K_1$. The second part of the formula takes care of the case where xy is not an edge in G, but we do have a subgraph $G|_{\{u,x,y\}}$ isomorphic to $K_2 \oplus K_1$ (these are the other two cases for $K_2 \oplus K_1$).

The formula above only forces vertices down. This is not enough to guarantee that on the down side we always obtain a complete bipartite graph (in fact you can always choose all vertices but u to go down, so pretty much every graph can arise there). To set things right we do the same as above, but now we construct a formula $down_G(v)$ for a vertex v, that tells us of which pairs of vertices at least one has to go to the "up" side. This formula only differs from $up_G(v)$ in the fact that every literal occurs complemented, because we have to exchange "up" with "down" (here $V' = V - \{v\}$):

$$down_G(v) = \bigwedge\{\overline{x} \vee \overline{y} \mid G(xv) = G(yv), G(xy) = 1, x, y \in V'\} \wedge \bigwedge\{\overline{x} \vee \overline{y} \mid G(xv) \neq G(yv), G(xy) = 0, x, y \in V'\}$$

Finally, we can construct the 2SAT formula

$$c_G(u, v) = up_G(u) \wedge down_G(v),$$

for different vertices u and v. In $c_G(u, v)$ a small optimization is possible by taking $V' = V - \{u, v\}$ in $up_G(u)$ and $down_G(v)$.

The above reasoning should make clear that every way of satisfying the formula guarantees that the graph G is split into two, arbitrarily connected, complete bipartite graphs. It does not yet guarantee that if the formula is not satisfiable, then the graph cannot be partitioned into two complete biparite graphs. This can be done by fixing u and to let v run through $V(G) - \{u\}$: if a graph has a bipartite switch, then there has to be a partition in which u goes up and one of the other vertices goes down. After fixing u to belong to the top complete bipartite graph, we have assumed all we can, and we simply have to try all other vertices for v.

We have now shown how the problem of deciding whether a switching class is bipartite can be reduced to finding a solution to a linear number of 2SAT problems. Since 2SAT can be solved in time linear in the size of the formula [1], and we construct each of the formulas in time quadratic in the number of vertices, we obtain an algorithm cubic in the order of the graph. □

Example 2. Take the graph G depicted in Figure 2. The selector $\{u_4\}$ gives us a bipartite graph.

The picture already implies that if we choose $u = u_1$ and $v = u_4$, then we should obtain a satisfiable formula $c(u_1, u_4)$, for which we need

$$up_G(u_1) = (u_2 \vee u_3)(u_3 \vee u_5)(u_6 \vee u_7)$$

and

$$down_G(u_4) = (\overline{u}_2 \vee \overline{u}_6)(\overline{u}_2 \vee \overline{u}_5)(\overline{u}_3 \vee \overline{u}_5)(\overline{u}_3 \vee \overline{u}_7)(\overline{u}_5 \vee \overline{u}_7)$$

Here we have given the "optimized" version. The reader can now verify that setting $u_3 = u_6 = 1$ and $u_2 = u_5 = u_7 = 0$ makes $c(u_1, u_4)$ true.

Choosing $u = u_1$ and $v = u_7$ yields a formula $c(u_1, u_7)$ which is not satisfiable. This is because both vertices are not connected to any vertex in the triangle $\{u_3, u_4, u_5\}$. We get

$$up_G(u_1) = (u_3 \vee u_4)(u_3 \vee u_5)(u_4 \vee u_5)\ldots$$

and

$$down_G(u_7) = (\overline{u}_3 \vee \overline{u}_4)(\overline{u}_3 \vee \overline{u}_5)(\overline{u}_4 \vee \overline{u}_5)\ldots$$

One of u_3, u_4 and u_5 must be assigned 1. Assume, without loss of generality, that this is u_3. But then $(\overline{u}_3 \vee \overline{u}_4)$ implies that $u_4 = 0$. This implies that $u_5 = 1$ (because of $(u_4 \vee u_5)$), but also that $u_5 = 0$ (by $(\overline{u}_3 \vee \overline{u}_5)$). The example shows that we may choose a "wrong" combination of u and v: one that does not yield a bipartite switch. However, it is guaranteed that whatever our choice for u: if the graph has a bipartite switch, then there exists a vertex $v \neq u$ for which the constructed formula is satisfiable.

6 Some Final Words

In this paper we have shown for three different graph problems that the same problem for switching classes is polynomial. From the above we may conclude that if G has a bipartite switch, then G is 4-colourable. Hence the algorithm implied in the theorem can be used as an approximative algorithm for this problem and will find a 4-colouring in polynomial time. There is no guarantee however that the colouring of G is minimal. More research is needed to determine to what extent this approximative algorithm is of use. Not surprisingly, in view of the P versus NP conjecture, there are graphs which are 4-colourable, but which do not have a bipartite switch.

References

1. B. Aspvall, M.F. Plass, and R.E. Tarjan. A linear-time algorithm for testing the truth of certain quantified boolean formulas. *Inf. Proc. Letters*, 8(3):121 – 123, 1979.
2. P.J. Cameron. Cohomological aspects of two-graphs. *Math. Z.*, 157:101 – 119, 1977.
3. D.G. Corneil and R.A. Mathon, editors. *Geometry and Combinatorics: Selected Works of J.J. Seidel.* Academic Press, Boston, 1991.
4. A. Ehrenfeucht, J. Hage, T. Harju, and G. Rozenberg. Complexity issues in switching of graphs. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Theory And Application Of Graph Transformations - TAGT '98*, volume 1764 of *Lecture Notes in Computer Science*, pages 59–70, Berlin, 2000. Springer-Verlag.
5. A. Ehrenfeucht, T. Harju, and G. Rozenberg. *The Theory of 2-Structures.* World Scientific, 1999.

6. A. Ehrenfeucht and G. Rozenberg. Dynamic labeled 2-structures. *Mathematical Structures in Computer Science*, 4:433–455, 1994.
7. J.L. Gross and T.W. Tucker. *Topological Graph Theory.* Wiley, New York, 1987.
8. J. Hage. *Structural Aspects Of Switching Classes.* PhD thesis, LIACS, 2001. http://www.cs.uu.nl/people/jur/2s.html.
9. J. Hage and T. Harju. A characterization of acyclic switching classes using forbidden subgraphs. Technical Report 5, Leiden University, Department of Computer Science, 2000. Submitted to Siam J. Disc. Math.
10. J. Kratochvíl, J. Nešetřil, and O. Zýka. On the computational complexity of Seidel's switching, in: Combinatorics, Graphs and Complexity (M.Fiedler and J.Nešetřil eds.) Proceedings 4th Czechoslovak Symposium on Combinatorics, Prachatice 1990. *Annals of Discrete Math.*, 51:161 – 166, 1992.
11. C.L. Mallows and N.J.A. Sloane. Two-graphs, switching classes and Euler graphs are equal in number. *SIAM J. Appl. Math*, 28:876 – 880, 1975.
12. J.J. Seidel. Graphs and two-graphs. In *Proc. 5th Southeastern Conf. on Combinatorics, Graph Theory, amd Computing*, Winnipeg, Canada, 1974. Utilitas Mathematica Publishing Inc.
13. J.J. Seidel. A survey of two-graphs. In *Colloquio Internazionale sulle Teorie Combinatorie (Rome,1973)*, volume I, pages 481–511, Rome, 1976. Acc. Naz. Lincei. Reprinted in [3].
14. J.J. Seidel and D.E. Taylor. Two-graphs, a second survey. In L. Lovasz and V.T. Sós, editors, *Algebraic Methods in Graph Theory (Proc. Internat. Colloq., Szeged, 1978)*, volume II, pages 689–711, Amsterdam, 1981. North-Holland. Reprinted in [3].
15. T. Zaslavsky. Biased graphs. I. Bias, balance, and gains. *J. Combin. Theory, Ser. B*, 47:32–52, 1989.

Confluence of Typed Attributed Graph Transformation Systems

Reiko Heckel, Jochen Malte Küster, and Gabriele Taentzer

University of Paderborn, Paderborn, Germany
Technical University of Berlin, Berlin, Germany
{reiko,jkuester}@upb.de, gabi@cs.tu-berlin.de

Abstract. The issue of confluence is of major importance for the successful application of attributed graph transformation, such as automated translation of UML models into semantic domains. Whereas termination is undecidable in general and must be established by carefully designing the rules, local confluence can be shown for term rewriting and graph rewriting using the concept of critical pairs. In this paper, we discuss typed attributed graph transformation using a new simplified notion of attribution. For this kind of attributed graph transformation systems we establish a definition of critical pairs and prove a critical pair lemma, stating that local confluence follows from confluence of all critical pairs.

1 Introduction

Graph transformation is increasingly popular as a meta-language to specify and implement visual modelling techniques, like the UML. It may be used for parsing visual languages [1] and for automated translation of visual models into code or semantic domains [6,19], or as a semantic domain itself [13,9]. Often, it is important to know whether the graph transformation system shows a functional behavior (is terminating and confluent) or if there are conflicts between rule applications that lead to true non-determinism. For example, functional behavior avoids the overhead of backtracking in the case of parsing and semantic ambiguity in the mapping of models to semantic domains.

For term rewrite systems confluence can be shown using the concept of critical pairs. Critical pairs which can be detected and analyzed statically, represent potential conflicts in a minimal context. If the rewrite system is terminating, confluence follows if all critical pairs can be joined [15].

This theory of critical pairs and confluence has been transferred to transformation systems on term graphs and hyper graphs [17,18]. However, in most applications of graph transformation to visual modelling techniques, *attributed* graphs are used to represent diagrams with textual, numerical, or layout information, semantic annotations, etc. To develop the theory of critical pairs and confluence in this case is the aim of this paper.

In particular, we shall be motivated by the problem of translating diagrams into a formal specification language for automatic verification. Such a mapping

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 161–176, 2002.
© Springer-Verlag Berlin Heidelberg 2002

can be described by attributed graph transformation rules based on the graphical presentation of the abstract syntax of the diagrams extended by semantic attributes that contain the results of the translation [10]. The translation has to be functional, i.e., terminating and confluent in order to ensure the existence of a unique result. As an example we present a translation of simple UML statecharts into CSP [11] for automated verification by means of a CSP model checker [8].

For this purpose, we introduce typed, vertex-attributed graph transformation systems. A critical pair lemma is established which states that a graph transformation system is locally confluent if all critical pairs are confluent. Thus, confluence can be shown by computing all critical pairs and demonstrating their confluence.

In the following, we first introduce typed attributed graph transformation and present, as a running example, rules for translating UML statecharts to CSP. Thereafter, critical pairs are defined and the critical pair lemma is stated. Lastly, the critical pairs in our example are discussed and the available tool support is described.

2 Typed Attributed Graph Transformation

Next, we present the *algebraic double-pushout (DPO) approach* [5] to the transformation of typed attributed graphs [2]. The two basic ingredients are graphs, representing object structures, and algebras representing pre-defined abstract data types. Attributed graphs occur at two levels: the type level (modelling a schema or class diagram) and the instance level (modelling an individual system snapshot).

Attributed graphs. By a *graph* we mean a directed unlabelled graph $G = \langle G_V, G_E, src^G, tar^G \rangle$ with a set of vertices G_V, a set of edges G_E, and functions $src^G : G_E \to G_V$ and $tar^G : G_E \to G_V$ associating to each edge its source and target vertex. A graph homomorphism $f : G \to H$ is a pair of functions $\langle f_V : G_V \to H_V, f_E : G_E \to H_E \rangle$ preserving source and target.

To speak about algebras throughout the paper, we assume a many-sorted signature $\Sigma = \langle S, OP \rangle$ consisting of a set of sort symbols $s \in S$ and a family of sets of operation symbols $op : s_1 \dots s_n \to s \in OP$ indexed by their arities.

Definition 1 (attributed graphs and morphisms). *An attributed graph (over Σ) is a pair $AG = \langle G, A \rangle$ of a graph G and a Σ-algebra A such that $\mid A \mid \subseteq G_V$, where $\mid A \mid$ is the disjoint union of the carrier sets A_s of A, for all $s \in S$, and such that $\forall\, e \in G_E : \ src(e) \notin \mid A \mid$. Let $Attr(AG) = \{e \in G_E \mid tar(e) \in \mid A \mid\}$, $Graph(AG) = G \setminus (\mid A \mid + Attr(AG))$ and $Alg(AG) = A$.*

An attributed graph morphism $f : \langle G_1, A_1 \rangle \to \langle G_2, A_2 \rangle$ *is a pair of a Σ-homomorphism $f_A = (f_s)_{s \in S} : A_1 \to A_2$ and a graph homomorphism $f_G = \langle f_V, f_E \rangle : G_1 \to G_2$ such that $\mid f_A \mid \subseteq f_V$, where $\mid f_A \mid = \bigcup_{s \in S} f_s$.*

Attributed graphs and graph morphisms form a category of Σ-attributed graphs $\mathbf{AGraph}(\Sigma)$. Often, we will fix the data algebra A in advance—in this case we also speak of a graphs and graph morphisms attributed over A.

Summarizing, data values are represented as vertices of graphs, henceforth called *data vertices* $d \in| A |$ to distinguish them from *object vertices* $v \in G_V \setminus | A |$. Object vertices are linked to data vertices by *attributes*, i.e., edges $a \in G_E$ with $src(a) = v$ and $tar(a) = d$. Edges between object vertices are called *links*. We have assumed that there are no edges from data vertices.

Compared with other notions of attributed graphs, like [14], where special attribute carriers are used to relate graph elements and data values, our presentation is simpler because attributed graphs are regarded as a special case of ordinary graphs. However, this limits us to attributed vertices.

Typed graphs. The concept of typed graphs [2] captures the well-known dichotomy between classes and objects, or between database schema and instance, in the case of graphs. Below, it is extended to attributed graphs.

Definition 2 (typed attributed graphs). *An* attributed type graph *over* Σ *is an attributed graph* $ATG = \langle TG, Z \rangle$ *over* Σ *where* Z *is the final* Σ*-algebra* Z *having* $Z_s = \{s\}$ *for all* $s \in S$.

An attributed instance graph $\langle AG, ag \rangle$ *over* ATG *is an attributed graph* AG *(over the same signature) equipped with an attributed graph morphism* $ag : AG \to ATG$.

A morphism of typed attributed graphs $h : \langle AG_1, ag_1 \rangle \to \langle AG_2, ag_2 \rangle$ *is a morphism of attributed graphs which preserves the typing, that is,* $ag_2 \circ h = ag_1$.

Thus, elements of Z represent the sorts of the signature which are included in TG as types for data vertices. In general, vertices and edges of TG represent vertex and edge types, while attributes in ATG are, in fact, attribute declarations. Given an attribute declaration $a \in ATG$ and an object vertex $v \in AG$ such that $ag(v) = src(a)$ we write $a(v)$ to denote the set of v's a-values $\{d \in| A | \mid \exists\, e \in AG_E.\ src(e) = v \wedge tar(e) = d\}$.

Instance graphs will be usually infinite, e.g. if the data type $\mathbb{N}$ of natural numbers is present, each $n \in \mathbb{N}$ will be a separate vertex. However, since the data type part will be kept constant during transformation, there is no need to represent this infinite set of vertices as part of the current state. The examples shown contain only those data vertices connected to some object vertex.

Sample type and instance graphs. For the translation of statecharts to CSP, the UML metamodel has to be flattened and inheritance must be removed by simulating it through additional attributes. In Fig. 2, an abridged metamodel for statecharts is shown as attributed type graph where types of model elements are depicted by rectangles while data types are shown as ellipses. Note that the inscriptions of the nodes and edges are the node and edge identities and no labels. The inscriptions of the data types such as String refers to the sort symbols in the signature given in Fig. 1.

The type graph contains cardinality constraints in UML-like notation, which restrict the number of in and outgoing edges of vertices. The formal treatment of such constraints, however, is beyond the scope of this paper.

SORTS

String, CSPEq, CSPExpr, Char

OPNS

empty: ->String
concat: String, String -> String
concat: String, Char -> String
empty: -> CSPExpr
stop: -> CSPExpr
[]: -> CSPExpr
->: string, CSPExpr -> CSPExpr
State: string -> CSPExpr
=: CSPExpr, CSPExpr -> CSPEq

Fig. 1. Signature for CSP translation

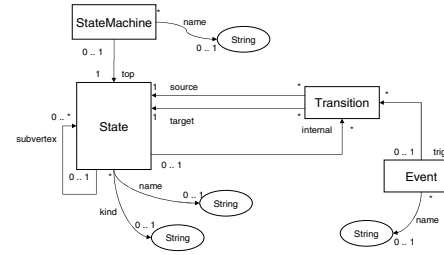


Fig. 2. Type graph for statecharts, derived from the UML metamodel

In Fig. 3, a simple instance graph for a statechart is shown. On the left, the formal representation is depicted with attribute values being modelled as vertices of the graph whereas on the right the UML-like syntax is given with the short-hand for attributes modelled inside a compartment of a class vertex. In the following, we will use this short-hand for space reasons and conformity with UML syntax.

Graph transformation. The DPO approach to graph transformation has originally been developed for vertex- and edge-labeled graphs [5]. Here, we present the typed version [2] extended to attributed graphs.

According to the DPO approach, graph transformation rules (also called graph productions), are specified by pairs of injective graph morphisms ($L \xleftarrow{l} K \xrightarrow{r} R$), called rule spans. The left-hand side L contains the items that must be present for an application of the rule, the right-hand side R those that are present afterwards, and the gluing graph K specifies the "gluing items", i.e., the objects which are read during application, but are not consumed. The transformation of graphs is defined by a pair of pushout diagrams, a so-called double pushout.

Definition 3 (DPO graph transformation). *Given an attributed type graph ATG and fixing a sort-indexed family of sets of variables $X = (X_s)_{s \in S}$, an ATG-typed graph transformation rule over X is a span of injective graph morphisms $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ over ATG such that L, K, R are attributed over $T_\Sigma(X)$ and l, r are identities on $T_\Sigma(X)$. If we are not interested in the gluing graph K we write $p : L \curvearrowright R$.*

A double-pushout (DPO) diagram *o is a diagram like below where (1), (2) are pushouts and top and bottom are rule spans. Given a rule p a* (direct DPO) transformation *from G to H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a DPO diagram where g, h are identities on $Alg(D)$.*

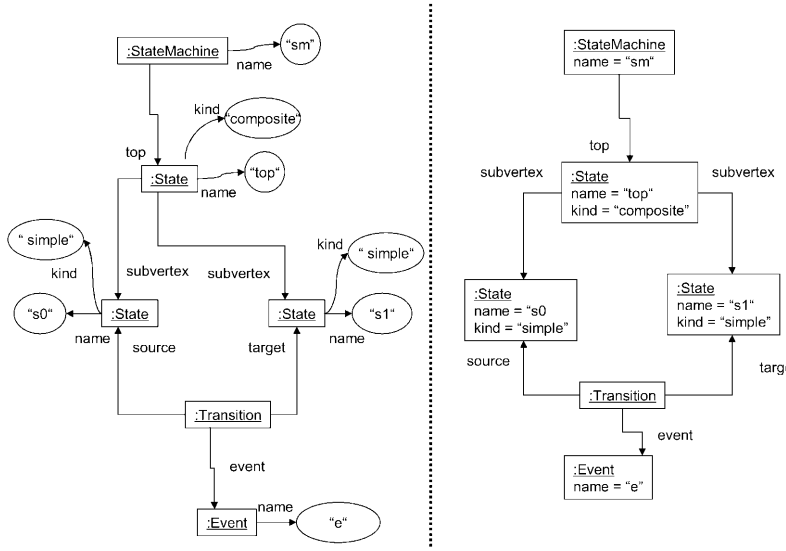


Fig. 3. Instance graph: formal vs UML-like syntax

$$\begin{array}{ccccc} L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ {\scriptstyle o_L}\downarrow & (1) & \downarrow{\scriptstyle o_K} & (2) & \downarrow{\scriptstyle o_R} \\ G & \xleftarrow[g]{} & D & \xrightarrow[h]{} & H \end{array}$$

The DPO diagram o is a categorical way of representing the occurrence of a rule in a bigger context. Operationally, it formalizes the replacement of a subgraph in a graph by two gluing diagrams, called pushouts. The left-hand side pushout (1) is responsible for removing the occurrence of $L \setminus l(K)$ in G, resulting in graph D. The right-hand side pushout (2) adds a copy of $R \setminus r(K)$ to D leading to the derived graph H.

The construction of pushout (1) requires that only objects in the image of K may be merged or (in the case of vertices) connected to edges in the context. This is reflected, respectively, in the *identification* and the *dangling condition* of the DPO approach, i.e. the *gluing condition*. Given a rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ and an occurrence $o_L : L \to G$ of the left-hand side, the existence of the pushout complement (1), and hence of a direct derivation[1] $G \overset{p(o)}{\Longrightarrow} H$ is characterized by the satisfaction of the gluing condition. The *identification condition* states that objects from the left-hand side may only be identified by the match if they also belong to the interface (and are thus preserved). The *dangling condition* ensures

[1] Pushout (2) always exists, since category $\frown Graph_{TG}$ is cocomplete due to the cocompleteness of category $\frown Graph$.

that graph D obtained by removing all objects that are to be deleted from G, is indeed a graph, i.e. no edges are left "dangling" without source or target node.

Definition 4 (graph transformation system). *A graph transformation system $GTS = (\Sigma, ATG, X, R)$ consists of a data type signature Σ, an attributed type graph ATG over Σ, a family of variables X over Σ, and a set of attributed graph transformation rules R over ATG and X.*

A transformation sequence $G_0 \overset{*}{\Longrightarrow} G_n = G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n$ in GTS is a sequences of consecutive transformation steps such that G_0 is typed over ATG and all rules p_i come from R.

Rules for mapping statecharts to CSP. Mapping rules for statecharts to the process algebra of Communicating Sequential Processes (CSP [11]) can be described by attributed graph transformation rules [6]. Such a rule consists of a UML metamodel instance extended by semantic attributes and control attributes. Within the semantic attributes, the actual computation of CSP expressions for the statechart is performed. Control attributes drive the order of rule applications. If an attribute is changed applying a rule, we use an assignment notation.

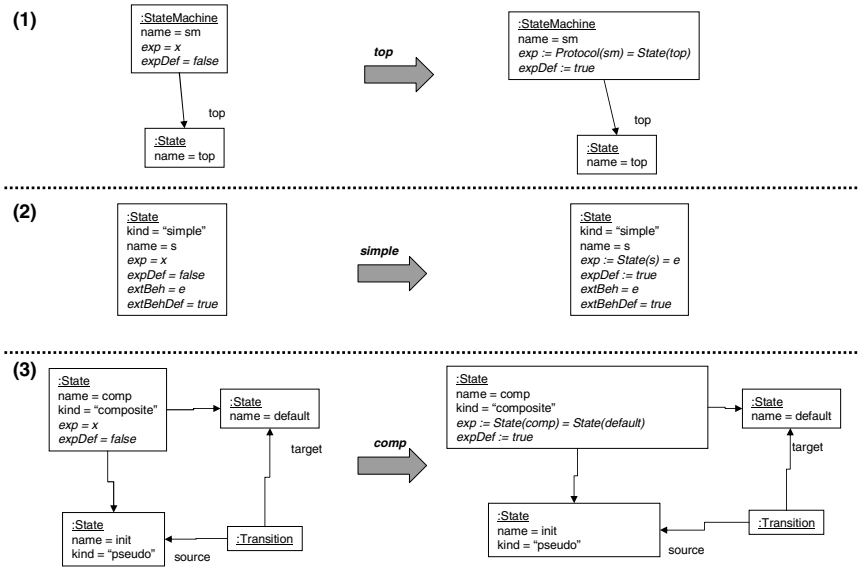


Fig. 4. Mapping rules for states

Consider, for example, rule (3) in Fig. 4 which defines the semantics of a composite (OR) state in terms of the semantics of its default state. The original attributes from the metamodel are represented in plain font whereas the semantic and control attributes are printed in italics. In this case, the composite state is annotated with a semantic attribute exp and a control attribute expDef.

Application of this rule leads to an exp set to a CSP expression defining the behavior of the composite state to be the behavior of the default state, and a change in the control attribute expDef from false to true, thereby hindering another application of this rule to the same composite state.

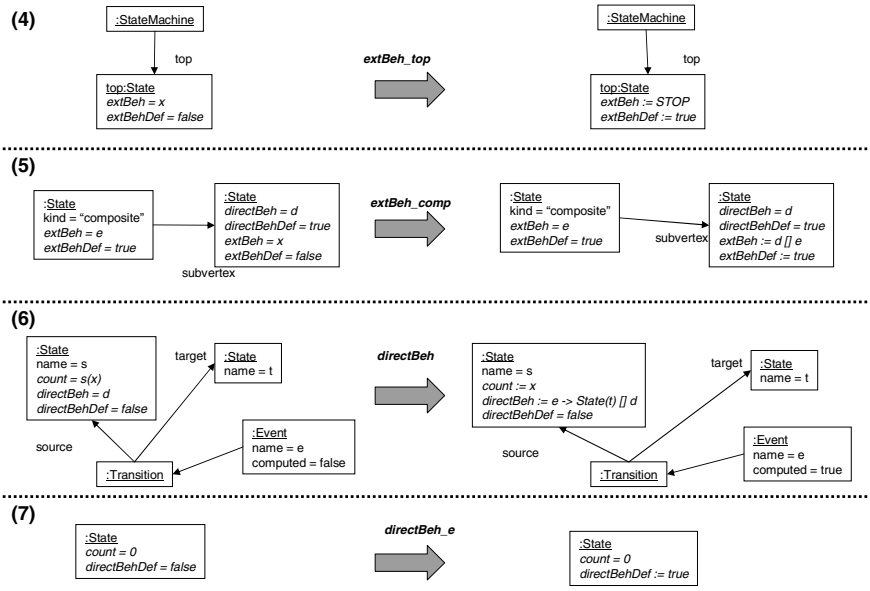


Fig. 5. Mapping rules for the behavior

In general, termination is undecidable for graph transformation systems. However, concerning our mapping rules, termination results from the following argument: Each rule is annotated with a finite number of control attributes. These control attributes take on a number of finite values (mostly true or false or a natural number). As each rule application either decreases a control attribute or changes its value from false to true, after a finite number of rule applications no rule will be applicable anymore, leading to termination of the mapping process.

3 Critical Pairs and Confluence

Independent transformations. The definition of parallel independence demands that the occurrences of two independent transformations do only share items which are preserved by both steps. Under this assumption, the local Church-Rosser theorem states that the two steps can be executed in any order with the same overall result [4].

Given two transformations $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ and $G \overset{p_2(o_2)}{\Longrightarrow} H_2$, $G \overset{p_1(o_1)}{\Longrightarrow} H_1$ is *(weakly) parallel independent* of $G \overset{p_2(o_2)}{\Longrightarrow} H_2$ if the occurrence $o_1(L_1)$ of the

left-hand side of p_1 is preserved by the application of p_2. This is the case if $o_1(L_1) \cap o_2(L_2 \setminus g_2(D_2)) = \emptyset$, that is, $o_1(L_1)$ does not overlap with objects that are deleted by p_2. If the two transformations are mutually independent, they can be applied in any order yielding the same result. In this case we speak of *parallel independence*. Otherwise, if one of two alternative transformations is not independent of the second, the second will disable the first. In this case, the two steps are *in conflict*.

Proposition 1 (local Church-Rosser theorem). *Given two parallel independent transformations* $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ *with* $H_1 \overset{h_1}{\longleftarrow} C_1 \overset{g_1}{\longrightarrow} G \overset{g_2}{\longleftarrow} C_2 \overset{h_2}{\longrightarrow} H_2$*, there are transformations* $H_1 \overset{p_2(o_2')}{\Longrightarrow} X$ *and* $H_2 \overset{p_1(o_1')}{\Longrightarrow} X$ *with* $o_1' = h_2 \circ g_2^{-1} \circ o_1$ *and* $o_2' = h_1 \circ g_1^{-1} \circ o_2$.

The local Church-Rosser theorem has been shown for colored graphs in [4]. Simply rephrasing its proof for typed graphs would lead to the proof of proposition 1.

In the case of attributed graph transformation, where attribute values are modelled as vertices and attribute links are edges, any modification of an attribute corresponds to a deletion of an attribute link and the generation of a new one. Therefore, two steps which modify the same attribute, are in conflict.

Critical pairs. A system is locally confluent if all conflicting pairs are *confluent*, that is, they are extendible by transformation sequences leading to a common successor graph. In order to check this in finite time, the potentially infinite set of conflicting pairs has to be reduced to a finite set of representatives. This is the aim of the construction of *critical pairs*, which produces all conflicting pairs of steps. A critical pair is *minimal*, i.e., it does not contain unnecessary context. It is also *syntactic*, meaning that it is attributed over a term algebra $T_\Sigma(X)$ or a quotient term algebra $T_\Sigma(X)/_\equiv$ for congruence relation $\equiv$ specified by a set of equational axioms or implemented by an equality predicate of an abstract data type.

Definition 5 (critical pairs). *Assume a graph transformation system* $GTS = (\Sigma, ATG, X, R)$*, a congruence* $\equiv \subseteq T_\Sigma(X) \times T_\Sigma(X)$*, and two rules* $p_1 : L_1 \curvearrowright R_1$ *and* $p_2 : L_2 \curvearrowright R_2$ *of* R *using disjoint subsets of variables of* X *in their attribute terms.*

A critical pair candidate for p_1 and p_2 wrt. $\equiv$ *is a pair of non-parallel independent transformations* $CP(p_1, k_1, K, p_2, k_2) = P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(k_2)}{\Longrightarrow} P_2$*, with* K *attributed over the quotient term algebra* $T_\Sigma(X)/_\equiv$.

A critical pair for p_1 and p_2 wrt. $\equiv$ *is a minimal element among the candidates w.r.t. the partial order* $\sqsubseteq$ *defined by* $CP(p_1, k_1, K, p_2, k_2) \sqsubseteq CP(p_1, k_1', K', p_2, k_2')$ *iff there exists a morphism* $k : K \to K'$*, injective on* $Graph(K)$ *and* $Attr(K)$ *(but not necessarily on* $Alg(K)$*), such that* $k_1' = k \circ k_1$ *and* $k_2' = k \circ k_2$.

For the graph structure, minimality means that no unnecessary context is present, i.e. k_1 and k_2 are jointly surjective. For the algebra part, the minimality

condition generalizes the idea of a most general unifier, that is, a substitution $\sigma : X \to T_\Sigma(X)$ with as little instantiation of variables as needed to equate two terms. Thus, $\sqsubseteq$ always has a set of minimal elements.

Next we consider three cases where the set of critical pairs for two given rules is finite (up to isomorphic copies) and can be effectively computed. In each case, we assume as given an overlapping $Graph(K)$ of the graph structures $Graph(L_1)$ and $Graph(L_2)$ of L_1 and L_2 with two graph morphisms $k_{i\,G} : Graph(L_i) \to Graph(K)$. The set of these overlappings is finite (up to isomorphism) if the graphical parts of L_1 and L_2 are both finite. The problem consists in checking if such an overlapping can be extended to the data type part, i.e., if a $T_\Sigma(X)/_\equiv$-attributed graph K exists with Σ-homomorphisms $k_{i\,A} : T_\Sigma(X) \to T_\Sigma(X)/_\equiv$ such that $\langle k_{i\,G}, k_{i\,A}\rangle : L_i \to K$ form attributed graph morphisms. We restrict our considerations to the case of single-valued (rather than multi-valued) attributes.

Case 1. First, we assume that the congruence $\equiv$ is trivial, i.e., it contains only the syntactic identities. In this case, $T_\Sigma(X)/_\equiv = T_\Sigma(X)$, that is, K is attributed over terms with variables of X. The attribute term for a vertex v in K is obtained by computing the most general unifier of all pre-images of v under k_1 and k_2.

More precisely, call $U(a, v) \subseteq T_\Sigma(X)$ the *unification set* for an attribute a of object vertex $v \in K$, given by $U(a, v) = \{t \mid (v = k_1(v_1) \wedge a(v_1) = t) \vee (v = k_2(v_2) \wedge a(v_2) = t)\}$, and enumerate the unification sets for all vertices v and all relevant attributes a as $U_1, \ldots, U_n$.

Now, a candidate $CP(p_1, k_1, K, p_2, k_2)$ is *attribute unifiable* if there exist substitutions σ_i such that $\sigma_1 = mgu(U_1)$ and, for all $j \in \{2, \ldots, n\}$, $\sigma_j = mgu(\sigma_{j-1}(U_j))$ where $mgu(U_i)$ computes the most general unifier of the set of terms U_i, if it exists.

If $CP(p_1, k_1, K, p_2, k_2)$ is attribute unifiable, the value of an attribute a for a vertex v in K is $\sigma_n(t)$ for any $t \in U(a, v)$. The algebra homomorphism part of both k_1 and k_2 is the free homomorphic extension to $T_\Sigma(X)$ of the same $\sigma_n : X \to T_\Sigma(X)$. If $CP(p_1, k_1, K, p_2, k_2)$ is not attribute unifiable, there is no critical pair based on this gluing of graphs.

Case 2. A second, more general variant allows a congruence $\equiv$ specified by a set of equational axioms, represented computationally by a confluent and terminating term rewrite system. In this case, the normal forms of this rewrite system can be used as unique representatives of their equivalence classes so that, effectively, a graph attributed over $T_\Sigma(X)/_\equiv$ can be represented as a $T_\Sigma(X)$-attributed graph. Since for normal forms, equivalence coincides with syntactic equality, we can reuse the construction of Case 1 by transforming the attributes terms in L_1 and L_2 to their normal forms, performing the unification, and attributing the graph K with the representatives of the equivalence classes of the resulting terms.

Case 3. Finally, we may allow any congruence which can be decided on ground terms, like the equivalence on CSP processes which is checked by the FDR tool [8]. In this case, attributes in the left-hand sides of the rules have to be

restricted to ground terms and variables, and to merge two attribute values we may either check their equivalence, if both are ground terms, or apply a substitution, if one is a variable.

The three cases can occur in combinations. In general we may use a different implementation for every sort of the algebra, and each of these implementations determines certain restrictions for the terms in the left-hand sides of rules. It shall be noted that these restrictions are not only relevant to the effective construction of critical pairs, but also to the transformation of graphs attributed over equivalence classes of terms. Here, unification is replaced by pattern matching of terms in the rules with (equivalence classes of) terms in the graphs to be transformed and, depending on the implementation of this equivalence, this pattern matching may be limited to purely syntactic matching in case 1, up to checking for equivalence of ground terms in case 3.

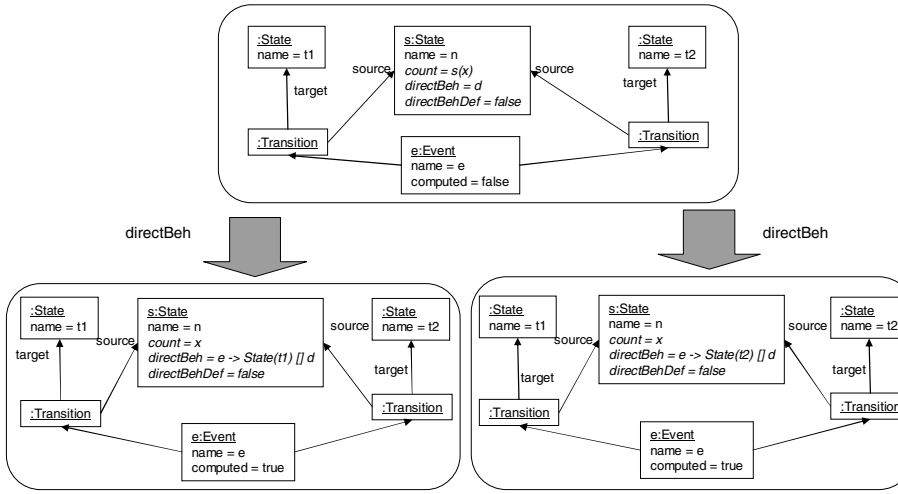


Fig. 6. A critical pair

A sample critical pair. In Fig. 6, two conflicting transformations on an overlapping graph of rule directBeh with itself are shown. Note that there are further overlapping graphs of rule directBeh with itself. In the present case, the gluing condition is satisfied as only attribute values are changed by the application of these rules. Furthermore, the transformations are not parallel independent, because the two rules overlap in state s and event e. Attribute *count* is changed (i. e. deleted and created), and therefore this is clearly a critical pair as far as the graphical structure is concerned. In order to decide whether this extends to the attribute part one has to consider the underlying equivalence relation on terms. In our case, for sort CSPEq, we assume an equivalence of CSP processes

like *failures equivalence* of processes, which can be checked by the FDR tool [8], thereby leading to case 3. We further assume for sort Bool that the equivalence relation is trivial (case 1) and for sort Nat a terminating and confluent term rewrite system (case 2). By construction of the overlapping graph, the critical pair candidate is minimal with respect to object vertices. We now compute the attributes of K. Concerning attribute computed of event e and directBehDef of state s, this is clearly the term false. Concerning attribute count, $s(x)$ and $s(y)$ are unified to $s(x)$. Finally, with respect to attribute directBeh, the two terms d and f are unified to d.

Embedding and completeness. Critical pairs cover all possible conflicting situations, which can be obtained by embedding the critical pairs into a larger context and instantiating their attributes. That means, to apply the same rules at essentially the same occurrence in a bigger graph.

Proposition 2 (completeness of critical pairs). *Assume a graph transformation system $GTS = (\Sigma, ATG, X, R)$, a congruence $\equiv \subseteq T_\Sigma(X) \times T_\Sigma(X)$, and two conflicting transformation steps $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ in GTS with $Alg(H_i) = Alg(G) = A$ such that there exists a Σ-homomorphism $m : T_\Sigma(X)/_\equiv \to A$. In this case, there exists a critical pair $P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(k_2)}{\Longrightarrow} P_2$ over $\equiv$ which embeds into the conflicting steps $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$.*

Proof. Sketch: Transformations $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ can be replayed on terms of $T_\Sigma(X)/_\equiv$ according to Σ-homomorphism m, since morphism $c : T_\Sigma(X) \to T_\Sigma(X)/_\equiv$ is unique and $m \circ c$ is equal on the algebra part of all occurrence morphisms in the given conflicting steps. Furthermore, composition and decomposition properties for pushouts in category $\frown Graph$ have to be used [4]. The conflicting transformations can be further reduced by cutting off unneeded context such that k_1 and k_2 are jointly surjective. That leads to $P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(k_2)}{\Longrightarrow} P_2$ where $Alg(P_i) = Alg(K) = T_\Sigma(X)/_\equiv$ and $P_i \subseteq H_i$, $K \subseteq G$. $P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(o_2)}{\Longrightarrow} P_2$ is a critical pair due to the minimality of graph and algebra parts.

An embedding theorem [3,12] answers the question, under which conditions a given transformation sequence $K_0 \overset{*}{\Longrightarrow} K_n$ can be replayed in a bigger context. In the double-pushout approach it is well-known that this is the case whenever the morphism $m : K_0 \to G_0$ satisfies the gluing condition wrt. the *derived rule* $K_0 \curvearrowright K_n$ summarizing the effect of the overall transformation sequence.

Definition 6 (derived production). *Given two spans $s = (G \overset{g}{\longleftarrow} D \overset{h_1}{\longrightarrow} H)$ and $t = (H \overset{h_2}{\longleftarrow} E \overset{i}{\longrightarrow} I)$, their composition $s;\ t = (G \overset{g \circ h_1'}{\longleftarrow} C \overset{i \circ h_2'}{\longrightarrow} I)$ is defined up to isomorphism by the pullback $D \overset{h_1'}{\longleftarrow} C \overset{h_2'}{\longrightarrow} E$ of $D \overset{h_1}{\longrightarrow} H \overset{h_2}{\longleftarrow} E$.*

For a direct transformation $g = G \overset{p(o)}{\Longrightarrow} H$ as in Def. 3, its derived production *$der(g)$ is defined by the bottom span of the DPO diagram $(G \overset{g}{\longleftarrow} D \overset{h}{\longrightarrow} H)$.*

Given a transformation sequence $h = (G_0 \overset{p_1(o_1)}{\Longrightarrow} \ldots \overset{p_n(o_n)}{\Longrightarrow} G_n)$ *with* $der(h_i)$ *the derived production of* $h_i = (G_{i-1} \overset{p_1(o_1)}{\Longrightarrow} G_i)$. *The derived production of the sequence* h *is defined as* $der(h) = der(h_1);\ \ldots;\ der(h_n)$.

It is obvious that the derived production is properly typed over ATG, if all original productions and graph G_0 are. Note that both morphisms of $der(g)$ are identities on the data algebras and that the derived production is unique up to isomorphism due to the pullback construction. The definition of derived productions can easily be generalized to transformation sequences of length greater than 2.

Proposition 3 (embedding of transformations). *Given a transformation sequence* $k = (K_0 \overset{p_1(k_1)}{\Longrightarrow} \cdots \overset{p_n(k_n)}{\Longrightarrow} K_n)$ *and a graph morphism* $m : K_0 \to G_0$, *then there is a transformation sequence* $g = (G_0 \overset{p_1(o_1)}{\Longrightarrow} \cdots \overset{p_n(o_n)}{\Longrightarrow} G_n)$ *with* $o_i = m \circ k_i$, *if and only if,* $der(k)$ *is applicable at* m, *i.e. there is a transformation* $G_0 \overset{der(k)(m)}{\Longrightarrow} G_n$.

The embedding theorem above has been shown in [16] for $n = 2$, but can be generalized to $n > 2$ straight forward. Moreover, it has been shown for colored graphs. The proof of proposition 3 would be a simple rephrasing for typed graphs.

Confluence. Embedding is relevant in the proof of the critical pair lemma below where it is shown that a graph transformation system GTS is locally confluent if all its critical pairs showing conflicting situations can be joined.

Definition 7 (confluence). *Two transformation sequences* $H_1 \overset{*}{\Longleftarrow} G \overset{*}{\Longrightarrow} H_2$ *are* confluent *if there are transformation sequences* $H_1 \overset{*}{\Longrightarrow} X$ *and* $H_2 \overset{*}{\Longrightarrow} X$.

Two transformations $H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ *with* $der(G \overset{p_i}{\Longrightarrow} H_i) = (G \overset{g_i}{\longleftarrow} D_i \overset{h_i}{\longrightarrow} H_i)$ *are* strongly confluent *if for* $(D_1 \overset{d_1}{\longleftarrow} D \overset{d_2}{\longrightarrow} D_2)$ *being the pullback of* $(D_1 \overset{g_1}{\longrightarrow} G \overset{g_2}{\longleftarrow} D_2)$ *and* $der(G \overset{p_1}{\Longrightarrow} H_i \overset{*}{\Longrightarrow} X) = (G \overset{c_i}{\longleftarrow} C_i \overset{x_i}{\longrightarrow} X)$ *there are morphisms* $e_i : D \to C_i$ *with* $g_i \circ d_i = e_i \circ c_i$ *for* $i = 1, 2$.

A graph transformation system $GTS = (\Sigma, ATG, X, R)$ *is* confluent w.r.t. a congruence $\equiv \subseteq T_\Sigma(X) \times T_\Sigma(X)$ *if for any* Σ*-algebra* A *satisfying this congruence,*[2] *all pairs of transformations* $H_1 \overset{*}{\Longleftarrow} G \overset{*}{\Longrightarrow} H_2$ *in* GTS *attributed over* A *are confluent.* GTS *is* locally confluent w.r.t. A *if the same holds for all pairs of the form* $H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$.

Strong confluence means that those graph objects preserved by transformations $G \Longrightarrow H_i$, are not deleted by transformations $H_i \overset{*}{\Longrightarrow} X$ for $i = 1, 2$. The congruence $\equiv$ plays the role of an equational specification for the data algebra A. However, it also covers cases where the congruence is not given in terms of equations, or where it is not equationally axiomatizable.

[2] An algebra A satisfies a congruence over terms with variables in X if for all assignments $\alpha : X \to A$, the free extension to $\bar{\alpha} : T_\Sigma(X)/_\equiv \to A$ is a homomorphism.

Confluence of a sample critical pair. The critical pair in Fig. 6 is not confluent: Assume that first the left hand rule application of directBeh is followed, setting attribute computed of the event to true. Thereby, any further application of rule directBeh to this event is not possible and as directBeh is the only rule adding term $e \to State(t2)$, there is no possibility of joining the two rules. Hence, we have found that our rule set is not confluent. This critical pair can be made confluent if we move attribute computed to the transitions.

Proposition 4 (critical pair lemma). *A graph transformation system* $GTS = (\Sigma, ATG, X, R)$ *is locally confluent w.r.t. a congruence* $\equiv \subseteq T_\Sigma(X) \times T_\Sigma(X)$ *if, for all pairs of rules* p_1, p_2 *in* R*, each critical pair* $P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(k_2)}{\Longrightarrow} P_2$ *over* $\equiv$ *is strongly confluent.*

Proof. Consider two direct transformations $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ with rules $p_i = L_i \curvearrowright R_i$ for $i = 1, 2$. There are the following cases:

1. $H_1 \overset{p_1(o_1)}{\Longleftarrow} G \overset{p_2(o_2)}{\Longrightarrow} H_2$ is parallel independent. Thus, there are transformations $H_1 \overset{p_1}{\Longrightarrow} X$ and $H_2 \overset{p_2}{\Longrightarrow} X$ due to Proposition 1.
2. $H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ is not parallel independent. According to Proposition 2 there is a critical pair $P_1 \overset{p_1(k_1)}{\Longleftarrow} K \overset{p_2(k_2)}{\Longrightarrow} P_2$ which embeds into $H_1 \overset{p_1}{\Longleftarrow} G \overset{p_2}{\Longrightarrow} H_2$ with morphisms $g : K \to G$ and $h_i : P_i \to H_i$ for $i = 1, 2$. Assuming that all critical pairs are confluent, there are two transformation sequences $K \Longrightarrow P_1 \Longrightarrow X$ and $K \Longrightarrow P_2 \Longrightarrow X$.
   Let $der(K \Longrightarrow P_i) = (K \overset{d_i}{\longleftarrow} D_i \overset{p_i}{\longrightarrow} P_i)$, $der(K \Longrightarrow P_i \Longrightarrow X) = (K \overset{c_i}{\longleftarrow} C_i \overset{x_i}{\longrightarrow} X)$, and $(D_1 \overset{dd_1}{\longleftarrow} D \overset{dd_2}{\longrightarrow} D_2)$ being the pullback of $(D_1 \overset{d_1}{\longrightarrow} K \overset{d_2}{\longleftarrow} D_2)$. Due to strong confluence and the construction of derived productions, there are morphisms $e_i : D \to C_i$ with $c_i \circ e_i = d_i \circ dd_i$ and $f_i : C_i \to D_i$. Furthermore, let *Boundary* be all nodes of K being in touch with edges of $G - K$. $der(G \Longrightarrow H_i)$ and $der(K \Longrightarrow P_i)$ contain[3] *Boundary*, since the gluing condition is satisfied for these transformations. Thus, there is a morphism $Boundary \to D$ due to pullback properties and $der(K \Longrightarrow P_i \Longrightarrow X)$ contain *Boundary*, i.e. it satisfies the gluing condition. By Proposition 3 there are transformations $G \overset{p_1(o_i')}{\Longrightarrow} H_i' \Longrightarrow Y$ with $o_i' = g \circ k_i$, thus $o_i' = o_i$ and hence H_i' is isomorphic to H_i.

As local confluence and termination imply confluence according to Newman's lemma [15], confluence of a terminating consistent graph transformation system can be shown by proving for all critical pairs the property of being confluent.

Restricting to vertex-preserving transformations and morphisms which are injective up to data vertices, the gluing condition is always satisfied for any transformation. Of course, this provides us with unrestricted embedding. Moreover in this case, confluence means always strong confluence. This restricted kind of transformations is the only one used in the running example. Thus, the critical pair lemma can easily be used.

[3] A production contains a graph G if there is a morphism $m : G \to K$.

Confluence of the example GTS. Computing critical pair candidates can be done by constructing all overlapping graphs of left hand sides of all combinations of rule pairs. In the following, we will check the example rule set for critical pairs.

Note that in this case the graph transformation system is vertex-preserving.

We first recall that so far the cardinality constraints specified in Fig. 2 have not been part of the formal treatment. However, such cardinality constraints can be seen as *negative constraints* as they require a certain structure not to be existent (i. e. that an object has more than one link to another object). The given graph transformation system is preserving these constraints because it only changes attribute values.

As a consequence, we do not have to show confluence for those critical pairs that do not have an overlapping graph fulfilling all negative constraints because those critical pairs will never be subgraphs of G. This simplifies tremendously the following discussion of critical pairs. However, it is important to note that in the case of positive constraints the set of critical pairs cannot be reduced to those where the overlapping graph fulfills the constraints. In order to show confluence of the complete rule set, we have to compute all critical pairs and show that each critical pair is confluent.

We first note that critical pairs only occur if two rules change the same attributes. Rule top changes only attributes of the state machine which are not changed or read by any other rule. Due to the constraints there is only one top state and therefore top cannot be overlapped with itself in the top state. Rules simple and comp change attributes exp and expDef, which are only changed by comp as well. However, there is no overlapping graph because a state cannot be a composite and simple state at the same time. Both rules extBehtop and extBehcomp change attributes extBeh and extBehDef. Due to the constraint that the top vertex is not a subvertex of any other state, there exists no overlapping graph. Rules directBeh and directBehe do not overlap because $s(x)$ cannot be unified with 0. Overlapping directBeh with itself at their transitions or target states is confluent (under the assumption that the attribute computed is shifted to the Transition class).

Tool support The complexity of computing critical pairs and proving their joinability arises the need for tool support. Currently, AGG [7] supports the computation of critical pairs for attributed graphs: All possible overlapping graphs are constructed and two rules are critical if they change the same attribute.

The attributed graph transformation implemented in AGG allows variables and constants in left-hand rules sides only. But this is not a restriction, since attribute conditions can be stated separately. However, unification on attribute values is not supported.

Furthermore, proving confluence by showing that all critical pairs are confluent also requires support. Here, an interactive approach that enables stepwise rule applications to a common successor graph could be followed, thereby avoiding the complexity of automated derivations.

Currently, AGG does not support type graphs and constraints. An additional possibility to specify graph constraints which can be used to check graphs and

to provide rules with post conditions such that consistent transformations are performed only, is under development. Having graph constraints available, the critical pair analysis can be made more efficient in the sense, that only those overlapping graphs are computed which satisfy the negative constraints. For most application we can expect that the set of critical pairs will become considerably smaller.

4 Conclusion

Confluence and termination of attributed graph transformation are important issues whenever attributed graph transformation is to be used in an automated way. In this paper, we have shown how confluence can be ensured for typed attributed graph transformation systems. Motivated by the translation of UML statecharts to CSP, typed attributed graph transformation systems have been introduced in order to represent transformations of diagrams based on a meta model-like representation. Then, the theory of critical pairs has been extended to typed attributed graph transformation systems. We have shown that an attributed graph transformation system is locally confluent if all its critical pairs are confluent. The concept of critical pairs has been applied to a concrete set of translation rules, thereby discovering an error leading to a non-confluent critical pair. Moreover, the issue of tool support based on AGG has been sketched.

Having now the critical pairs analysis technique for attributed graph transformation at hand, also the other applications mentioned in the introduction already, can be checked for functional behavior in future. In [9], functional requirements are described by UML use cases refined by activity and collaboration diagrams. Using graph transformation as semantic domain here, graph rules formalism the functional requirements to a system. The critical pair analysis can be used to find out conflicts and dependencies between different use cases.

Another application of graph transformation where functional behavior is of importance, is parsing of visual diagrams. Allowing free editing of visual diagrams, they have to be parsed to be sure that they belong to some visual language. Analyzing the critical pairs of parsing rules, and applying conflict-free rules first, increases the efficiency of graph parsing [1].

References

1. P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rom.
2. A. Corradini, U. Montanari, and F. Rossi. Graph processes. In *Fundamenta Informaticae*, volume 26 (3,4), pages 241–266, 1996.
3. H. Ehrig. Embedding theorems in the algebraic theory of graph grammars. In *LNCS 56*, pages 245–255. Springer, 1977.

4. H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Graph Grammars and their Application to Computer Science and Biology*. Springer LNCS 73, 1979.
5. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
6. G. Engels, R. Heckel, and J. M. Küster. Rule-based specification of behavioral consistency based on the UML meta-model. In M. Gogolla and C. Kobryn, editors, *Proc. 4th Intl. Conference on The Unified Modeling Language (UML '02), Toronto, Canada, October, 2001*, volume 2185 of *LNCS*, pages 272–287. Springer, 2001.
7. C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999. See also `http://tfs.cs.tu/berlin.de/agg`.
8. Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*, 1997.
9. J. H. Hausmann, R. Heckel, and G. Taentzer. Detection of Conflicting Functional Requirements in a Use Case-Driven Approach. In *Proc. 24th Intl. Conference on Software Engineering*, Orlando, FL, 2002. ACM/IEEE Computer Society.
10. R. Heckel, J.M. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In H.-J. Kreowski, editor, *Proc. ETAPS'02 Workshop on Application of Graph Transformation (AGT'02), Grenoble, France*, April 2002.
11. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
12. H.-J. Kreowski. *Manipulation von Graphmanipulationen*. PhD thesis, FB13, 1978.
13. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *Proc. UML 2001, Toronto, Kanada*, volume 2185 of *LNCS*. Springer-Verlag, 2001.
14. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In *Term Graph Rewriting: Theory and Practice*, pages 185–199. John Wiley & Sons Ltd, 1993.
15. M. H. A. Newman. On theories with a combinatorial definition of 'equivalence'. In *Annals of Mathematics*, 43 (2), pages 223–243, 1942.
16. J. Padberg and G. Taentzer. Embedding of derivations in high-level replacement systems. Technical Report 93/9, Technical University of Berlin, Computer Science Department, 1993.
17. D. Plump. Hypergraph Rewriting: Critical Pairs and Undecidability of Confluence. In M.R Sleep, M.J. Plasmeijer, and M. C.J.D. van Eekelen, editors, *Term Graph Rewriting*, pages 201–214. Wiley, 1993.
18. D. Plump. Term graph rewriting. In G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*, pages 3 – 62. World Scientific, 1999.
19. D. Varro, G. Varro, and A. Pataricza. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming*, 44(2), 2002.

Abstraction and Control for Shapely Nested Graph Transformation

Berthold Hoffmann

Technologiezentrum Informatik, Universität Bremen
Postfach 330 440, D-28334 Bremen
hof@tzi.de

Abstract. Shapely nested graph transformation is the computational model for DiaPlan, a language for programming with graphs that represent diagrams. It supports nested structuring of graphs, structural graph types (*shapes*), and graph variables. In this paper, we extend the model by two concepts that are essential for programming: *abstraction* allows compound transformations to be named and parameterized, and *control* allows the order of rule application to be specified. These concepts combine neatly with the underlying computational model, and preserve its rule-based and *graph*-ical nature.

1 Introduction

Graph transformation defines a computational model $\langle \mathcal{G}, \Rightarrow_T \rangle$ for a class $\mathcal{G}$ of graphs, by a transformation relation $\Rightarrow_T \subset \mathcal{G} \times \mathcal{G}$ that is induced by some finite set T of graph transformation rules. Various computational models of that kind are studied in [20]. If graph transformation shall be used for specification and programming, the scale of practical systems requires that the computational model $\langle \mathcal{G}, \Rightarrow_T \rangle$ is extended by concepts for structuring and typing: its data $\mathcal{G}$ should be *structured* in a nested way; its programs T should be *encapsulated* in modules; and, *abstraction mechanisms* should allow to structure the computations $\Rightarrow_T$. Furthermore, *control mechanisms* should allow to eliminate unwanted nondeterminism in the rule-based definition of $\Rightarrow_T$, and a *type discipline* should detect inconsistencies in its data and programs.

Shapely nested graph transformation [3,12] supports nested structuring of graphs, and comes with a structural type discipline (*shapes*). Analoguously to term rewrite rules, its rules may contain *variables* so that transformation steps may move, delete or duplicate subgraphs of arbitrary size. Shapely nested graph transformation is the computational model of DiaPlan, a language for programming with graphs that is currently being designed by Frank Drewes (Umeå), Mark Minas (Erlangen), and the author [11,13]. DiaPlan shall complement DiaGen [17], a tool for generating editors that handle the *syntax* of diagram languages, by a language and tool for programming the *semantics* of diagram languages.

In this paper, we extend shapely nested graph transformation by concepts that are essential for the design of DiaPlan: We propose an *abstraction* con-

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 177–191, 2002.
© Springer-Verlag Berlin Heidelberg 2002

cept that allows to name and parameterize compound transformations in predicate definitions, and provide *control* of the evaluation order by an overall strategy (depth-first innermost evaluation), as well as by user-definable completion clauses and applicability conditions. The extension shall be seamless so as to preserve the rule-based and graphical nature of graph transformation.

The rest of the paper is structured as follows. Sections 2 to 4 recall ingredients of shapely nested graph transformation: (nested) graphs, shapes (types), and transformation. This is done as far as it is essential for defining the major programming concepts proposed in the paper: abstraction (in Section 5), and control (in Section 6). In Section 7, we compare our concepts to those in related languages, and outline some further research.

Acknowlegdements. I wish to thank Frank Drewes and Mark Minas for clarifying discussions about the design of DiaPlan, and the reviewers for their advice to remove technical details from sections 2–4.

2 Graphs

Our notion of graphs is tailored to programming: edges may connect an arbitrary number of nodes to model relations of any arity; nodes and edges may contain graphs in a nested fashion so that recursively structured values can be represented. We also distinguish a sequence of interface nodes at which graphs may be glued together. This extends the *nested graphs* of [3,4,12] where only edges can contain graphs.

Let $\mathcal{C}$ be a typed alphabet of *constant names* with an *arity function* $\text{arity}\colon \mathcal{C} \to \mathcal{C}^*$.[1] The set $\mathcal{G}$ of *graphs* consists of sixtuples $G = \langle O, E, \text{lab}, \text{ass}, N, p\rangle$ over a finte set O of *top-level objects* with a subset $E \subset O$ of *edges* and a complementary set $V = O \setminus E$ of *nodes*; the function $\text{lab}\colon O \to \mathcal{C}$ *labels* top-level objects by constant names; the function $\text{ass}\colon E \to V^*$ *associates* top-level edges to sequences of top-level nodes; N is a family of (possibly empty) graphs $G.o \in \mathcal{G}$ contained in the objects $o \in O$ (their *direct components*); the node sequence $p \in V^*$ designates the *points* of G.[2]

We require that the point sequence p, and the association sequences $\text{ass}(e)$ (for $e \in E$) do not contain repetitions, and that edges respect arity, i.e. satisfy:

$$\text{lab}^*(\text{ass}(e)) = \text{arity}(\text{lab}(e)) \text{ for all edges } e \in E$$

This corresponds to a well-known normal form of graphs that does not restrict the expressiveness of the concepts defined below. (See [10] for details.)

We call an object $o \in O$ *atomic* if $G.o$ is empty, or *compound* otherwise. An edge e labelled by c is called a *c-edge*. The *handle graph* $\langle c\rangle$ of a constant name c consists of an atomic c-edge that is associated to atomic points. G is called *plain*

[1] S^* is the set of *finite sequences* over some set S, including the *empty sequence* ε.

[2] Precisely, $\mathcal{G}$ has to be defined by induction over the nesting depth of objects, see [4].

if it contains no compound objects, and we define $\text{arity}(G) = \text{lab}^*(p)$. Thus $\langle c \rangle$ is plain and has the arity of the label c.

The sequences $\Omega_G = \{\varepsilon\} \cup \{o\,\omega \mid o \in O, \omega \in \Omega_{G.o}\}$ define *positions* of objects in a nested graph. They are used to select the *nested component* $G.\,\omega$ at some position $\omega \in \Omega_G$. The plain graph $G(\omega)$ is the nested component $G.\,\omega$ without its direct components.

Two graphs G and H are *isomorphic*, written $G \cong H$, if there is a bijective function $m\colon O_G \to O_H$ between their sets of top objects so that labels, associations, and points are preserved, and all corresponding components $G.\,o$ and $H.\,m(o)$ $(o \in O_G)$ are isomorphic, recursively.

Example 1 (Graphs). The figures of this paper contain graphs of some kind. We use the following conventions when drawing graphs: nodes are depicted as circles or ovals, and edges as boxes that are drawn around their label; the contents of a node or edge (if not empty) is also drawn inside. The association of an edge e to a sequence $v_1 \ldots v_k$ of nodes is depicted by a bundle of lines from e to the v_i, which are called *tentacles*. Atomic binary edges are drawn like directed edges, as arrows from their first to their second associated node; their label is written aside. The "invisible label" $\sqcup$ is omitted in figures (to model unlabelled nodes and edges).

Note the difference to notions of *hierarchical graphs* that are used for system modeling [1,6], where the objects of an underlying graph (which is plain, according to our definition) are grouped in packages that may form a hierarchy, and may have interfaces (designating some objects of a package as "public"). In hierarchical graphs, several packages may *share* an object (or a subpackage), and the associations of an edge may cross package borders. Our nesting concept forbids sharing and border-crossing edges, and is thus *compositional*: every component of a nested graph can be replaced, independently of the other ones. This is essential for programming.

3 Shapes

Usually, not every graph is a meaningful value, but only those that have a particular *shape*: chain graphs, for instance, must have a linear structure. The term *shape analysis* has come into use for inferring properties of the pointer-based data structures in imperative programs [21]. On the more abstract level of graphs, we *define* the shape of graphs (by edge replacement) so that it can be checked statically.

Edge Replacement. Let G be graph with an edge $e \in G(\omega)$. The *replacement* of e in $G.\,\omega$ at the position $\omega \in \Omega_G$ by some graph U with atomic points and $\text{arity}(U) = \text{arity}(lab((e))$ is defined as follows: Unite $G.\,\omega$ and U disjointly, redirect all tentacles at the points of U to the corresponding associated nodes of e, remove e and U's points, and insert the resulting graph for $G.\,\omega$.

This way of edge replacement is a straight-forward extension of hyperedge replacement [10] to nested graphs.

Shape Rules. Let S be a typed alphabet of *shape names* disjoint with $\mathcal{C}$. $(\mathcal{C} \cup S)$-labelled graphs are called *syntax graphs* if their S-edges are atomic.

Let Σ be a finite set of *shape rules* of the form $s ::= R$, consisting of a shape name s and a syntax graph R of the same arity. Σ *directly derives* a syntax graph G to a syntax graph H, written $G \Rightarrow_\Sigma H$, by replacing an s-edge in G by the graph R of some shape rule $s ::= R \in \Sigma$. The reflexive and transitive closure of $\Rightarrow_\Sigma$ is denoted by $\Rightarrow^*_\Sigma$.

For the rest of this paper, we fix a finite set Σ of shape rules over the shape names S, and use it to specify the shape of graphs.

Example 2 (Shape Rules for Chain Graphs). The rules in Figure 1 define the shape of *chain graphs*. (We use "|" to seperate alternative rules for the shape name γ.) Two atomic points designate the begin and end of a chain, and every node in between contains an item graph. In the examples of this paper, the item graphs (with shape named ι) are series-parallel graphs as defined in [2, Section 2.2]. Figures 4 and 7 below contain chain and item graphs that are shaped according to these rules.

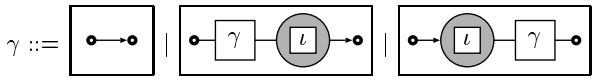


Fig. 1. Shape rules for chain graphs

The rules for γ are *ambiguous*: If we removed the rightmost shape rule, γ would still generate the same set of chain graphs. However, with this rule, we may define transformation rules in a more general way. (See Example 4 below.)

Σ-graphs. For rules, we need special edges that denote variable parts in a graph. For that purpose, we consider a typed alphabet X of *variable names* disjoint with $\mathcal{C}$ and S. We assume that every variable name is associated with a *shape name* $\text{shape}(x) \in S$.

Let G be a $(\mathcal{C} \cup X)$-labelled graph where all *variables* (the X-edges) are atomic. G is called a Σ*-graph* if $\langle s \rangle \Rightarrow^*_\Sigma \text{shape}(G)$ for some shape name $s \in S$, where $\text{shape}(G)$ is the syntax graph obtained by relabelling every variable name in G by its shape name. Then we write $\Sigma \vdash G : s$. (Σ-graphs may have several shapes, because with a "chain" rule like $s ::= \langle s' \rangle$, $\Sigma \vdash G : s'$ implies $\Sigma \vdash G : s$.)

$\mathcal{G}_\Sigma(X)$ denotes the set of Σ*-graphs*, and $\mathcal{G}_\Sigma$ denotes the set of *constant* Σ*-graphs*, which contain no variables.

It is decidable whether a graph satisfies some shape rules Σ or not (see [3]):

Theorem 1. *The question "$\Sigma \vdash G : s$?" is decidable.*

Data types of functional and logical languages are tree-like. They can be defined by shape rules with unary shape names only. But also data structures

with sophisticated sharing, like cyclic lists, or leaf-connected trees, can be defined in a way that is not possible in imperative languages. (See [8] for a similar specification of such types in *Structured Gamma*.)

Edge replacement can only define graph shapes of bounded node degree. However, this restriction can be overcome without sacrificing decidability if we allow rules similar to the embedding rules used in the DiaGen system [17]. Then also shapes like that of all control flow graphs can be defined.

4 Transformation

Graphs are transformed by *matching* a pattern P, and rewriting this match with a replacement R. In the case of nested graph transformation [12], rules consist of Σ-graphs P and R, and transformation is defined like term rewriting [14]: P is embedded into some context C, after substituting its variables by appropriate graphs; transformation yields a graph where R is embedded into the same context C, after instantiating its variables with the same substitution. (The variable concept is inspired by [19].) Here we just consider "shapely" nested graph transformation. The "unshaped" case just forgets about typing. (See [12] for details.)

Context Embedding. A Σ-graph C containing a single variable e in some plain component $C(\omega)$ ($\omega \in \Omega_C$) is an *s-context* if e has the shape $s \in S$ and is associated to atomic nodes. The *embedding* of a Σ-graph U with $\Sigma \vdash U : s$ in C is denoted as $C[U]$ and defined as follows: Unite $C.\,\omega$ and U disjointly, redirect all tentacles at the associated nodes $\mathrm{ass}(e)$ to the corresponding points of U, remove e with its associated nodes, and assign the result to $P.\,\omega$. (Context embedding preserves the points of the inserted Σ-graph, and removes the associations of the replaced variable, while edge replacement does it the other way round. Otherwise, both operations are equal.)

Example 3 (Substitution and Context). Figure 2 shows a γ-context (if we assume that the variable name H has shape γ), and a substitution for two variables X and C that are nullary and binary, respectively.

Variable Instantiation. A function $\sigma\colon X \to \mathcal{G}_\Sigma(X)$ is a *substitution* if $\Sigma \vdash \sigma(x) : \mathrm{shape}(x)$ for all $x \in X$.

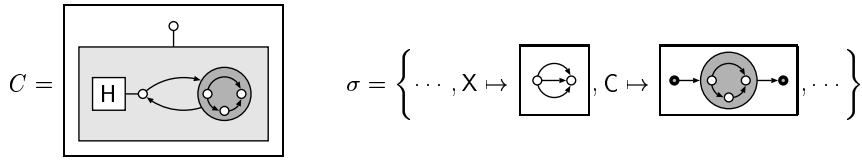


Fig. 2. A context and two substitution pairs

The *instantiation* of a Σ-graph P according to σ is obtained by the simultaneous replacement of all x-variables in P by the Σ-graph $\sigma(x)$. The resulting *instance* is denoted by $P\sigma$. (The order of replacement is irrelevant as edge replacement is commutative and associative.)

Transformation Rules and Steps. The definition of rules and their application is similar as in term rewriting [14]. Therefore, we also require the same properties as for term rewrite rules: their patterns must not be variables, as such rules apply to every graph so that transformation diverges, and their replacements must not contain variables that do not occur in their pattern, since then arbitrary subgraphs may be created "out of thin air".

A *(transformation) rule* $t = P/R$ consists of two Σ-graphs P and R so that $\Sigma \vdash P : s, R : s$ for some $s \in S$, where the *pattern* P is not a variable handle, and only variable names from P occur in the *replacement* R. Then t *transforms* a graph G into another graph H, written $G \Rightarrow_t H$, if the instances of P and R according to a substitution σ can be embedded into some s-context C so that $G \cong C[P\sigma]$ and $H \cong C[R\sigma]$.

Graph transformation preserves shapes. (See [12] for the straightforward proof.)

Theorem 2. *If $\Sigma \vdash G : s$ for some $s \in S$ and $G \Rightarrow_t H$, then $\Sigma \vdash H : s$.*

Hence, shapes set up a *type discipline* that can be *statically checked*: Theorem 1 allows to confirm whether a transformation rule t consists of Σ-graphs or not. If this is true, and a graph G has been checked to be shaped, Theorem 2 guarantees that every transformation step $G \Rightarrow_t H$ yields a shaped graph H. After the step ("at runtime") type checking is not necessary.

Example 4 (Chain Graph Transformation). Figure 3 shows a rule e that *enters* an item graph at the end of a chain graph, and a rule r that *removes* the first item graph from a chain graph. The variable names C and X have shape(C) $= \gamma$ and shape(X) $= \iota$, respectively.

Note that the second recursive rule in Figure 1 is needed to derive the shape of the chain graph in r's pattern. Without that ambiguous shape rule, r had to be defined by recursive traversal of the chain graph.

Figure 4 shows a transformation via r, using the context and substitution in Figure 2.

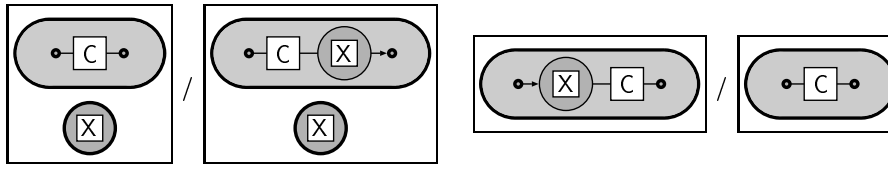


Fig. 3. Chain rules for entering and removing item graphs

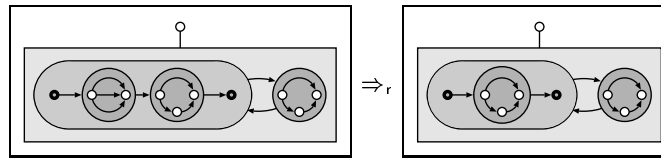


Fig. 4. A removing transformation

Variables make graph transformation quite expressive: a single step may affect subgraphs of arbitrary size: rule e *duplicates* a member node with its entire contents, since the variable name X occurs twice in its replacement graph; rule r *deletes* a member node, again with its contents, since X does not occur in is replacement graph. Let e^{-1} denote the inverse rule of e where pattern and replacement are interchanged. Then e^{-1} requires to *compare* arbitrarily large subgraphs: it applies only to a host graph like G, where both X-variables in its pattern match isomorphic subgraphs. This allows to express rather complex applicability conditions. Implementations of shapely nested graph transformation may forbid such rules by requiring that their pattern is *linear*, i.e. that every variable name occurs at most once.

Constructing Transformation Steps. In [3] we have discussed how transformation steps can be constructed. Here we just note that for some Σ-graph G, every transformation $G \Rightarrow H$ and its result H is uniquely determined by a *redex* $\rho = \langle t, m, \sigma \rangle$, consisting of the rule $t = P/R$ used, an occurrence morphism m indicating the place where the skeleton of its pattern P (i.e. without its variables) occurs in G, and the matching substitution σ.

5 Abstraction

In programming languages, *abstraction* means to name and parameterize compound computation tasks so that they can afterwards be called (with different arguments) just like elementary computations. For graph transformation, we thus need a concept for naming and parameterizing sequences of graph transformation steps. We extend the set $\mathcal{C}$ of constant names (denoting data) by names that denote abstractions. Abstractions are called *predicates* (not functions) because their evaluation may fail, and may be nondeterministic, i.e. yield more than one result. Every predicate is defined by a set of rules that may contain predicates in their replacement graphs by which other abstractions are called.

Predicate Definition. We consider a typed alphabet Q of *predicate names* that is disjoint to $\mathcal{C}$, $\mathcal{S}$, and X. Let G be a graph labelled by $Q \cup \mathcal{C} \cup X$. $\overline{G}$ denotes the *data* of G, i.e. G without all predicate edges (the edges labelled by Q). G is an *expression* if $\overline{G}$ is a Σ-graph, and if its predicate edges are atomic. An expression G has the *shape* of $\overline{G}$. $\mathcal{E}_\Sigma(X)$ denotes the set of expressions, and $\mathcal{E}_\Sigma$ the set of

constant expressions (without variables). Thus $\mathcal{E}_\Sigma(X) \supset \mathcal{G}_\Sigma(X)$ and $\mathcal{E}_\Sigma \supset \mathcal{G}_\Sigma$. We extend substitutions so that they map variable names to expressions of the same shape, and extend contexts and instances to be expressions.

An expression G is a *p-pattern* if it contains exactly one predicate edge e, which is labeled by the predicate name p, and occurs on its top-level $G(\varepsilon)$. The *definition* of a predicate name $p \in Q$ consists of a finite, nonempty set T_p of rules $t = P/R$ such that P is a p-pattern, and R is an expression. A *program* consists of a set $\mathcal{T} = \bigcup_{p \in Q} T_p$ of predicate definitions. (Examples of predicate definitions are given in the next section, after introducing concepts for control.)

For constant expressions G and H, we write $G \Rightarrow_\mathcal{T} H$ if there is an *evaluation step* via a transformation $t \in \mathcal{T}$, and $G \Rightarrow^*_\mathcal{T} H$ if there is an *evaluation sequence* of $n \geqslant 0$ consecutive steps. A constant expressions G is a *normal form* if no rule in $\mathcal{T}$ applies to G, and *terminal* if G is a Σ-graph. A program $\mathcal{T}$ is called *terminating* if every evaluation sequence leads to a normal form after finitely many steps. $\mathcal{T}$ is *uniquely normalizing* if every graph G evaluates to at most one normal form.

The nodes associated to a predicate edge designate its *parameters*. If such a node is compound, its contents is a *graph parameter* (as in Example 5 below); if the contents contains predicate edges, it is a *predicate parameter* (as in Example 6 below). Predicates may also match and modify the "local context" $G.\omega$ where a predicate edge e occurs if a part of one of their patterns is not contained in their parameter nodes. (However, such rules do not occur in this paper.)

Predicate Evaluation. The evaluation of a program over a constant input expression can be imagined as constructing an *evaluation tree* Δ that is defined as follows: Its top level $\Delta(\varepsilon)$ is a tree of *evaluation states* that contain constant expressions, and are connected by atomic binary edges labelled with redices ρ. Δ may be infinite unless the evaluation relation $\Rightarrow_\mathcal{T}$ is terminating. Its root v_0 represents the *initial state* and contains the constant input expression. If $\Delta(\varepsilon)$ contains an atomic binary ρ-edge from a state v to a state v', then $\Delta.v \Rightarrow_t \Delta.v'$ via some redex ρ of t. A state $v \in \Delta(\varepsilon)$ is *complete* if there is a ρ-edge from v to some state v' for every redex ρ in the expression $\Delta.v$. Δ is *complete* if all its states are complete. Then, a leaf v in Δ contains an expression $\Delta.v$ that is in normal form; if $\Delta.v$ is a Σ-graph, it is a *result* of Δ, otherwise v is called a *blind alley* of the evaluation. (Practical implementations will *not* construct Δ, but just update the input expression G; this simplistic assumption shall only make discussion easier.)

As programs are nondeterministic in general, we define their semantics by an *evaluation function* $eval_\mathcal{T} \colon \mathcal{E}_\Sigma \to \mathcal{G}^*_\Sigma$ that enumerates a sequence of results one after the other, in a nondeterministic way. This function initializes the evaluation tree Δ by the initial state v_0 that contains the input expression G. Then edges and successor states for evaluation steps $\Delta.v \Rightarrow_t \Delta.v'$ are added until every state is complete. Whenever $\Delta.v'$ is a Σ-graph, it is returned as a result.

Depth-First Innermost Evaluation. The evaluation function $eval_\mathcal{T}$ is nondeterministic in several repects:

1. Which is the *actual state* $\hat{v} \in \Delta(\varepsilon)$ where the evaluation shall be continued?
2. Which is the *actual call*, i.e. the predicate edge $\hat{e}$ in $\Omega_{\Delta.\hat{v}}$ where the next occurrence shall be sought?
3. Which is the *actual rule* $\hat{t} \in \mathcal{T}$ that shall be applied at this edge?
4. Which *actual redex* $\hat{\rho} = \langle \hat{t}, \hat{m}, \widehat{\sigma} \rangle$ of $\hat{t}$ shall be used to transform $\Delta.v$?

Such a degree of nondeterminism is not only inefficient, but also confusing for programmers. Therefore we propose a general *evaluation strategy* that reduces nondeterminism:

1. States in $\Delta(\varepsilon)$ are totally ordered by age. The actual state $\widehat{v}$ is the most recently inserted state that is incomplete. This strategy is known as *depth-first search* in logic programming. (*Breadth-first search*, the opposite strategy, has the advantage to determine every normal form of the input expression, but only after exploring *all* shorter evaluations, which is very inefficient. So we accept the possibility that depth-first search may diverge due to a non-terminating rule although the input expression has a normal form.)
2. We order the predicate edges in a state v by age in the first place, and by nesting depth in the second place. This order is partial as a transformation step may introduce several predicate edges on the same nesting level. The actual call $\hat{e}$ is an innermost of the newest predicate edges in $\Delta.\widehat{v}$. This strategy corresponds to *innermost evaluation* (or *eager evaluation*) in functional programming. (Again, eager evaluation does not always find all normal forms, unlike the complementary strategy of *lazy evaluation*. We prefer it because it chooses an evaluation order that is more intuitive for programmers.)
3. Rules are ordered as they appear in the program. The actual rule $\hat{t}$ is the first rule for which a redex $\rho = \langle \hat{t}, m, \sigma \rangle$ has an occurrence m containing the actual call $\hat{e}$ so that no ρ-edge issues from $\hat{v}$.
4. The actual rule $\hat{t}$ may have many redices containing $\hat{e}$. These redexes may have different occurrences m that *overlap* with each other, and/or matching substitutions σ that *compete* with each other.
   Neither occurrences nor substitutions can be ordered in a canonical way. In [3] we consider conditions ensuring that the matching substitution $\widehat{\sigma}$ is *uniquely determined* by $\hat{t}$ and m. Our running example satisfies these conditions. (The rules `e` and `r` have no variable on top level, and at most one in each of their nested components. See [3, Theorem 1] for details.)

In the following, we assume that substitutions are uniquely determined as discussed in 4. The general evaluation strategy then leaves two sources of nondeterminism in the refined evaluation function $eval_{\mathcal{T}}$:

- Several predicate edges may be chosen as the actual call $\hat{e}$.
- The actual rule $\hat{t}$ may have several *overlapping* occurrences that contain the actual call.

The second source can only be avoided by careful design of shape and transformation rules. The applicability conditions proposed below allow to control the first source of nondeterminism.

6 Control

In this section we propose concepts by which programmers may control evaluation beyond the general strategy. Completion clauses allow to handle blind alleys in the evaluation, and applicability conditions order the predicates inserted by an evaluation step.

Completion Clauses. A program $\mathcal{T}$ is *sufficiently complete* [9] if every normal form of $\Rightarrow_{\mathcal{T}}$ is terminal, i.e. does not contain a Q-edge. Only if all predicate definitions are sufficiently complete, every graph has a complete evaluation. (Sufficient completeness does not suffice alone, however: evaluation must *terminate* as well.)

Sufficient completeness is a desirable property of predicate definitions; it is not so easy to achieve, however. For instance, we could define a predicate remove based on the single rule r of Example 4, by associating a remove-edge to the compound chain node in its pattern. (See the two expressions in Figure 5.) This predicate would not be complete, as its rule applies only if its parameter is a non-empty chain graph; otherwise, evaluation gets stuck in a blind alley. In programming languages, there are different ways to handle blind alleys:

- *Logical languages* consider them as *failure*. Then *backtracking* returns to the evaluation step where p was called, and tries another evaluation step starting from there.
- *Functional languages* consider such programs to be *erroneous*. Then evaluation *throws an exception* that can be caught (by *exception handlers*) in the context where the predicate has been called.

As evaluation can be nondeterministic, we cannot restrict ourselves to the functional interpretation alone. For conceptual completeness, we even allow a third way of completion: *success* is the complement of *failure*; it allows to continue evaluation. (Figure 6 shows a predicate using this kind of completion.)

Technically, success, failure and exceptions are signaled by edges labelled with built-in predicate names $+$, $-$, and $\bot_1 \ldots \bot_k$ $(k > 0)$ respectively. Success and failure edges are always nullary, but exceptions may have parameters (to be used by the exception handlers). Completion clauses are patterns of these predefined edges. They are added at the end of every predicate definition, following the symbol "$/\!/$".

A predicate definition may furthermore contain *exception handlers* (to catch exceptions thrown by the predicates called in its rules). They are specified by *exceptional rules* for $\bot_i$-patterns $(1 \leqslant i \leqslant k)$. Exception rules start with "?", and occur just before the completion clause.

If there is no (more) actual rule $\hat{t}$ with an occurrence m containing the actual call $\hat{e}$, the completion clause in the corresponding predicate definition is executed as follows:

- For *success* $(+)$, a fresh evaluation state v' is added with a $+$-edge from $\widehat{v}$ to v' so that $\Delta.\, v'$ contains $\Delta.\, v$ without $\hat{e}$, and evaluation continues.

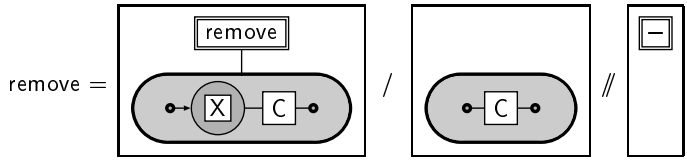


Fig. 5. The predicate remove

- For *failure* $(-)$, backtracking determines the ancestor state $\bar{v}$ of $\widehat{v}$ where $\hat{e}$ has been introduced. Evaluation continues with the next redex for the actual call of $\bar{v}$.
- For *exceptions* $(\bot_i)$, interrupt handling determines the closest ancestor state $\bar{v}$ of $\widehat{v}$ with an exceptional clause for $\bot_i$. This clause is evaluated, and recorded in Δ by an $\bot_i$-edge from $\bar{v}$ to a new state v' where evaluation continues.

Example 5 (A Predicate with Completion Clause). In Figure 5 we define a remove predicate based on the single rule r of Example 4, with a completion clause that specifies that the predicate fails if its ordinary rule cannot be applied.

A functional specification of remove could raise an exception $\bot_{\mathsf{ec}}$ (signalling an *empty chain*) that could then be handled by predicates calling remove.

Applicability Conditions. So far, the predicates inserted by an evaluation step may be evaluated in an arbitrary innermost order. However, it is often reasonable to consider some predicates as *applicability conditions* that have to be evaluated first, in order to make sure that this rule shall be applied, and evaluate the remaining predicates only then.

We thus distinguish a subset A of the predicate edges in R as *applicability predicates*, and give their evaluation priority over the rest. Applicability predicates are drawn as predicate edges with a dashed outer border. This simple concept suffices to make the local evaluation order in a replacement graph deterministic. (Rules can be split up so that A and $\Omega_R \setminus A$ contain at most one predicate edge each.)

Predicate Variables. As a last extension of rules, we consider "higher-order" variables that may be bound to computations instead of data. These variables have to be distinguished only because their substitutions are different.

Let $Y \subset X$ denote a subset of *predicate variable names*, which we draw as boxes with double borders. In a substitution σ, a predicate variable $y \in Y$ may be mapped onto an expression $\sigma(y)$ that does not contain constant names, and satisfies $\mathrm{arity}(\sigma(y)) = \mathrm{arity}(y)$. Predicate variables may be used to program combinators in a functional style.

Example 6 (A Control Combinator). Figure 6 shows a control combinator "!" that normalizes a chain graph denoted by C according to some unary predicate

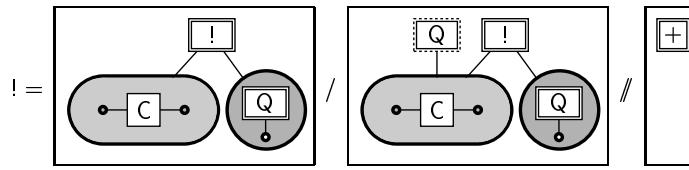


Fig. 6. The control combinator !

denoted by the predicate variable Q. It evaluates Q as an applicability predicate, and calls itself recursively as long as this succeeds.

For the termination of !, it is crucial that Q is an applicability predicate: otherwise, the evaluation of ! could loop in its recursive rule without ever evaluating Q.

Figure 7 shows an evaluation of ! with remove as a predicate parameter. The evaluation is deterministic: it empties the chain graph given as the first parameter. (The combinator ! is not deterministic in general because it might be applied to a nondeterministic predicate Q.)

Other common control structures can be defined by similar combinators. Note, however, that combinators cannot be defined as easily as in functional languages, because shapes like γ are not *polymorphic*, and graph variables have a fixed arity. So the !-combinator only applies to chain graphs and to unary chain predicates.

7 Conclusions

We have extended shapely nested graph transformation, a powerful model for computing with graphs that are recursively structured and shaped, by concepts for abstraction and control that shall become part of the language DiaPlan: predicates name and parameterize compound transformations, a global evaluation strategy (depth-first innermost) restricts nondeterminism, completion clauses cut off blind alleys of the evaluation, and applicability conditions control the order of predicate evaluation. The concepts are inspired by the way how term rewriting [14] is extended to functional programming languages.

Related Work. We concentrate our discussion of related concepts to Progres [22], the (so far) most successful and comprehensive programming language based on graph transformation. Progres *productions* are simlar to our transformation rules. They specify basic operations that are named, and may be parameterized by nodes, edges, and attribute values (also by types). Progres *procedures* call these productions (and other procedures), using a rich language of deterministic and nondeterministic control structures that is textual. Procedures are named and may be parameterized as well. So there is a clear separation

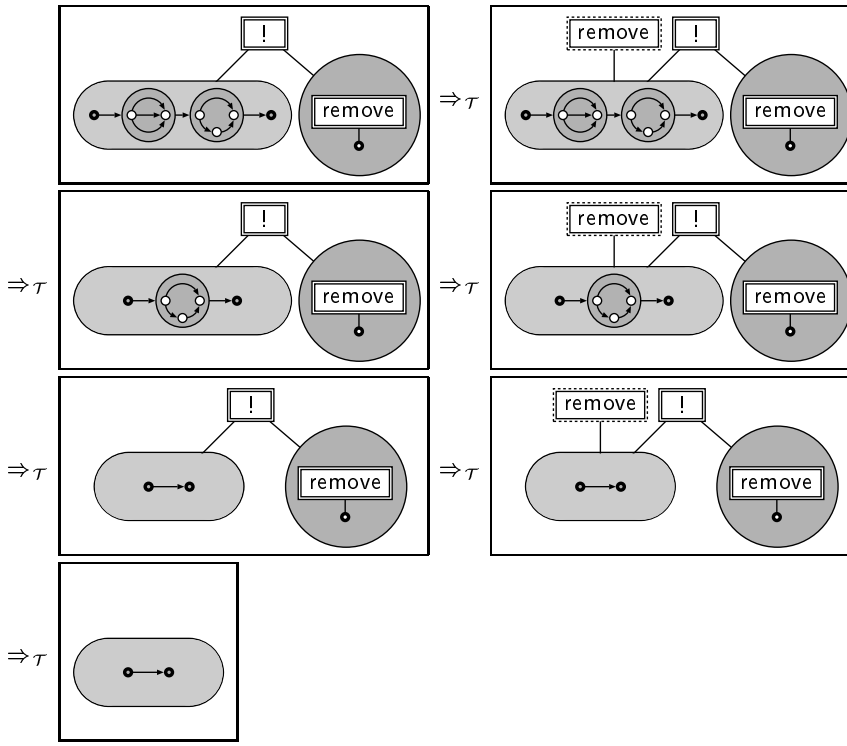


Fig. 7. An evaluation of ! with remove

between the graphical, rule-based specification of basic operations, and the textual, procedural programming of procedures. In contrast to that, the predicates proposed in this paper are just defined by slightly extended rules. This is more regular, and we find it more intuitive.

Another weakness of PROGRES lies in its underlying computational model, which does not support graph structuring. So productions and procedures always operate on one large plain graph. This is not satisfactory as program structuring has to be accompanied with data structuring in order to be effective. Our predicates profit from the nesting concept: their parameters may be graphs, and even by predicate expressions (contained in their associated nodes), not just "pointers" in a global graph.

Two other pieces of related work shall briefly be mentioned. AGG [7] is a prototyping rather than a programming system; it does not provide abstraction, and only rudimentary control structures. GRACE [15,16], a generic framework for structuring given graph transformation models, also separates rules from abstractions (called *transformation units*).

We are not aware of any other graph- and rule-based programming or specification language that supports recursive structuring and typing of graphs, and integrates abstraction and control seamlessly in a rule-based fashion.

Future Work. Some concepts of DiaPlan are still open, e.g. encapsulation and concurrency, and the concepts mentioned in this paper need further thought as well.

We want to specify whether predicates may fail or not, whether they are uniquely normalizing, or nondeterministic, and whether they have an *effect* on the local context where they are applied, or not (or only if they succeed). We want to distinguish parameter *modes* (*in*, *out*, and *inout*) in order to specify *data flow*. Then we can characterize common programming paradigms, like functions (non-failing effect-free predicates without inout-parameters) or methods (effect-free predicates with one inout-parameter, the *receiver* object). This will make programs more transparent, and enhance their implementation.

If a functional style of programming shall be supported, we need to check programs for *unique normalization*. There is some hope that results concerning confluence and termination of term rewriting [14] and plain graph transformation [18] can be combined for that purpose.

And, last but not least, DiaPlan must be *implemented*.

References

1. G. Busatto. *An Abstract Model of Hierarchical Graphs and Hierarchical Graph Transformation*. Dissertation, Universität Paderborn, June 2002.
2. F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In Rozenberg [20], chapter 2, pages 95–162.
3. F. Drewes, B. Hoffmann, and M. Minas. Constructing shapely nested graph transformations. In H.-J. Kreowski and P. Knirsch, editors, *Proc. Int'l Workshop on Applied Graph Transformation (AGT'02)*, 2002. 107–118.
4. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64(2):249–283, 2002.
5. G. Engels, H. Ehrig, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. II: Applications, Languages, and Tools*. World Scientific, Singapore, 1999.
6. G. Engels and R. Heckel. Graph transformation as a conceptual and formal framework for system modelling and evolution. In U. Montanari, J. Rolim, and E. Welz, editors, *Automata, Languages, and Programming (ICALP 2000 Proc.)*, number 1853 in Lecture Notes in Computer Science, pages 127–150. Springer, 2000.
7. C. Ermel, M. Rudolf, and G. Taentzer. The Agg approach: Language and environment. In Engels et al. [5], chapter 14, pages 551–603.
8. P. Fradet and D. Le Métayer. Structured Gamma. *Science of Computer Programming*, 31(2/3):263–289, 1998.
9. J. V. Guttag and J. J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–51, 1978.
10. A. Habel. *Hyperedge Replacement: Grammars and Languages*. Number 643 in Lecture Notes in Computer Science. Springer, 1992.
11. B. Hoffmann. From graph transformation to rule-based programming with diagrams. In M. Nagl, A. Schürr, and M. Münch, editors, *Int'l Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, number 1779 in Lecture Notes in Computer Science, pages 165–180. Springer, 2000.

12. B. Hoffmann. Shapely hierarchical graph transformation. In *Proc. IEEE Symposia on Human-Centric Computing Languages and Environments*, pages 30–37. IEEE Computer Press, 2001.
13. B. Hoffmann and M. Minas. Towards rule-based visual programming of generic visual systems. In N. Dershowitz and C. Kirchner, editors, *Proc. Workshop on Rule-Based Languages*, Montréal, Quebeq, Canada, Sept. 2000.
14. J. W. Klop. Term rewriting systems. In S. Abramsky, D. M. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 1–116. Oxford University Press, 1992.
15. H.-J. Kreowski and S. Kuske. Graph transformation units and modules. In Ehrig et al. [5], chapter 15, pages 607–638.
16. S. Kuske. *Transformation Units – A Structuring Pronciple for Graph Transformation Systems.* Dissertation, Universität Bremen, Fachbereich Mathematik u. Informatik, 2000.
17. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
18. D. Plump. *Computing by Graph Rewriting.* Habilitationsschrift, Universität Bremen, 1999.
19. D. Plump and A. Habel. Graph unification and matching. In J. E. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, number 1073 in Lecture Notes in Computer Science, pages 75–89. Springer, 1996.
20. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. I: Foundations.* World Scientific, Singapore, 1997.
21. M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, 1998.
22. A. Schürr, A. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [5], chapter 13, pages 487–550.

Hyperedge Substitution in Basic Atom-Replacement Languages*

Renate Klempien-Hinrichs

Universität Bremen, Fachbereich Mathematik/Informatik
Postfach 33 04 40, D–28334 Bremen, Germany
rena@informatik.uni-bremen.de

Abstract. We introduce basic atom replacement in hypergraphs, which allows us to combine node-rewriting steps and hyperedge-rewriting steps within one type of grammar. The generated languages are closed under hyperedge substitution, thereby providing a structured way to build complex hypergraph languages from simple ones. In particular, there are basic atom-replacement languages which cannot be generated by any type of context-free hypergraph grammar.

1 Introduction

Where in a graph, an edge distinguishes a pair of nodes (the source node and the target node of the edge), a hyperedge in a hypergraph distinguishes an arbitrarily long sequence of nodes. Hence, a graph can be seen as a special hypergraph. Generalising the concept of Chomsky grammars from strings to hypergraphs, the application of a production consists of *choosing*, in the host hypergraph, a subhypergraph to be rewritten that corresponds to the left-hand side of the production, *removing* that subhypergraph to yield the remainder of the host hypergraph, *inserting* disjointly the right-hand side of the production, and *linking* the replacing hypergraph with the remainder.

Among string languages, the context-free ones are of particular interest since given a derivation in a context-free grammar, one can reorder derivation steps by the context-freeness lemma or, equivalently, represent the derivation as a derivation tree. This is essential to prove structural properties of context-free string languages such as e.g. closure under substitution and the pumping lemma [1] or Parikh's theorem [11]. For structures more complex than strings, axioms to classify a grammar as context-free are given in [3].

Steering towards context-free hypergraph grammars, the left-hand side of an applied production can be expected to consist of one nonterminal symbol, and the replaced subhypergraph to be an atomic item of the host hypergraph that is labelled with that symbol. In contrast to strings, a hypergraph has two types of atomic items: nodes and hyperedges. Consequently, there are two different

* This research was partially supported by the EC TMR Network GETGRATS (General Theory of Graph Transformation Systems) and the ESPRIT Basic Research Working Group APPLIGRAPH (Applications of Graph Transformation).

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 192–206, 2002.
© Springer-Verlag Berlin Heidelberg 2002

types of context-free grammars, depending on whether nodes or hyperedges are rewritten (cf. Engelfriet [5]). In confluent node-rewriting grammars [6,8], nodes of the replacing hypergraph can be connected by hyperedges to neighbours of the rewritten node. Hyperedge-rewriting grammars [4] are based on the gluing approach: distinguished nodes of the replacing hypergraph are identified with the incident nodes of the rewritten hyperedge.

With node rewriting, the number of hyperedges can grow very fast, while with hyperedge rewriting, every hyperedge can be treated individually. The separated handle-rewriting hypergraph grammars by Courcelle et al. [2] may be seen as an attempt to combine both advantages in one context-free approach: a handle is a hyperedge together with its incident nodes, and when a handle is rewritten, nodes of the replacing hypergraph can be connected by hyperedges to neighbours of the handle. However, the class of their generated languages is incomparable with that of hyperedge-rewriting languages [2], and both of them are included in the class of confluent node-rewriting languages [9].

In this paper, we propose basic atom-replacement grammars, i.e. grammars that combine node-rewriting steps with hyperedge-rewriting steps so that they cannot interfere with one another. These grammars specify a class of hypergraph languages that is closed under hyperedge substitution. As a consequence, complex hypergraph languages can be specified with (confluent) basic atom-replacement grammars in a structured way. Examples are the set GRAPHS of all graphs, ACYCLIC of all acyclic graphs, HAMILTONIAN of all hamiltonian graphs, CONNECTED of all connected graphs, and the set TOURNAMENTS of all tournaments. While the closure result itself is not very hard to obtain, its importance is underlined by the fact that none of the above-mentioned languages can be generated by the known types of context-free hypergraph grammars.

The paper is organised as follows. In Section 2, hypergraphs, node rewriting and hyperedge rewriting are presented. Basic atom replacement is introduced and its claim to context-freeness discussed in Section 3. In Section 4, hyperedge substitution in hypergraphs is defined and the closure of classes of hypergraph languages under this operation is studied. Section 5 contains some concluding remarks.

2 Node Rewriting and Hyperedge Rewriting

The natural numbers are denoted by $\mathbb{N}$, the positive integers by $\mathbb{N}_+$, and for $n \in \mathbb{N}$ let $[n] = \{1, 2, \ldots, n\}$. In particular, $[0] = \emptyset$. The powerset of a set A is denoted by $\mathcal{P}(A)$. Given two sets A, B, the natural extension of a mapping $f\colon A \to B$ to sets and strings, respectively, is $f\colon \mathcal{P}(A) \to \mathcal{P}(B)$ and $f^*\colon A^* \to B^*$, respectively.

Throughout the paper, we will assume Σ to be a finite set of symbols (the *labels*), which is partitioned into four sets, $\Sigma = N_V \cup N_E \cup T_V \cup T_E$. We call $N = N_V \cup N_E$ the set of *nonterminal* symbols, $T = T_V \cup T_E$ the set of *terminal* symbols, $\Sigma_V = N_V \cup T_V$ the set of *node* labels, and $\Sigma_E = N_E \cup T_E$ the set of *hyperedge* labels.

A *hypergraph* over Σ is a tuple $H = (V_H, E_H, lab_H, att_H)$ where V_H and E_H are disjoint finite sets of nodes and hyperedges, respectively, $lab_H \colon V_H \cup E_H \to \Sigma$ assigns a label to nodes and hyperedges such that $lab_H(V_H) \subseteq \Sigma_V$ and $lab_H(E_H) \subseteq \Sigma_E$, and $att_H \colon E_H \to {V_H}^*$ assigns a (possibly empty) sequence $att_H(e)$ of *attachment nodes* to each hyperedge $e \in E_H$. An *atom* is a node or a hyperedge.

In a picture of a graph, a node is drawn as a circle ○ and a hyperedge as a square □, with its tentacles represented by numbered lines. A hyperedge ○–1–□–2–○ may also be drawn as an arrow ○⟶○. The label and name of an atom are written next to and within that atom, respectively; thus ⓥ $_a$ represents an a-labelled node v.

For a hyperedge $e \in E_H$ with $att_H(e) = v_1 \dots v_n$, we denote the ith *incident* node of e by $att_H(e, i) = v_i$ (where $i \in [n]$ is said to be a *tentacle* of e) and the *rank* of e by $rank_H(e) = n$. An atom x with label $lab_H(x) = \sigma$ is also called *σ-labelled*.

Let H and H' be hypergraphs. A *hypergraph isomorphism* $f \colon H \to H'$ is a pair $f = (f_V, f_E)$ of bijections $f_V \colon V_H \to V_{H'}$ and $f_E \colon E_H \to E_{H'}$ such that $lab_{H'}(f_V(v)) = lab_H(v)$ for all $v \in V_H$, and $lab_{H'}(f_E(e)) = lab_H(e)$ and $att_{H'}(f_E(e)) = f_V^*(att_H(e))$ for all $e \in E_H$. If an isomorphism $f \colon H \to H'$ exists, then H and H' are *isomorphic*, denoted $H \cong H'$. For a (*concrete*) hypergraph H, the class $[\![H]\!] = \{H' \mid H \cong H'\}$ of hypergraphs isomorphic to H is also called an *abstract* hypergraph. (In the sequel, the notation $[\![\ ]\!]$ will also be used for isomorphy classes of more complex objects.)

For the replacement of nodes, we consider hypergraphs with embedding, i.e. hypergraphs that are extended by a so-called connection relation. An element of a connection relation can recognize a hyperedge incident to the replaced node (by the label of the hyperedge, the tentacles attached to that node, and the labels of the other attachment nodes) and specifies how a connecting hyperedge is to be created.

Figure 1 illustrates how a hypergraph with embedding replaces a node: The X-labelled node v in the left hypergraph is replaced with a hypergraph with embedding (H, C) where H contains in particular two nodes u_1 and u_2 and C consists of the connection instructions $(c, b\Diamond b/c, u_2 3 u_1 1)$ and $(c, b\Diamond b/d, 1u_2)$. Both instructions recognize the c-labelled hyperedge e of rank 3 in the left hypergraph: the first incident node of e is labelled b, the second is v itself as indicated by the *occurrence* symbol $\Diamond$, and the third is again labelled b. The first instruc-

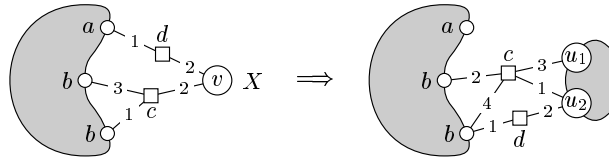


Fig. 1. Replacement of a node

tion then specifies the creation of a c-labelled hyperedge of rank 4, with u_2 as the first incident node, $att(e,3)$ as the second, u_1 as the third, and $att(e,1)$ as the fourth. Similarly, the second instruction specifies the creation of a d-labelled hyperedge linking $att(e,1)$ and u_2. These two new hyperedges are shown in the right hypergraph. Finally, the d-labelled hyperedge on the left is not described by any connection instruction in C and is therefore deleted.

Formally, let $\Diamond$ be a new symbol, i.e. $\Diamond \notin \Sigma$. A *hypergraph with embedding* over Σ is a pair (H, C) where H is a hypergraph over Σ (with $V_H \cap \mathbb{N}_+ = \emptyset$) and the *connection relation* $C \subseteq (\Sigma_E \times (\Sigma_V \cup \{\Diamond\})^*) \times (\Sigma_E \times (\mathbb{N}_+ \cup V_H)^*)$ is a finite set of *connection instructions* of the form $(\alpha, x_1 \cdots x_m/\beta, y_1 \cdots y_n)$ such that for each $i \in [n]$, $y_i \in \mathbb{N}_+$ implies $1 \leq y_i \leq m$ and $x_{y_i} \in \Sigma$. Two hypergraphs with embedding (H, C) and (H', C') are isomorphic if there is an isomorphism $f\colon H \to H'$ that extends to the connection relations, i.e. $C' = \{(\alpha, x_1 \cdots x_m/\beta, \bar{f}_V^*(y_1 \cdots y_n)) \mid (\alpha, x_1 \cdots x_m/\beta, y_1 \cdots y_n) \in C\}$, where $\bar{f}_V(y_i) = f_V(y_i)$ if y_i is a node and $\bar{f}_V(y_i) = y_i$ otherwise. A hypergraph with empty embedding $(H, \emptyset)$ will be considered the same as the hypergraph H.

Let H_1 be a hypergraph containing a node v and (H_2, C_2) a hypergraph with embedding disjoint from H_1. The hypergraph $H_3 = H_1[v/(H_2, C_2)]$ is obtained as a result of the following basic steps:

1. REMOVE v together with all incident hyperedges from H_1, yielding the remainder H_1^- of H_1.
2. ADD H_2 to H_1^-.
3. CONNECT H_2 and H_1^- with hyperedges according to the connection relation C_2: for all hyperedges $e \in E_{H_1}$ incident with v and all connection instructions $(\alpha, x_1 \cdots x_m/\beta, y_1 \cdots y_n) \in C_2$ such that
   - $lab_{H_1}(e) = \alpha$,
   - $rank_{H_1}(e) = m$, and
   - for all $i \in [m]$, $x_i = \begin{cases} \Diamond & \text{if } att_{H_1}(e,i) = v \\ lab_{H_1}(att_{H_1}(e,i)) & \text{otherwise,} \end{cases}$

   add a hyperedge e' with
   - $lab_{H_3}(e') = \beta$,
   - $rank_{H_3}(e') = n$, and
   - for all $j \in [n]$, $att_{H_3}(e', j) = \begin{cases} y_j & \text{if } y_j \in V_{H_2} \\ att_{H_1}(e, y_j) & \text{if } y_j \in \mathbb{N}_+. \end{cases}$

For the replacement of hyperedges, we consider hypergraphs with gluing, i.e. hypergraphs that are extended by a so-called gluing relation. An element of a gluing relation checks that the attachment nodes of the replaced hyperedge have suitable labels and specifies which nodes of the replacing hypergraph are to be identified with these incident nodes.

Figure 2 illustrates how a hypergraph with gluing replaces a hyperedge: The Y-labelled hyperedge e in the left hypergraph is replaced with a hypergraph with gluing (H, G) where H contains in particular two nodes u_1 and u_2 and G consists of the gluing instruction $(aabb/u_1u_1u_2u_2)$. This instruction leads to the identification of the first and second incident node of e with u_1, and the third

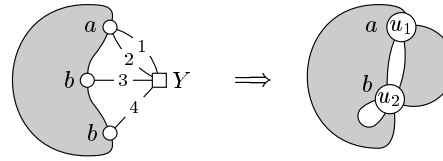


Fig. 2. Replacement of a hyperedge

and fourth incident nodes of e with u_2. Note that thereby $att(e, 3)$ and $att(e, 4)$ are identified with one another. Symmetrically, two tentacles of the replaced hyperedge to the same node could lead to the identification of two nodes in the replacing hypergraph.

Formally, a *hypergraph with gluing* over Σ is a pair (H, G) where H is a hypergraph over Σ and the *gluing relation* $G \subseteq \bigcup_{i\in\mathbb{N}} \Sigma_E^i \times V_H^i$ is a finite set of *gluing instructions* of the form $(\gamma_1 \cdots \gamma_k / v_1 \cdots v_k)$ such that $\gamma_1 \cdots \gamma_k = lab_H^*(v_1 \cdots v_k)$. Two hypergraphs with gluing (H, G) and (H', G') are isomorphic if there is an isomorphism $f\colon H \to H'$ that extends to the gluing relations, i.e. $G' = \{(\gamma_1 \cdots \gamma_k / f_V^*(u_1 \cdots u_k)) \mid (\gamma_1 \cdots \gamma_k / u_1 \cdots u_k) \in G\}$. A hypergraph with empty gluing $(H, \emptyset)$ will be considered the same as the hypergraph H.

Let H_1 be a hypergraph containing a hyperedge e and (H_2, G_2) a hypergraph with gluing disjoint from H_1. The hypergraph $H_3 = H_1[e/(H_2, G_2)]$ is obtained as a result of the following basic steps:

1. REMOVE the hyperedge e (but not its incident nodes) from H_1, yielding the remainder H_1^- of H_1.
2. ADD H_2 to H_1^-.
3. GLUE H_1^- and H_2 according to the gluing relation G_2: for all gluing instructions $(\gamma_1 \cdots \gamma_k / u_1 \cdots u_k) \in G_2$ with $lab_{H_1}^*(att_{H_1}(x)) = \gamma_1 \cdots \gamma_k$, identify, for all $i \in [k]$, $att_{H_1}(e, i)$ and u_i.

It is convenient to consider the replacement of an atom x in a hypergraph (H_1, rel_1) *with embedding or with gluing* with a hypergraph (H_2, rel_2) with embedding (if x is a node) or gluing (if x is a hyperedge), yielding a hypergraph $(H_3, rel_3) = (H_1, rel_1)[x/(H_2, rel_2)]$ with embedding or gluing, respectively, such that $H_3 = H_1[x/(H_2, rel_2)]$. The definition of rel_3 is straightforward, so that an informal description instead of the lengthy technical details may suffice here.

- If (H_2, rel_2) is a hypergraph with gluing, then the nodes of H_1 incident to the hyperedge x may be identified with nodes of H_2 and thereby change their names. Whenever such a node of H_1 occurs in the connection or gluing relation rel_1, the new name has to occur instead in rel_3; otherwise, rel_3 is the same as rel_1.
- If (H_2, rel_2) is a hypergraph with embedding and (H_1, rel_1) is a hypergraph with gluing, then rel_3 equals rel_1.
- If both (H_2, rel_2) and (H_1, rel_1) are hypergraphs with embedding, then every connection instruction in rel_1 in which the node x occurs has to be modified

so that if it recognizes a hyperedge e and specifies the creation of a hyperedge e' incident to x, and a connection instruction in rel_2 recognizes e' and specifies the creation of a hyperedge e'', rel_3 contains the concatenation of these instructions, recognizing e and directly specifying the creation of e''. Otherwise, rel_3 is the same as rel_1. (For the details see Definition 2.6 in [10].)

3 Basic Atom Replacement

In a grammar that combines node rewriting steps with hyperedge rewriting steps, some undesirable effects may occur when a nonterminal node and a nonterminal hyperedge are incident. In a step rewriting the node, the hyperedge can be deleted or multiplied or get a terminal label. In a step rewriting the hyperedge, the node can be identified with another node incident with that hyperedge. In order to avoid these effects, we define *basic* hypergraphs with embedding or with gluing, respectively.

A hypergraph is *nonterminal disjoint* if a nonterminal node and a nonterminal hyperedge are never incident. A hypergraph with embedding (H, C) is *basic* if H is nonterminal disjoint and for all connection instructions $(\alpha, x_1 \cdots x_m / \beta, y_1 \cdots y_n) \in C$, α is terminal, and β is nonterminal only if all nodes specified by $y_1 \cdots y_n$ are terminal, i.e. if for all $j \in [n]$, either $y_j \in V_H$ and $lab_H(y_j) \in T$, or $y_j \in \mathbb{N}_+$ and $x_{y_j} \in T$. A hypergraph with gluing (H, G) is *basic* if H is nonterminal disjoint and G specifies only terminal nodes, i.e. $G \subseteq T_E^* \times V_H^*$.

The following lemma implies that node and hyperedge replacements do not interfere with each other if they are carried out involving only basic hypergraphs with embedding or with gluing, respectively.

Lemma 1. *Let (H_1, rel_1) be a basic hypergraph with embedding or with gluing, respectively. If either x is a node in H_1 and (H_2, rel_2) is a basic hypergraph with embedding disjoint from H_1, or x is a hyperedge in H_1 and (H_2, rel_2) is a basic hypergraph with gluing disjoint from H_1, then $(H_1, rel_1)[x/(H_2, rel_2)]$ is a basic hypergraph with embedding or with gluing, respectively.*

Now basic atom-replacement grammars can be defined, i.e. hypergraph grammars in which node-rewriting and hyperedge-rewriting steps can be combined. Moreover, the axiom may be a basic hypergraph with gluing, so that the language generated by such a grammar can be substituted for hyperedges as will be defined in the next section.

A *basic atom-replacement grammar* is a tuple $AG = (N, T, P, (Z, G_Z))$ where N and T are finite, disjoint sets of nonterminal and terminal labels respectively, P is a finite set of *productions* of the form $X ::= (R, rel)$ such that either $X \in N_V$ and (R, rel) is a basic hypergraph with embedding over $N \cup T$, or $X \in N_E$ and (R, rel) is a basic hypergraph with gluing over $N \cup T$, and the *axiom* is (Z, G_Z), a basic hypergraph with gluing over $N \cup T$.

A basic hypergraph with gluing (H, G) *derives directly* to a basic hypergraph with gluing (H', G') by applying a production $p = (X ::= (R, rel)) \in P$ to an X-

labelled atom x of H,[1] denoted $(H, G) \Rightarrow_{x,p} (H', G')$ or $(H, G) \Rightarrow_P (H', G')$, if $(H', G') = (H, G)[x/(R, rel)]$. As usual, $\Rightarrow_P^*$ denotes the reflexive and transitive closure of $\Rightarrow_P$. Moreover, the set

$$S(AG) = \{(H, G) \mid (Z, G_Z) \Rightarrow_P^* (H, G)\}$$

contains the *sentential forms* of AG, and

$$L(AG) = \{[\![(H, G)]\!] \mid (H, G) \in S(AG) \text{ is labelled over } T\}$$

is its *generated language*. Note that the generated language contains *abstract* hypergraphs with gluing; we will call any set of abstract hypergraphs with gluing over T a *hypergraph language* and denote the set of these languages by $\mathcal{L}_T$.

Example 1 (basic atom-replacement grammars). Let $N_V = \{X\}$, $N_E = \{Y\}$, $T_V = \{a\}$, and $T_E = \{b\}$. As each set contains a unique symbol, they will not be written in pictures. Rather, nonterminal items are drawn with bold lines.

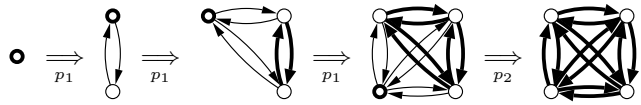


Fig. 3. Deriving complete graphs with node rewriting

Figure 3 illustrates that one can generate the complete graphs with terminally labelled nodes and nonterminally labelled edges starting from the axiom $\circ X$ and using the node-rewriting productions p_1 and p_2:

$$X \underset{p_1}{::=} \left(\begin{array}{c} u \\ \\ v \end{array}, \left\{ \begin{array}{l} (b, \Diamond a/b, u2)\,, (b, a\Diamond/b, 1u), \\ (b, \Diamond a/Y, v2), (b, a\Diamond/Y, 1v) \end{array} \right\} \right)$$

$$X \underset{p_2}{::=} \Big(\textcircled{v},\ \big\{(b, \Diamond a/Y, v2), (b, a\Diamond/Y, 1v)\big\} \Big)$$

On this basis, let us specify three basic atom-replacement grammars AG_1, AG_2, and AG_3.

The grammar AG_1 turns every nonterminal edge of one of the complete graphs into a sequence of two edges with the help of the edge-rewriting production p_3, which it contains in addition to p_1 and p_2:

$$Y \underset{p_3}{::=} \left(\begin{array}{c} u \\ \\ v \end{array}, \{(aa/uv)\} \right)$$

[1] We assume (R, rel) to be disjoint from (H, G); if that is not the case, an isomorphic copy of (R, rel) is used.

Instead of exactly two edges, a chain of arbitrarily many edges can be achieved in AG_2, which contains productions p_1, p_2 and in addition p_4, p_5 as follows:

$$Y \underset{p_4}{::=} \left(\begin{array}{c} u \\ \downarrow \\ v \end{array}, \{(aa/uv)\} \right) \qquad Y \underset{p_5}{::=} \left(\begin{array}{c} u \\ \searrow \\ \circ \\ \swarrow \\ v \end{array}, \{(aa/uv)\} \right)$$

In grammar AG_3, one can repeatedly choose a nonterminal edge and derive it into a complete graph whose every node is the target (or source, respectively) of an edge with the same source (or target, respectively) as the edge from which it was derived. For this, AG_3 differs from AG_2 only in that production p_5 is turned into p_6 where the new node (and none of the edges) is nonterminal:

$$Y \underset{p_6}{::=} \left(\begin{array}{c} u \\ \searrow \\ \bullet \\ \swarrow \\ v \end{array}, \{(aa/uv)\} \right)$$

The beginning of a sample derivation in this grammar is shown in Figure 4. Note that as soon as one edge is generated, the production p_2, while having a completely terminal right-hand side graph, will make any incident edge nonterminal. Consequently, p_4, which just changes the label of a hyperedge from nonterminal to terminal, is the only terminating production. ∎

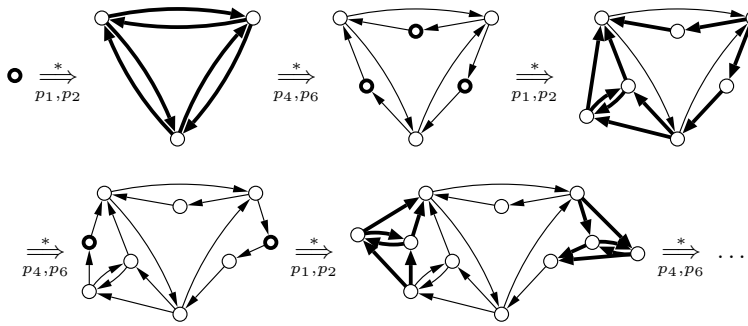


Fig. 4. A derivation in the basic atom-replacement grammar AG_3 of Example 1

Basic atom replacement is designed so that only atoms can be replaced and the replacement of one atom cannot inhibit the replacement of another. These properties suggest that basic atom-replacement grammars might be context-free. According to the axiomatic definition by Courcelle [3], a grammar is *context-free* if it is *associative* and *confluent*.

Associativity means that whenever a sentential form contains an atom x_1 to be rewritten with a production p_1 and the right-hand side of that production

contains an atom x_2 to be rewritten with a production p_2, then the result is the same whether first x_1 is replaced and then x_2, or whether x_1 is replaced with a production obtained from p_1 by replacing therein x_2.

General atom-replacement grammars are not associative, see Example 5.22 in [10]. For basic atom-replacement grammars, however, Lemma 1 implies that all sentential forms are nonterminal disjoint, which in turn ensures associativity.

Lemma 2. *Basic atom-replacement grammars are associative.*

Confluence means that whenever a sentential form contains two atoms x_1, x_2 and there are two productions p_1, p_2 with p_i applicable to x_i, then the result of the two derivation steps must be the same independent of the order in which the steps are executed. In general, this is not the case for node-rewriting grammars (see e.g. Example 1.10 in [6]), and neither for the more general basic atom-replacement grammars. Hyperedge rewriting on its own, however, is naturally context-free, and with the help of Lemma 1 one can show that this property transfers to a sequential independence of hyperedge-rewriting steps in basic atom-replacement derivations.

Lemma 3. *Let (H, G) be a basic hypergraph with gluing containing a hyperedge e and another atom x, and let p_e, p_x be two productions with p_e applicable to e and p_x applicable to x. Then the derivations $(H, G) \Rightarrow_{e,p_e} (H_e, G_e) \Rightarrow_{x,p_x} (H_{ex}, G_{ex})$ and $(H, G) \Rightarrow_{x,p_x} (H_x, G_x) \Rightarrow_{e,p_e} (H_{xe}, G_{xe})$ have the same result, i.e.*

$$(H_{ex}, G_{ex}) = (H_{xe}, G_{xe}).$$

Concerning context-free hypergraph grammars, it is known that confluent node rewriting subsumes hyperedge rewriting in such a way that every hyperedge-rewriting step can be simulated by a corresponding node-rewriting step [9]. This suggests that there is no point in adding hyperedge rewriting to confluent node rewriting. Yet, the generative power of confluent node rewriting is strictly less than that of confluent basic atom replacement.

Theorem 1. *The set of confluent basic atom-replacement languages properly includes the set of confluent node-rewriting languages, which in turn properly includes the set of hyperedge-rewriting languages.*

Proof. We have to show the existence of a confluent basic atom-replacement language that cannot be generated by any confluent node-rewriting grammar.

Consider the grammar AG_1 from Example 1. It is basically a linear (and therefore confluent) node-rewriting grammar generating all complete graphs, enlarged by a hyperedge-rewriting rule to split every edge in two and insert a new node in between. Altogether, AG_1 is confluent. Moreover, the size of the generated graphs, measured in the number of nodes, grows quadratically: as a complete graph with n nodes has $n^2 - n$ edges, adding a node for each edge yields a total of $n + (n^2 - n) = n^2$ nodes.

On the other hand, the size of the graphs (or hypergraphs) in a confluent node-rewriting language is a semilinear set (cf. Parikh's theorem [11]). Hence, $L(AG_1)$ cannot be generated by a confluent node-rewriting grammar. □

Although Lemmas 2 and 3 seem to imply that confluent basic atom-replacement grammars must be context-free in the sense of Courcelle [3], the existence of such a language where the size of the members is not a semilinear set contradicts this intuition. In fact, consider the derivation in AG_1 shown in Figure 3. Even though the same production is used in the first three derivation steps, the number of new nonterminal atoms, and more specifically of new nonterminal edges, varies depending on the number of edges incident to the rewritten node. From a context-free grammar, however, one expects that the number of nonterminal items generated in applications of the same production be constant. This means that AG_1 is not *nonterminal-preserving* (and hence not even a grammar) in the sense of Courcelle [3].

The reason that a basic atom-replacement grammar can generate a varying number of nonterminal hyperedges in distinct applications of the same production lies in the possibility to turn terminal hyperedges into nonterminal ones during a node-rewriting step. If this is forbidden, then indeed adding hyperedge rewriting to confluent node rewriting does not augment the generative power.

4 Closure under Hyperedge Substitution

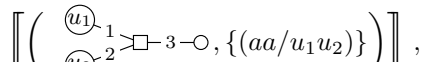
We define hyperedge substitution similarly to Habel [7, Chapter III, Section 1], but for abstract node-labelled hypergraphs (with gluing).

Let $\Gamma \subseteq T$. A mapping $subst\colon \Gamma \to \mathcal{L}_T$ assigning to every symbol in Γ a language of basic hypergraphs with gluing is called *hyperedge substitution*. It is extended to an abstract hypergraph $[\![(H, G)]\!]$ with gluing by defining

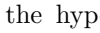
$$\begin{aligned} subst([\![(H,G)]\!]) = \{&[\![(H,G)[e_1/(H_1,G_1)]\dots[e_n/(H_n,G_n)]]\!] \mid \\ &e_1,\dots,e_n \text{ are the hyperedges in } H \text{ with label in } \Gamma, \\ &[\![(H_i,G_i)]\!] \in subst(lab_H(e_i)) \text{ for all } i \in [n], \\ &H \text{ and all the } H_i \text{ are mutually disjoint}\}. \end{aligned}$$

A hyperedge substitution $subst\colon \Gamma \to \mathcal{L}_T$ is *finite* if for all $\gamma \in \Gamma$, $subst(\gamma)$ is finite. It is a *hyperedge homomorphism* if all the $subst(\gamma)$ contain exactly one element.

Example 2 (hyperedge substitution). Let $T_V = \{a\}$ and $T_E = \{b\}$ (as the terminal labels are unique, they can be omitted from pictures), and let H be the complete graph with three nodes and six edges. Defining $subst\colon \{b\} \to \mathcal{L}_T$ as the hyperedge homomorphism assigning to b the set containing as unique element



$subst([\![H]\!])$ consists of the hypergraph shown in Figure 5.

Considering the hyperedge substitution $subst'\colon \{b\} \to \mathcal{L}_T$ assigning to b the infinite set containing all basic hypergraphs with gluing $[\![(H', \{(aa/u_1u_2)\})]\!]$ where H' is a chain of edges of the form  ,  , $subst'([\![H]\!])$ contains, among others, the graphs shown in Figure 6. ∎

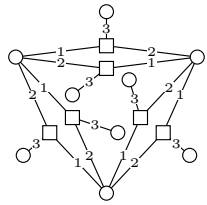


Fig. 5. $subst([\![H]\!])$

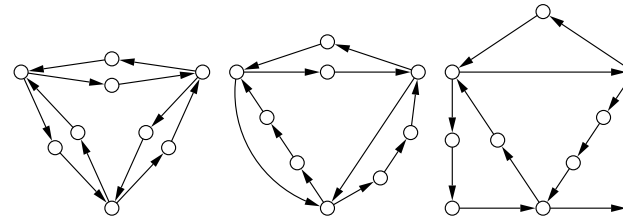


Fig. 6. Three hypergraphs in $subst'([\![H]\!])$

With hyperedge substitution, many interesting graph and hypergraph languages can be built from simple languages in a structured way.

Example 3 (construction of hypergraph languages). Let $T_V = \{a\}$ and $T_E = \{b, c\}$. Moreover, let $selectedge\colon \{b\} \to L$ be the finite substitution with L containing the two basic hypergraphs with gluing

$$[\![(\ \textcircled{u_1}\ \ \textcircled{u_2}\ , \{(aa/u_1u_2)\})]\!] \quad \text{and} \quad [\![(\ \textcircled{u_1} \xrightarrow{b} \textcircled{u_2}, \{(aa/u_1u_2)\})]\!]\,;$$

in other words, *selectedge* allows us to arbitrarily select or delete b-labelled edges.

Extending *selectedge* to the graph languages

(i) COMPLETE of complete graphs with b-labelled edges only,
(ii) O-TOURNAMENTS of ordered tournaments with b-labelled edges only (a tournament is a graph with exactly one edge between every two distinct nodes, and it is ordered if it may be seen as a representation of a strict order),
(iii) $\text{COMPLETE}_{\text{HAMILTONIAN}}$ of complete graphs with a directed hamiltonian cycle distinguished by c-labelled edges, and
(iv) $\text{COMPLETE}_{\text{SPANTREE}}$ of complete graphs with a spanning tree distinguished by c-labelled edges, respectively,

yields the graph languages

(i) GRAPHS of *all* graphs,
(ii) ACYCLIC of all acyclic graphs,
(iii) HAMILTONIAN of all hamiltonian graphs, and
(iv) CONNECTED of all connected graphs, respectively.

If desired, *selectedge* can be modified to substitute a b-labelled edge for every c-labelled edge, so that the hamiltonian cycles in (iii) and the spanning trees in (iv) are no longer distinguished.

As a last example, consider the substitution $selectdir\colon \{b\} \to L'$ with L' containing the two basic hypergraphs with gluing

$$[\![(\ \textcircled{u_1} \xrightarrow{b} \textcircled{u_2}, \{(aa/u_1u_2)\})]\!] \quad \text{and} \quad [\![(\ \textcircled{u_1} \xleftarrow{b} \textcircled{u_2}, \{(aa/u_1u_2)\})]\!]\,,$$

so that one can arbitrarily change the direction of a b-labelled edge. Then extending *selectdir* to the language O-TOURNAMENTS yields the set TOURNAMENTS of *all* tournaments. ∎

As, unlike [7], the extension of hyperedge substitution is defined here for abstract hypergraphs, we have to verify that the result of the extension of a hyperedge substitution depends neither on the sequence in which the hyperedges are replaced, nor on the concrete choice of the replacing hypergraphs with gluing, nor on the concrete choice of the hypergraph whose hyperedges are replaced.

Lemma 4. *The extension of a hyperedge substitution to abstract hypergraphs is well defined.*

From Habel [7, Chapter III, Theorem 1.2], we know:

Fact 1. *Hyperedge-rewriting languages are closed under hyperedge substitution.*

In contrast, confluent node-rewriting languages are not closed even under the simplest type of hyperedge substitution.

Proposition 1. *Confluent node-rewriting languages are not closed under hyperedge homomorphism.*

Proof. The set COMPLETE of complete graphs can be generated by a confluent (even linear) node-rewriting grammar (cf. Example 1). With the hyperedge homomorphism assigning to the unique edge label the right-hand side of the production p_3 in Example 1, we obtain a set of graphs where the number of nodes grows as n^2 (cf. the proof of Theorem 1). However, the the number of nodes of the graphs (or hypergraphs) in a confluent node-rewriting language is a semilinear set. □

Although basic atom-replacement grammars are in general not context-free, one can prove by their associativity (see Lemma 2) and the sequential independence of hyperedge-rewriting steps (see Lemma 3) that the generated languages are closed under hyperedge substitution. Moreover, hyperedge substitution is compatible with confluence.

Theorem 2. *The following classes of hypergraph languages are effectively closed under hyperedge substitution: (1) basic atom-replacement languages and (2) confluent basic atom-replacement languages.*

Proof. Let $AG = (N, T, P, (Z, G_Z))$ be a basic atom-replacement grammar and, for some $\Gamma \subseteq T$, $subst\colon \Gamma \to \mathcal{L}_T$ a hyperedge substitution such that $subst(\gamma)$ is generated by a basic atom replacement grammar $AG_\gamma = (N_\gamma, T, P_\gamma, (Z_\gamma, G_\gamma))$. Moreover, let $\hat{\Gamma} = \{\hat{\gamma} \mid \gamma \in \Gamma\}$ be another set of symbols; they will be used to label the hyperedges that are replaced by the substitution. Without loss of generality, assume that $\hat{\Gamma}$ and the sets of nonterminal labels of all grammars are mutually disjoint.

Construct a basic atom-replacement grammar $AG' = (N', T, P', (Z', G'_Z))$ that generates $subst(L(AG))$, as follows. The nonterminal symbols of AG' are all the nonterminal symbols of AG and the AG_γ's, together with the symbols in $\hat{\Gamma}$. The new axiom (Z', G'_Z) is obtained from the old axiom (Z, G_Z) by changing every (terminal) label $\gamma \in \Gamma$ of a hyperedge in Z whose incident nodes are terminal into its nonterminal counterpart $\hat{\gamma}$. The productions of AG' consist of

- a modified version of the productions in P where in the right-hand sides, every occurrence of a label $\gamma \in \Gamma$ for a hyperedge (or specification of a hyperedge in a connection instruction) that is incident to terminal nodes only is changed into $\hat{\gamma}$,
- a production $\hat{\gamma} ::= (Z_\gamma, G_\gamma)$ for every $\gamma \in \Gamma$, and
- all the productions of the AG_γ's.

Clearly, in AG' one can derive any hypergraph in $L(AG)$, with the difference that for $\gamma \in \Gamma$, γ-labelled hyperedges are now labelled $\hat{\gamma}$. Moreover, every hypergraph in $L(AG_\gamma)$ can be derived in AG' starting from the axiom (Z_γ, G_γ), so that by the associativity of AG', rewriting a $\hat{\gamma}$-labelled hyperedge into (Z_γ, G_γ) and then executing a derivation "in AG_γ" is equivalent to replacing the hyperedge with a hypergraph in $L(AG_\gamma)$. Hence, $subst(L(AG)) \subseteq L(AG')$.

For the other inclusion, consider any derivation in AG' ending in a terminal hypergraph. By the sequential independence of hyperedge-rewriting steps and the fact that each $\hat{\gamma}$-labelled hyperedge must be incident to terminal nodes only, one can reorder the steps in this derivation so that first come all rewriting steps using productions obtained from those in AG, and then, for every hyperedge labelled $\hat{\gamma}$ with $\gamma \in \Gamma$, all steps rewriting that hyperedge and its descendants. Again by the associativity of AG', this amounts to saying that the hyperedge is replaced with a hypergraph in $L(AG_\gamma)$. Altogether, the result of the whole derivation lies in $subst(L(AG))$, and consequently $L(AG') \subseteq subst(L(AG))$.

Finally, if AG and all the AG_γ's are confluent, then so is AG' because rewriting steps cannot interact across the "border" of terminal nodes with which any $\hat{\gamma}$-labelled hyperedge is incident. □

With Theorem 2, it is easy to show that the languages GRAPHS, ACYCLIC, HAMILTONIAN, CONNECTED, and TOURNAMENTS from Example 3 are confluent basic atom-replacement languages. This is because there are confluent node-rewriting grammars generating the languages COMPLETE, O-TOURNAMENTS, $\text{COMPLETE}_{\text{HAMILTONIAN}}$, and $\text{COMPLETE}_{\text{SPANTREE}}$, respectively, and the finite languages of hypergraphs with gluing used for the substitutions are hyperedge-rewriting languages. Note, however, that these five atom-replacement languages cannot be generated by any known type of context-free hypergraph grammar.

5 Conclusion

Basic atom replacement combines the concepts of node-rewriting and hyperedge-rewriting steps in one framework. Thus it is possible to generate a large number of hyperedges in a single step, as well as to transform single hyperedges in a direct way. Moreover, the combination of node-rewriting steps with hyperedge-rewriting steps in one grammar is made in such a way that the grammar remains associative, and hyperedge rewriting stays sequentially independent. As a consequence, one can show the class of basic atom-replacement languages, as well as its subclass of confluent basic atom-replacement languages, to be closed under hyperedge substitution.

However, even the restriction to confluent basic atom-replacement grammars does not yield a context-free rewriting device because these grammars are not necessarily nonterminal preserving. While in [3], devices that lack this property are not called grammars at all, the example of confluent basic atom-replacement grammars suggests a more general viewpoint. In fact, due to the confluence and associativity of these grammars, it should be possible to extend the notion of derivation trees and define a kind of *derivation graphs* in a meaningful way. The vertices of a derivation graph might keep track of hyperedges as well as nodes so that one can represent the transition of a terminal hyperedge into a nonterminal one. It would be interesting to see which derivation tree-based results known for context-free grammars carry over to confluent and associative grammars. For example, one might find a tool to prove that a hypergraph language is not a confluent basic atom-replacement language.

It may be possible to find a classification of basic atom-replacement languages that is linked to their closure under hyperedge substitution. Owing to the associativity of basic atom-replacement grammars, the effect of rewriting a hyperedge and subsequently all its nonterminal descendants can also be achieved by hyperedge substitution. Hence, a derivation can be seen as a number of node-rewriting steps, followed by a hyperedge substitution, followed again by node-rewriting steps, and so forth; the derivation in Figure 4 was arranged in this way. In other words, a basic atom-replacement language can be generated by an iteration of hyperedge substitutions with node-rewriting languages. We conjecture that there may be a hierarchy of languages based on the number of iterations it takes minimally to generate a certain language. A prospective candidate to prove this conjecture is the grammar AG_3 given in Example 1 and variants thereof. With its productions, one can repeatedly replace an edge by a complete graph whose every node is connected to the source and target of the replaced edge. It may well be that the language obtained from the set of complete graphs by twice substituting edges in this way cannot be constructed if maximally one substitution cycle is allowed.

Finally, let us mention that there are probably many other interesting graph and hypergraph languages that are not context-free, but have a simple specification as (confluent) basic atom-replacement languages.

References

1. Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–177, 1961.
2. Bruno Courcelle, Joost Engelfriet, and Grzegorz Rozenberg. Handle-rewriting hypergraph grammars. *Journal of Computer and System Sciences*, 46:218–270, 1993.
3. Bruno Courcelle. An axiomatic definition of context-free rewriting and its application to NLC graph grammars. *Theoretical Computer Science*, 55:141–181, 1987.
4. 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. I: Foundations*, chapter 2, pages 95–162. World Scientific, 1997.

5. Joost Engelfriet. Context-free graph grammars. In G. Rozenberg and A. Salomaa, editors, *Handbook of Formal Languages. Vol. III: Beyond Words*, chapter 3, pages 125–213. Springer, 1997.
6. Joost Engelfriet and Grzegorz Rozenberg. Node replacement graph grammars. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, chapter 1, pages 1–94. World Scientific, 1997.
7. Annegret Habel. *Hyperedge Replacement: Grammars and Languages*, volume 643 of *Lecture Notes in Computer Science*. Springer, 1992.
8. Renate Klempien-Hinrichs. Node replacement in hypergraphs: Simulation of hyperedge replacement, and decidability of confluence. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Proc. Fifth Intl. Workshop on Graph Grammars and Their Application to Comp. Sci.*, volume 1073 of *Lecture Notes in Computer Science*, pages 397–411. Springer, 1996.
9. Renate Klempien-Hinrichs. The generative power of context-free node rewriting in hypergraphs. *Grammars*, 2:211–221, 1999.
10. Renate Klempien-Hinrichs. *Context-free Hypergraph Grammars. Node and Hyperedge Rewriting with an Application to Petri Nets*. Books on Demand, Norderstedt, 2001. Doctoral thesis, Universität Bremen, 2000.
11. Rohit J. Parikh. On context-free languages. *Journal of the Association for Computing Machinery*, 13:570–581, 1966.

Distributed Graph Transformation Units*

Peter Knirsch and Sabine Kuske

Universität Bremen, Fachbereich 3
Postfach 33 04 40
D-28334 Bremen, Germany
{knirsch,kuske}@informatik.uni-bremen.de

Abstract. Transformation units are a structuring principle for graph transformation systems. In this paper we introduce distributed transformation units that can be used to model distributed graph transformation systems. A distributed transformation unit consists of a set of local transformation units which are connected via interface units. Semantically, a distributed transformation unit transforms distributed graphs consisting of a set of local graphs connected via interface graphs, in such a way that every local graph with its interfaces is transformed by a local transformation unit. Since the interface graphs can be modified concurrently by various local transformation units we introduce a concurrent semantics of transformation units. The presented concepts are illustrated with a running example of a simple game where two different actors access a common game board randomly.

1 Introduction

Graph transformation [25,7,8] combines graphs with rules and has been thoroughly studied during the last three decades. It allows the specification and programming of dynamic systems by representing system states as graphs and state changes as applications of graph transformation rules. One benefit of this approach is the visual specification of a system together with a well-defined formal semantics.

In practice, graph transformation systems may become large and difficult to manage. On the one hand there may be so many graph transformation rules within one system that structuring concepts for dividing them into small units of sets of rules become necessary. In the last years, a series of such structuring concepts for graph transformation systems has been developed [13]. In this paper we concentrate on the structuring concept of transformation units [18,17,21]. A transformation unit consists of a set of graph transformation rules, a set of imported transformation units, a control condition, an initial graph class expression, and a terminal graph class expression. Semantically, it transforms initial graphs into terminal graphs by interleaving rule applications with transformations of imported units such that the control condition is satisfied. One outstanding feature of transformation units is their approach independence meaning that

* This work has been supported by the *Deutsche Forschungsgemeinschaft* and the ESPRIT Working Group Applications of Graph Transformation (APPLIGRAPH).

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 207–222, 2002.
© Springer-Verlag Berlin Heidelberg 2002

they can be used for all graph transformation approaches (see [25] for an overview on the major graph transformation approaches). On the other hand, the graphs to be transformed by a graph transformation system may be so large that they should be split up into smaller more understandable pieces. The structuring of graphs as well as the transformation of structured graphs has been studied in a series of papers (see e.g. [24,10,28], [29,3,5]).

In this paper we combine the concept of transformation units with a structuring concept for graphs to overcome both above mentioned burdens of large systems. More precisely, we show how distributed graphs can be transformed by distributed transformation units. The distributed graphs, we consider, consist of a set of local graphs that are connected by interface graphs. Local graphs with interfaces are transformed concurrently by the local transformation units of a distributed transformation unit according to synchronization rules and the control condition of the local units. For the transformation of interface graphs the local units use the rules and imported units provided by so-called interface transformation units. Since interface graphs can be transformed concurrently by different local units, a concurrent semantics for transformation units is also presented in this paper. The semantics of a distributed transformation unit is a binary relation on distributed graphs. It contains every pair of distributed graphs where the second one is obtained from the first one by a distributed graph transformation. transformation units which allows transformations of interface graphs. The open semantics is useful if a distributed unit will be combined with still unknown distributed units in a later stage of system development.

The work presented in this paper constitutes a first step in specifying complex distributed and concurrent systems in a structured and approach independent way. It is a straight forward extension of the concept of structuring with transformation units and is meant to contribute to the GRACE initiative (cf. [27,1,16]).

The example used to demonstrate the main concepts of distributed graph transformation is a simple toy example – a game where two actors try to traverse a game board visiting every place without knowledge of each other. Each place can be occupied by one actor at the time only. Each actor has a representation of the board and of its current location exclusively. Only in its private data it can mark visited places. All information needed are represented graphically in a distributed graph. Changes to the common game board which is located in the interface graph of the two actors' local graphs are performed by the application of synchronization rules, which are part of the distributed transfromation unit, to guarantee consistency. The game ends when all places are marked visited.

This paper is organized as follows. In Sect. 2, we recall the concept of transformation units together with its interleaving semantics. In Sect. 3, a concurrent semantics for transformation units is presented which allows to apply rules and imported transformation units concurrently to the actual graph. Sect. 4 introduces distributed transformation units and their semantics. In Sect. 5 related work is shortly touched. The paper ends with the conclusion in Sect. 6.

2 Transformation Units

The main purpose of transformation units is to divide large sets of graph transformation rules into smaller ones in a structured and systematic way. A graph transformation approach provides syntactic and semantic ingredients of transformation units. In the original sense it consists of a class of graphs, a class of rules, a class of control conditions, and a class of graph class expressions. The control conditions are used to regulate the rule application process which in general is non-deterministic. With graph class expressions one can specify sets of graphs which satisfy certain properties. This is useful if rules should be applied only to certain graphs.

In this paper we extend the notion of a graph transformation approach by a subgraph relation and a union operator for graphs. This is motivated by the fact that in the case of concurrent and distributed graph transformation, a transformation unit may apply rules and imported transformation units in parallel to different subparts of a graph. In every graph transformation step a graph is splitted into subgraphs. After transforming the subgraphs the resulting graphs are joined again into a single graph (cf. also [6]). The splitting and joining of graphs can be defined based on the subgraph relation and the union operator provided by the underlying graph transformation approach.

Definition 1 (Graph transformation approach). A *graph transformation approach (with subgraph relation)* is a tuple $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ where

- $\mathcal{G}$ is a class of *graphs* containing $\emptyset$, called the *empty graph*.
- $\subseteq$ is a partial order on $\mathcal{G}$ called *subgraph relation* such that for all $G \in \mathcal{G}$, $\emptyset \subseteq G$. Moreover, for all $G, G' \in \mathcal{G}$ the following holds. Let M be the set of all subgraphs of G and G', i.e. $M = \{H \in \mathcal{G} \mid G \supseteq H \subseteq G'\}$. Then there is exactly one largest graph $H \in M$, i.e. $H' \subseteq H$ for all $H' \in M$. The graph H will be denoted by $G \cap G'$.
- $\cup$ is a commutative and associative gluing operator for graphs which combines two graphs G and G' into a new one in such a way that $G, G' \subseteq G \cup G'$, and for all $G'' \in \mathcal{G}$ with $G, G' \subseteq G''$, $G \cup G' \subseteq G''$.
- $\mathcal{R}$ is a class of *rules*.
- $\Rightarrow$ is a *rule application operator* yielding a binary relation $\Rightarrow_r \subseteq \mathcal{G} \times \mathcal{G}$ for each $r \in \mathcal{R}$.
- $\mathcal{C}$ is a class of *control conditions* such that each $C \in \mathcal{C}$ specifies a binary relation $SEM(C) \subseteq \mathcal{G} \times \mathcal{G}$.
- $\mathcal{E}$ is a class of *graph class expressions* such that each $X \in \mathcal{E}$ specifies a set $SEM(X) \subseteq \mathcal{G}$.

In order to stress that a graph G has a subgraph I, G is also denoted by the pair $(G \supseteq I)$.

Our example is based on the well studied double pushout approach [4]. The underlying class of graphs consists of directed node labelled graphs. In our example a graph transformation rule is a tuple $r = (L, K, R)$ of graphs, where

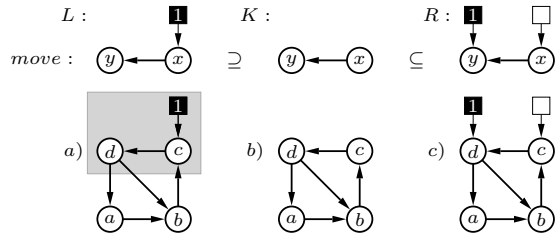


Fig. 1. An application of the rule *move* to an instance of a game board

L is the left-hand side of r, R the right-hand side, and K is a subgraph of L and R. In the top of Fig. 1 an example of a rule *move* is shown the left-hand side of which consists of a graph where node x is marked with a black flag. In the right-hand side the black flag is moved to node y and replaced by a white flag showing that this node has already been visited. The common subgraph K contains two nodes labelled with the variables x and y and an edge pointing from the x-labelled to the y-labelled node[1].

Roughly speaking, the application of a rule (L, K, R) to a graph G replaces an image of the left-hand side L in G by the right-hand side R such that the common part K is kept. Fig. 1c shows the result of applying the rule *move* to the graph in Fig. 1a where the variables x and y are instantiated with c and d, respectively. The rule *move* means that one actor (black flag) moves from the place labelled x to the place y and marks its old place visited (white flag).

As control condition we use term r, i.e. a rule or an imported transformation unit, which means that r is applied once and the term $r!$ which means that rule r must be applied as long as possible as well as the sequential composition of both concepts. The graph class expressions of our example are *all* which specifies the set of all graphs, *marked* meaning that each node has an attached flag and *unmarked*, i.e. no node is marked.

A transformation unit consists of an initial and a terminal graph class expression, a set of rules, a control condition, and a set of imported transformation units. The semantics of transformation units is interleaving in the sense that local rules and imported transformation units are applied sequentially starting with initial graphs and ending up with terminal graphs such that the control condition is satisfied.

Definition 2 (Transformation unit with interleaving semantics).

1. A *transformation unit* over an approach $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ is a tuple $tu = (I, U, R, C, T)$ where $I, T \in \mathcal{E}$, U is a set of transformation units over $\mathcal{A}$, $R \subseteq \mathcal{R}$, and $C \subseteq \mathcal{C}$. The components of a transformation unit $tu = (I, U, R, C, T)$ are also denoted by I_{tu}, U_{tu}, R_{tu}, C_{tu}, and T_{tu}, respectively, and the set of all transformation units over $\mathcal{A}$ is denoted by $\mathcal{T}_{\mathcal{A}}$.

[1] Using variables can be interpreted as a shorthand description of the set of all rules with sensibly instantiated labels and therefore as the set of all possible moves.

2. Let $SemRel: U \to 2^{\mathcal{G}\times\mathcal{G}}$ [2]. The *interleaving semantics* of $tu = (I, U, R, C, T)$ w.r.t. *SemRel* is defined by $INTER(tu, SemRel) = SEM(I) \times SEM(T) \cap SEM(C) \cap (\bigcup_{u\in U} SemRel(u) \cup \bigcup_{r\in R} \Rightarrow_r)^*$.

In this paper, we assume that the import structure of transformation units is acyclic. Transformation units with arbitrary import structure are studied in [19].

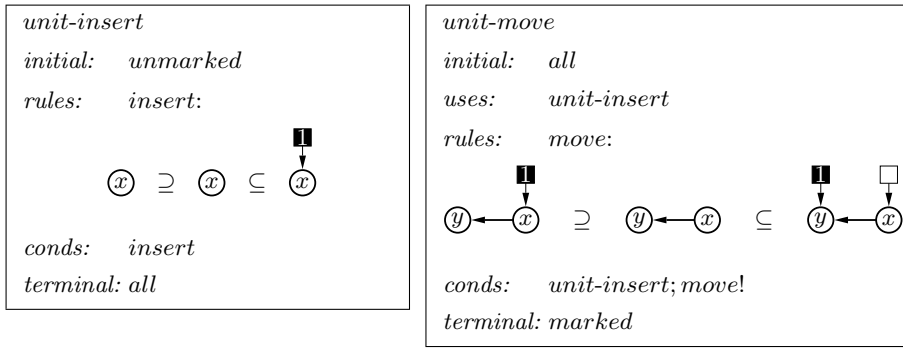


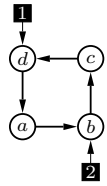
Fig. 2. An example of two transformation units *unit-insert* and *unit-move*

An example of two transformation units is shown in Fig 2. Unit *unit-insert* takes as initial graphs all unmarked graphs. Terminal graphs are all graphs. The unit just inserts a black flag attached to a node labelled with x. Unit *unit-move* imports *unit-insert*. It first inserts the actor flag and then moves it around marking visited places, i.e. nodes. Terminal graphs are those where all nodes of the game board have flags attached.

3 A Concurrent Semantics for Transformation Units

A distributed transformation unit consists of a set of local units and a set of interface units. It transforms distributed graphs which consist of a set of local graphs and a set of interface graphs. In every transformation step local units transform local graphs and interface graphs in such a way that every local graph is either transformed by exactly one local transformation unit or remains unchanged. The interface graphs may be transformed concurrently by various local transformation units. For the transformation of an interface graph local units use the rules and the imported units of the corresponding interface unit. Such concurrent transformations of interface graphs are valid if they belong to the concurrent semantics of the interface units. In this section we present a concurrent semantics of transformation units which is used later on for the definition of the semantics of the interface units of distributed transformation steps.

[2] $2^{\mathcal{G}\times\mathcal{G}}$ denotes the powerset of $\mathcal{G} \times \mathcal{G}$.



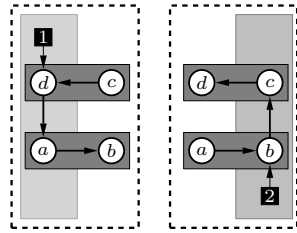


Fig. 3. A game board and a split depicted by different backgrounds

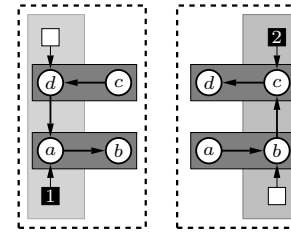


Fig. 4. Result of a concurrent rule application to a split

The concurrent semantics of a transformation unit allows to apply imported transformation units and rules simultaneously to a graph G. To this aim G is splitted into a set of subgraphs. Such a set is called a split of G. The parts where graphs of a split overlap are called boundary graphs. They have a kind of *read-only* permission and cannot be changed, whereas all other parts of a split can be transformed in parallel. The manipulation of boundary graphs is not allowed because their parallel transformation by different rules or imported transformation units may cause conflicts. This is for example the case if one rule or unit deletes a node of a boundary graph whereas another rule or unit preserves the node.

In Fig. 3 an example of a split, here of a game board, with two subgraphs is shown. On the left there is the original graph that is then decomposed into two subgraphs visualized by the dashed borderlines. Components of the split are shown by the different shaded backgrounds. The boundary graph, visualized with the dark background, is not allowed to be modified. It is easy to see that changing graphs of a split exclusively while preserving the boundary prevents us from causing unwanted side effects. The result of two concurrent applications of rule *move*, and a similar rule for 2-labelled nodes, is shown in Fig. 4.

Definition 3 (Split). Let $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be a graph transformation approach. A *split* of a graph $G \in \mathcal{G}$ is a set $\{(G_1 \supseteq I_1), \ldots, (G_m \supseteq I_m)\}$ with $G_i, I_i \in \mathcal{G}$ $(i = 1, \ldots, m)$ such that $G_1 \cup \cdots \cup G_m = G$ and for $i, j = 1, \ldots, m$ with $i \neq j$, $G_i \cap G_j \subseteq I_i \cap I_j$.

In a concurrent transformation step a transformation unit applies rules and imported units to different components of some split such that the boundary graphs are preserved. In Fig. 4 the above introduced rule *move* is applied concurrently to both subgraphs of the split. Because the rule just changes nodes in both subgraphs that are not in the boundary the resulting split can sensibly be rejoined again.

The concurrent semantics of *tu* comprises the iteration of concurrent transformation steps starting from an initial graph of *tu* and ending up in a terminal graph of *tu* such that the control condition is satisfied.

Definition 4 (Concurrent semantics).

1. Let $tu = (I, U, R, C, T)$ be a transformation unit over a graph transformation approach $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$, let $SemRel: U \to 2^{\mathcal{G}\times\mathcal{G}}$ and let $G, G' \in \mathcal{G}$. Then there exists a *concurrent transformation step* from G to G' in tu with respect to $SemRel$, denoted by $G \Rightarrow^{conc}_{tu,SemRel} G'$ if there are a split $s = \{(G_1 \supseteq I_1), \ldots, (G_m \supseteq I_m)\}$ of G and a split $s' = ((G'_1 \supseteq I_1), \ldots, (G'_m \supseteq I_m))$ of G' such that for $i = 1, \ldots, m$, $(G_i, G'_i) \in \bigcup_{u\in U} SemRel(u) \cup \bigcup_{r\in R} \Rightarrow_r$.
2. The concurrent transformation step $G \Rightarrow^{conc}_{tu,SemRel} G'$ is also denoted by $G \Rightarrow^{conc}_{tu} G'$ if $SemRel$ is known from the context. In order to point out which rules and units are applied in the step, we write $G \Rightarrow^{conc}_{B} G'$ where $B = \{b_1, \ldots, b_m\} \subseteq U \cup R$ such that for $i = 1, \ldots, m$, $G_i \Rightarrow_{b_i} G'_i$ if $b_i \in R$ and $(G_i, G'_i) \in SemRel(b_i)$ if $b_i \in U$.
3. The *concurrent semantics* $CONC(tu)$ of tu w.r.t. $SemRel$ is defined as $CONC(tu, SemRel) = (SEM(I) \times SEM(T)) \cap (\Rightarrow^{conc}_{tu,SemRel})^* \cap SEM(C)$.

4 Distributed Transformation Units

A distributed transformation unit mainly consists of a set of local transformation units which are applied concurrently to a distributed graph. A distributed graph consists of a set of local graphs with interfaces. The interfaces represent shared data and are connected to local graphs by boundary graphs. The local graphs represent private data. The local transformation units of a distributed unit are applied concurrently to the local graphs of a distributed graph. In such a distributed transformation step, every local transformation unit may transform also the interface of a local graph. This means that the interface graphs can be rewritten by several transformation units at the same time as long as the boundary graphs are preserved.

4.1 Distributed Graphs

The basic distribution structure of distributed graphs and distributed transformation units is reflected by a bipartite graph connecting a set of local nodes with a set of interface nodes (cf. also [29]).

Definition 5 (Distribution structure).

1. A *distribution structure* is a tuple $NG = (LocV, IntV, Att)$ where $LocV$ is a set of *local vertices*, $IntV$ is a set of *interface vertices* and $Att \subseteq LocV \times IntV$ is a set of edges attaching local vertices to interface vertices. The set $LocV \cup IntV$ is denoted by V_{NG}. The components of NG are denoted by $LocV_{NG}$, $IntV_{NG}$, and Att_{NG}.
2. For every $v \in LocV$ the *distribution structure induced by* v is equal to $ind_{NG}(v)$ with $LocV_{ind_{NG}(v)} = \{v\}$, $IntV_{ind_{NG}(v)}$ is the set of all interface vertices connected to v in NG, and $Att_{ind_{NG}(v)}$ consists of all edges of NG attached to v.

A distributed graph over a distribution structure NG consists of a local graph for every local vertex of NG, an interface graph for every interface vertex, and a boundary graph for every edge in NG. Boundary graphs are graphs that connect local graphs with their interface graph. This means that the boundary graph of every edge e in NG must be a common subgraph of the graphs associated with the source and the target of e.

Definition 6 (Distributed graph).

1. Let $NG = (LocV, IntV, Att)$ be a distribution structure and let $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be a graph transformation approach. A *distributed graph* over $\mathcal{A}$ and NG is a pair of mappings $DG = (graph, boundary)$ with $graph\colon V_{NG} \to \mathcal{G}$, $boundary\colon Att \to \mathcal{G}$ such that $boundary((l,i)) = graph(l) \cap graph(i)$ for every $(l,i) \in Att$ and $graph(l) \cap graph(i) = \emptyset$ if $(l,i) \notin Att$. The set of all distributed graphs over NG and $\mathcal{A}$ is denoted by $DIST(NG, \mathcal{A})$.
2. For every $v \in LocV$ the *induced distributed graph* $ind_{DG}(v)$ is equal to $(graph_v, boundary_v)$ where $graph_v = graph|_{V_{ind_{DG}(v)}}$ and $boundary_v$ is equal to $boundary|_{Att_{ind_{DG}(v)}}$.

Fig. 5 shows a distributed graph. The distribution structure is symbolized on top of the shaded parts. The outer vertices are local vertices and the inner vertex is an interface vertex. The vertices of the distribution structure get assign the graphs beneath. Accordingly, the left and the right graph are local graphs and the graph in between is the interface graph. The actors of this simple game have a representation of the game board in there local graphs indicating by the edges the moves that are eventually possible and by white flags which places they already visited. Up to now they did not visit any place and therfore there are no white flags in the local graphs. If a move is possible also depends on the opponent who may occupy and therefore block a place to go. The complete information which places are occupied at the moment is stored in the interface graph. White flags symbolize free places, black flags occupied ones. Information on possible routes to take, i.e. edges, are not part of the interface graph. Therefore, the boundary graph consists of the places of game board (darkest background) only.

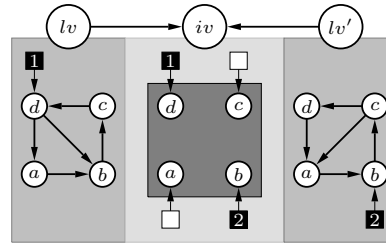


Fig. 5. A distributed graph

In a distributed transformation step, every local graph together with its interfaces is transformed by a (local) transformation unit such that the boundaries

are preserved. The manipulation of boundaries may produce dangling edges in the interface and in the local graphs. Therefore it is forbidden.

The local parts of the distributed graph are the main sources of information for both players. They are their own representation of knowledge about the state of the game board. Both only know where they are located. The interface represents a global view to the board that must be updated if something changes locally. This is not in the scope of ordinary rules or transformation units because we need some kind of synchronisation.

4.2 Distributed Transformation Units

A distributed transformation unit *dtu* is defined over a distribution structure NG and transforms distributed graphs over NG. More precisely, it contains a local transformation unit for every local vertex in NG and an interface unit for every interface vertex in NG. Hence, for any distributed graph DG over NG, every local vertex of NG relates a local transformation unit of *dtu* with a local graph of DG and an interface unit of *dtu* with an interface graph of DG. Interface units are transformation units as in Definition 2. Local transformation units transform local graphs and interfaces. For the transformation of a local graph, they employ local (private) graph transformation rules and imported transformation units whereas for the transformation of interface graphs they make use of the rules and imported units of the corresponding interface units. For this purpose, every local transformation unit *ltu* provides a set of synchronization rules that determine which rules or imported units of *ltu* can be applied in parallel with rules or imported units of interface transformation units. Hence, synchronization rules couple rules or imported units of local transformation units with rules or imported units of interface transformation units[3]. More precisely, a synchronization rule assigns rules or a transformation units to the vertices of the distribution structure induced by some local vertex of NG. This is done in such a way that the rules or transformation units assigned to the interface vertices be part of the corresponding interface units. The rules or units assigned to the local vertex can be arbitrary.

In addition to synchronization rules, a local transformation unit provides an initial graph class expression, a local control condition, and a terminal graph class expression. The local control condition prescribes, roughly speaking, in which order the synchronization rules should be executed[4]. Moreover, every local unit *ltu* induces a core unit which only takes into account the initial and the terminal graph class expression of *ltu*, and the rules and transformation units assigned to local nodes by the synchronization rules.

[3] The synchronization rules presented here are different from those in [23] which in general do not require parallel independence of rule applications.

[4] In general, one can take as control condition for a local network graph the class of control conditions of the underlying graph transformation approach (see also [20]) but with a different semantics. This semantics has to allow more transformations of interface graphs than those specified in the synchronization rules, because other local units also have access to the interface.

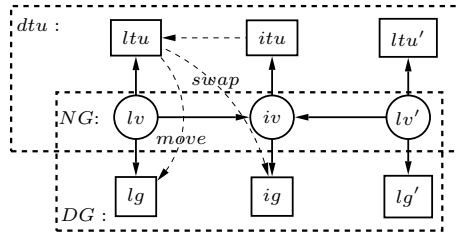
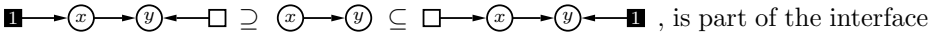


Fig. 6. A schema of a distributed transformation unit

In Fig.6 a schematic view of a distributed graph DG over a distribution structure NG and a corresponding distributed transformation unit dtu over NG is shown. The local transformation unit ltu applies two rules synchronously. Rule $move$ is part of the local transformaton unit ltu, which is the unit $unit\text{-}move$ in the example, and is applied to the local graph lg. Rule $swap$, given by $swap$:  , is part of the interface unit itu and is applied to the interface. According to our running example this synchonisation rule enable us to move the flag in the local part and swap the state of the nodes in the interface graph at the same time.

Definition 7 (Distributed transformation unit).

1. Let $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$ be a graph transformation approach and let $NG = (LocV, IntV, Att)$ be a distribution structure. A *distributed transformation unit* over $\mathcal{A}$ and NG is a mapping dtu with $dtu(v) \in \mathcal{T}_{\mathcal{A}}$ for every $v \in IntV$, and for every $v \in LocV$, $dtu(v) = (I, S, C, T)$ where $I, T \in \mathcal{E}$, S is a set of *synchronization rules over* $ind_{NG}(v)$, i.e. a set of partial mappings $s\colon V_{ind_{NG}(v)} \rightharpoonup \mathcal{R} \cup \mathcal{T}_{\mathcal{A}}$ such that for every $i \in IntV_{ind_{NG}(v)}$ and every $s \in S$, $s(i) \in U_{dtu(i)} \cup R_{dtu(i)}$, and C is a control condition specifying a binary relation $SEM(C) \subseteq DIST(ind_{NG}(v), \mathcal{A}) \times DIST(ind_{NG}(v), \mathcal{A})$. The components of $dtu(v)$ are also denoted by $I_{dtu(v)}$, $S_{dtu(v)}$, $C_{dtu(v)}$, and $T_{dtu(v)}$.
2. For every $v \in V_{NG}$, $dtu(v)$ is called an *interface unit* of dtu if $v \in IntV$; otherwise it is called a *local transformation unit.*
3. For every $v \in LocV$ the *induced distributed transformation unit w.r.t. dtu* is defined by $ind_{dtu}(v) = dtu|V_{ind_{NG}(v)}$. Every local unit $dtu(v)$ $(v \in LocV)$ induces a *core unit* $core(dtu(v)) \in \mathcal{T}_{\mathcal{A}}$ defined by $(I_{dtu(v)}, U, R, true, T_{dtu(v)})$ with $U = \{s(v) \mid s \in S_{dtu(v)}\} \cap \mathcal{T}_{\mathcal{A}}$, $R = \{s(v) \mid s \in S_{dtu(v)}\} \cap \mathcal{R}$, and $SEM(true) = \mathcal{G} \times \mathcal{G}$.

In the following, we assume that every imported unit u of an interface or core unit specifies a binary relation $SemRel(u)$ on the underlying graph class, denoted by $\Rightarrow_u$. This could be the interleaving or the concurrent semantics or any other binary relation on graphs.

4.3 Distributed Transformation Steps

A distributed transformation unit dtu over a distribution structure NG transforms distributed graphs over NG. More precisely, a distributed graph trans-

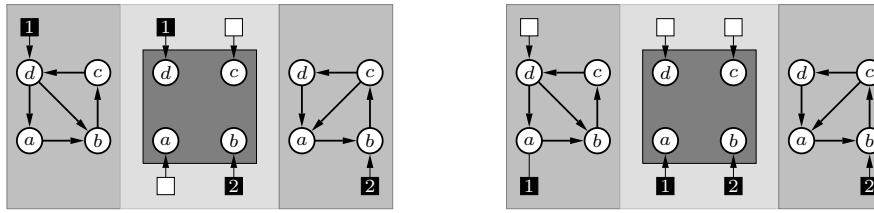


Fig. 7. Synchronization rules force synchronous rule applications to local and interface graphs

formation is a sequence of distributed graph transformaiton steps. Given a distributed graph DG over NG, a distributed graph transformation step from DG consists of the parallel transformation of the local graphs of DG together with their interfaces. This means that every local unit *ltu* of *dtu* may transform the distributed graph induced by the local graph related to *ltu* via the distribution structure NG. The transformations are performed according to the synchronization rules of the local units. In the running example (cf. Fig.7) both actors try to move their flag, i.e. to apply the rule move. Here synchronization is needed. Not only the black flags in the local parts have to be moved but also at the same time the flags in the interface. This is the main purpose of synchronisation rules. Note that because of the white flags the rule *swap* never yields a result where two black flags coincide if we start form a proper graph. This must be granted by initial and terminal graph class expressions.

Definition 8 (Distributed transformation step). Let *dtu* be a distributed transformation unit over $NG = (LocV, IntV, Att)$ and $\mathcal{A} = (\mathcal{G}, \subseteq, \cup, \mathcal{R}, \Rightarrow, \mathcal{C}, \mathcal{E})$. Let $\{l_1, \ldots, l_n\} \subseteq LocV$ and for $i = 1, \ldots, n$, let $s_i \in S_{dtu(l_i)}$. Let $DG, DG' \in DIST(NG, \mathcal{A})$. Then there is a *distributed transformation step* $DG \Rightarrow^{dist}_{dtu} DG'$ via $s_1, \ldots, s_n$ if for all $e \in Att$, $boundary_{DG}(e) = boundary_{DG'}(e)$ and if the following holds.

- For $i = 1, \ldots, n$, $graph_{DG}(l_i) \Rightarrow_{s_i(l_i)} graph_{DG'}(l_i)$.
- For every $v \in IntV_{ind_{NG}(l_i)}$ $(i = 1, \ldots, n)$, $graph_{DG}(v) \Rightarrow^{conc}_{M(v)} graph_{DG'}(v)$ with $M(v) = \{s(v) \mid s \in defined(v)\}$ where $defined(v) = \{s \in \{s_1, \ldots, s_n\} \mid v \in DOM(s)\}$ [5].
- For every $v \in V_{NG} - \cup_{i=1,\ldots,n} DOM(s_i)$, $graph_{DG}(v) = graph_{DG'}(v)$.

An open distributed transformation step from DG to DG' differs from a distributed transformation step in the transformation of the interface graphs, because not only transformations prescribed by synchronization rules can be applied to an interface graph but also any other rule or unit contained in the interface unit related to that interface graph.

Definition 9 (Open distributed transformation step). Let *dtu* be a distributed transformation unit over $NG = (LocV, IntV, Att)$ and an approach $\mathcal{A}$.

[5] $DOM(s_i)$ denotes the domain of s_i.

Let $\{l_1, \ldots, l_n\} \subseteq LocV$ and for $i = 1, \ldots, n$, let $s_i \in S_{dtu(l_i)}$. Let $DG, DG' \in DIST(NG, \mathcal{A})$. Then there is an *open distributed transformation step* $DG \Rightarrow^{open}_{dtu} DG'$ via $s_1, \ldots, s_n$ if for all $e \in Att$, $boundary_{DG}(e) = boundary_{DG'}(e)$ and if the following holds.

- For $i = 1, \ldots, n$, $graph_{DG}(l_i) \Rightarrow_{s_i(l_i)} graph_{DG'}(l_i)$.
- For every $v \in LocV - \{l_1, \ldots, l_n\}$, $graph_{DG}(v) = graph_{DG'}(v)$.
- For every $v \in IntV - \cup_{i=1,\ldots,n} DOM(s_i)$, $graph_{DG}(v) \Rightarrow^{conc}_{dtu(v)} graph_{DG'}(v)$.
- For every $v \in IntV_{ind_{NG}(l_i)}$ $(i = 1, \ldots, n)$, $graph_{DG} \Rightarrow^{conc}_{M(v)_+} graph_{DG'}(v)$ where $M(v)_+$ is some subset of $U_{dtu(v)} \cup R_{dtu(v)}$ such that $M(v) \subseteq M(v)_+$.

4.4 Distributed Semantics

Now we can define the distributed and the open distributed semantics of distributed transformation units based on its (open) distributed transformation steps. Every pair (DG, DG') of distributed graphs over NG is in the distributed semantics of a distributed transformation unit *dtu* over NG, if DG is an initial distributed graph of *dtu*, DG' is a terminal distributed graph of *dtu*, DG' is obtained from DG by a sequence of distributed transformation steps such that the control conditions of the distributed unit are satisfied. A distributed graph DG is an initial graph of *dtu* if for every vertex v in NG, the graph assigned to v is in the semantics of the initial graph class expression of the unit assigned to v. Analogously, DG is a terminal graph of *dtu* if for every vertex v in NG, the graph assigned to v is specified by the initial graph class expression of the unit assigned to v. The open distributed semantics of a distributed transformation unit allows additional applications of rules and units of the interface transformation units to interface graphs

Definition 10 (Distributed semantics). Let *dtu* be a distributed transformation unit over $NG = (LocV, IntV, Att)$ and $\mathcal{A}$. Let $DG, DG' \in DIST(NG, \mathcal{A})$ such that the following holds.

- For all $v \in V_{NG}$, $(graph_{DG}(v), graph_{DG'}(v)) \in SEM(I_{dtu(v)}) \times SEM(T_{dtu(v)})$.
- For each $v \in IntV$, $(graph_{DG}(v), graph_{DG'}(v)) \in SEM(C_{dtu(v)})$.
- For each $v \in LocV$, $(ind_{DG}(v), ind_{DG'}(v)) \in SEM(C_{dtu(v)})$.

Then (DG, DG') is in the *distributed semantics* $DIST(dtu)$ of *dtu* if $(DG, DG') \in (\Rightarrow^{dist}_{dtu})^*$, and (DG, DG') is in the *open semantics* $OPEN(dtu)$ of *dtu* if $(DG, DG') \in (\Rightarrow^{open}_{dtu})^*$.

It can be easily shown that the distributed semantics of a distributed transformation unit is a subset of its open semantics. Moreover, the semantics of a distributed transformation unit *dtu* is based on the concurrent semantics of its interface units and the predefined semantics of the core units. This is stated in the next observation.

Observation 1 Let *dtu* be a distributed transformation unit over a distribution structure $NG = (LocV, IntV, Att)$. Let $(DG, DG') \in OPEN(dtu)$. Then

for every $v \in IntV$, $(graph_{DG}(v), graph_{DG'}(v)) \in CONC(dtu(v), SemRel)$, and for all $v \in LocV$, $(graph_{DG}(v), graph_{DG'}(v)) \in INTER(core(dtu(v)), SemRel)$ where $SemRel(u) = \Rightarrow_u$ for every $u \in U_{dtu(v)}$

In a distributed transformation step from DG to DG' every local component of DG is transformed according to an open distributed transformation step. As mentioned before, for a distributed graph DG over NG and a distributed transformation unit dtu over NG, we can associate with every local node v of NG the distributed graph $ind_{DG}(v)$ and the distributed units $ind_{dtu}(v)$. The next observation states that the distributed semantics of dtu restricted to $ind_{DG}(v)$ is contained in the open distributed semantics of $ind_{dtu}(v)$. Moreover, the distributed semantics of $ind_{dtu}(v)$ is a subset of the distributed semantics of of dtu restricted to $ind_{DG}(v)$. Finally, the open semantics of dtu restricted to $ind_{DG}(v)$ is contained in the open semantics of $ind_{dtu}(v)$.

Observation 2 Let dtu be a distributed transformation unit over a distribution structure $NG = (LocV, IntV, Att)$ and $\mathcal{A}$ and let $DG, DG' \in DIST(NG, \mathcal{A})$. Then for every $v \in LocV$,

$$DIST(ind_{dtu}(v)) \subseteq DIST(dtu)|ind_{DG}(v) \subseteq OPEN(dtu)|ind_{DG}(v) \subseteq OPEN(ind_{dtu}(v)).$$[6]

5 Related Work

This section provides a brief overview of different approaches to distributed and concurrent systems in the field of graph transformation. In [8], concurrency, parallelism, and distribution aspects are modelled by graph transformation. In particular, [22] presents how distributed algorithms can be implemented with graph relabeling systems. In [14] actor grammars and ESM systems are presented. Actor grammars model actor systems and ESM systems are a generalization of actor grammars. Actor systems are systems of inherently distributed actors concurrently interacting by exchanging asynchronous messages. In [2] various concurrent semantics for the double pushout approach are reviewed and compared. In [23] graph transformation is applied to model concurrent constraint programs, distributed systems with synchronization, and the π-calculus. In [29] distributed graphs can be transformed globally or locally by graph transformation rules. A distributed graph consists of a network graph the nodes of which are equipped with local graphs. This work bases on [28] where the main concepts are introduced. One difference between network graphs and our distribution structure is that the edges are labelled with graph morphisms. A concept of Open Graph Transformation Systems is given in [12]. Here open systems are specified

[6] For a relation $Rel \subseteq DIST(NG, \mathcal{A}) \times DIST(NG, \mathcal{A})$ and $v \in LocV$, $Rel|ind_{DG}(v)$ consists of all pairs $(dg_1, dg_2) \in DIST(ind_{DG}(v), \mathcal{A})$ such that $dg_i = ind_{DG_i}(v)$ $(i = 1, 2)$ for some $(DG_1, DG_2) \in Rel$.

by a loose semantics allowing to model reactivity. In [26] it is shown how graph transformation can be used to describe Petri Nets, statecharts, parallel logic programming, and systems of processes. Concerning interface graphs our open semantics defines a comparable system behaviour. In [15] graph multiset transformation is presented which is inspired by the concepts of genetic algorithms and DNA computing and is closely related to our concurrent semantics of transformation units. Finally, transformation of structured graphs is considered in e.g. [5,9].

6 Conclusion

In the presented approach to distributed graph transformation distributed graphs and distributed transformation systems are represented as labelled bipartite graphs consisting of local vertices and interface vertices. In the first case the local vertices are labelled with local graphs and the interface vertices with interface graphs; the edges are labelled with boundary graphs, i.e. common subgraphs of interfaces and local graphs that cannot be transformed. In the second case interface vertices are equipped with transformation units and local vertices with tuples consisting of an initial graph class expression, a terminal graph class expression, a set of synchronization rules, and a control condition. Semantically, a distributed transformation unit transforms distributed graphs. We have defined an open and a closed semantics of distributed transformation units which is based on the concurrent semantics of the interface units and the interleaving semantics of the units induced by the tuples assigned to the local vertices of the underlying network graph. In contrast to the standard semantics, the open semantics allows transformations of interface graphs not specified in the synchronization rules of the distributed units.

There remain at least the following further topics to be studied.

- There is a lot of work to be done in future concerning related work in the vast field of distributed systems. Especially the concept of distributed transformation units should be compared with other existing approaches to distributed graph transformation (cf. [8]).
- This approach constitutes a nice means to model multiagent systems. In a next step properties of multiagent systems and other features important to artifical intelligence should be investigated on this formal basis.
- Concrete control conditions for the local transformation units should be defined. As stated before, these could mainly be the same as for transformation units but with a more general semantics.
- In order to be practically applicable, an implementation of distributed transformation units is needed. (For the non-distributed case there exists a very first prototype [11].)
- By extending the network graphs with graph transformation rules one obtains a means to modify the net structure of distributed systems and distributed graphs (cf. [28]). This allows, for example in the case of agent systems, to specify the termination or generation of agents at runtime or to modify the structure of the graph they act on.

References

1. 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.
2. Paolo Baldan, Andrea Corradini, Hartmut Ehrig, Michael Löwe, Ugo Montanari, and Francesca Rossi. Concurrent semantics of algebraic graph transformations. In Ehrig et al. [8], pages 107–185.
3. Giorgio Busatto, Gregor Engels, Katharina Mehner, and Annika Wagner. A framework for adding packages to graph transformation systems. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 352–367, 2000.
4. 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 [25].
5. Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 2002. To appear; short version in Proc. *FOSSACS 2000*, LNCS 1784.
6. Hartmut Ehrig, Paul Boehm, Udo Hummert, and Michael Löwe. Distributed parallelism of graph transformations. In *Proc. Graph-Theoretic Concepts in Conmputer Science*, volume 314 of *lncs*, pages 1–19, 1988.
7. 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, Singapore, 1999.
8. 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, Singapore, 1999.
9. Gregor Engels and Reiko Heckel. Graph transformation as unifying formal framework for system modeling and model evolution. In *Proc. ICAPL 2000*, 2000.
10. Gregor Engels and Andy Schuerr. Encapsulated hierachical graphs, graph types, and meta types. In *SEGRAGRA'95*, volume 2 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1995.
11. Martin Faust. The GRACEland web page. URL: `http://www.informatik.uni-bremen.de/theorie/GRACEland`, 1998.
12. Reiko Heckel. *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems.* PhD thesis, TU Berlin, 1998.
13. Reiko Heckel, Gregor Engels, Hartmut Ehrig, and Gabriele Taentzer. Classification and comparison of module concepts for graph transformation systems. In Ehrig et al. [7], pages 639–689.
14. Dirk Janssens. Actor grammars and local actions. In Ehrig et al. [8], pages 57–106.
15. Hans-Jörg Kreowski. A sight-seeing tour of the computational landscape of graph transformation. In Wilfried Brauer, Hartmut Ehrig, Juhani Karhumäki, and Arto Salomaa, editors, *Formal and Natural Computing. Essays Dedicated to Grzegorz Rozenberg*, volume 2300 of *LNCS*, pages 119–137. Springer, 2002.

16. 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*. Elsevier, 2002. 13 pages.
17. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units and modules. In Ehrig et al. [7], pages 607–638.
18. Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.
19. 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.
20. Sabine Kuske. More about control conditions for transformation units. In Hartmut Ehrig, Gregor Engels, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. Theory and Application of Graph Transformations*, volume 1764 of *LNCS*, pages 323–337, 2000.
21. Sabine Kuske. *Transformation Units—A structuring Principle for Graph Transformation Systems*. PhD thesis, University of Bremen, 2000.
22. Igor Litovski, Yves Métivier, and Éric Sopena. Graph relabelling systems and distributed algorithms. In Ehrig et al. [8], pages 1–56.
23. Ugo Montanari, Marco Pistore, and Francesca Rossi. Modeling concurrent, mobile and coordinated systems via graph transformations. In Ehrig et al. [8], pages 189–268.
24. Terrence W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In Volker Claus, Hartmut Ehrig, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science and Biology*, volume 73 of *LNCS*, pages 389–400, 1979.
25. Grzegorz Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific, Singapore, 1997.
26. H.-J. Schneider. Describing systems of processes by means of high-level replacement. In Ehrig et al. [8], pages 401–450.
27. Andy Schürr. Programmed graph transformations and graph transformation units in GRACE. In Janice E. Cuny, Hartmut Ehrig, Gregor Engels, and Grzegorz Rozenberg, editors, *Proc. Graph Grammars and Their Application to Computer Science*, volume 1073 of *LNCS*, pages 122–136, 1996.
28. Gabriele Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, Shaker Verlag, 1996.
29. Gabriele Taentzer, Manuel Koch, Ingrid Fischer, and Victor Volle. Distributed graph transformation with application to visual design of distributed systems. In Ehrig et al. [8], pages 269–340.

Describing Policies with Graph Constraints and Rules

Manuel Koch[1] and Francesco Parisi-Presicce[2]

[1] Freie Universität Berlin, Germany
mkoch@inf.fu-berlin.de
[2] Univ. di Roma La Sapienza, Italy
parisi@dsi.uniroma1.it

Abstract. Policies are used to control the behavior of complex systems. In order to support reasoning about the behavior, it is necessary to have a precise specification of the policy, better if described in an intuitive visual formalism. Graphical constraints and graph transformation rules provide such a combination of a natural yet semantically sound formalization. Furthermore, the evolution of policies, the combination of policies and the stepwise development of policies can be given a solid foundation and their analysis and verification can benefit from the use of existing general graph transformation engines.

1 Introduction

Policies are used to control the behavior of complex systems by a set of rules that define the choices in the individual and collective behavior of the entities in the system. The separation of the policy from the system implementation allows dynamic policy modifications to obtain a new system behavior without any re-implementation [16]. The use of a formal language to specify policies is a prerequisite for an effective analysis of conflicts within and between policies and their resolution to achieve consistency.

We propose a uniform and precise framework based on graph transformations [14] for the specification and uniform treatment of policies. Graph transformations provide an intuitive visual description for the manipulation of graph structures as they occur in policy-based systems. The proposed framework allows the specification of policy rules, of constraints for the prohibition (unwanted parts of system states) and of constraints for the obligation (required parts of system states) of system states. The constructive rules of the framework build the acceptable system states, the declarative constraints provide additional useful information when developing the policy through successive refinement steps.

The intention of the framework is to help in the design of a customized policy by providing an intuitive visual representation along with a formal theory to develop and modify a policy in a systematic way. The framework supports policy evolution that reacts to changing requirements, policy composition for constructing complex policies from basic policies, and policy refinement of abstract policies to more concrete policies. The composition of policies considers a

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 223–238, 2002.
© Springer-Verlag Berlin Heidelberg 2002

black box approach in which the policy rules and constraints of the basic policies are unknown and clear box approaches in which the policy specifications are known.

We have used the framework for the specification of security policies for access control [10,9], but the framework is not restricted to access control. In this paper, we use as an example a specification for a workflow policy taken from [2].

Section 2 fixes the notation of the graph transformation approach used in the paper; section 3 defines the formal framework to specify policies; section 4 is concerned with obligation policies. Section 5 contains the concepts for policy composition and section 6 for policy refinement. Section 7 contains a summary and points to future work.

2 Notation

This section briefly introduces the notation used for graph transformations.

The node and edge types are specified in a *type graph*. A type graph is a graph, where each of its nodes represents a node type and each of its edges an edge type. A type graph is a pattern for a class of graphs. A graph G is called *typed* into a type graph TG, if there is a total morphism $t : G \to TG$ and if $t_N(n) = t_N(n')$, then n and n' are nodes of the same type; similarly for edges. Nodes and edges are labeled by partial labeling functions $l_N : N \to LS$ and $l_E : E \to LS$ with labels taken from a set LS.

All graph morphisms must preserve the type, but not necessarily the label of a graph.

A rule is an injective (partial) morphism $r : L \to R$. A *negative application condition* (NAC) for a rule $r : L \to R$ consists of a set $A(r)$ of pairs (L, N), where L is a subgraph of N. The part $N \setminus L$ represents a structure that must not occur in a graph G for the rule to be applicable. In the figures, we represent (L, N) with N, where the subgraph L is drawn with solid and $N \setminus L$ with dashed lines.

A rule $r : L \to R$ with a NAC $A(r)$ is applicable to a graph G if L occurs in G via a total morphism $m : L \to G$ and for each (L, N) in $A(r)$ it is not possible to extend m to a morphism $N \to G$. We use the single pushout approach [14] for the application of a rule at a given match.

A (positive or negative) *graphical constraint*, or just constraint, is given by a total morphism $c : X \to Y$. A total morphism $p : X \to G$ *satisfies* a positive (negative) constraint c if there exists (does not exist) a total morphism $q : Y \to G$ so that $c \circ q = p$. A graph G *satisfies* a constraint c if each total morphism $p : X \to G$ satisfies c, i.e., G satisfies a positive constraint if each occurrence of X in G can be extended to Y, while G satisfies a negative constraint c if each occurrence of X in G cannot be extended to Y. In the remainder of this paper, we depict only the forbidden graph X in the figures for the special case of negative constraints in which $c : X \to X$ is the identity.

3 Policy Framework

A *policy framework* is a conceptual framework for the specification of access control models based on graph transformations. A policy framework specifies an access control model by 1) a type graph that provides the type information of the entities of the policy, 2) a set of rules that specify the policy rules and build the accepted system states, 3) a set of positive graphical constraints for the wanted substates and 4) a set of negative graphical constraints to specify unwanted substates. Graphical constraints (negative and positive) can express the constraints in [15] as well as the dynamic constraints in [2]. In any implementation of an access control policy, the constraints of the policy framework are not needed since the only acceptable states are those explicitly built by the implemented rules. The declarative constraints, however, provide useful information when developing an access control policy through successive refinement steps, or when trying to predict the behavior of a policy.

We present next the policy framework for a simplified version of the workflow example introduced in [2]. The workflow represents a tax refund process and consists of four tasks to be executed sequentially:

- task $T1$: A clerk prepares a check for a tax refund.
- task $T2$: A manager can approve or disapprove the check. This task must be performed by two different managers.
- task $T3$: The decisions of the managers are collected and the final decision is made. The manager who collects the results must be different from those executing task $T2$. Her/his decision is a consequence of the outcome of task $T2$, i.e., (s)he does not decide about the tax refund.
- task $T4$: A clerk issues (if both managers approved) or voids (if one manager disapproved) the check based on the result of task $T3$. The clerk who issues or voids the check must be different from the clerk who prepared the check.

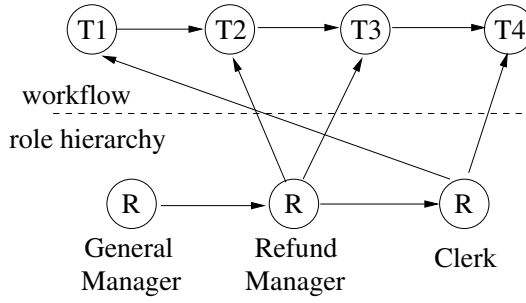


Fig. 1. Workflow, role hierarchy and role authorization for tasks.

Figure 1 shows the specification of the workflow by a graph. Each task is associated with one ore more roles, which are the only ones authorized to execute

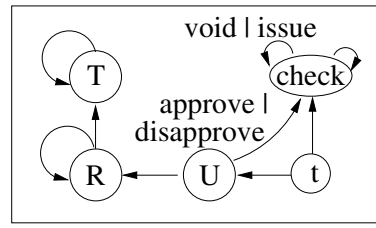


Fig. 2. The type graph for the workflow policy.

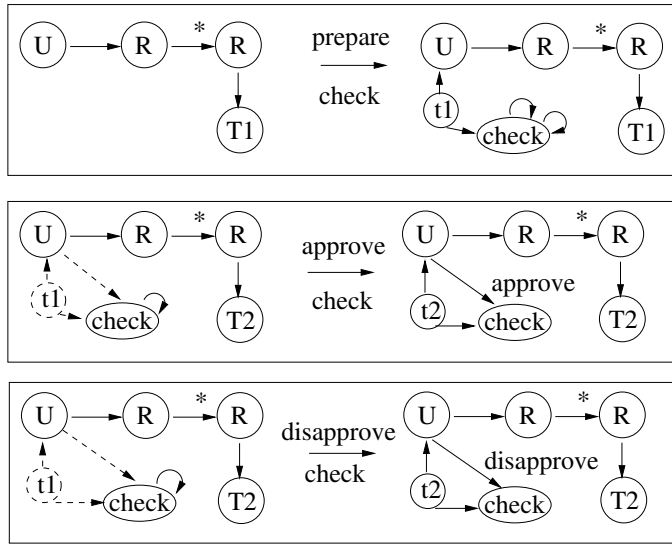


Fig. 3. Task 1 and Task 2.

a task. The assignment of roles to the tasks they can execute is specified by edges from the role to the task. Roles may be related in a role hierarchy which is represented by a role graph as shown in figure 1. An edge from a role R_i to a role R_j specifies that R_i dominates R_j in the sense that any user in role R_i may execute all tasks to which R_j is associated. For example, the role *General Manager* can execute all tasks, the role *Clerk* can only execute task $T1$ and $T4$.

The type graph for the workflow policy framework in figure 2 contains the nodes T for tasks and R for roles. The edge from the R node to the T node represents the assignment edge for roles to tasks. Users are represented by nodes of type U, a check by a node of type *check*. There can be an unlabeled loop at a check node (used for counting the decisions made for a check) or a loop labeled *void* or *issue* (used to indicate whether the check is voided or issued). The edge between user nodes U and role nodes R represents the assignment of a user to a role. Nodes of type t connecting a user node with a check node specify the tasks executed on the check by a user. An edge from the user node U to the

check node may be labeled by *approve* or *disapprove* specifying whether a user approved or disapproved a check.

The rules of the policy framework for the workflow access control specification are shown in figures 3,4 and 5. The rule *prepare check* specifies task $T1$, in which a new check is created. Any user who is playing a role which dominates a role associated to task $T1$ may create the check. A $*$-labeled edge between nodes A and B represents a (possibly empty) path from A to B. Since task $T1$ is associated to the role *Clerk*, checks can be created by users in any role. A node $t1$ is added between the user and the *check* node to indicate that the user has executed task $T1$ on the check. The *check* node has two loops since two managers have to approve or disapprove the check in task $T2$.

The rules *approve check* and *disapprove check* model the decision of a manager to approve or disapprove a check, respectively. The rules can be applied only to users who play a role which dominates a role associated to task $T2$. According to figure 1, these roles are *General Manager* and *Refund Manager*. If a manager approves a check, an edge with label *approve* between the user and the check node is introduced, while if the manager disapproves the check, the edge is labeled with *disapprove*. The node $t2$ between the user and the check node identifies the user that has executed task $T2$ on this check. A loop from the check node is deleted, since a decision for this check was made. If there is no loop attached to a check, the rules are not applicable anymore. The NAC prevents both users from deciding twice for the same check and users who executed task $T1$ from deciding for the check.

The rules *decide issue* and *decide void* specify the final decision of a manager based on the decisions in task $T2$. Both rules can be applied only to users who play a role dominating a role associated to task $T3$. These roles are *General Manager* and *Refund Manager*. The rules require two users who have decided about the check. If the variable X can be substituted with *approve*, the check is approved and the check node gets an *issue*- labeled loop. If one manager disapproved (i.e., there is an edge from a user labeled *disapprove* to the check), the check is voided and the loop at the check node carries the label *void*. The NAC prevents a user who has executed task $T2$ on this check from making the final decision.

The rules *issue* and *void* specify the issuing of the check (if the check node has a loop labeled *issue*) or the voiding of the check (if the check node has a loop with label *void*). In both cases, the node $t4$ indicates that the user has executed task $T4$ on the check. To prevent the user who executed task $T1$ on this check from executing this task on the check, a NAC is added. The rule can be applied to all users in roles which dominate a role associated to $T4$. For both rules, the color of the *check* node is changed to black to indicate the end of the workflow for this check. If the tax refund is finished, the check node is deleted by the rule *end tax refund*. All connected nodes *ti* are deleted as well. The double-circled node t_i is a set node which matches all nodes t_i in a graph.

In addition to the rules of the policy framework, the graphical constraints can be specified. In [2], Bertino et al. give examples of (mainly dynamic) constraints.

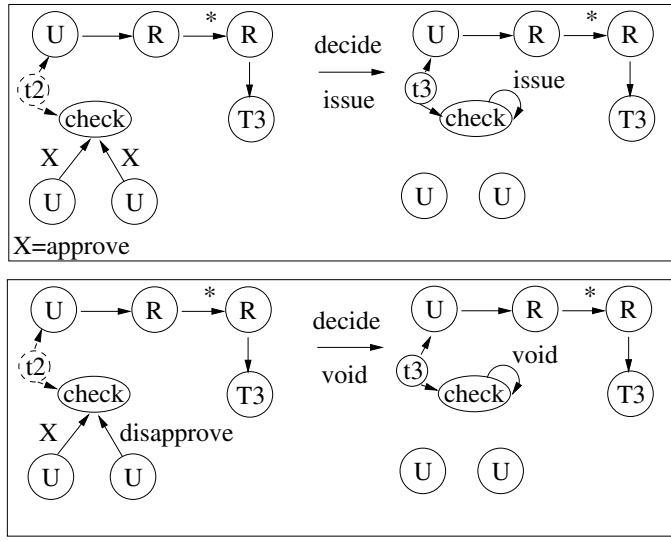


Fig. 4. Task 3.

We choose three to express them by graphical constraints. The remaining constraints in [2], however, can be expressed, too.

C1 Task $T2$ must be executed by a role dominating the roles that execute tasks $T1$ and $T4$, unless $T1$, $T2$ and $T4$ are executed by the role *General Manager*.
C2 If a user belongs to role *Clerk* and has performed task $T1$ then (s)he cannot perform task $T4$.
C3 If a user has performed task $T2$ then (s)he cannot perform $T3$.

Figure 6 shows the graphical constraints for C1, C2 and C3. The constraint for C1 is positive, the other two are negative constraints. The graphical constraint for C1 requires that if a user in role *Refund Manager* has executed task $T2$ on a check, then the role *Refund Manager* dominates the roles of the users who executed $T1$ and $T4$. This is specified by the two required +-labeled edges (i.e., a nonempty path) from the *Refund Manager* role to the other two roles, since a nonempty path from role R to role R' models the domination of R' by R. This is a graphical constraint only for the role *Refund Manager*, since the role *General Manager* may execute $T1, T2$ and $T4$ and the role *Clerk* cannot execute $T2$. The negative graphical constraint for C2 models the requirement that no user who plays the role *clerk* has executed $T1$ and $T4$. The graphical constraint for C3 forbids a graph where a user has executed both $T2$ and $T3$.

A policy framework has therefore 3 kinds of constraints: a static constraint in the form of the type graph, preconditions represented by the left hand side of the rules, and invariants represented by positive and negative graphical constraints.

A security policy framework is *coherent* if the policy rules create only graphs which satisfy both the negative and the positive constraints. Coherence of a pol-

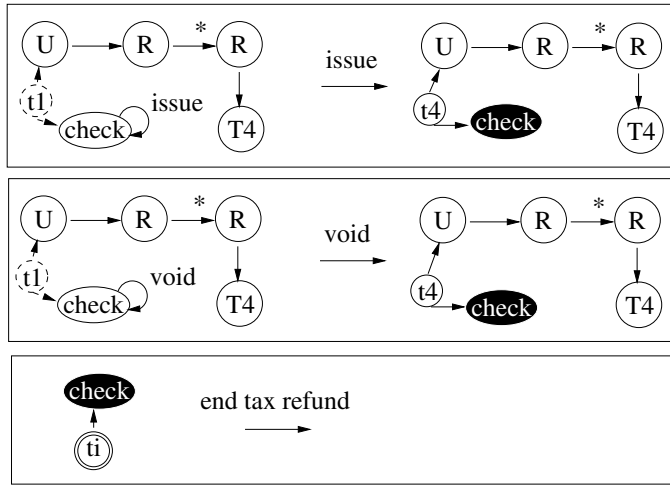


Fig. 5. Task 4.

icy framework is not generally given, but the graph rules which may construct graphs that do not satisfy constraints can be statically detected and automatically modified so that coherence is ensured [12].

4 Obligation Policies

The policy framework defined in the previous section describes an access control (or other kind of) policy by specifying when a rule r MAY be applied (matching in the state graph of the left hand side of a rule r satisfying the NAC) and WHAT is the effect of applying such a rule. The decision of WHEN to apply a particular rule to change the state is external to the framework. This is adequate for access control policies, but not sufficient for more general security management policies, where it is necessary to be able to specify that a system MUST change according to certain rules, when a particular condition is satisfied.

Obligation policies specify the actions that must be performed by managers within the system when certain events occur [5]. From the viewpoint of security management, obligation policies specify the actions that must be performed by a manager when security violations occur.

As an example, we consider the system login of a user. If the same user enters two times in a row the incorrect password, this may be a system attack and the system administrator logs the user in a special security log-file. We assume a possible system attack already in the case of two failed attempts, but any other number could be chosen as well. The graph rules for login are shown in Figure 7. A black user node represents a user who is not yet logged into the system, a white user node represents a user logged in. The rule *login* specifies the entering of the password (represented by the Pw node attached to the user node). If the

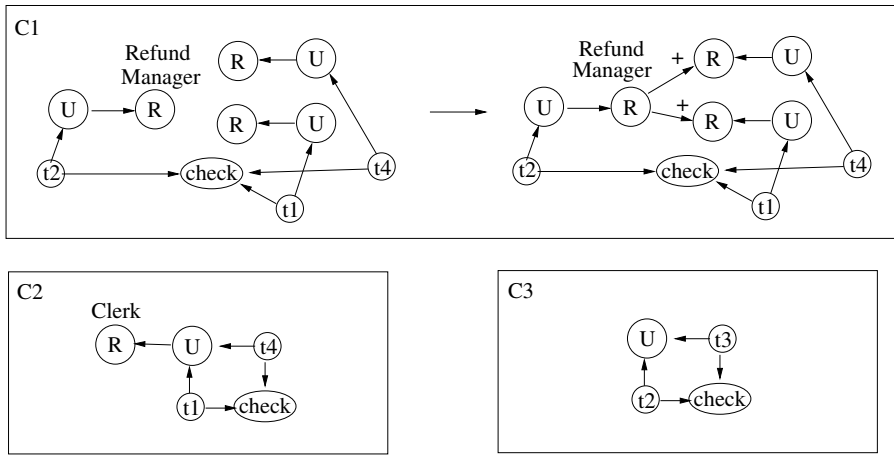


Fig. 6. Graphical constraints for C1,C2 and C3.

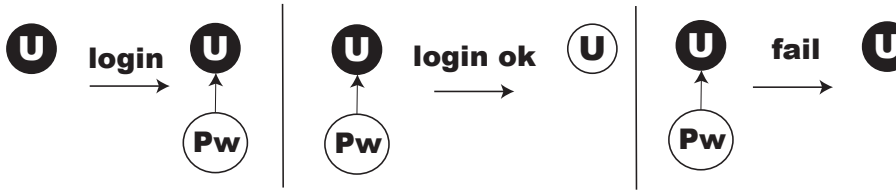


Fig. 7. Rules for user login.

correct password is entered, the rule *login ok* changes the user node color from black to white and the user enters the system. If the password is incorrect, the user node stays black.

To detect a system attack, we consider the derivation sequence which leads to the actual system state, since a derivation sequence stores the history of events, (i.e., rule applications). For the login attack example, we assume one derivation sequence for each user in which only login rule applications (i.e., rules in figure 7) are considered. These derivation sequences are subsequences of the global derivation sequence (i.e., in which all rules are applied). A derivation sequence with respect to a single user facilitates the specification of a login system attack. If the derivation sequence for a user contains the rule application sequence *login, fail, login, fail*, the administrator must be informed and (s)he has to perform counter measures.

Two parts have to be specified in an obligation policy: on the one hand, the event which requires a special treatment by a manager (in our example the event of two failed logins), and on the other hand, the special treatment itself (in our example, the logging of the user). Therefore, an obligation policy is specified by

a set of pairs $OP = \{(\Phi, \mathcal{R})\}$, where Φ is a graph-interpreted temporal formula [6] and $\mathcal{R}$ is a set of rules. The temporal formula specifies the event, the rules in $\mathcal{R}$ the special treatment that must be performed if the event occurs. In our login example, the formula specifies the rule sequence *login, fail, login, fail* and $\mathcal{R}$ contains the rule for the manager to log the user.

Graph-interpreted temporal logic is an extension of the classical propositional fragment of linear time temporal logic [17]. We give next a brief and mostly informal introduction to graph-interpreted temporal logic which explains the concepts used in this paper. We refer the reader to [6] for a detailed explanation of graph-interpreted temporal logic.

Definition 1 (graph formula). *Let* Q *be a (countable) set of graphs. A* graph formula *is a term generated by the following syntax*

$$\Phi ::= Q | \neg\Phi | \Phi_1 \wedge \Phi_2 | \mathbf{Prev}\Phi$$

We let $\Phi, \Phi_1 \ldots$ *range over the set* TF *of graph formulas.*

The operators $\neg$ and $\wedge$ are the usual ones for negation and conjunction. The operator **Prev** constitute instead the temporal part of the logic. Roughly, the semantics is given in terms of an evaluation over derivation sequences, the *previous-time*-operator **Prev** demands that a formula holds in the immediate predecessor state. We use a past-time temporal logic, since we want to check events (i.e., rule applications) happened in the history of the system.

Figure 8 shows the temporal formula for two consecutive failed logins of the same user. The first two subformulas specify a user who has currently not entered a password, but (s)he has entered it one step before. Since the user node is currently still black, the password was incorrect and the rule *fail* was applied. The last two parts of the formula specify a login fail that the user has caused before the fail described in the first part of the formula. In order to keep track of graphical elements during the derivation sequence, we use variables to denote the same graphical elements in different graphs of the formula. For example, the variable x indicates that the user is the same in all graphs of the graph formula.

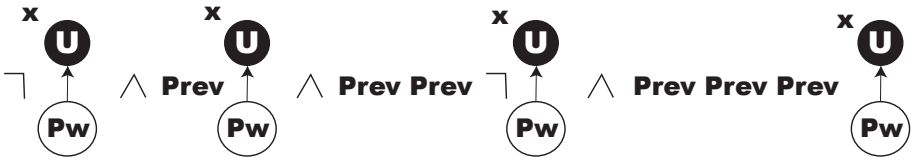


Fig. 8. The temporal formula for two failed login attempts of the same user.

Model checkers can automatically check the formula with respect to a derivation sequence [6]. Since the derivation sequences are finite, this check is always possible.

When a derivation sequence satisfies a graph formula Φ of a pair $(\Phi, \mathcal{R})$ in an obligation policy OP, the manager must apply the rules in the set $\mathcal{R}$ to handle the situation specified in Φ. The rules in $\mathcal{R}$ are applied as long as possible. In our example, whenever a user has two consecutive failed login attempts the manager simply logs the user in a log-file. The rule in Figure 9 specifies this logging and is the only member of the set $\mathcal{R}$ in the example. The user node is labeled with the variable x to specify that the rule has to be applied to the user who has performed the two login fails detected by the formula. The node SL specifies the security log file in which the administrator puts the potential attackers. The negative application condition ensures that the rule is applied once.

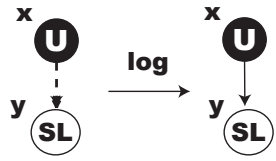


Fig. 9. The repair rule.

A role node can be additionally added to rules in $\mathcal{R}$ to model specific subjects (that may be humans or automated manager components) that must perform the actions specified in the rules. In addition, the "as-long-as-possible" semantics for the application of rules in $\mathcal{R}$ could be controlled by rule expressions as introduced in section 5.2.

5 Combination of Policies

When developing complex systems, a typical approach is to decompose the problem into subproblems, solve the subproblems and then combine the partial solutions to construct the overall solution. A "horizontal" structuring mechanism is needed that allows the definition of independent separate parts, each with its own semantics, to be completed with interconnection mechanisms to compose the subpolicies in different ways depending on the objective.

There are (at least) three different ways in which separate policies can be combined, depending on the level of abstraction of the subpolicies:

1. shuffling (or clear box)
2. black box
3. nuts and bolts

We are going to describe how the different ways can be modelled in our framework of graph transformation rules. As an example, we use the policy in section 3, denoted by policy A in the sequel, and the following policy, denoted

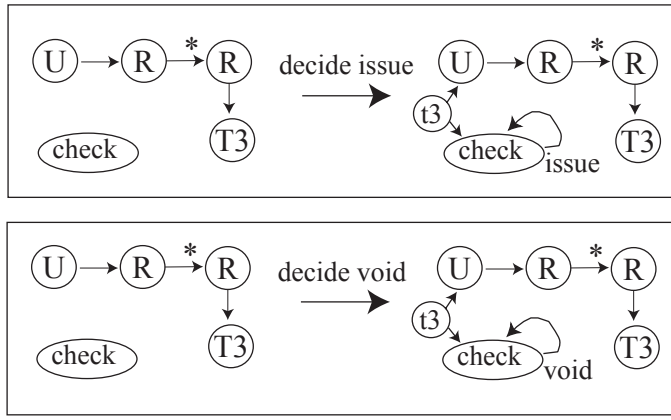


Fig. 10. Another policy.

by policy B: In this policy, the task $T2$ is omitted and task $T3$ is different in the sense that a manager can decide by her-/himself whether the check is approved or disapproved, i.e. no other decisions must be collected. The rules common in both policies are *prepare check*, *issue*, *void* and *end tax refund* (cf. section 3). Policy B contains additionally the rules in figure 10 which specify that any user playing a role that can perform task $T3$ can decide about the check. No previously made decisions of other users are necessary.

5.1 Shuffling

In the shuffling approach, policies are clear boxes. The components of a policy (in the case of our policy framework, the set of graph rules and the sets of positive and negative graphical constraints) maintain their individuality but can be combined to form the components of the new policy. For example, the new policy may have the rules common to the original two policies and the constraints of both, or the union of the original sets of rules and only the common positive and negative constraints. In our example policies A and B, we combine them to a policy $A + B$ by the union of their rules and constraints. In such a framework, it is often the applicability of a rule that determines which (sub)policy is being applied. The two (or more) original policies now "compete": constraints in different policies may be in conflict (this can be harmless or critical, depending on whether there is redundancy or not), a rule may be in conflict with another rule or with a constraint from a different policy. In the combined policy $A + B$ we have a rule conflict between the rules *decide issue(A)*[1] and *decide issue(B)* since they do partly the same in both policies, but under different conditions. In the rule *decide issue(A)*, a final decision to issue a check must be based on two previously made decisions which are not required by the rule *decide issue(B)*.

[1] The expression rule(X) specifies that the rule stems from policy X.

Having access to the components of the policy framework, these conflicts can be detected statically (i.e., before the combined policy is put into effect) and can be resolved by means of metapolicies, giving raise to coherent composed policies. Conflict detection and resolution is discussed in detail in [12]. Among the advantages of such an approach to policy combination is the possibility to enforce metapolicies of the form "more specific rules take precedence over more generic ones" when the rules come from different subpolicies (presumably already enforced within the same policy). To this end, a new negative application condition can be added to the more generic rule to prevent its applicability when the more specific one is applicable. That a rule $r_1 : L_1 \to R_1$ is more specific than a rule $r_2 : L_2 \to R_2$ can easily be checked statically: if there is a total morphism $h : L_2 \to L_1$, then whenever r_1 is applicable to G via a match $m : L_1 \to G$, then so is r_2 via a match $m \circ h : L_2 \to G$. For example, the rule *decide issue(A)* in figure 4 is a specialization of the rule *decide issue(B)* in figure 10. Therefore, the rule *decide issue(B)* would get a negative application condition that prevents its applicability if there are already decisions made for the check.

Notice that in this approach the rules themselves are not changed, but their applicability may be reduced by adding NAC's. The effect of the rule remains the same as before the combination. The original policies are subpolicies of the combined one. Alternatives to the metapolicy described here can be found in [11,9].

5.2 Nut and Bolts

In the "nuts and bolts" view of combinations, the entities that form the components, i.e., the individual rules and constraints, can be combined to form the new rules and the new constraints. The effect of the new rules may bear no resemblance to the rules of the subpolicies, unlike the combination discussed in the previous subsection. Rules are combined using rule expressions [4,7].

A rule expression E, built up over a certain set of operators, specifies a set of sequences of rule names, denoted by $seq(E)$. If one can apply the rules to a graph in the order given in such a sequence, the sequence is *applicable* to the graph and the graph is transformed. A rule expression is applicable to a graph, if at least one sequence in $seq(E)$ is applicable to it.

Given a set $Names$ of rule names from which rule expressions are constructed, a rule expression is a term generated by the following syntax, where we distinguish between expressions E interpreted as rules to be applied and expressions BE to be interpreted as boolean values resulting from the applicability of rules:

- basic operators:
  $E ::= Names \mid E_1; E_2 \mid \mathbf{null} \mid E_1 \cup E_2 \mid E[\nu] \mid$
  $\mathbf{if}\ BE\ \mathbf{then}\ E_1\ \mathbf{else}\ E_2\ \mathbf{end} \mid$
  $\mathbf{while}\ BE\ \mathbf{do}\ E\ \mathbf{end} \mid \mathbf{asLongAsPossible}\ E\ \mathbf{end}$
- boolean operators:
  $BE ::= BE_1\ \mathbf{and}\ BE_2 \mid BE_1\ \mathbf{implies}\ BE_2 \mid$
  $BE_1\ \mathbf{or}\ BE_2 \mid BE_1 = BE_2 \mid BE_1\ \mathbf{xor}\ BE_2 \mid \mathbf{a}(E) \mid \mathbf{na}(E)$

Most of the operators presented above have the obvious meaning: operators **a** and **na** test the applicability and non-applicability, respectively, of the rule. The expression $E[\nu]$ is the expression E with the renaming described by ν. The operator **asLongasPossible** applies a rule corresponding to the rule expression to a graph as long as it is applicable.

The use of rule expressions allows a closer control on the "sequencing" of the application of rules. It is possible to replace two rules that *can* be applied one after the other with their sequential composition so as to *force* the effect of the two successive applications. Similarly, it is possible to enforce the simultaneity of two rules from different policies by replacing them with the appropriate rule expression using $\cup$ and renaming.

In our example, the set *Names* contains the rule names of the policies A and B. If we want to specify again the higher priority of the rule *decide issue(A)*, we use the following rule expression.

if(**a**(*decide issue(B)*) **and** **na**(*decide issue(A)*)) **then** *decide issue(B)* **end**.

This approach is similar to the one presented in [8] where four methods for combining policies are classified: concatenation (with control dependence between two policies), parallel application (for policies that do not conflict), selection (with different policies receiving multiple types of results of another policy), repetition (for the continued application of a policy until a condition is met). These methods can readily be described using rule expressions extended to sets of rules rather than to individual rules.

5.3 Black Box

In this approach, policies are treated as black boxes in that they are used without any knowledge of their internal structure. In the combined policy, the decision on which subpolicy applies is made 'a priori' using the external interconnection language. It is not possible to defer to the subpolicy the decision on which is the appropriate rule to be used in a given context. This approach can be modelled in the graph transformation approach with the use of transformation units [13].

A *transformation unit* over a given graph transformation approach is a system $TU = (I, U, R, C, T)$ where

- I and T are graph class expressions with associated sets of graphs $SEM(I)$ and $SEM(T)$ which are the graphs the unit can operate upon (as input) and those returned by the unit (as export), respectively.
- U is a set of identifiers representing other units available in the system
- R is a set of rules
- C is a control condition that imposes restrictions on causal dependencies among rules

This is the approach to policy combination in [3] and in [18]. In the former, Bonatti et. al propose an algebra for a policy composition framework. The approach is set-based and policies are represented as expressions of the algebra and

the composition of policies is specified through given algebra operators such as union, intersection etc. The set-theoretic semantics of the algebra allows to reason about a policy specification and their properties. In [18], a policy is viewed as a relation over a set of authorizations of the form $(subject, object, actionterm)$; the approach is state based (similar to our approach where a state graph represents a state) and a policy transforms a state into a state. Policy combinations are also described by operators, in this case on relations (accounting for non determinism) rather than on ground authorization terms as in [3], and sequential composition is explicitly included.

Graph transformation units can be used to model policy combinations in a uniform way. Whereas in [3,18] there are two levels of description, explicit *operators* to be applied to *policies*, we can use **a policy** represented by a unit to incorporate the knowledge of how to combine the other (sub)policies. This "top" unit has an empty set R of local rules, a set U of identifiers for the policies (the policy 'variables' in the expressions in [3,18]) and a control condition C to represent a term built using the operators. This control condition can be represented using a rule expression built as described in 5.2.

This idea of using a unit to combine pre-existing units can be extended by allowing the "top" unit to be general, with local rules and constraints I and T on input and output graphs. This in turn can be used to structure a policy in layers, in a hierarchy of units based on the 'functional' dependencies represented in the sets of identifiers U. This requires an extension of the notion on unit in [13] by adding new components in the form of sets of positive and negative constraints. We expect all the results on units to still hold in this extended version.

This approach to policy combination does not need knowledge of the inner workings of the policies. It is possible to detect inconsistencies among the subpolicies as they are executed and a decision at a metalevel can be made to resolve them. What is not possible is to detect conflicts statically and to modify the policies to remove them.

6 Refinement of Policies

A particular kind of evolution of a policy is given by a refinement step in which policy changes derive from the need to add details and to specify the behavior at a lower level. Depending on the level of abstraction, a task represented by a single step (rule) may correspond to a combination of tasks at a lower level. A refinement step consists of replacing a rule with an expression in such a way that a direct (one-step) derivation using the single rule generates the same state produced via the derivation sequence (multiple-steps) corresponding to the interpretation of the rule expression. It can be shown, that refinement via rule expressions constructed with the syntax in subsection 5.2 preserve derivation sequences in the sense mentioned above. Refinement can be viewed as the decomposition of high level rules into the combination of lower level rules. The combination can be encapsulated into a unit, allowing the application of each rule at a lower level to produce "temporary" state graphs that need not satisfy

the invariant constraints, provided that the unit is treated as a transaction. Negative application conditions of a high level rule can be decomposed, using the structure of the rule expression, into the appropriate NAC's for the lower level rules.

7 Concluding Remarks

We have presented a formalism to specify policies where states are represented by graphs and their evolution by graph transformations. A policy is formalized by four components: a type graph, positive and negative constraints (a declarative way of describing what is wanted and what is forbidden) and a set of rules (an operational way of describing what can be constructed). In [12] is shown, how graph rules can be automatically modified before run-time so that they construct only graphs that satisfy the constraints.

An approach similar to ours can be found in [5], which proposes a language-based approach for the specification of policies for distributed object systems. It is based on Ponder, a declarative object-oriented language featuring composite policies and meta-policies. Among the major differences between our graph-based formalism and the one in [5] is the lack of a formal semantics for Ponder. Our framework could be used to give a formal semantics and to view Ponder as a 'low level' (in the sense of easier to implement) language that can be used to realize our policy frameworks. The requirements for a policy language identified on page 19 in [5] are support for access control and delegation, structuring techniques, composite policies, analysis of conflicts, extensibility for new types of policies, and comprehensible and easy to use by policy users. We have not addressed explicitly delegation and extensibility may not be its strongest suit, but we believe that the intuitive visual representation of graphs goes a long way towards the last requirement.

A tool, based on a generic graph transformation engine, is under development to assist in the specification of policy frameworks. An aim of the tool is the support of a systematic detection (i.e., which rule may destroy which constraint) and resolution (i.e., modification of rules) of conflicts in the specified policy.

References

1. *Policies for Distributed Systems and Networks*, number 1995 in LNCS. Springer, 2001.
2. Elisa Bertino, Elena Ferrari, and Vijay Atluri. The Specification and Enforcement of Authotization Constraints in Workflow Management Systems. *ACM Trancactions on Information and System Security*, 2(1):65–104, February 1999.
3. P. Bonatti, S. de Capitani di Vimercati, and P. Samarati. A Modular Approach to Composing Access Control Policies. In *Proc. of 7th ACM Conference on Computer and Communication Security*, pages 164–173. ACM, 2000.
4. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. Consistency checking and visualization of OCL constraints. In A. Evans and S. Kent, editors, *UML 2000*, number 1939 in LNCS, pages 294–308. Springer, 2000.

5. N. Damianou, N. Dulay, E. Lupu, and M. Sloman. The Ponder Policy Specification Language. In *Proc. of POLICY 2001* [1], pages 18–38.
6. F. Gadducci, R. Heckel, and M. Koch. A fully abstract model for graph-interpreted temporal logic. In H. Ehrig, G. Engels, H.J. Kreowski, and G. Rozenberg, editors, *Proc. of 6th Int. Workshop on Theory and Graph Transformation*, number 1764 in LNCS. Springer, 2000.
7. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Formal software specification with refinement and modules for typed graph transformation systems. *Journal of Computer and System Sciences*. to appear.
8. Y. Kanada. Taxonomy and Description of Policy Combination Methods. In *Proc. of POLICY 2001* [1], pages 171–184.
9. M. Koch, L. V. Mancini, and F. Parisi-Presicce. On the Specification and Evolution of Access Control Policies. In S. Osborne, editor, *Proc. 6th ACM Symp. on Access Control Models and Technologies*, pages 121–130. ACM, May 2001.
10. M. Koch, L.V. Mancini, and F. Parisi-Presicce. A Formal Model for Role-Based Access Control using Graph Transformation. In F.Cuppens, Y.Deswarte, D.Gollmann, and M.Waidner, editors, *Proc. of the 6th European Symposium on Research in Computer Security (ESORICS 2000)*, number 1895 in Lect. Notes in Comp. Sci., pages 122–139. Springer, 2000.
11. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Foundations for a graph-based approach to the Specification of Access Control Policies. In F.Honsell and M.Miculan, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, Lect. Notes in Comp. Sci. Springer, March 2001.
12. M. Koch, L.V. Mancini, and F. Parisi-Presicce. Conflict Detection and Resolution in Access Control Policy Specifications. In M.Nielsen and U.Engberg, editors, *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS 2002)*, number 2303 in Lect. Notes in Comp. Sci., pages 223–237. Springer, 2002.
13. H.-J. Kreowski and S. 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*, volume 2, chapter 15, pages 607–638. 1999.
14. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations. Vol. I: Foundations*. World Scientific, 1997.
15. R. S. Sandhu. Role-Based Access Control. In *Advances in Computers*, volume 46. Academic Press, 1998.
16. M. S. Sloman. Policy Driven Management for Distributed Systems. *Network and Systems Management*, 2(4):333–360, 1994.
17. C. Stirling. Modal and Temporal Logics. In *Background: Computational structures*, volume 2, pages 477–563. Clarendon Press, 1992.
18. D. Wijesekera and S. Jajodia. A policy algebra for access control: the propositional case. In *Proc. of the 8th ACM Conference on Computer and Communication Security*, pages 38–47. ACM Press, November 2001.

Computer Aided Multi-paradigm Modelling to Process Petri-Nets and Statecharts

Juan de Lara[1] and Hans Vangheluwe[2]

[1] ETS Informática
Universidad Autónoma de Madrid
Madrid Spain,
Juan.Lara@ii.uam.es
[2] School of Computer Science
McGill University, Montréal
Québec, Canada
hv@cs.mcgill.ca

Abstract. This paper proposes a *Multi-Paradigm* approach to the modelling of complex systems. The approach consists of the combination of meta-modelling, multi-formalism modelling, and modelling at multiple levels of abstraction. We implement these concepts in AToM^3, A Tool for Multi-formalism, Meta-Modelling. In AToM^3, modelling formalisms are modelled in their own right at a meta-level within an appropriate formalism. AToM^3 uses the information found in the meta-models to automatically *generate tools* to process (create, edit, check, optimize, transform and generate simulators for) the models in the described formalism. Model processing is described at a meta-level by means of models in the graph grammar formalism. As an example, meta-models for both syntax and semantics of Statecharts (without hierarchy) and Petri-Nets are presented. This includes a graph grammar modelling the transformation between Statecharts and Petri-Nets.

1 Introduction

Complex systems are characterized, not only by a large number of components, but also by the diversity of these components. This often implies that the components are described in different formalisms. Several approaches are possible to deal with this variety:

1. A single *super-formalism* may be constructed which subsumes all the formalisms needed in the system description. In most cases, this is neither possible nor meaningful.
2. Each system component may be modelled using the most appropriate formalism and tool. In the *co-simulation* approach, each component is subsequently simulated with a formalism-specific simulator. Interaction due to component coupling is resolved at the trajectory level (simulation output data). Questions about the overall system can only be answered at the state trajectory level. It is no longer possible to answer symbolic, high-level questions which could be answered within the individual components' formalisms.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 239–253, 2002.
© Springer-Verlag Berlin Heidelberg 2002

3. In *multi-formalism* modelling, as in co-simulation, each system component may be modelled using the most appropriate formalism and tool. However, a single formalism is identified into which each of the component models may be symbolically transformed [18]. The formalism to transform to depends on the question to be answered about the system. The Formalism Transformation Graph [18] (FTG, see Figure 1) proposed by the authors suggests DEVS [20] as a universal common formalism for simulation purposes. It is easily seen how multi-formalism modelling subsumes both the super-formalism approach and the co-simulation approach.

In order to make the multi-formalism approach applicable, we still have to solve the problem of interconnecting a plethora of different tools, each designed for a particular formalism. Also, it is desirable to have highly problem-specific formalisms and tools. The time needed to develop these is usually prohibitive. We tackle this problem by means of *meta-modelling*. Using a meta-layer of modelling, it is possible to model the modelling formalisms themselves. Using the information in these meta-layers, it is possible to generate customized tools for models in the described formalisms. The effort required to construct a tool for modelling in a formalism tailored to particular applications thus becomes minimal. Furthermore, when the generated tools use a common data structure to internally represent the models, transformation between formalisms is reduced to the transformation of these data structures.

In this article, we present AToM^3 [3], a tool which implements the ideas presented above. AToM^3 has a meta-modelling layer in which different formalisms are modelled. From the meta-specification (in the Entity Relationship formalism extended with constraints), AToM^3 generates a tool to process models described in the specified formalism. Models are represented internally using *Abstract Syntax Graphs*. As a consequence, transformations between formalisms is reduced to graph rewriting. Thus, the transformations themselves can be expressed as graph grammar models [5]. Graph grammars have been used in highly diverse areas such as graphical editors, code optimization and computer architecture [7]. Here we use them in the context of multi-paradigm modelling (for formalism transformation) and present an example of transforming Statechart models (without hierarchy)into behaviourally equivalent Petri-Nets.

The rest of the paper is organized as follows: section 2 introduces Multi-Paradigm Modelling and section 3 presents other related approaches. Section 4 gives an overview of the multi-paradigm modelling tool AToM^3. In section 5, we show an example of model manipulation in AToM^3. Finally, section 6 presents conclusions and future work.

2 Multi-paradigm Modelling

Computer Automated Multi-Paradigm Modelling is an emerging field which addresses and integrates three orthogonal directions of research:

1. *Multi-Formalism modelling*, concerned with the coupling of and transformation between models described in different formalisms. In Figure 1, a part of the "formalism space" is depicted in the form of an FTG. The different formalisms are shown as nodes in the graph. The arrows denote a homomorphic relationship "can be mapped onto". The mapping consists of transforming a model in the source formalism into a behaviourally equivalent one in the target formalism.

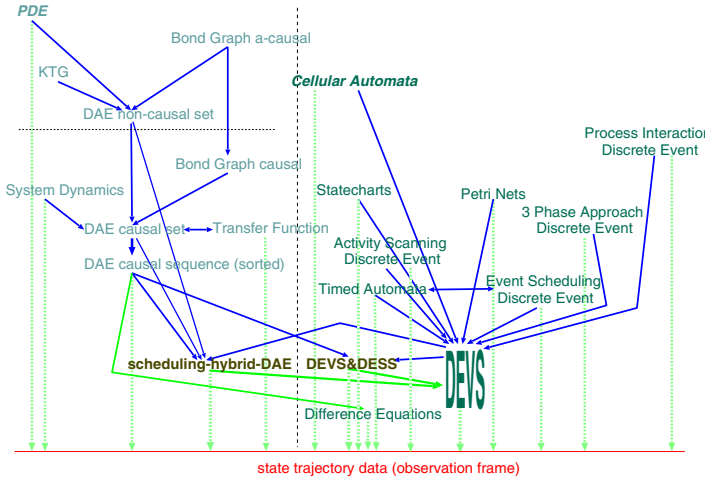


Fig. 1. Formalism Transformation Graph (FTG).

In our approach, we allow the specification of composite systems by coupling heterogeneous components expressed in different formalisms. For the analysis of its properties, most notably, its behaviour, the composite system must be assessed by looking at the *whole* multi-formalism system. For appropriate processing (simulation, code generation, etc.) of the composite model, its components may have to be transformed to a common formalism, which can be found in the FTG. As we will see later, formalisms are meta-modelled and stored as graphs. Thus, the transformations denoted by the arrows of the FTG can be modelled as graph grammars.

2. *Model Abstraction*, concerned with the relationship between models at different levels of abstraction.
3. *Meta-Modelling* (models of models), which is the process of modelling formalisms. Formalisms are described as models using meta-formalisms. The latter are nothing but formalisms expressive enough to describe other formalisms' syntax and semantics. Examples are the Entity Relationship formalism or UML class diagrams. A model of a meta-formalism is called a meta-meta-model; a model of a formalism is called a meta-model. Table 1 depicts the levels considered in our meta-modelling approach.

Table 1. Meta-Modelling Levels.

Level	Description	Example
Meta-Meta-Model	Model that describes a formalism that will be used to describe other formalisms.	Description of Entity-Relationship Diagrams, UML class Diagrams
Meta-Model	Model that describes a simulation formalism. Specified under the rules of a certain Meta-Meta-Model	Description of Deterministic Finite Automata, Ordinary differential equations (ODE)
Model	Description of an object. Specified under the rules of a certain Meta-Model	$f'(x) = -\sin x, f(0) = 0$ (in the ODE formalism)

To be able to fully specify modelling formalisms, the meta-level formalism may have to be extended with the ability to express constraints (limiting the number of meaningful models). For example, when modelling a Determinsitic Finite Automaton, different transitions leaving a given state must have different labels. This cannot be expressed within Entity-Relationship diagrams alone. Expressing constraints is most elegantly done by adding a constraint language to the meta-modelling formalism. Whereas the meta-modelling formalism frequently uses a graphical notation, constraints are concisely expressed in textual form. For this purpose, some systems [17] (including ours) use the Object Constraint Language OCL [14] used in the UML. As AToM^3 is implemented in the scripting language Python [16], arbitrary Python code may also be used.

3 Other Approaches

A similar approach to our vision of Multi-Paradigm Modelling (although oriented to the software engineering domain) is ViewPoint Oriented Software Development [8]. See also [19] and [13] for other approaches.

There are other visual tools to describe formalisms using meta-modelling, among them DOME [4], Multigraph [17], MetaEdit+ [11] and KOGGE [6]. Some of these allow one to express formalism semantics by means of a textual language (KOGGE for example uses a language similar to Modula-2). Our approach is quite different, as we express such semantics by means of graph grammar models. We believe that graph grammars are a natural, declarative, and general way to express transformations. As graph grammars are highly amenable to graphical representation, they are superior to a purely textual language. Also, none of the tools consider the possibility of "translating" models between different formalisms. There are various languages and systems for graph grammar manipulation, such as PROGRES [15], GRACE [9] and AGG [1]. None of these have a meta-modelling layer.

Our approach is original in the sense that we combine the advantages of meta-modelling (to avoid explicit programming of customized tools) and graph transformation systems (to express tool behaviour and formalism transformation) for multi-paradigm modelling. Our main contribution is in the field of multi-paradigm modelling [18], as we have a general means to transform models between different formalisms.

4 AToM^3: An Overview

AToM^3 is a tool written in Python [16] which uses and implements the concepts presented above. Its architecture is shown in Figures 2 and 3. In both figures, models are represented as white boxes, having in their upper-right hand corner an indication of the meta-...model they were specified with. It is noted that in the case of a graph grammar model, to convert a model in formalism F_{source} to a model in formalism F_{dest}, it is necessary to use the meta-models of both F_{source} and F_{dest} together with the meta-model of graph grammars.

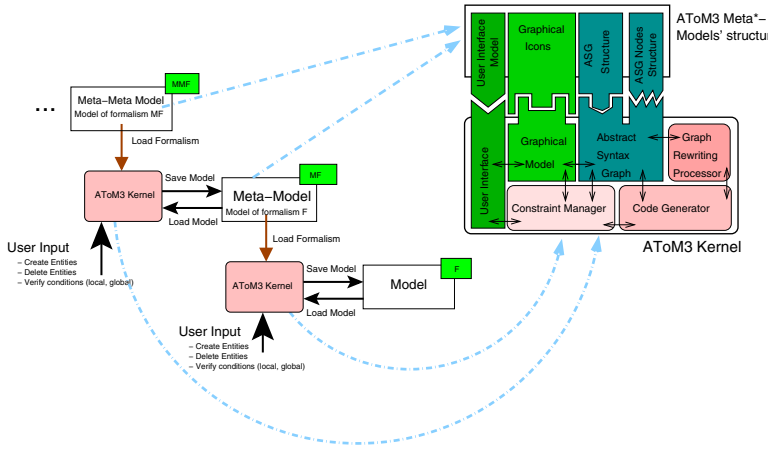


Fig. 2. Meta-... Modelling in AToM^3.

The main component of AToM^3 is the *Kernel*, which is responsible for loading, saving, creating and manipulating models (at any meta-level, via the *Graph-Rewriting Processor*), as well as for generating code for customized tools. Both meta-models and meta-meta-models can be loaded into AToM^3 as shown in Figure 2. The first kind of models allows constructing valid models in a certain formalism, the second kind are used to describe the formalisms themselves. Models, meta-models and meta-meta-models are all stored as *Abstract Syntax Graphs* whose nodes and links are typed, and their relationships are subject to constraints dictated by the formalism under which these models were defined.

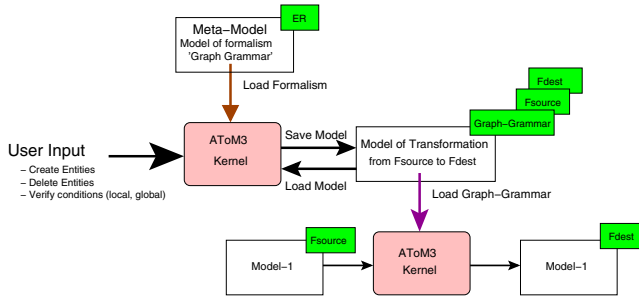


Fig. 3. Model Transformation in AToM^3.

The ER formalism extended with constraints is available at the meta-meta-level. It is perfectly possible to define other meta-meta-formalisms using ER, such as UML class diagrams. Constraints can be specified as OCL or Python expressions, and the designer must specify when (pre- or post- and on which event) the condition must be evaluated. Events can be *semantic* (such as editing an attribute, or connecting two entities) or *graphical* (such as dragging, dropping, or moving an object).

When modelling at the meta-meta-level, the entities which may appear in a model must be specified together with their attributes. We will refer to this as the *semantic* information. For example, to define the Petri-Nets Formalism, it is necessary to define both *Places* and *Transitions*. Furthermore, for *Places* we need to add the attributes *name* and *number of tokens*. For *Transitions*, we need to specify their *name* attribute.

In the meta-model, it is also possible to specify the *graphical appearance* of each entity when instantiated at the lower meta-level. For example, for Petri-Nets, we can choose to represent *Places* as circles with the *number of tokens* inside the circle and the *name* beside it, and *Transitions* as thin rectangles with the *name* beside them. That is, we can specify how some semantic attributes are displayed graphically. Constraints can also be associated with the graphical entities.

The meta-meta-information is used by the *Kernel* to generate some Python files (see the upper-right corner of Figure 2), which, when loaded by the *Kernel*, allows the processing of models in the defined formalism. These files include a model of the user interface presented when the formalism is loaded. This model follows the rules of the *"Buttons"* formalism, and by default contains a "create" button for each object found in the meta-model. For the case of the Petri-Net formalism, it contains buttons to create *Places*, *Transitions*, and the connections between them. This model can be modified using AToM^3 to for example add buttons to execute graph grammars on the current model or to delete unwanted buttons. When a formalism is loaded, the *Kernel* interprets the user interface model, to create and place the actual widgets and associate them with the appropriate actions.

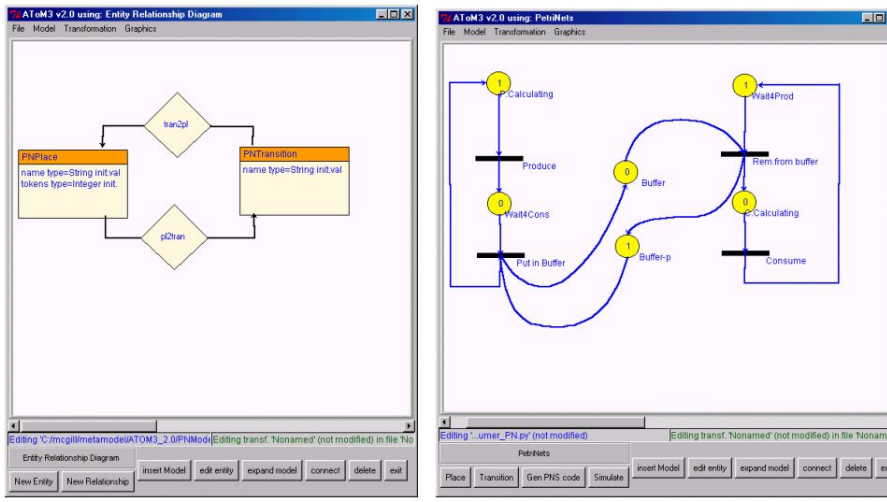


Fig. 4. Meta-Model of the Petri-Net Formalism (left) and Generated Tool to Process Petri-Net Models, with a Producer-Consumer model (right).

Figure 4 shows an example of meta-modelling at work. It shows AToM^3 being used to describe the Petri-Net formalism (left), and the automatically generated tool (from the previous description) to process Petri-Nets. This tool can be further extended by specifying Petri-Nets manipulations, such as operational semantics (simulation) by means of graph grammar models, expressed at the meta-level.

For the implementation of the Graph Rewriting Processor, we have used a generalization of the algorithm given in [5], in which we allow non-connected graphs in Left Hand Sides (LHS) in rules. It is also possible to define a sequence of graph grammars to be applied to the model. This is useful, for example to couple grammars to convert a model into another formalism, and then apply an optimizing grammar. Often, for clarity and efficiency reasons, graph grammars are divided into different independent parts.

Rule execution can either be continuous (no user interaction) or step-by-step whereby the user is prompted after each rule execution. As the LHS of a rule can match different subgraphs of the host graph, we can also control whether the rule must be applied to all the matching subgraphs (if disjoint), if the user can choose one of the matching subgraphs interactively, or whether the system chooses one at random.

As in grammars for formalism transformations we have a mixing of entities belonging to different formalisms, it must be possible to open several meta-models at the same time (see Figure 3). Obviously, the constraints of the individual formalism meta-models are meaningless when entities in different formalisms are present in a single model. Such a model may come to exist during the intermediate stages of graph grammar evaluation when transforming a model

from one formalism into another. It is thus necessary to disable evaluation of constraints during graph grammar processing (i.e., all models are reduced to Abstract Syntax Graphs). At the end of the execution of a graph grammar for formalism transformation, the Kernel checks if the resulting model is valid in some of the currently open formalisms, and closes the other formalisms.

5 Model Manipulations

In this section we present some examples of the kind of model manipulations performed in AToM^3. These manipulations complement the default Kernel capabilities. When a tool is generated, it can be used to build, load and save models and check whether they are correct. Graph grammars are a way to enrich these functionalities. Some examples of the uses of graph grammars in AToM^3 are:

- Formalism transformation: an example will be presented in subsection 5.1.
- Model optimization: these transformations reduce the complexity of a model.
- Simulator specification: allow us to express the operational semantics of a formalism.
- Code generation: permit the generation of code for a specific tool.

5.1 Formalism Transformation

A formalism transformation takes a model m_1 in a formalism F_{source} and converts it into a model m_1' but expressed in formalism F_{dest}. The FTG (see Figure 1) shows behaviour preserving transformations between formalisms. We have implemented several of these transformations in AToM^3 with graph grammars. Some of the reasons to perform formalism transformations are:

- In a composite model with components described in different formalisms, it may be possible to transform each component into some formalism reachable (in the FTG) from the formalisms of each component. Then, once the composite model has all its components described in the same formalism, we can simulate it.
- To solve problems that are easier to solve in some other formalism. For example, in the case of a model in the Statecharts formalism, for which we want to determine whether it can deadlock. As this kind of analysis is well known for Petri-Nets, we can transform the Statecharts model into the Petri-Nets formalism, and then solve the problem in that domain. We will provide such a transformation (limited to non-hierarchical Statecharts) in this section.

The Statecharts formalism is widely used for modelling reactive systems. Statecharts can be described as an elaboration of the HiGraphs [10] semantics as well as of State Automata. HiGraphs allow for hierarchy (blobs can be inside blobs), parallelism (a blob can be divided into concurrent Orthogonal components), Blobs are connected via hyperedges. Our meta-model for Statecharts is thus composed of the following entities:

- *Blobs*. These have a *name* and represent the states in which the system can be. A state may have one or more *Orthogonal components* inside. We represent *Blobs* as rectangles.
- *Orthogonal components*. These have a *name* and a Statechart inside. We will represent *Orthogonal components* as rectangles with rounded, dashed lines. As we will see later, *"insideness"* is represented as a relationship at the meta-level.
- *Initial state*. These kind of entities mark the state in which the system enters when reaching a certain *Orthogonal component*.

The following relationships are also included in the meta-model:

- *Hyperedge*. This relationship implements a directed hyperedge and is used to connect a group of *Blobs*. It contains the following attributes:
  - *Event*, which is a list of the events that must occur for the transition to take place;
  - *Broadcast_Event*, which is a list of the events to broadcast if the transition takes place;
  - and *Actions* which stores Python code to be executed when the transition takes place.

  Events can be global, or can be directed to a particular *Orthogonal component*. Also included in the *Event* list is a guard: the condition of a particular *Orthogonal component* being in a certain state.
- *has_Inside*. This is a relationship between *Orthogonal components* and *Blobs*. It expresses the notion of hierarchy: *Blobs* are inside *Orthogonal components*.
- *composed_of*. This is a relationship between *Blobs* and *Orthogonal components*. It expresses the notion of hierarchy in the other direction, meaning that *Blobs* are composed of one or more *Orthogonal components*.
- *has_Initial*. This relationship express the notion of hierarchy between *Orthogonal components* and *Initial states*.
- *iconnection*. This relationship allows the connection of an *Initial state* and a *Blob*

As relationships *has_Inside*, *composed_of* and *has_Initial* are a means to express hierarchy they are drawn as invisible links.

Some actions have been added to the meta-model to move all the *Blobs* inside of *Orthogonal components* when the latter are moved. Also, when an *Orthogonal component* is placed inside a *Blob*, this is enlarged to accomodate it. Figure 5 shows the meta-model and the generated tool for modelling Statecharts.

In this section we will present a graph grammar to transform non-hierarchical Statecharts (without OCL constraints) into Petri-Nets. This is, a Statechart model can be composed of several *Orthogonal components*, which can contain *Blobs*, but the converse is not allowed. The model in Figure 5 has this restriction.

This transformation has been implemented in four different, independent graph grammars, as there are four well defined steps for completing the transformation, and there is no need to evaluate the applicability of the other rules, as none of them will be applicable. This makes the transformation process faster and easier to debug. The four graph grammars are:

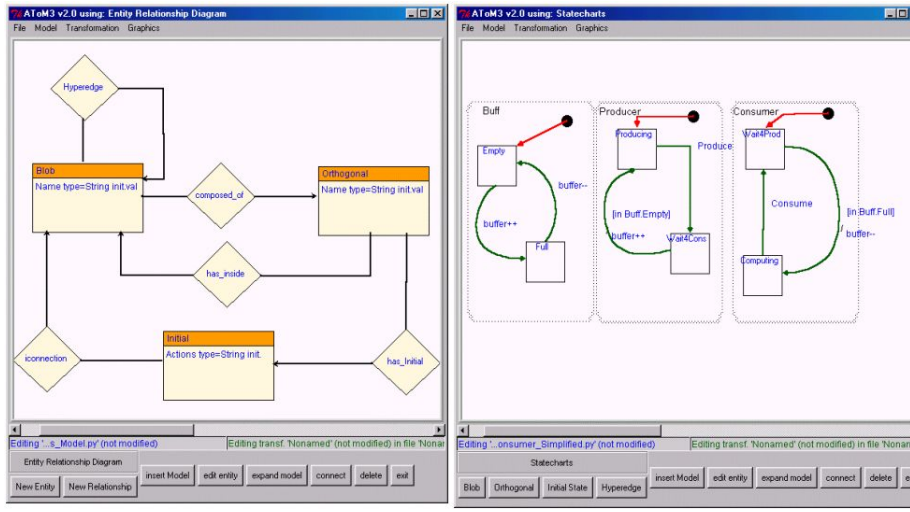


Fig. 5. The Meta-Model (left) and the generated tool for Statechart modelling. Used here to model the Producer-Consumer Problem (right).

1. The first graph grammar identifies the events in the Statechart by looking at the events each *hyperedge* has in its lists. It creates a *Place* for each event. In our approach, each type of event in the system is assigned a *Place*. These are considered *interfaces* in which the system puts a token whenever the event takes place. For each *Orthogonal component* in the Statechart model, the graph grammar creates *"local"* versions of the global events. Each global event is connected to the local events via a unique *Transition*, in such a way that when the global event is present, the token is broadcast to all the local events of the *Orthogonal components*. This global/local distinction is useful when an output event has to be sent:
   - If the event has to be sent to a particular *Orthogonal component*, a token will be placed in its corresponding local event *Place*.
   - If it has to be broadcast to the whole system, it will be placed in the global event *Place*.

   Note also that the local event *Places* are connected to a *Transition* whose purpose is to eliminate the token if it has not been consumed immediatelly after the event took place. This graph grammar is shown in Figure 6.
   It can also be observed how the local events are connected to their corresponding *Orthogonal component*.
2. The second graph grammar creates a *Place* for each *Blob* and moves the *Hyperedges* from the *Blobs* to the *Places*. Figure 7 shows this graph grammar. The way the transfer of the *hyperedges* from the *Blobs* to the *Places* is done is by keeping a connection from the *Blob* to the associated *Place*. The graph grammar also locates the initial state and thus puts a token in the corresponding *Place* (see rule 4). Once all the *hyperedges* are moved, the

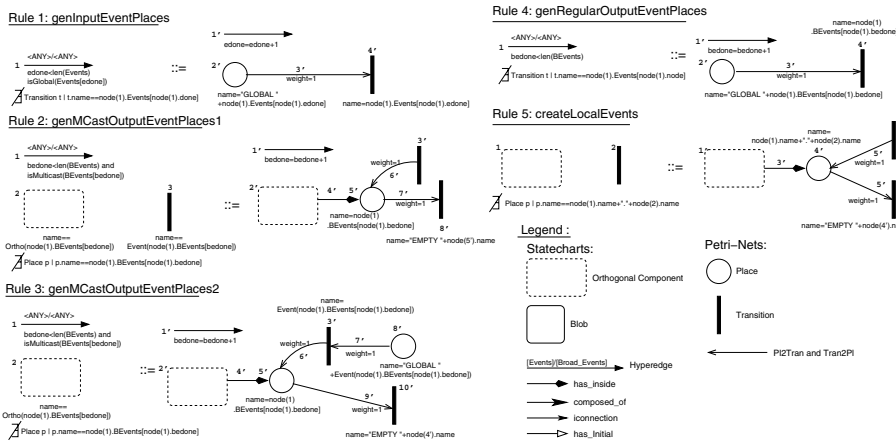


Fig. 6. First Graph Grammar: creates the Local and Global Event Places.

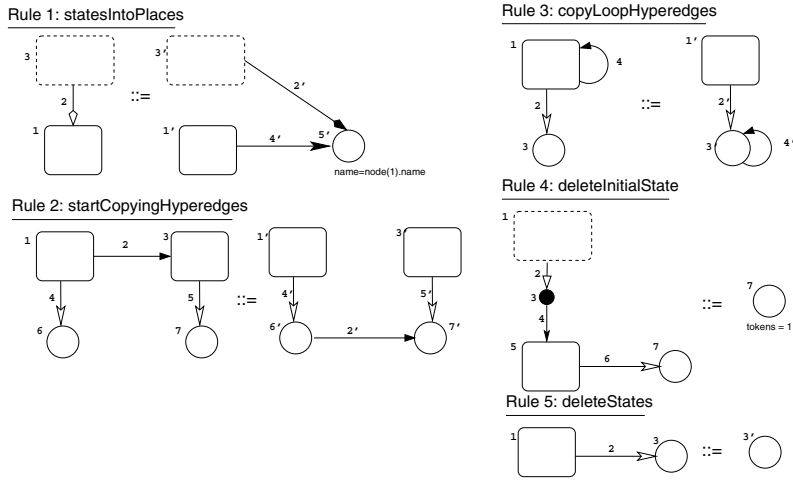


Fig. 7. Second Graph Grammar: creates the Local and Global Event Places.

Blobs can be erased. During the execution of the Petri-Net, in the *Places* that represent the states of the system, we will have a number of tokens equal to the number of *Othogonal components*.

3. The third graph grammar converts the *hyperedges* into *Transitions* connected to the appropriate *Places*. Figure 8 shows some of the rules of this graph grammar (the implementation is composed of nine rules). The way to proceed is first to identify the *Place* associated with the first event of the list of events of the *hyperedge* and then make a connection between this event and an itermediate *Transition* which is created in the process, and then eliminate the event from the list. A similar process has to be followed for the lists of

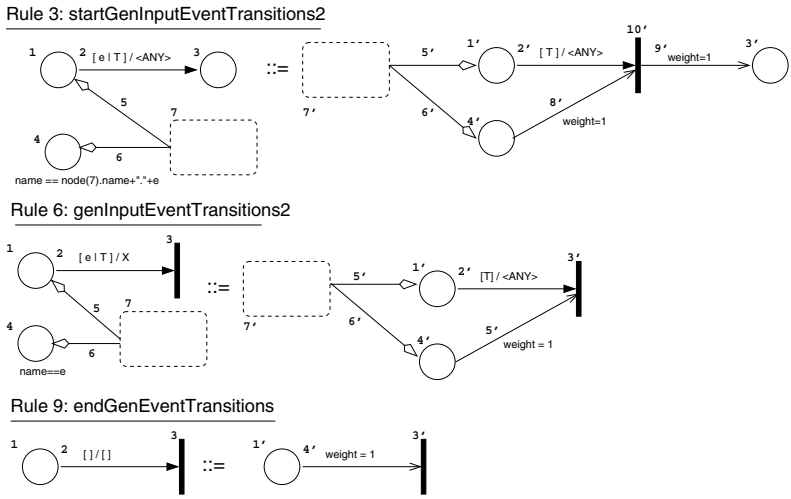


Fig. 8. Third Graph Grammar (not Completelly Shown): eliminates Hyperedges.

events to be broadcast. The process finishes when both lists of events are empty. Observe that we used a Prolog-like notation for identifying the first element of the lists of events and its subsequent elimination.
Other rules of this graph grammar deal with the case when an *Orthogonal* component is in a certain state; and with the case of an event to be sent to a particular *Orthogonal component*.

4. The fourth graph grammar simply removes the *Orthogonal components*.

The results of the application of this graph grammar to the Statechart model in Figure 5 is shown in Figure 9 (after applying a graph grammar for simplifying Petri-Nets, not shown in this paper).

There are other approaches to the conversion of Statecharts into Petri-Nets. In [12], a manual method is proposed, but it is not systematic. Basically, a human Petri-Net and Statecharts expert would have to understand the behaviour of the Statechart model and then model an equivalent Petri-Net. The possibility of further simplification and manipulation of the model is not considered. Also, events are represented as *Transitions* whereas in our approach, we insert *interface Places* to represent *Global* and *Local* events. This facilitates the reuse of the Petri-Net models and makes it possible to send events to particular orthogonal components.

6 Conclusions and Future Work

In this paper we have discussed the advantages of a multi-paradigm approach when modelling complex systems. Meta-Modelling means explicitly model the formalisms. It allows for the automatic generation of customized tools. Formal-

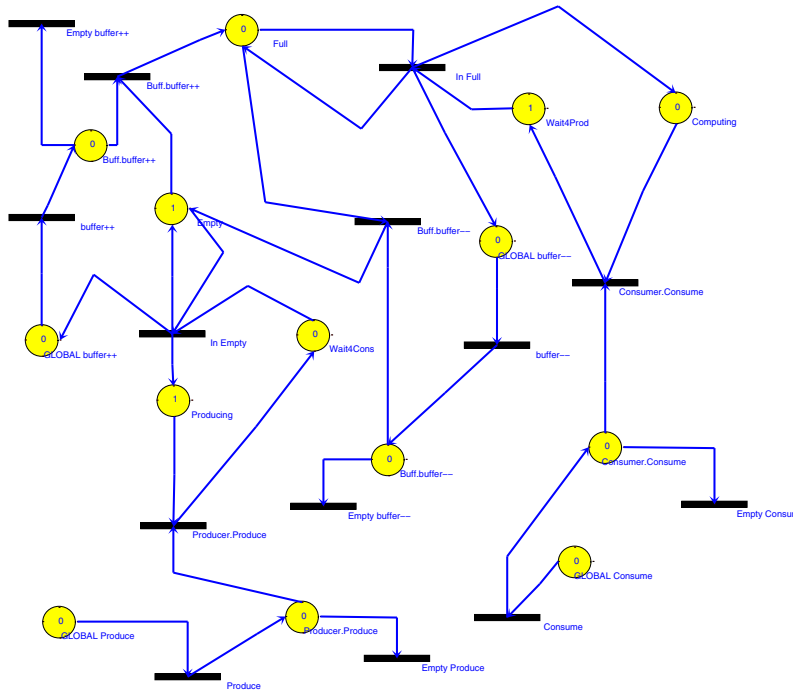


Fig. 9. Result of the application of the graph grammar to the Producer-Consumer Statechart.

ism transformation permits the translation of models between formalisms to solve problems that are easier to solve in other formalisms.

We have presented AToM^3, a meta-modelling tool able to generate customized, formalism-specific tools. As models are stored in the form of graphs, AToM^3 can manipulate them using graph grammars. Users can define –in a visual, high level way– graph grammars to manipulate their models without having to modify or have knowledge about the AToM^3 kernel code. In particular, we have shown how the user can analyze a Statechart model by converting it into a Petri-Net. Some other graph grammars could then be applied to reduce its complexity, and to simulate it or to generate code for a specific Petri-Net tool for further processing. Note how this process can be completely automated from the appropriate graph grammars. The process could also be made invisible to the Statecharts modeller, who could be only interested in knowing –in the case of the example presented– whether the Buffer can be made to exceed its capacity.

The advantages of using an automated tool for generating customized model-processing tools are clear: instead of building the whole application from scratch, it is only necessary to specify –in a graphical manner– the kind of models we will deal with. The processing of such models can be expressed by means of graph grammars, at the meta-level. Our approach is also highly applicable if we want

to work with a slight variation of some formalism, where we only have to specify the meta-model for the new formalism and a tranformation into a "known" formalism (one that already has a simulator available, for example). We then obtain a tool to model in the new formalism, and are able to convert models in this formalism into the other for further processing. We have not only described formalisms commonly used in the simulation of dynamical systems, but we have also described formalisms such as Data Flow Diagrams and Structure Charts used for the structured description of software.

In the future, we plan to describe nother meta-meta-models in terms of the current one (Entity-Relationship), in particular UML class diagrams. For this purpose, relationships between classes such as *inheritance* should be described. Thanks to our meta-modelling approach, we will be able to describe different subclassing semantics and their relationship with subtyping. Furthermore, as the semantics of inheritance will be described at the meta-level, code can be generated in non-object-oriented languages. We also want to explore the automatic proof of behavioural equivalence between two models in different formalisms by bi-simulation. This may help in validating that a graph grammar for formalism transformation is correct. In a short term, we are working in a transformation of full Statecharts (with hierarchy) into Petri-Nets.

Acknowledgements

This paper has been partially sponsored by the Spanish Interdepartmental Commission of Science and Technology (CICYT), project number TEL1999-0181. Prof. Vangheluwe gratefully acknowledges partial support for this work by a National Sciences and Engineering Research Council of Canada (NSERC) Individual Research Grant. The authors would like to thank three anonymous referees for their useful comments.

References

1. AGG Home page: *http://tfs.cs.tu-berlin.de/agg/*
2. Blonstein, D., Fahmy, H., Grbavec, A. 1996. *Issues in the Practical Use of Graph Rewriting.* LNCS 1073, Springer, pp.38-55.
3. de Lara, J., Vangheluwe, H. 2002 *AToM^3: A Tool for Multi-Formalism Modelling and Meta-Modelling.* In European Conferences on Theory And Practice of Software Engineering ETAPS'02, Fundamental Approaches to Software Engineering (FASE). Lecture Notes in Computer Science 2306, pp.: 174 - 188. Springer-Verlag. AToM^3 home page: *http://moncs.cs.mcgill.ca/MSDL/research/projects/ATOM3.html*
4. DOME guide. *http://www.htc.honeywell.com/dome/*, Honeywell, 1999.
5. Dorr, H. 1995. *Efficient Graph Rewriting and its implementation.* LNCS 922, Springer.
6. Ebert, J., Sttenbach, R., Uhe, I. *Meta-CASE in Practice: a Case for KOGGE* Proceedings of the 9th International Conference, CAiSE'97, Barcelona. LNCS 1250, 203-216, Berlin, 1997. See KOGGE home page at: *http://www.uni-koblenz.de/∼ist/kogge.en.html*

7. Ehrig, H., Engels, G., Kreowski, H.-J., and Rozenberg, G. 1999. *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 2: Applications, Languages, and Tools*. World Scientific.
8. Finkelstein, A., Kramer, J., Goedickie, M. 1990. *ViewPoint Oriented Software Development* Proc, of the 3rd Workshop on Software Engineering and its Applications, Tolouse.
9. GRACE Home page: *http://www.informatik.uni-bremen.de/theorie/GRACEland/GRACEland.html*
10. Harel, D. On visual formalisms. *Comm. of the ACM*, 31(5):514–530, 1988.
11. Kelly, S., Lyytinen, K., Rossi, M. *MetaEdit+: A fully configurable Multi-User and Multi-Tool CASE and CAME Environment* In Advanced Information System Engineering; LNCS 1080. Berlin, Springer 1996. See MetaEdit+ Home page at: *http://www.MetaCase.com/*
12. King, P., Pooley, R. *Using UML to Derive Stochastic Petri Net Models* In Davies and Bradley Editors. UKPEW'99, Proc. 15th UK Performance Engineering Workshop. Bristol. pp.: 45-56.
13. Niskier, C., Maibaum, T., Schwabe, D. 1989 *A pluralistic Knowledge Based Approach to Software Specification* 2nd European Software Engineering Conference, LNCS 387, Springer, pp.:411-423.
14. OMG Home Page: *http://www.omg.org*
15. PROGRES home page: *http://www-i3.informatik.rwth-aachen.de/research/projects/progres/main.html*
16. Python home page: *http://www.python.org*
17. Sztipanovits, J., et al. 1995. *"MULTIGRAPH: An architecture for model-integrated computing"*. In ICECCS'95, pp. 361-368, Ft. Lauderdale, Florida, Nov. 1995.
18. Vangheluwe, H. *DEVS as a common denominator for multi-formalism hybrid systems modelling.* In *IEEE Symposium on Computer-Aided Control System Design*, pp.:129–134. IEEE Computer Society Press, September 2000.
19. Zave, P., Jackson, M. 1993. *Conjunction as Composition* ACM Transactions on Software Engineering and Methodology 2(4), 1993, 371-411.
20. Zeigler, B., Praehofer, H. and Kim, T.G. *Theory of Modelling and Simulation: Integrating Discrete Event and Continuous Complex Dynamic Systems*. Academic Press, 2nd ed., 2000.

Using Graph Transformation as the Semantical Model for Software Process Execution in the *APSEE* Environment

Carla A. Lima Reis[1,2], Rodrigo Quites Reis[1,2], Marcelo M. de Abreu[1], Heribert Schlebbe[3], and Daltro J. Nunes[1]

[1] PPGC-UFRGS, Federal University of Rio Grande do Sul, Porto Alegre, RS, Brazil
clima@inf.ufrgs.br, quites@computer.org, mmabreu@terra.com.br, daltro@inf.ufrgs.br
[2] Department of Informatics, Federal University of Pará, Belém, Pará, Brazil
[3] Faculty of Computer Science, University of Stuttgart, Stuttgart, Germany
schlebbe@informatik.uni-stuttgart.de

Abstract. Software Process Technology recently evolved to automate software process management by providing specialized languages and environments to control the human performance in software development activities. This paper presents *APSEE* as an executable visual language for software process modeling. The underlying meta-model was specified using graph grammars, which successfully derived a Java-based implementation. Finally, this experience is discussed with respect to the provided management mechanism to handle dynamic changes in executing processes.

1 Introduction

Process-Centered Software Engineering Environments (PSEEs) constitute a special kind of Software Engineering Environments (SEEs) emerged to support the rigorous definition of software processes, aiming to analyze, simulate, enact (or execute) and reuse processes definition [9]. Research groups and software industry developed a number of PSEEs during the last decade, including: EPOS, ADELE/TEMPO, SPADE, PEACE+ and E3 [5].

Complex software development involves activities performed by a number of developers with different technical skills. Automated control of activities execution is possible using a process model (a formal description of the software process which constitutes a partially ordered set). Many kinds of information must be integrated in this model in order to identify who, when, how and why the steps are performed during software development [17]. Thus, Process Modeling Languages (PMLs) were developed to describe and manipulate these models. Process models also provide a firm basis for process monitoring and simulation, where the behavior of a process can be analyzed in advance through dry runs (i.e. without real performers or tools). Finally, process models are also useful for verification purposes, where one can formally prove the properties of interest.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 254–269, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The vast majority of existing PMLs constitute diagrammatic visual languages, providing graphical representation for activities and ordering dependencies. Process manipulation is also often supported in a graphical way by existing editors which rely on modern user interface paradigms to display specialized process views. Process models are usually described to be automatically executed by a Process Engine (a key component in the architecture of modern PSEEs). Thus, its interpretation has to be in accordance to the language semantics [7].

Process Execution has been an important topic of research of both Software Engineering and Workflow Management areas. Some important topics considered in this context are the proposal of more flexible languages and execution mechanisms [8]. Therefore, to increase flexibility, the PML must provide a rich syntax and semantics, while supporting dynamic user interaction (which can interfere on the prescribed flow of execution). In spite of the importance of this topic, most of the existing PMLs do not provide formal syntax and semantics. Therefore, some critical process execution features are described informally, relying only on textual descriptions and complex, low-level programming languages.

A recent technique to support the formal specification of visual languages (VL) has been proposed by Bardohl [4] [2]. Within this approach, called GENGED, a VL is specified by an alphabet and a grammar where the abstract and concrete syntaxes are distinguished. According to Bardohl, the use of a graph grammar based approach provides a more natural and visual formalism for the specification of a VL.

This paper presents some of the key aspects of a software process execution mechanism called *APSEE*. *APSEE* is a uniform adaptive meta-model that integrates process modeling, visualization, instantiation and simulation facilities, which has been successfully experienced to describe a number of specialized instances (see [23], for examples). This text describes the main model characteristics and its underlying execution semantics—which was specified by using graph transformation [6] (combined with algebraic specification). This semantical model forms the basis for the development of a prototype implemented in the Java language.

The paper is organized as follows. Section 2 provides detailed motivations for the proposed meta-model and discusses the use of graph grammars as a semantical model for this approach. Section 3 presents the proposed *APSEE* model, including the *APSEE-PML* visual language specification that is used to describe an execution example. Section 4 discusses the related work and section 5 presents the final remarks.

2 Motivation

2.1 The Need for Flexibility in Software Process Execution

Software process models are peculiar, since they aim to describe the work of specialized people performing creative tasks. The current software development dynamics constitute an obstacle to foresee all the resulting product characteristics in advance. Software processes are people-oriented processes, being unpredictable by nature. Thus, process models are intrinsically susceptible to change,

and it implies that the underlying execution mechanism must handle incomplete process models and on-the-fly changes accordingly. In addition, the flow choices among alternate process paths can also be dynamically evaluated during execution, being based on previous activities' behavior or results. However, most of existing PSEEs are rigid, and do not provide an adequate level of flexibility in executing processes.

The lack of flexible support for dynamic process management has been recognized to be one of the main reasons for the low acceptance of existing PSEEs [8]. In addition, most of existing process execution mechanisms does not offer integrated support for the wide variety of tools and services to automate process analysis and improvement (e.g., simulation and reuse facilities). This clearly constitutes an obstacle to the wide use of Process Technology to automate software development in current software organizations; hence, flexibility is an important goal of this work.

2.2 Graph Grammars: Some Reasons for This Choice

The complexity of software process execution mechanism implies in the selection of a suitable formalism. The literature describes a number of experiences on using formal and semi-formal approaches with a variety degree of success (see for example, [11]).

Graph Grammars (GGs) emerged in the 1970s as a promising alternative to standard textual formal methods. GGs evolved as many methods and tools are available to support its use in a wide variety of domains [6][24]. Regarding Process Technology, a remarkable experience is described by Dynamite [28], which applied a graph rewriting technique to specify its underlying meta-model. The experience on its design was a valuable hint about the feasibility on using GGs to provide formal semantics for sophisticated process management mechanisms, in a graphical and intuitive way. The use of a graph grammar based approach can be justified by a number of factors listed below:

- In order to achieve a reasonable degree of integration of the services provided by a process management infrastructure, an **unified and complex meta-model** is required. The meta-model contains the interrelated information about the processes, its activities and dependencies and also about the software development organization, with the corresponding organizational and technological features. Additional requirements for modern process management mechanisms (e.g., simulation and reuse) also contribute to increase the meta-model complexity.
- Process execution is realized by **complex fine-grain transformations on process states** which affect the associated information (e.g., allocated resources and developers' agenda). Thus, the execution operations must guarantee the model consistency during these transformations. Also, there is a high degree of **parallelism** during process execution. The use of a formal method is needed to describe this behavior through an abstract and precise way. In addition, the use of a precise graphical notation has the potential to facilitate the prototype derivation.

- Some specific process execution requirements also constitute important factors to justify the use of a GG based approach. The management of **partially incomplete process descriptions and the required support for dynamic modifications** constitute challenging requirements that must be satisfied by an execution mechanism which, on the other hand, must also enforce models' consistency. Thus, the modeling of transformation rules can help the definition of consistent states, which are useful for inconsistency detection and management.
- The need to **integrate the execution to the organization model** also constitutes a key issue in the provided mechanism. Since both process execution and the organizational models constitute complex data structures, the graph transformation modeling paradigm can be useful to categorize specific occurrences, encapsulating important events that directly affects the associated organization components.

3 The Proposed *APSEE* Model

APSEE is part of a larger work where process technology is investigated in order to increase the level of automation on process management. Thus, this section presents the proposed model, the adopted formalism and the underlying infrastructure.

3.1 The Underlying *APSEE* Meta-model

While a complete description and evaluation of the *APSEE* meta-model is out of the scope of this paper, this section presents an overview of the underlying infrastructure w.r.t. its historical roots and design characteristics.

APSEE is a software framework to automate software process management that evolved from a Software Process Engine [13] originally proposed for the PROSOFT environment; hence *APSEE* is a PROSOFT component. PROSOFT is a formal object-based software development paradigm that is currently implemented as a Java-based SEE[1]. *APSEE* evolved to handle the dynamic and evolving characteristics of flexible software process management, including the support for the description of processes in multiple levels of detail, process simulation, improvement and reuse. Nowadays, *APSEE* is the underlying integration framework for a number of models to support process simulation, instantiation, execution, improvement and reuse.

The *APSEE* meta-model describes a process as a partially ordered collection of activities. A graphical PML is available, providing graphical representation for the proposed set of language constructs. A dynamic set of control mechanisms is available, delivering flexible synchronization on activity connections.

[1] PROSOFT is developed in cooperation between PPGC-UFRGS (Porto Alegre, Brazil) and Uni-Stuttgart (Germany) [25] within the Graphit Project supported by CNPq and DLR.

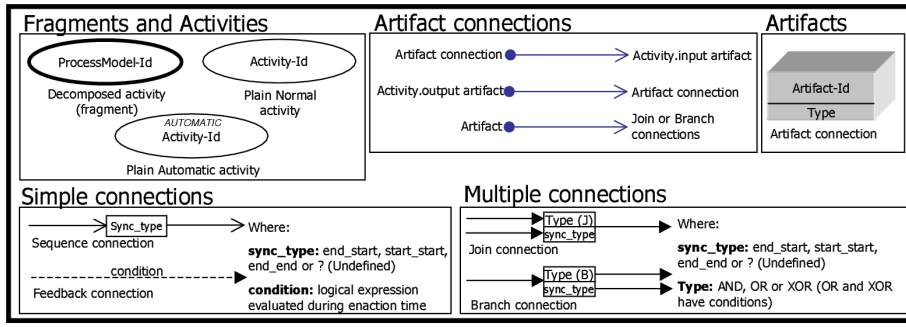


Fig. 1. Graphical representation for the main *APSEE* Process Modeling Language constructs.

The *APSEE* design was influenced by the need to provide support for explicit separation of concerns during process modeling. According to [20], a process designer deals with a multi-dimensional reality that mixes information about the Project (i.e., instantiation details like organization-specific roles and detailed schedule), the Environment (the set of software tools needed to support an activity) and the Process itself. The adopted model extends this proposal by implementing additional dimensions: People (including details about role types, agents, groups and abilities), Software (produced, consumed and transformed software artifacts), Resource (describing consumable, exclusive and shareable support resources) and Policies (syntactical and instantiation aspects that constitute crosscutting properties to process components [15], [22]). Each dimension describes independently defined components bound together during process modeling. The authors believe that this constitutes an important characteristic that distinguishes the proposed approach.

3.2 The *APSEE-PML* Visual Process Modeling Language

APSEE-PML is the graphical language used for process modeling in the *APSEE* environment. The underlying *APSEE* meta-model includes information about processes, which are visually represented through the language alphabet and interpreted by the execution mechanism. The visual symbols of the alphabet are showed in figure 1. These symbols are modeled as vertices of the graph schemes (presented in the following subsections). A number of specification methods were used to specify *APSEE* types, including the Algebraic-PROSOFT formalism [18] (for the definition of the required abstract data types) and graph grammars.

As noted by Bardohl [4], there are two levels of Visual Language descriptions: while the "abstract syntax level describes the logical meaning", the "concrete syntax level is used for the layout". Indeed, the combination of the abstract and concrete syntaxes is named Visual Syntax. With respect to the *APSEE-PML* specification, both the data type algebraic specification and a graph-schema mutually complement each other in order to completely describe the language's Visual Syntax. This text does not include the algebraic specification for the

required data types, which is presented in [16], while the corresponding graph-schema is discussed in the following section.

An *APSEE-PML* process model instance is essentially composed by activities and connections. Activities can be decomposed (by describing a new process model) or plain. The plain activity description includes information about the required roles, agents, and resources, the consumed and produced software, also describing relevant execution-related information (e.g., deadlines, states and execution events). Activities can also be automatic, that is, do not require people or resources.

The dependency between activities is modeled through Connections denoting data and control flow. **Simple** connections indicate a dependency between two activities. Simple Sequence (the start or end of the first activity enables the second one) and Simple Feedback (the end of the origin enables the re-activation of the target) are subtypes of simple connections. There are three types of possible dependencies for Simple Sequence connections: *end-start*, where the second activity depends on the end of the first one to start; *start-start*, where the second activity depends on the beginning of the first one to start; and *end-end*, where it is possible to stop the second activity execution only after the end of the first one [10].

Simple Feedback connections (between two activities) are associated to logical conditions that enable the re-activation of a previous executed activity. Thus, activity re-activation can be defined in advance (during modeling time), or manually included (execution time) in response to a specific abnormal event (re-activated activities are versioned, including the respective produced artifacts). **Multiple Branch** and **Join** connections are available in three types: AND, (inclusive-) OR and XOR. Finally, **artifact** connections denote produced, consumed and transformed software artifacts (denoting the data flow). Artifact connections are useful to explicitly define the use/consume of artifacts (documents, diagrams, code, and so on) relation between activities and/or multiple connections.

3.3 The Graph-Schema

Because of the inherent complexity of a process management mechanism, the *APSEE-PML* graph-schema is a large diagram interconnecting a number of components. Therefore, this section presents some specific parts of the complete graph-schema[2] The graph-schema symbols are represented according to the visual symbols presented in figure 1. Some data types that are not graphically available for the *APSEE-PML* user (i.e., do not have a graphical representation like the Process, Process Model, APSEE Manager, Agent and Group types) were included in the graph-schema since they play important "behind the scenes" tasks related to its execution.

Each symbol is associated to a name (at the top of the symbol) and a separated attribute part. Data attributes are typed elements that are used to rep-

[2] The interested reader can find a complete description of the graph-schema in [14].

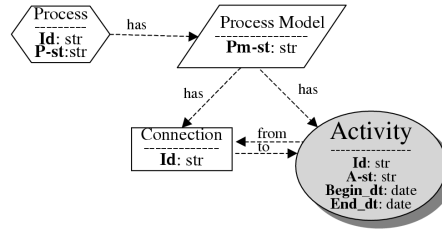


Fig. 2. The Graph-schema elements related to the composition of a process model.

resent and reason about the process state. The included arrows represent graph arcs, with specific meanings: *dashed* arrows denote relationship links, while the *solid* labeled arrows represent message calls (they are used in the interaction part of the graph schema presented in figure 5). For simplification purposes, all the specific message call relationships from the original model were replaced by generalizations.

The **essential** part of the graph-schema is presented in figure 2, where a software process has a process model composed by activities and connections. The *P-st* attribute (the *process* execution state) can assume the following values: *Not-Started*, *Enacting* or *Finished*. The *Pm-st* attribute (corresponding to the *process model* state, in figure 2), in turn, is associated to the state in the process model lifecycle: *Null* (in modeling time), *Requirements* (when only a textual description for the process is provided), *Abstract* (a process model with no links to specific organization' instances), *Instantiated* (the process model is ready for execution, including the people and resources involved), *Enacting*, *Finished* or *Mixed* (when the process model is composed by fragments in different process modeling states). Finally, the possible states for *Activity* instances (the *A-st* attribute) are: *Null* (in modeling time/no execution data yet), *Waiting* (when the associated dependency is not satisfied), *Ready* (when it is ready to start), *Active* (the agents are performing the activity), *Paused*, *Finished*, *Cancelled* (the activity was cancelled before starting), and *Failed* (the activity failed during its execution).

The **Connection** part of the graph-schema is presented in figure 3, where the ***Is-A*** relationship can be understood as inheritance, similar to the Object-Oriented paradigm. The ***Is-A*** relationship simplifies the graph-schema and the corresponding transformation rules, since attributes and relationships are inherited to the node's subtypes. For instance, when a rule's left side contains references to the main activity type (***Activity***), then it can be replaced by any of the associated sub-types (i.e., decomposed—*Activity_dec*, normal—*Activity_Normal* and automatic—*Automatic*). Analogous behavior happens to Multiple Connection and its subtypes.

Also in figure 3, the **From** and **To** edges describe the origin and target of the connection. The **To** relationship between connection and activity is redefined for Branch multiple connections (*Branch Or* and *Branch XOr*), since they require an additional condition for each successor. Other important attributes

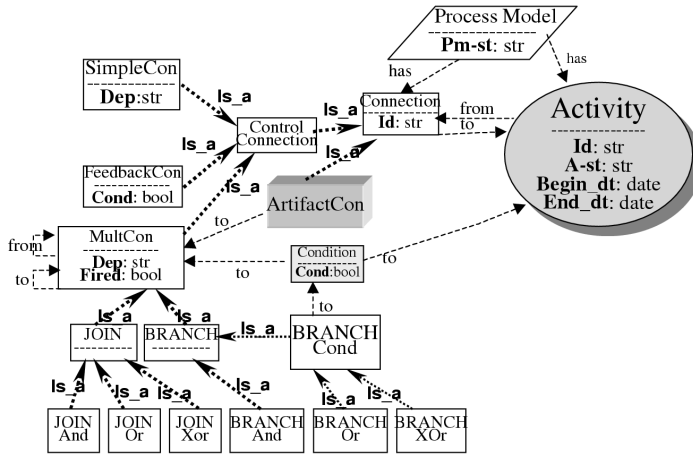


Fig. 3. Part of the *APSEE-PML* graph-schema representing the Connection model.

include: the dependency of simple and multiple connections (*dep*, which can assume *end-start*, *start-start* or *end-end*), and the condition for feedback connections and Branch-Cond successors (*cond*, defining a logical expression to be evaluated during execution). Simple (*SimpleCon*), feedback (*FeedbackCon*) and multiple connections (*MultCon*) were also included in figure 1.

The **Activity** part of the graph-schema is shown in figure 4. The ***Is-a*** relationships are used to inherit attributes and relationships from the *Activity* symbol. In the graph schema, the symbols for representing the activity types are distinguished to facilitate the development of GG rules. The decomposed activity (*Activity-Dec*) type is defined by a process model, which in turn, has connections and activities, but this relationship is transitively refined by the *Sub-task* relationship. *ActVersion* represents the activity versions created by the system when the activity has to be re-executed.

Concerning the interactive nature of process execution and the integration of both process and organizational models, the graph-schema has to include some additional components that are not graphically described in the provided graphical PML notation (e.g., the Agents, Agendas, Groups and Resources). The *APSEE-Manager* component is also included, since it controls the system interactions. Figure 5 presents the graph schema for the system **interaction** specification. This part is needed because some rules fire in response to user events (which can influence other information).

3.4 Rules for Generation and Execution of *APSEE-PML* Processes

After defining the graph-schema, it is necessary to declare how to create a valid language sentence and also how to execute a process instance. The GENGED approach for the generic description of visual languages [3] allows the definition

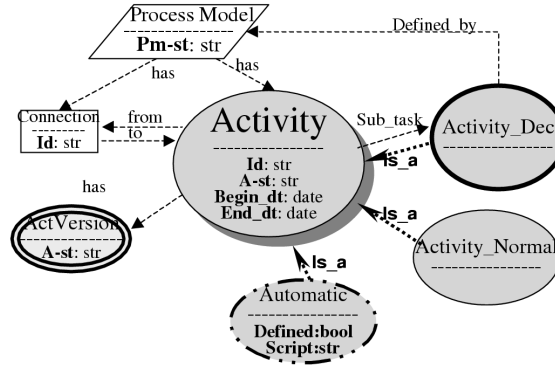


Fig. 4. Part of the *APSEE-PML* graph-schema representing the Activity model.

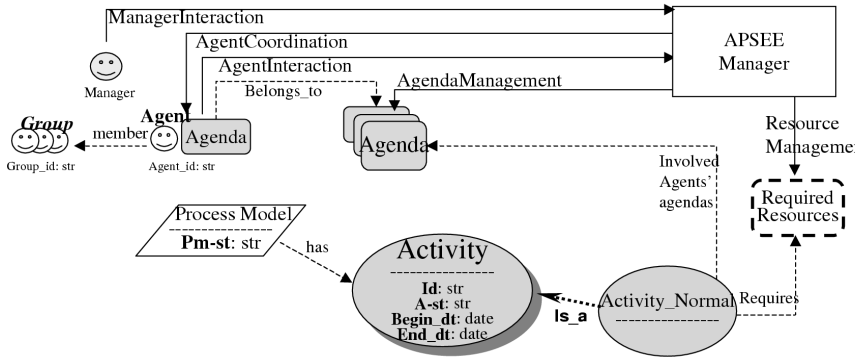


Fig. 5. The part of *APSEE-PML* specification representing the Interaction model.

of a visual grammar to generate the sentences of the language and a set of behavior rules for defining the transformations that changes the system and model states. Thus, the *APSEE-PML* visual grammar is responsible to create process model instances and also enforces their consistency when dynamic "on the fly" modifications are required, while behavior rules represent the execution semantics.

More than 160 rules were described to handle process execution completely; a separate set of more than 200 rules were described for the generation grammar (which includes consistency verifications and dynamic modifications). In some cases, Negative Application Conditions (NACs) were used to describe a condition that must be negative to enable the evaluation of a rule's left-side condition.

Figure 6 shows a generation rule example for inserting a simple connection between two generic activities. For preventing the inclusion of cycles (which would imply in a *deadlock*), one NAC states that the target activity must not have a

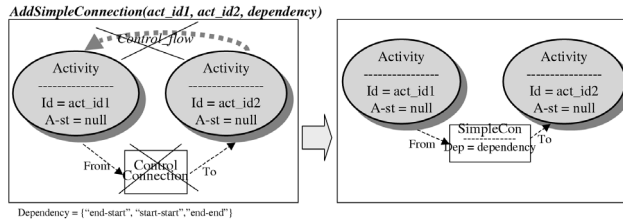


Fig. 6. Rule *AddSimpleConnection* for inclusion of a simple connection between activities.

direct/indirect control flow[3] to the first one (origin), while an additional NAC assumes that there is no control connection between the source and target activities. This rule is restricted to non-executed activities (i.e., *A-st* must assume the *null* value); the rules for dynamic modification are presented in section 3.5.

Strictly from the point of view of process-activity synchronization, the provided process execution strategy takes into account: the activities and their fragments (sub-activities); the associated connections dependencies, the resource availability; and the activities' pre- and post-conditions[4]. Figure 7 shows the behavior rules for the process execution start, which is explicitly asked by the manager. Figure 9 shows how the transition from *waiting* to *ready* of an activity state is done considering the dependencies stated in the NACs of figure 8.

3.5 Support for Dynamic Ad-hoc Changes

Differently from the majority of workflow approaches, where dynamic change is often restricted to the reconciliation of many executing cases to a new process, the proposed approach deals with changes for specific cases, which is commonly called ad-hoc changes [1]. The main problem with ad-hoc changes is to handle consistency in a specific executing process instance, avoiding changes that can introduce deadlocks or affect past occurred activities.

In this work we propose a set of rules for dealing with consistency checks in response to a process change request. Figure 10 shows rules that extend the previously presented rule in figure 6 w.r.t. execution-time details (i.e., both activities are executing), in a simplified way. Additional rules are described in [16].

3.6 Execution Example

An illustrative example using the current *APSEE* prototype is presented in figure 11. The left part of the figure presents an instantiated fictitious process sample

[3] Control flow detection rules are detailed in detail in [16]. Informally, they search for the occurrence of a direct or indirect path between activities composed by control connections.

[4] Current organization and processes statuses (i.e., process and product metrics) can also influence execution as described in [16].

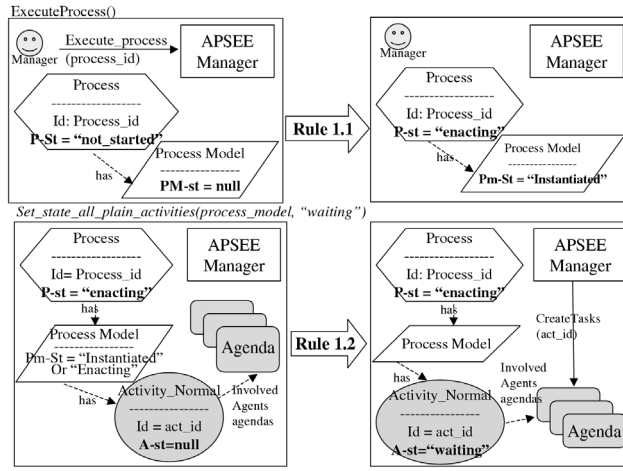


Fig. 7. Rules for the start of process execution and agenda initialization.

composed by four connected normal activities. The right side presents the same process under execution, just after its start, highlighting the activities that are ready to execute. This example needs the activation of rules such as the rules presented in figure 7 (rules 1.1 and 1.2). Rule 1.1 is fired as a consequence of the *execute_process* message call, while rule 1.2 includes the respective activities in the involved agents' agendas. For the application of these rules we consider the single-pushout approach [24] as the underlying formalism. The agenda window shown in the figure 12 depicts the information displayed for one involved agent.

4 Related Work

According to Garg [9], due to the lack of a widely accepted standard for the PSEE Architecture Design, it is difficult to compare and evaluate the experiences presented in the specialized literature. Existing PSEEs tend to concentrate on software process modeling and execution phases. This constitutes an obstacle to provide integrated support for software process management. For example, one immediate consequence is that organizational aspects are often treated in a simplistic way, which does not distinguish (passive) support resources from (active) agents (e.g., [12] and [21]).

From the point of view of dynamic control flow constructs, we can consider the TrigsFlow environment [10] as one of the related works. TrigsFlow provides a visual PML which supports *end-start*, *start-start* and *end-end* dependencies, through Branch and Join connectors. These constructs are semantically mapped to textual Event-Condition-Action rules that are interpreted by the execution mechanism. However, TrigsFlow does not allow the inclusion of conditions based on the current status of process execution and feedback connections are not supported, as stated in [10].

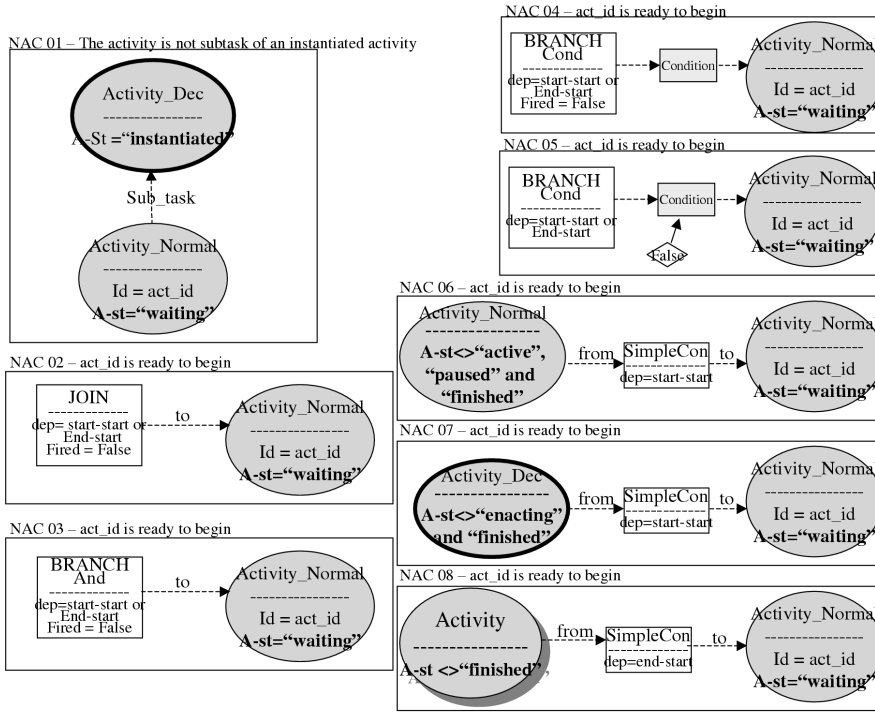


Fig. 8. Negative Application Conditions for *SearchForReadyActivities* Rule (next figure).

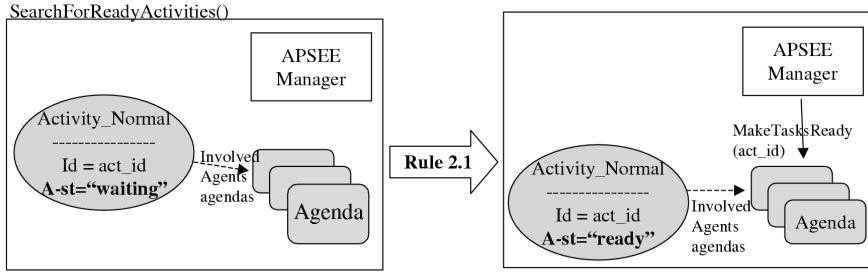


Fig. 9. Rule for transition of states of a normal activity (The NACs are presented in figure 8).

The previously cited Dynamite environment was developed as an application of graph rewriting techniques through the PROGRES language [11] [26]. However, this specification does not detail the user-system interaction, focusing only in the model's internal consistency by observing the data and control flows. The underlying Dynamite meta-model does not include organization

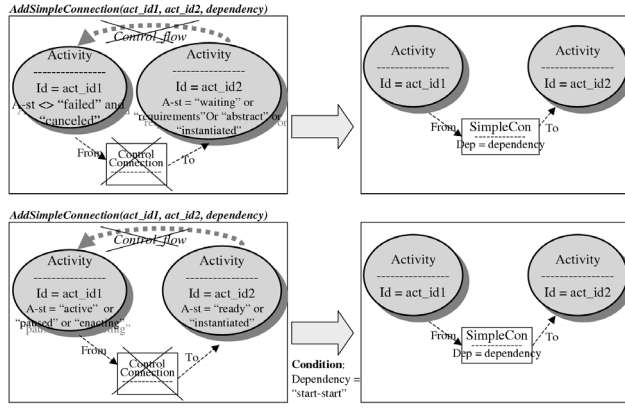


Fig. 10. Additional *AddSimpleConnection* rules to handle change during execution.

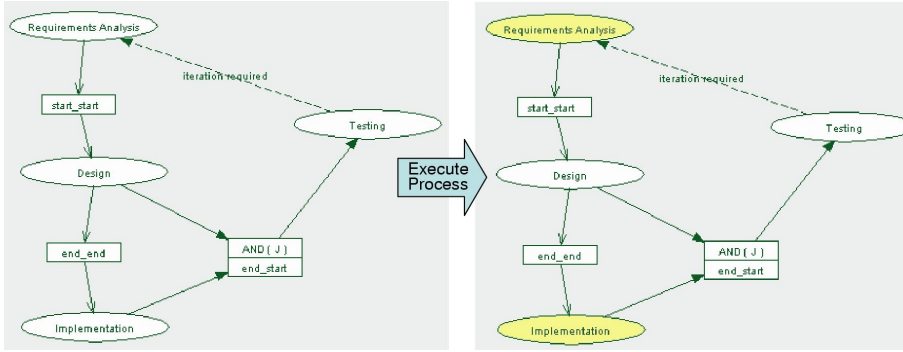


Fig. 11. Enactment example: "Process Execution start"

details, and sequence *end-start* constitute the only supported control connection construct (multiple connections are not allowed). Thus, the provided set of execution-related transformation rules is restricted to the definition of artifact use by activity instances. Feedback connections are supported in a limited sense by allowing a process manager to manually define which activities must be reactivated in an executing process instance.

Finally, the novel contribution of the proposed *APSEE* meta-model and execution mechanism is the establishment of an integration framework for a variety of services and tools. The proposed set of rules also treats agent and resources as first-order elements that decisively influence process performance and instantiation [15]. *APSEE* also advances by promoting a richer meta-model, which in one way can facilitate process modeling, and in the other hand dramatically increases the underlying language semantics complexity. In addition, the proposed *APSEE* semantics define a solution for a number of challenging problems faced by the current PSEE design.

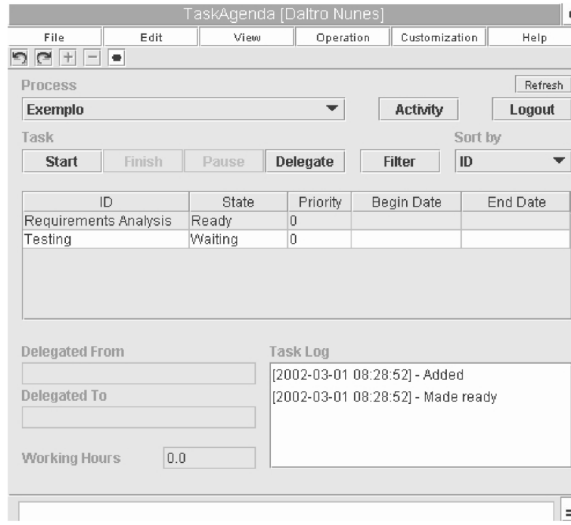


Fig. 12. The *APSEE* TaskAgenda with two activities assigned for the agent.

5 Final Remarks

This paper briefly presented the design of a flexible and integrated software process meta-model through the use of graph grammars (GG). The experience shows that GG constitute a viable alternative to the development of this complex software system and successfully guided the current system implementation as a component of PROSOFT environment. Hence, the experience can be faced as a complex and successful case study for GENGED approach [2] [4].

The proposed model was not tested in a specific graph transformation tool, like PROGRES [26] or AGG, described in [6]. The main goal was to use the GG approach as a formal specification design method combined with algebraic specification to derive implementation in a specific environment (PROSOFT). This experience showed the feasibility of this combination in the context of process execution.

The introduction of the **Is-a** relationship in the graph-schema greatly simplified the rules and decreased its quantity. The priority of evaluation was always given to the more specific case (e.g., a rule containing the *activity_normal* type is evaluated first than a rule with the *activity* type).

The definition of conflict-free transformation rules is a challenging task due to the high number of possible states for a given process model. However, the NACs helped us to minimize this problem by describing compact solutions for most of the cases.

Although the actual effect on using the cited formalisms on *APSEE* development was not empirically evaluated, the experience showed that developers with no previous knowledge on the graph grammar notation were able to successfully

use the GG-based specifications to derive the corresponding Java-PROSOFT methods in a shorter period of time compared to similar trials using standard algebraic methods (as described in [16]). In fact, the adopted formalism strongly benefits from the close relationship between the visual PML notation and the underlying execution mechanisms: during early system's implementation, it supported progressive enhancements to the PML notation in an adequate way.

The *APSEE* system has been applied to a number of general-purpose and domain-specific processes in small software developments (described in [16], [23] and [22]). The flexibility of the proposed model was obtained through the language structure (with many kinds of connections) and also through the dynamic modification rules. Many extensions were proposed to the *APSEE* framework, including an Agent-based Simulator [27], a Process Visualization and Monitoring framework and an integrated process reuse workbench [22]. Finally, it is important to note that the proposed meta-model and semantics intends to be a step towards increasing the flexibility and automation degrees in PSEEs, and this step can be useful for increasing software quality and reducing the product time to market.

References

1. Aalst, W.M.P. van der. Generic Workflow Models: How to handle dynamic change and capture management information. International Conference on Cooperative Information Systems (COOPIS′99). Edinburger, Scotland, Sept. 1999.
2. Bardohl, R.; Taentzer, G.; Minas, M.; Schürr, A. Application of Graph Transformation to Visual Languages. In: Handbook of Graph Grammars and Computing by Graph Transformations, Volume 2: Applications, Languages and Tools, World Scientific, 1999.
3. Bardohl, R.; Ehrig, H.; Ermel, C. Generic Description, Behavior and Animation of Visual Modeling Languages. Dagstuhl Seminar 00411 Semi-Formal and Formal Specification Techniques for Software Systems, October 9th-13th, 2000.
4. Bardohl, R. GenGED - Visual Definition of Visual Languages based on Algebraic Graph Transformation. PhD Thesis. Tec. Universität Berlin. Kovac Verlag, Hamburg, 2000.
5. Derniame, J.; Kaba, B.; Wastell, D. (Eds.). Software Process. Lecture Notes in Computer Science, Vol. 1500. Springer, 1999.
6. Ehrig, H.; Engels, G.; Kreowski, H-J.; Rozenberg, G. Handbook of Graph Grammars and Computing by Graph Transformation: Applications, Languages and Tools. Volume 2. World Scientific, Singapore, 1999.
7. Feiler, P.; Humphrey, W. Software Process Development and Enactment. 2nd International Conference on the Software Process. IEEE Press, Feb.1993.
8. Fuggetta, A. Software Process: A Roadmap. In: Finkelstein (Ed.), Future of Software Engineering. ACM Press 2000.
9. Garg, P. K.; Jazayeri, M. Process-Centered Software Engineering Environments. Los Alamitos: IEEE CS Press, 1996.
10. Kappel, G et al. Coordination in workflow management systems: A rule based approach. In Conen, W.; Neuman, G. (Eds.) Coordination technology for collaborative applications. Lecture Notes in Computer Science, Vol. 1364, 1997.

11. Krapp, C.A. An Adaptable Environment for the Management of Development Processes. Ph.D. Thesis. Aachen University. Germany, 1998.
12. Lerner, B. et al. Modeling and Managing Resource Utilization in Process, Workflow and Activity Coordination. Technical Report. University of Massachusetts Aug., 2000.
13. Lima Reis, C.A.; Reis, R.Q.; Nunes, D. J. Dynamic Software Process Manager for the PROSOFT Software Engineering Environment. Symposium on Software Technology (SoST'98). Buenos Aires: Sadio/European Software Institute. Sept. 1998.
14. Lima Reis, C.; Reis, R.; Abreu, M.; Nunes, D. APSEE: Um modelo formal e flexível para execução de processos de software. 5th Iberoamerican Workshop on Requirements Engineering and Software Environments (IDEAS 2002). La Habana, Cuba, April 2002.
15. Lima Reis, C.A. "Resource Instantiation Policies in Software Process Environments". *26th Annual International Computer Software and Applications Conference* (COMPSAC'02). Oxford, England, IEEE CS Press, August 2002.
16. Lima Reis, C.A. APSEE: A Flexible Software Process Enactment System. Ph.D. Thesis, Porto Alegre: PPGC-UFRGS, 2002 (to appear, in Portuguese).
17. Lonchamp, J. A Structured Conceptual and Terminological Framework for the Software Process Engineering. 2nd International Conference on the Software Process, IEEE CS, Mar.1993.
18. Nunes, D.J. The Algebraic-PROSOFT Software Development Paradigm. Technical Report. Porto Alegre: PPGC-UFRGS, 1996. (`http://www.inf.ufrgs.br/$\sim$prosoft`)
19. Paulk, M.; Weber, C.; Curtis, B. The Capability Maturity Model. Addison-Wesley Publishing Co., 1994.
20. Perry, D. E. Practical Issues in Process Reuse. 10th International Software Process Workshop, 10. (ISPW'10). IEEE Press, France, June 1996.
21. Podorozhny, R.et al. Modeling Resources for Activity Coordination and Scheduling. 3rd International Conference on Coordination Models and Languages. April 1999. Lecture Notes in Computer Science, Vol. 1594. Springer-Verlag.
22. Reis, R.Q. et al. Automatic Verification of Static Policies on Software Process Models. Annals of Software Engineering. Special Volume on Process-Based Software Engineering. V.14. Kluwer Academic Publishers, Oct.2002 (to appear).
23. Reis, R.Q. et al. Towards a Software Process Model to Support the Design of Mobile Computing Applications. 6th World Conference on Integrated Design and Process Technology, Pasadena (USA), 2002.
24. Rozenberg, G. (Ed.). Handbook on Graph Grammars: Foundations. Vol. 1. World Scientific, Singapore, 1997.
25. Schlebbe, H.; Schimpf, S. Reengineering of Prosoft in Java. Technical Report. Fakultät Informatik, Universität Stuttgart, Germany. Oct.1997.
26. Schürr, A.; Winter, A.J.and Zündorf, A. Graph Grammar Engineering with PROGRES. In W. Schäfer and P. Botella, (eds.) 5th European Software Engineering Conference, Spain, 1995. Lecture Notes in Computer Science, Vol. 989. Springer-Verlag.
27. Silva, F.; Lima Reis, C.; Reis, R.; Nunes, D. A Model for Software Process Simulation based on Cooperative Agents. 13th Brazilian Symposium on Software Engineering. Florianópolis, Oct. 2001.
28. Westfechtel, B. Models and Tools for Managing Development Processes. Lecture Notes in Computer Science, Vol. 1646. Springer-Verlag, 1999.

Graph-Based Reengineering of Telecommunication Systems

André Marburger and Bernhard Westfechtel

Aachen University of Technology,
Department of Computer Science III,
Ahornstrasse 55,
52074 Aachen, Germany
{marand,westfechtel}@cs.rwth-aachen.de
http://www-i3.informatik.rwth-aachen.de

Abstract. The E-CARES project addresses the reengineering of large and complex telecommunication systems. Within this project, graph-based reengineering tools are being developed which support not only the understanding of the static structure of the software system under study. In addition, they support the analysis and visualization of its dynamic behavior. The E-CARES prototype is based a programmed graph rewriting system from which the underlying application logic is generated. Furthermore, it makes use of a configurable framework for building the user interface. By means of reuse and generators, the effort of implementing the E-CARES prototype is reduced considerably.

1 Introduction

The E-CARES[1] research cooperation between Ericsson Eurolab Deutschland GmbH (EED) and Department of Computer Science III, RWTH Aachen, has been established to improve the reengineering of complex legacy telecommunication systems. It aims at developing methods, concepts, and tools to support the processes of understanding and restructuring of this special class of embedded systems. The subject of study is Ericsson's Mobile-service Switching Center (MSC) for GSM-networks called AXE10. The AXE10 software system comprises approximately 10 million lines of code spread over circa 1,000 executable units.

Maintenance of long-lived, large, and complex telecommunication systems is a challenging task. In the first place, maintenance requires understanding of the actual system. While design documents are available at Ericsson which do support understanding, these are informal descriptions which cannot be guaranteed to reflect the actual state of implementation. Therefore, tools are urgently needed which make the design of the actual system available on-line.

In E-CARES, a prototypical reengineering tool is being developed which addresses these needs. Currently, it assumes that the systems under study are written in PLEX, a proprietary programming language that is extensively used

[1] **E**ricsson **C**ommunication **AR**chitecture for **E**mbedded **S**ystems [1]

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 270–285, 2002.
© Springer-Verlag Berlin Heidelberg 2002

at Ericsson. However, we intend to support other programming languages — e.g., C — as well so that the prototype may handle multi-language systems. The prototype represents the structure of the studied system graphically at different levels of abstraction. Since telecommunication systems are highly dynamic, structural representations are not sufficient for program understanding. Therefore, tool support focuses particularly on understanding the behavior by visualizing and animating traces, constructing state diagrams, etc.

Internally, telecommunication systems are represented by various kinds of graphs. The structure of these graphs and the effects of graph operations are formally in PROGRES [2], a specification language which is based on programmed graph transformations. From the specification, code is generated which constitutes the core part of the application logic. In addition, the E-CARES prototype includes various parsers and scripts to process textual information, e.g., source code. At the user interface, E-CARES offers different kinds of textual, graphical, and tree views which are implemented with the help of UPGRADE [3], a framework for building graph-based applications.

The rest of this paper is structured as follows: Section 2 gives an overview of E-CARES. Section 3 describes the functionality of the E-CARES prototype. Section 4 is dedicated to the underlying graph-based specification. Related work is compared in Section 5. Section 6 concludes the paper.

2 Overview

2.1 GSM Basics

The MSCs are the "heart" of a GSM-Network as shown in the simplified sketch in Figure 1. A MSC provides the "services" a person can request by using a mobile phone, e.g. a simple phone call, a phone conference, or a data call, as well as additional infrastructure like authentication. Each MSC is supported by several Base Station Controllers (BSC), each of which controls a set of Base Station Transceivers (BTS). The interconnection of MSCs and the connection to other networks (e.g. public switched telecommunication networks) is provided by gateway MSCs (GMSC). In fact, the MSC is the most complex part of a GSM-network. A MSC consists of a mixture of hardware (e.g. switching boards) and software units. This makes the AXE10 a heterogenous system. In our research we have focused on the software part of this system.

2.2 Brief Introduction into PLEX

The executable units of the AXE10 software system are implemented in Ericsson's in-house programming language PLEX (**P**rogramming **L**anguage for **EX**changes). PLEX is an asynchronous concurrent real-time language designed for programming of telecommunication systems. The programming language has a *signaling paradigm* as the top execution level. That is, only events can trigger code execution. Events are programmed as signals.

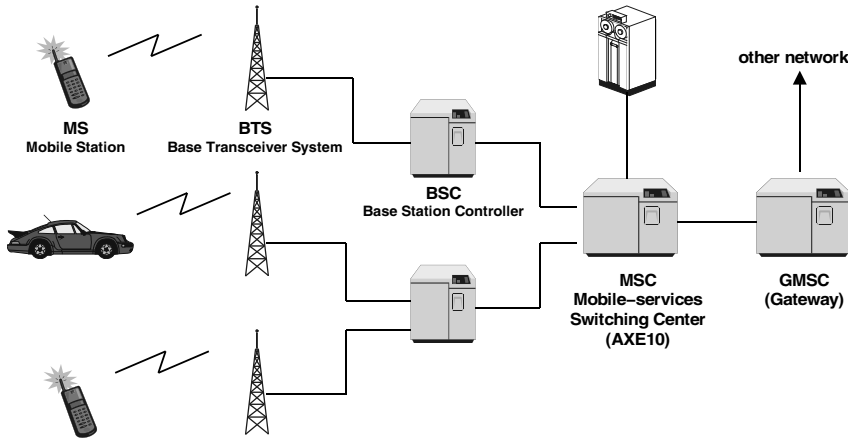


Fig. 1. Simplified sketch of a GSM network

Using the PLEX language a software system consists of one or more software units named *blocks*. A block is a quantity of code comprising several "functions" (executable parts of a block), which are compiled together and have data in common. Blocks have data encapsulation, that is, a block's data cannot be accessed by other blocks. Internally, a block is divided into multiple sectors (e.g., the declare sector for variable declarations and the program sector for the application logic). On a more abstract level, blocks are virtually grouped into *subsystems*, and subsystems are combined to so-called *application modules*.

Although PLEX does not support any further structuring within document files, we have identified some additional structuring via coding conventions in the program sector. At the beginning of the program sector all signal reception statements (*signal entries*) of a software unit are coded. After these signal entry points, a number of *labeled statement sequences* follows. The bottom part of the program sector consists of *subroutines*.

The control flow inside a unit is provided by `goto` and `call` statements. The `goto` statement is used to jump to a label of a labeled statement sequence. Subroutines are accessed by means of `call` statements. Both `goto` and `call` statements are parameter-less. That is, they affect only the control flow, but not the data flow.

Inter-block communication and data transport is provided by different kinds of *signals*. As every PLEX unit has data-encapsulation, signals are able to carry data. Therefore, signals may affect both the control flow and the data flow.

At run-time, every block can create several instances (processes). This again is not a feature of the PLEX programming language but achieved through implementation tricks and coding conventions. Therefore, these instances are managed by the block and not by the run-time environment.

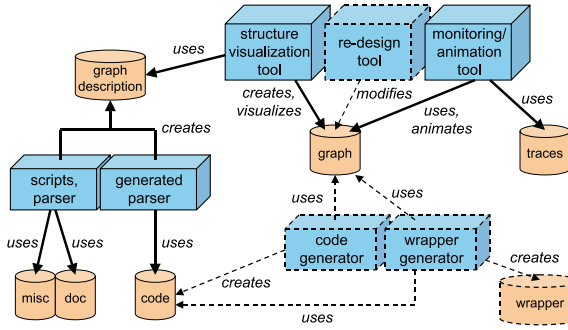


Fig. 2. Prototype architecture

2.3 E-CARES Prototype

In general, software reengineering comprises three phases [4]. The first phase is called *reverse engineering*. The reverse engineering is concerned with stepwise abstraction from source code and system comprehension. In the second phase (*restructure phase*), changes are performed on different levels of abstraction (e.g. code level, architecture level, etc.). The third phase is dedicated to *forward engineering* to introduce new parts to a system. So far, work has focused on the reverse engineering phase.

The basic "architecture" of the E-CARES prototype is outlined in Figure 2. We obtained three different sources of information for the static analysis of a PLEX system. The first one is the *source code* of the system. It is considered to be the core information as well as the most reliable one. Via code analysis (parsing) a number of structure documents is generated from the source code, one for each software unit. Theses structure documents form a kind of textual graph description.

The second and the third source of information are *miscellaneous documents* (e.g. product hierarchy description) and the systems *documentation*. As far as the information from these sources is computer processable, we use parsers and scripts to extract additional information. This additional information is stored in structure documents as well.

The structure *visualization tool* reads the textual graph descriptions that where created during code analysis etc. and creates an attributed graph. This attributed graph is the basis for any other investigation. It is extended stepwise with additional information e.g. resulting from sources two and three or interaction with human domain experts.

There are two possibilities to obtain "dynamic information", using an emulator or querying a running AXE10. In both cases, the result is a list of events plus additional information in a temporal order. This "dynamic information" is a fourth source of information. Traffic cases can be identified, fed into the *monitoring and animation tool*, and simulated there by traversing the structure graph. A trace simulated by the monitoring and animation creates a graph of

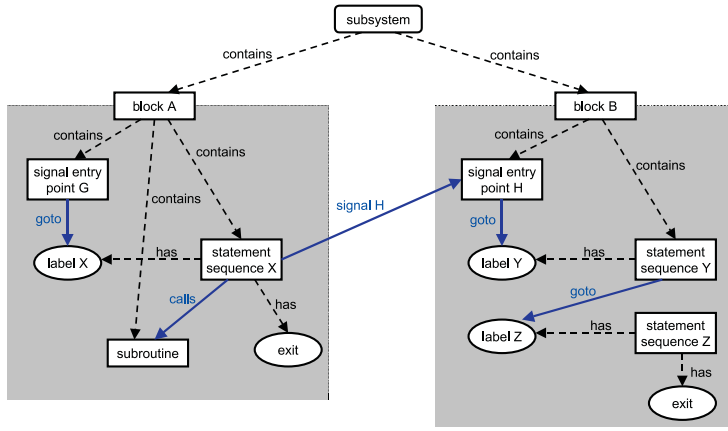


Fig. 3. Cut-out of a structure graph

interconnected block instances that is connected to the static structure graph. This helps software architects to identify components of a system that take part in a certain traffic case. Therefore, "dynamic information" allows us to analyze a system regarding different runtime conditions.

The dashed parts of Figure 2 represent planned extensions of the current reverse and re-engineering prototype. The re-design tool will be used to map structure graph elements to elements of a modeling language (e.g. ROOM [5], SDL [6]). This will result in an architecture graph that can be used to perform architectural changes to the AXE10 system. The *code generator* is planned to generated PLEX code according to changes in the structure graph and/or the architecture graph. The *wrapper generator* is intended to enable reuse of existing parts of the AXE10 system written in PLEX in a future switching system at Ericsson that is written in a different programming language, e.g. C++.

3 Functionality

3.1 Structure

The static structure of a PLEX program is represented internally by a *structure graph*. Figure 3 shows a small (and simplified) example illustrating the kind of information contained in a structure graph. In the example, there is a subsystem which contains two blocks A and B. The subgraphs for these blocks are created with the help of the PLEX parser. Initially, each block is parsed independently; subsequently, inter-block communication is added. The subgraph for a block — the block structure graph — contains nodes for signal entry points, labels, (contiguous) statement sequences, subroutines, exit statements, etc. Thus, the block structure graph shows which signals may be processed by the block, which statement sequences are executed to process these signals, which subroutines are

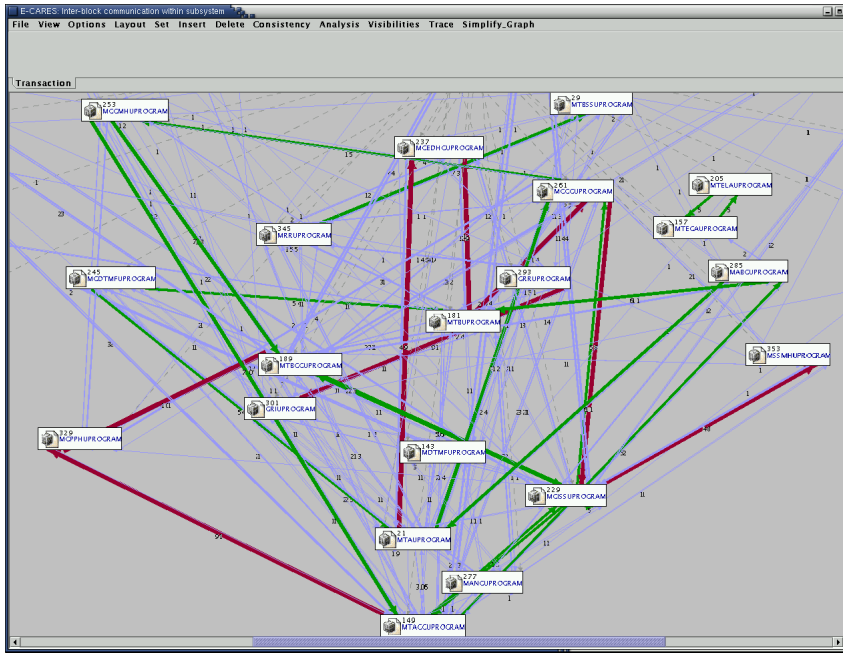


Fig. 4. Block Communication with weights

used for processing, etc. In addition, the block structure graph initially contains nodes representing outgoing signals. Subsequently, a global analysis is carried out to bind outgoing signals to receiving blocks based on name identity. In our example, a signal H is sent in the statement sequence X of block A. This signal is bound to the signal entry point of block B. From the signal edges between statement sequences and signal entry points, more coarse-grained communication edges may be derived (between blocks and eventually between subsystems).

Externally (at the user interface), the structure graph is represented by multiple *views*. The product hierarchy is displayed in a *tree view*. Furthermore, there is a variety of *graphical views* which display the structure graph at different levels of abstraction (internals of a block, block communication within a subsystem, communication between subsystems). The user may select among a set of different, customizable layout algorithms to arrange graphical representations in a meaningful way. He may also collapse and expand sets of nodes to adjust the level of detail. Graph elements representing code fragments are connected to the respective source code regions, which may be displayed on demand in *text views* (i.e., a code viewer is integrated into the E-CARES prototype).

The structure graph also contains quantitative data based on different *metrics*. For example, the graph view of Figure 4 shows the communication of blocks within one subsystem. Communication edges are weighted by the number of signal entry points acting as receivers of signals from the sending block. Numbers,

colors, and thickness are used to display communication weights. In this way, the user may distinguish "communication highways" from "secondary roads". We are currently experimenting with other metrics and their visual representation. For example, the size of a block node may vary according to the size of its source code, red-colored block nodes may indicate faulty blocks for which a large number of bugs had to be fixed, etc.

3.2 Behavior

Structural information alone is by no means sufficient to understand complex telecommunication systems. In fact, these systems are highly dynamic. Depending on the actual traffic, instances of blocks are created dynamically. For example, consider management of a call between two communication partners using mobile phones. A connection between these partners is established along which the data for the phone call are transmitted. This is achieved by creating a *link chain* of communicating block instances. Therefore, it is essential to understand how link chains are established, how they are managed, how the block instances involved in a link chain communicate which each other, etc.

To some extent, behavioral information may be created by static analysis. For example, this refers to the management of link chains. In AXE-10, there is a component which is responsible for managing link chains. A block participating in a link chain has to register with this component. To this end, a special statement is executed which is supported by the underlying runtime system. Therefore, a link chain is established as follows: A block instance indicates its participation in a link chain to the central manager component. Subsequently, it send signals to other block instances which may join the link chain. Thus, (potential) link chains may be recognized by static control flow analysis.

As another example of static analysis, consider the creation of *state diagrams* from the source code[2]. Reverse engineering of state diagrams relies on some coding assumptions (methods of use). In particular, it is assumed that the state of a block is captured in some state variable defined as an enumeration type. A variable of some enumeration type is considered a state variable if it contains elements obeying certain naming conventions (e.g., strings such as `Idle` or `Seized` indicate state variables). When a state variable has been recognized, the corresponding state diagram may be reconstructed by analyzing signal entries, conditions querying the state variable, assignments to the state variable, etc.

In addition, the E-CARES prototype makes use of *dynamic information* collected from the running system. During execution, *traces* are captured which log the actual communication between block instances at runtime. Traces are stored in text files which are parsed and visualized in a step-wise manner. Figure 5 illustrates that traces may be visualized in two ways. On the left, a trace is represented by a collaboration diagram. Each node of the diagram stands for a block instance; edges represent block communications (which are labeled with the signals they represent numbered in the order of occurrence in the trace).

[2] This feature has not been implemented yet.

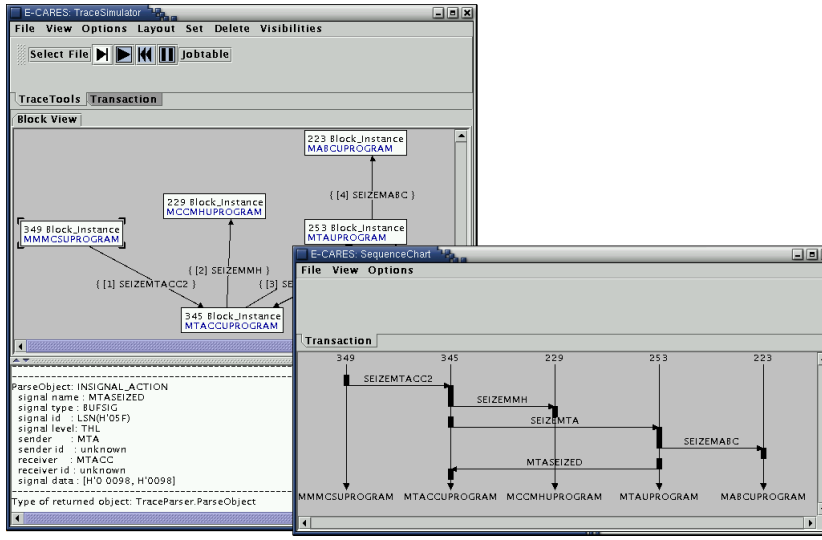


Fig. 5. Trace simulator

On the right, the same trace is shown as a sequence diagram. Each vertical line corresponds to a block instance; horizontal lines are used to display block communications. The lower left window shows the current step in the trace. In addition to the views shown in the screenshot, the user may view the job tables, which arrange signals waiting for processing into queues of different priorities. Internally, traces are connected to the structure graph. In this way, the user may switch between trace visualizations and the structure graph. In addition, some consistency checks may be performed (e.g., whether a block occurring in the trace is contained in the structure graph or whether there is a communication edge corresponding to a signal occurring in the trace).

4 Specification

The analysis of a software system, the *reverse engineering* part of a reengineering process, leads to a number of graphs or graph like descriptions of the systems structure, control flow, and data flow. We specify these graphs with the help of a programmed graph rewriting system. In the sequel, we discuss cut-outs of this specification, which comprises about 125 pages PROGRES code.

4.1 Graph Scheme

In Figure 6 the basic graph scheme we set up for our reengineering tools is described in a notation similar to UML class diagrams [7]. The node classes `FILE`, `ITEM`, and `ERROR` build the basis of the graph scheme. The node class `ERROR` has been introduced to be able to annotate errors with respect to structure or

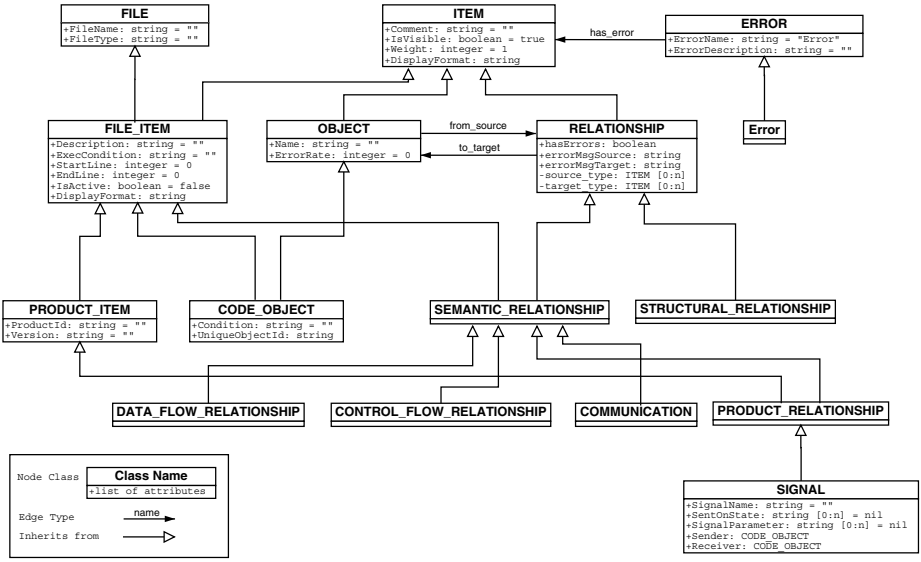


Fig. 6. Graph scheme

semantics of the graph or graph elements[3]. Nodes of type `ERROR` are connected to nodes of type `ITEM` via a `has_error` edge.

The two classes `OBJECT` and `RELATIONSHIP`, both derived from `ITEM`, divide the majority of nodes that occur in a structure graph into two groups. All node types derived from `OBJECT` are used to abstract from different kinds of fragments in the source code of an analyzed system. Node types derived from `RELATIONSHIP` serve to describe relationships between code fragments. Starting from the source object, a relationship is represented by the concatenation of a `from_source` edge, a node of the corresponding subclass of `RELATIONSHIP`, and an edge of type `to_target`. This *edge-node-edge* construct allows to simulate attributed edges which we are lacking. In the remainder, we sometimes use the term *edge* if we refer to such a construct.

The node classes `OBJECT` and `RELATIONSHIP` are further refined to be able to differ between different kinds of code fragments and relationships. For example, the structure graph we use combines structural information (`STRUCTURAL_RELATIONSHIP`), control flow information (`CONTROL_FLOW_RELATIONSHIP`), and data flow information (`DATA_FLOW_RELATIONSHIP`) in a single graph. This allows complex graph transformations and graph queries that utilize all three kinds of information at once. Some examples of these graph transformations are described in the following section.

[3] Sometimes it is necessary to allow intermediate inconsistencies in a graph, e.g. during manual changes to the graph. In other cases, the correctness of a structure graph depends on a user's focus and interest.

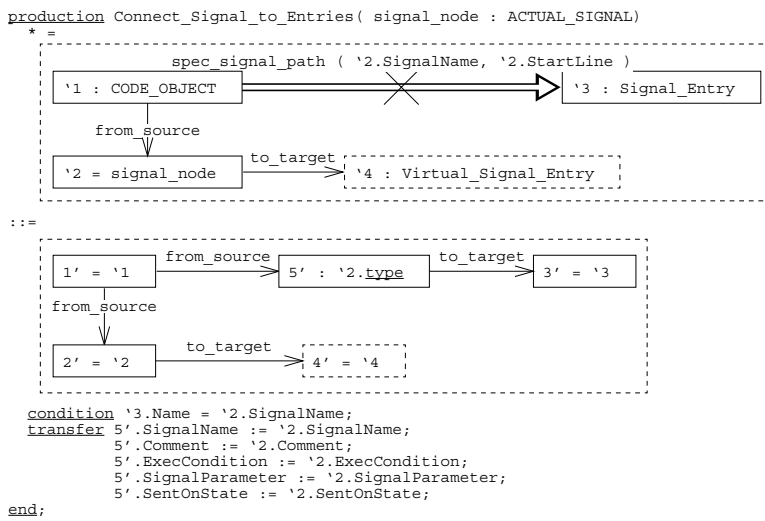


Fig. 7. Graph transformation to connect a signal to the corresponding signal entries

4.2 Graph Transformations

To illustrate the functionality of our specification, three examples of transformation rules are given below. Each rule is a representative of a different part of the tool specification. Figure 7 shows a transformation rule for connecting a signal to the corresponding signal entries in the current structure graph. This rule represents the structure graph creation part of the specification. The signal node to be connected is supplied as a parameter, which is used to match node 2 in the left hand side. Though a signal is supposed to be a relationship and therefore should be represented by an edge, we use edge-node-edge constructs to simulate attributed edges (see Section 4.1). A signal is connectable to a certain signal entry if there is no such signal "edge" between the sending node and that signal entry yet. This circumstance is expressed by the crossed out path expression between nodes 1 and 3. Node 4 is an optional node. This is illustrated by the dashed box. It is used as an intermediate anchor for signals just inserted. If a match of the left hand side of the rule in Figure 7 is found this match is replaced as indicated in the right hand side of the rule. That is, a new signal "edge" is inserted between the sending node 1 and the signal entry node 3. According to the graph scheme in Figure 6, this new signal "edge" consists of a `from_source` edge, a node of the same type as node 2 in the left hand side, and a `to_target` edge. All other nodes are replaced identically.

The condition statement below the right hand side of the transformation rule in Figure 7 formulates an additional requirement for a valid match of the left hand side. A match is valid if and only if the value of the `Name` attribute of node 3 if equal to the `SignalName` attribute of node 2.

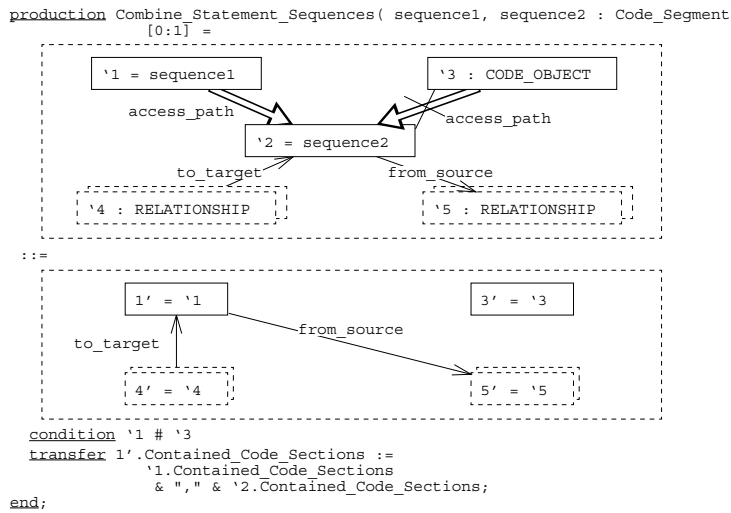


Fig. 8. Graph transformation to combine two statement sequences

The transfer of attributes from the original signal node 2 to the new signal node 5 that is indicated by the `transfer` statements at the bottom of the rule, makes node 5 a true copy of node 2.

Normally, the transformation rule in Figure 7 would only connect a signal to a single appropriate signal entry. To be more precise, the signal would be connected to an arbitrary appropriate signal entry specified by the first match of the left hand side. This is a matter of the PROGRES system. Matches in PROGRES are non-deterministic. But the star operator in the heading of the transformation rule forces the PROGRES execution machinery to find all valid matches for the left hand side and to perform the right hand side for all of them. Thus, the signal supplied by the transformations parameter is connected to all corresponding signal entries.

The transformation rule in Figure 8 is a representative of the structure graph simplification facility of our reengineering tool. This rule combines two statement sequences if the second one is only accessible from the first statement sequence. Again, the statement sequences in question are supplied as parameters of the transformation rule. These parameters are used to match nodes 1 and 2 of the left hand side. The sets of optional nodes represented by the node sets 4 and 5 (double dashed boxes) in the left hand side of the rule indicate that there might be other incoming and outgoing relationships to and from the second statement sequence (e.g. sent signals). The condition noted down below the left hand side of the transformation rule forces matches of node 3 to be different from node 1.

The right hand side of the transformation rule in Figure 8 shows that statement sequence 2 is deleted from the structure graph but its context is reconnected to the statement sequence that is represented by node 1. Additionally, the fact that the first statement sequence "consumed" the second one is indicated by the

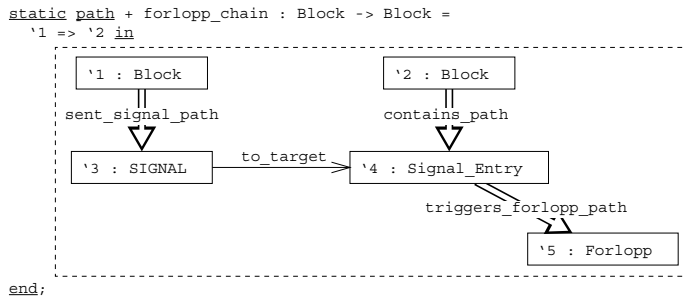


Fig. 9. Static path expression to establish forlopp link chains

`transfer` statement at the bottom of the rule. A special attribute collects the names of all statement sequences that are represented by node 1.

Figure 9 shows a different kind of graph transformation, a (static) path expression. Paths are simple graph queries that only consist of a left hand side. In case of static paths, there is a side effect that materializes all matches of the left hand side in form of edges between the source node and the target node of a path. Materialized paths can be visualized in our graph visualization tool. Therefore, we use static paths e.g. in the analysis part of the specification to gather additional information on a system that is not obtainable via code parsing.

The path expression in Figure 9 is used to detect blocks in the analyzed system that are inter-connected in a so-called *forlopp link chain* at runtime. Normally, the information on forlopp link chains is only available at runtime. But, using this path expression we are able to predict, which blocks of the AXE10 software system are able to take part in a certain forlopp chain. The path corresponding expression is defined to start and end at a node of type `Block`. Furthermore, it is defined that node 1 in the left hand side is the starting node of the path. Accordingly, node 2 is the target node of the path. Blocks 1 and 2 take part in the same forlopp chain, if there is a path from block 1 via a signal to a forlopp action in block 2.

In Figure 10 a transaction is presented that illustrates the connection of a trace to the structure graph as described in Section 3.2. It performs a single step of a trace file. That is, the transaction is used to transform a single signal action in a trace file into corresponding graph transformations. A signal action references a sending block instance, a receiving block instance, and a signal. Furthermore, each signal action is part of a specific trace. This information is again supplied as parameters of the transaction.

The transaction `Trace_Step_Block` is divided into three parts. The first parts checks whether the counterparts of the two block instances referenced by a signal action are already elements of the current structure graph. If not, the transaction `Add_or_Get_Node` inserts appropriate block nodes into a special part of the structure graph and annotates them as formerly missing. This enables a user on the one hand to use the trace animation tool without having an appropriate

```
transaction + Trace_Step_Block( traceId : string ; senderBlockName : string ;
                                senderInstanceId : string ; signalName : string ;
                                executionNumber : integer ; receiverBlockName : string ;
                                receiverInstanceId : string )
  [0:1] =
  use
    senderBlock, receiverBlock : Block
    (* ... and other local declarations *)
  do
    (* ------------------------ changes to structure graph ------------------------ *)
      Add_or_Get_Node ( Block, senderBlockName & "UPROGRAM", out senderBlock )
    & Add_or_Get_Node ( Block, receiverBlockName & "UPROGRAM", out receiverBlock )
    & Add_or_Extend_Block_Communication ( senderBlock, receiverBlock, signalName,
                                          out communicationEdge )
    & Activate ( senderBlock, receiverBlock, communicationEdge )
    (* -------------------------- changes to trace graph -------------------------- *)
    & Suspend_Working_Instance ( traceId, signalName )
    & Add_or_Get_Block_Instance ( traceId, senderBlock, senderInstanceId,
                                  out senderInstance )
    & Switch_ExecStatus_Of_Instance ( senderInstance, "WaitingOrSuspended" )
    & Add_or_Get_Block_Instance ( traceId, receiverBlock, receiverInstanceId,
                                  out receiverInstance )
    & Switch_ExecStatus_Of_Instance ( receiverInstance, "Working" )
    & Insert_Block_Instance_Communication ( traceId, senderInstance, receiverInstance,
                                            signalName, out instanceCommunicationEdge )
    & executionId := ("[" & string ( executionNumber ) & "] " & signalName)
    & instanceCommunicationEdge.SignalExecutionOrder :=
            (instanceCommunicationEdge.SignalExecutionOrder or executionId )
    (* ------------------------ changes to structure grap ------------------------ *)
    & currentSubsystem := senderBlock.=contained_in_subsystem_path=>
    & Propagate_Activation ( currentSubsystem )
    & receiverSubsystem := receiverBlock.=contained_in_subsystem_path=>
    & Propagate_Activation ( receiverSubsystem )
    & Propagate_Activation ( communicationEdge )
  end
end;
```

Fig. 10. Transaction to support animation of traces

structure graph. On the other hand, the separation of formerly missing parts in a special part of the structure graph gives quick access to this kind of inconsistencies and allows controlled corrections.

The second part of the transaction first switches the execution status of the block instance currently working to `suspended`. Next the node that corresponds to sending block instance is either added to or fetched from the current trace graph. The execution status of this block instance is either set to `waiting` or it is set to `suspended`. The value depends on the kind of signal action processed (combined signal or single signal). The corresponding information is obtained by querying the structure graph. This procedure is repeated for the receiving block instance node. But, its execution status is set to `working`. After the block instance communication edge between the sending and the receiving block instance has been inserted, an execution identifier for the current signal is calculated. This identifier is added to the attribute `SignalExecutionOrder` of the communication edge. Finally, in the third part of the transaction, the information about changes of the execution status of parts of the structure graph and trace graph is propagated from the block level to the more abstract levels (subsystems etc.).

5 Related Work

Reengineering of telecommunication systems has been studied only rarely so far. Much work has been performed in business applications written in Cobol [8–10]. Telecommunication systems differ from business applications with respect

to their dynamic nature, real-time requirements, their underlying computational paradigm, etc. Therefore, reengineering tools are required which specifically address these characteristics. Reengineering tools for business applications are usually based on a data-centered approach and support e.g. the porting of Cobol programs into an object-oriented architecture. In contrast, the E-CARES prototype follows a process-centered approach since it considers the active components of a telecommunication system and their communication behavior.

Graphs and graph rewriting systems play an essential role in E-CARES. In the following, we examine reengineering tools from a tool builder's perspective. We compare E-CARES to a set of other graph-based reengineering tools.

Rigi [11, 12] is an interactive toolkit with a graphical workbench which can be used for reengineering. Internally, a software system is represented by a set of entities and relationships. Externally, program understanding is supported by graph visualization techniques. Rigi is also used in the *Bauhaus* project [13, 14], whose aim is to develop methods and techniques for automatic or semi-automatic architecture recovery, and to explore languages to describe recovered architectures. In contrast to E-CARES, Rigi (and thus Bauhaus) is not based on a high-level specification language. Rather, graphs are accessed through a procedural interface, which makes coding of graph algorithms more painstaking.

The *GUPRO* project [15, 16] is concerned with the development of a generic environment for program understanding. GUPRO is based on the meta CASE tool KOGGE [17]. Internally, programs are represented as graphs. GUPRO offers parsers for several languages, including COBOL and C. Different kinds of analyzes may be specified with the help of a graph query language, but graph transformations cannot be specified in a declarative way. Moreover, GUPRO offers a textual user interface, while E-CARES provides graphical tools.

VARLET [18, 19] addresses the problem of database reengineering. In VARLET, a relational schema is transformed into an object-oriented one; triple graph grammars [20] provide the underlying theoretical foundation. VARLET tools have been specified and implemented with the help of PROGRES. E-CARES addresses a different application domain. Up to now, we only address reverse engineering, i.e., the system under study is not modified. For the re-design of telecommunication systems, we intend to use triple graph grammars, as well, as we did in a previous project in the domain of business applications [21].

FUJABA [22] is a CASE tool for UML which supports round-trip engineering from UML to Java and back again. While reverse engineering of class diagrams is well understood and implemented in a couple of commercial tools (e.g., Together and Rational Rose), Fujaba also recovers collaboration diagrams from Java source code [23]. In Fujaba, collaboration diagrams are not used as examples of computations; rather, they are executable and may be considered as generalizations of graph transformations. The E-CARES prototype is based on a different specification language (PROGRES), addresses PLEX rather than Java programs, and deals with the application domain of telecommunication systems, which has not been considered in Fujaba so far.

6 Conclusion

We have presented the E-CARES prototype for reengineering of telecommunication systems. The E-CARES prototype is based on a programmed graph rewriting system acting as an operational specification from which code is generated. In this paper, we have tried to provide some insight into this specification, which, however, cannot be presented without some context information. In general, we believe that reengineering is an application domain which is well-suited for graph rewriting. There are other reengineering tools which are also based on graphs, but most of them lack the support provided by a high-level specification language and force the tool developer to perform low-level programming.

So far, the implementation supports only reverse engineering, i.e., it aids in system understanding, but not yet in system restructuring. Current and future work addresses (among others) the following topics: reverse engineering of state diagrams, application of metrics, multi-language support (e.g., C or SDL in addition to PLEX), selection of an architectural description language which is independent of the used programming languages (ROOM seems to be a good candidate), re-design, and source code transformation.

References

1. Marburger, A., Herzberg, D.: E-CARES research project: Understanding complex legacy telecommunication systems. In: Proceedings of the 5th European Conference on Software Maintenance and Reengineering, Lisbon, Portugal, IEEE Computer Society Press (2001) 139–147
2. Schürr, A., Winter, A., Zündorf, A.: The PROGRES approach: Language and environment. [24] 487–550
3. Jäger, D.: Generating tools from graph-based specifications. Information Software and Technology **42** (2000) 129–140
4. Chikofsky, E.J., Cross II, J.H.: Reverse engineering and design recovery: A taxonomy. IEEE Software **7** (1990) 13–17
5. Selic, B., Gullekson, G., Ward, P.T.: Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc., Reading, Massachusetts (1994)
6. Ellsberger, J., Hogrefe, D., Sarma, A.: SDL - Formal Object-oriented Language for Communicating Systems. Prentice Hall (1997)
7. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified Modeling Language User Guide. Addison Wesley, Reading, Massachusetts (1999)
8. Sneed, H.M.: Migration of procedurally oriented COBOL programs in an object-oriented architecture. In: Proceedings of the International Conference on Software Maintenance, IEEE Computer Society Press (1992) 105–116
9. Cremer, K.: A tool supporting the re-design of legacy applications. In Nesi, P., Lehner, F., eds.: Proceedings of the Second Euromicro Conference on Software Maintenance and Reengineering, Florence, Italy, IEEE Computer Society Press (1998) 142–148
10. Canfora, G., Cimitile, A., Lucia, A.D., Lucca, G.D.: Decomposing legacy systems into objects: An eclectic approach. Information and Software Technology **43** (2001) 401–412

11. Müller, H.A., Wong, K., Tilley, S.R.: Understanding software systems using reverse engineering technology. In: The 62nd Congress of L'Association Canadienne Francaise pour l'Avancement des Scienes (ACFAS), Montreal, Canada (1994)
12. Storey, M.A.D., Müller, H.A., Wong, K.: Manipulating and documenting software structures. In: P. Eades and K. Zhang (eds.) Software Visualisation. Volume 7 of Series on Software Engineering and Knowledge Engineering., World Scientific Publishing (1996)
13. Girard, J.F., Koschke, R.: Finding components in a hierarchy of modules: A step towards architectural understanding. In: Proceedings of the International Conference on Software Maintenance 1997, Bari, Italy, IEEE Computer Society Press (1997) 58–65
14. Koschke, R.: Atomic Architectural Component Recovery for Program Understanding and Evolution. PhD thesis, Institute of Computer Science, University of Stuttgart, Stuttgart, Germany (2000)
15. Kullbach, B., Winter, A., Dahm, P., Ebert, J.: Program comprehension in multi-language systems. In: Proceedings of the 4th Working Conference on Reverse Engineering, Honolulu, Hawaii, IEEE Computer Society Press (1998)
16. Kamp, M.: Managing a multi-file, multi-language software repository for program comprehension tools - a generic approach. In: Proceedings of the 6th International Workshop on Program Comprehension, Ischia, Italy, IEEE Computer Society Press (1998) 64–71
17. Ebert, J., Süttenbach, R., Uhe, I.: Meta-CASE in practice: A case for KOGGE. In Olive, A., Pastor, J.A., eds.: Proceedings 9th International Conference on Advanced Information Systems Engineering (CAiSE'97). LNCS 1250, Barcelona, Spain, Springer-Verlag (1997) 203–216
18. Jahnke, J., Zündorf, A.: Applying graph transformations to database re-engineering. [24] 267–286
19. Jahnke, J.H.: Managing Uncertainty and Inconsistency in Database Reengineering Processes. PhD thesis, University of Paderborn, Department of Mathematics and Computer Science, Paderborn, Germany (1999)
20. Schürr, A.: Specification of graph translators with triple graph grammars. In Mayr, E., Schmidt, G., Tinhofer, G., eds.: Proceedings WG '94 Workshop on Graph-Theoretic Concepts in Computer Science. LNCS 903, Herrsching, Germany, Springer-Verlag (1994) 151–163
21. Cremer, K.: Graph-based reverse engineering and reengineering tools. In Nagl, M., Schürr, A., Münch, M., eds.: AGTIVE — Applications of Graph Transformations with Industrial Relevance. LNCS 1779, Castle Rolduc, The Netherlands, Springer-Verlag (1999) 95–110
22. Zündorf, A.: Rigorous Object-Oriented Development. PhD thesis, University of Paderborn, Paderborn, Germany (2002) Habilitation thesis.
23. Niere, J., Wadsack, J., Zündorf, A.: Recovering UML diagrams from Java code using patterns. In Jahnke, J., Ryan, C., eds.: Proceedings of the 2nd Workshop on Soft Computing Applied to Software Engineering, Twente, The Netherlands, Centre for Telematics and Information Technology (2001)
24. Ehrig, H., Engels, G., Kreowski, H.J., Rozenberg, G., eds.: Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools. Volume 2. World Scientific, Singapore (1999)

Formalising Behaviour Preserving Program Transformations

Tom Mens[1], Serge Demeyer[2], and Dirk Janssens[2]

[1] Postdoctoral Fellow of the Fund for Scientific Research - Flanders
Programming Technology Lab, Vrije Universiteit Brussel,
Pleinlaan 2, B-1050 Brussel, Belgium
tom.mens@vub.ac.be
[2] Department of Mathematics and Computer Science, Universiteit Antwerpen,
Universiteitsplein 1, B-2610 Antwerpen, Belgium
{serge.demeyer,dirk.janssens}@ua.ac.be

Abstract. The notion of refactoring —transforming the source-code of an object-oriented program without changing its external behaviour— has increased the need for a precise definition of refactorings and their properties. This paper introduces a graph representation of those aspects of the source code that should be preserved by a refactoring, and graph rewriting rules as a formal specification for the refactoring transformations themselves. To this aim, we use type graphs, forbidden subgraphs, embedding mechansims, negative application conditions and controlled graph rewriting. We show that it is feasible to reason about the effect of refactorings on object-oriented programs independently of the programming language being used. This is crucial for the next generation of refactoring tools.

1 Introduction

Refactorings are software transformations that restructure an object-oriented program while preserving its behaviour [1,2,3]. The key idea is to redistribute instance variables and methods across the class hierarchy in order to prepare the software for future extensions. If applied well, refactorings improve the design of software, make software easier to understand, help to find bugs, and help to program faster [1].

Although it is possible to refactor manually, tool support is considered crucial. Tools such as the *Refactoring Browser* support a semi-automatic approach [4], which is recently being adopted by industrial strength software development environments (e.g., VisualWorks, TogetherJ, JBuilder, Eclipse[1]). Other researchers demonstrated the feasibility of fully automated tools [5]; studied ways to make refactoring tools less dependent on the implementation language being used [6] and investigated refactoring in the context of a UML case-tool [7].

Despite the existence of such tools, the notion of *behaviour preservation* is poorly defined. This is mainly because most definitions of the behaviour of an

[1] see http://www.refactoring.com/ for an overview of refactoring tools

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 286–301, 2002.
© Springer-Verlag Berlin Heidelberg 2002

object concentrate on the run-time aspects while refactoring tools must necessarily restrict themselves to the static description as specified in the source-code. Refactoring tools typically rely on an abstract syntax tree representation of the source-code and assert pre- and postconditions before and after transforming the tree [8]. This representation contains details about the control flow of a program, which are largely irrelevant when specifying the effects of a refactoring on the program structure. Moreover, an abstract syntax tree is necessarily dependent on the programming language being used, while refactorings should be defined independently of the programming language [6].

For these reasons, a lightweight graph representation of the source code is probably more appropriate for studying refactorings. Such a representation should not bother with the details necessary for sophisticated data- and control flow analysis or type inferencing techniques, since these are necessarily dependent on the programming language. Instead it should focus on the core concepts present in any class-based object-oriented language –namely classes, methods and variables– and allow us to verify whether the relationships between them are preserved. Moreover, it should enable a transparant yet formal specification of the refactorings, as direct manipulations of the graph representation.

Therefore, this paper presents a *feasibility study* to see whether graph rewriting can be used to formalise what exactly is preserved when performing a refactoring. Section 2 introduces the concept of refactorings by means of a small motivating example and presents several types of behaviour that should be preserved. In Section 3 we introduce the typed graph representation of the source code and formalise two selected refactorings ("encapsulate field" and "pull up method") by graph rewriting productions. In Section 4 we use this formalisation to guarantee the preservation of well-formedness and certain types of behaviour. Section 5 concludes the paper with the lessons learned regarding the feasibility of graph rewriting as a formal basis for the behaviour preserved during refactoring.

2 Motivating Example

As a motivating example, this paper uses a simulation of a Local Area Network (LAN). The example has been used successfully by the Programming Technology Lab of the Vrije Universiteit Brussel and the Software Composition Group of the University of Berne to illustrate and teach good object-oriented design. The example is sufficiently simple for illustrative purposes, yet covers most of the interesting constructs of the object-oriented programming paradigm (inheritance, late binding, super calls, method overriding). It has been implemented in Java as well as Smalltalk. Moreover, the example follows an incremental development style and as such includes several typical refactorings. Thus, the example is sufficiently representative to serve as a basis for a feasability study.

2.1 Local Area Network Simulation

In the initial version there are 4 classes: *Packet*, *Node* and two subclasses *Workstation* and *PrintServer*. The idea is that all *Node* objects are linked to each other

in a token ring network (via the *nextNode* variable), and that they can *send* or *accept* a *Packet* object. *PrintServer* and *Workstation* refine the behaviour of *accept* (and perform a super call) to achieve specific behaviour for printing the *Packet* (lines 18–20) and avoiding endless cycling of the *Packet* (lines 26–28). A *Packet* object can only *originate* from a *WorkStation* object, and sequentially visits every *Node* object in the network until it reaches its *addressee* that accepts the *Packet*, or until it returns to its *originator* workstation (indicating that the *Packet* cannot be delivered).

Below is some sample Java code of the initial version where all constructor methods have been omitted due to space considerations. Although the code is in Java, other implementation languages could serve just as well, since we restrict ourselves to core object-oriented concepts only.

```
01 public class Node {
02   public String name;
03   public Node nextNode;
04   public void accept(Packet p) {
05     this.send(p); }
06   protected void send(Packet p) {
07     System.out.println(name + nextNode.name);
08     this.nextNode.accept(p); }
09   }

10 public class Packet {
11   public String contents;
12   public Node originator;
13   public Node addressee;
14   }

15 public class PrintServer extends Node {
16   public void print(Packet p) {
17     System.out.println(p.contents); }
18   public void accept(Packet p) {
19     if(p.addressee == this) this.print(p);
20     else super.accept(p); }
21   }

22 public class Workstation extends Node {
23   public void originate(Packet p) {
24     p.originator = this;
25     this.send(p); }
26   public void accept(Packet p) {
27     if(p.originator == this) System.err.println("no destination");
28     else super.accept(p); }
29   }
```

This initial version serves as the basis for a rudimentary LAN simulation. In subsequent versions, new functionality is incorporated incrementally and the object-oriented structure is refactored accordingly. First, logging behaviour is

added which results in an "extract method" refactoring ([1], p110) and an "encapsulate field" refactoring ([1], p206). Second, the *PrintServer* functionality is enhanced to distinguish between ASCII- and PostScript documents, which introduces complex conditionals and requires an "extract class" refactoring ([1], p149). The latter is actually a composite refactoring which creates a new intermediate superclass and then performs several "pull up field" ([1], 320) and "pull up method" ([1], p322) refactorings. Finally, a broadcast packet is added which again introduces complex conditionals, resolved by means of an "extract class", "extract method", "move method" ([1], p142) and "inline method" ([1], p117).

2.2 Selected Refactorings

Fowler's catalogue [1] lists seventy-two refactorings and since then many others have been discovered. Since the list of possible refactorings is infinite, it is impossible to prove that all of them preserve behaviour. However, refactoring theory and tools assume that there exist a finite set of *primitive refactorings*, which can then freely be combined into *composite refactorings*. For this feasability study, we restrict ourselves to two frequently used primitive refactorings, namely "encapsulate field" and "pull up method". The preconditions for these two object-oriented refactorings are quite typical, hence they may serve as representatives for the complete set of primitive refactorings.

EncapsulateField. Fowler [1] introduces the refactoring *EncapsulateField* as a way to encapsulate public variables by making them private and providing accessors. In other words, for each public variable a method is introduced for accessing ("getting") and updating ("setting") its value, and all direct references to the variable are replaced by dynamic calls (`this` messages) to these methods.
Precondition. Before creating the new accessing and updating methods on a class C, a refactoring tool should verify that no method with the same signature exists in any of C's ancestors and descendants, C included. Otherwise, the refactoring may accidentally override (or be overridden by) an existing method, and then it is possible that the behaviour is not preserved.

PullUpMethod. Fowler [1] introduces the refactoring *PullUpMethod* as a way to move similar methods in subclasses into a common superclass. This refactoring removes code duplication and increases code reuse by inheritance.
Precondition. When a method m with signature s is pulled up into a class C, it implies that all methods with signature s defined on the direct descendants of C are removed and replaced by a single occurrence of m now defined on C. However, a tool should verify that the method m does not refer to any variables defined in the subclass. Otherwise the pulled-up method would refer to an out-of-scope variable and then the transformed code would not compile. Also, no method with signature s may exist on C, because a method that is overwritten accidentally may break the existing behaviour.

2.3 Behaviour Preservation

Since we take a lightweight approach to source code refactoring here, we only look at notions of behaviour preservation that can be detected statically and do not rely on sophisticated data- and control-flow analysis or type inferencing techniques. The general idea is that, for each considered refactoring, one may catalog the types of behaviour that need to be preserved. For the feasibility study of this paper, we concentrate on three types of behaviour preservation that are important and non-trivial for the two selected refactorings. Section 4 discusses to which extent the selected refactorings satisfy these preservation properties:

A refactoring is *access preserving* if each method implementation accesses at least the same variables after the refactoring as it did before the refactoring. These variable accesses may occur transitively, by first calling a method that (directly or indirectly) accesses the variable. A refactoring is *update preserving* if each method implementation performs at least the same variable updates after the refactoring as it did before the refactoring. A refactoring is *call preserving* if each method implementation still performs at least the same method calls after the refactoring as it did before the refactoring.

Table 1. Node type set $\Sigma = \{C, B, V, S, P, E\}$ and edge type set $\Delta = \{l, i, m, t, p, e, \bullet, d, a, u\}$

node type	description	examples
C	**C**lass	*Node*, *Workstation*, *PrintServer*, *Packet*
B	method **B**ody	`System.out.println(p.contents)`
V	**V**ariable	*name*, *nextNode*, *contents*, *originator*
S	method **S**ignature	*accept*, *send*, *print*
P	formal **P**arameter of a message	*p*
E	**E**xpression in method body	`p.contents`
edge type	**description**	**examples**
$l: S \to B$	dynamic method **l**ookup	`accept(Packet p)` has 3 method bodies
$i: C \to C$	**i**nheritance	`class PrintServer extends Node`
$m: V \to C$	variable **m**embership	variable *name* belongs to *Node*
$B \to C$	method **m**embership	method *send* is implemented in *Node*
$t: P \to C$	message parameter **t**ype	`print(Packet p)`
$V \to C$	variable **t**ype	`String name`
$S \to C$	signature return **t**ype	`String getName()`
$p: S \to P$	formal **p**arameter	`send(Packet p)`
$E \to E$	actual **p**arameter	`System.out.println(nextNode.name)`
$e: B \to E$	**e**xpression in method body	`if (p.addressee==this) this.print(p); else super.accept(p);`
$\bullet: E \to E$	cascaded expression	`nextNode.accept(p)`
$d: E \to S$	**d**ynamic method call	`this.send(p)`
$a: E \to P$	parameter **a**ccess	`p.originator`
$E \to V$	variable **a**ccess	`p.originator`
$u: E \to V$	variable **u**pdate	`p.originator = this`

3 Formalising Refactoring by Graph Rewriting

3.1 Graph Notation

The graph representation of the source code is rather straightforward. Software entities (such as classes, variables, methods and method parameters) are represented by *nodes* whose label is a pair consisting of a name and a node type. For example, the class *Packet* is represented by a node with name *Packet* and type C (i.e., a C-node). The set $\Sigma = \{C, B, V, S, P, E\}$ of all possible node types is clarified in Table 1. Method bodies (B-nodes) have been separated from their signatures (S-nodes) to make it easier to model late binding and dynamic method lookup, where the same signature may have many possible implementations. B-nodes (method bodies) and P-nodes (formal parameters) have an empty name.

Relationships between software entities (such as membership, inheritance, method lookup, variable accesses and method calls) are represented by *edges* between the corresponding nodes. The label of an edge is simply the edge type. For example, the inheritance relationship between the classes *Workstation* and *Node* is represented by an edge with type i (i.e., an i-edge) between the C-nodes *Workstation* and *Node*. The set $\Delta = \{l, i, m, t, p, e, \bullet, d, a, u\}$ of all possible edge types is clarified in Table 1. For m-edges (membership), the label is often omitted in the figures.

Using this notation, an entire program can be represented by means of a single typed graph. Because the graph representation can become very large, we only display those parts of the graph that are relevant for the discussion. For example, Figure 1 only shows the graph representation of the static structure of the LAN simulation. (For B-nodes, a name has been put between parentheses to make the graph more readable.)

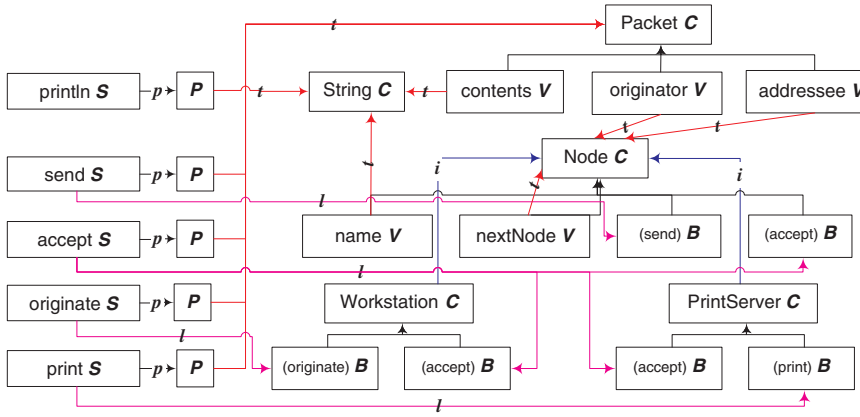


Fig. 1. Static structure of LAN simulation

A method body is represented by a structure consisting of E-nodes connected by edges that express information about dynamic method calls and variable invocations (accesses and updates). For example, Figure 2 represents the method bodies in class *Node*. The method body of *send* contains a sequence of two subexpressions, which is denoted by two e-edges from the B-node to two different E-nodes. The second subexpression `nextNode.accept(p)` is a cascaded method call (represented by a •-edge) consisting of a variable access (represented by an a-edge to the V-node labelled *nextNode*) followed by a dynamic method call with one parameter (represented by a d-edge and corresponding p-edge originating from the same E-node).

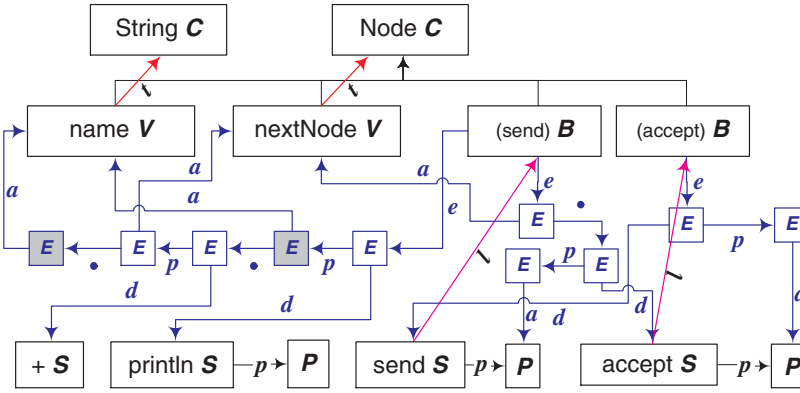


Fig. 2. Method bodies of class *Node*

We have deliberately kept the graph model very simple to make it as language independent as possible. It does not model Java-specific implementation features such as: Java interfaces; explicit references to `this`; constructor methods; control statements (such as `if`, `for`, etc.); Java modifiers (such as `static`, `abstract`, `protected`, `final`); inner classes; threads; exceptions.

3.2 Well-Formedness Constraints

On top of the graph representation, we need to impose constraints to guarantee that a graph is well-formed in the sense that it corresponds to a syntactically correct program. These *well-formedness constraints* are essential to fine-tune our graph notation to a particular programming language (in this case Java). We use two mechanisms to express these constraints: a *type graph* and *forbidden subgraphs*.

The notion of a *type graph* (or graph schema) is formally presented in [9,10]. Intuitively, a type graph is a meta-graph expressing restrictions on the instance graphs that are allowed. Formally, a graph is well-formed only if there exists a graph morphism into the type graph: a node mapping and edge mapping that

preserves sources, targets and labels. For node labels, only the type component, as introduced in Table 1, is taken into account. Figure 3 displays the type graph needed for our particular graph representation.

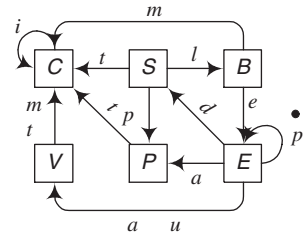


Fig. 3. Type Graph

Because type graphs alone are insufficient to express all constraints that we are interested in, we use a second mechanism called *forbidden subgraphs* to exclude illegal configurations in a graph. A graph G satisfies the constraint expressed by a forbidden subgraph F if there does not exist an injective graph morphism from F into G. To specify forbidden subgraphs we use *path expressions* of the form $a \xrightarrow[exp]{} b$, where a and b belong to the node type set Σ of Table 1, and exp is a regular expression over the edge type set Δ of Table 1. [2] A graph that contains more than one path expression denotes the set of all graphs obtained by substituting these edges in the obvious way.

Some typical examples of well-formedness constraints, needed to guarantee that a refactoring does not lead to an ill-formed graph, are given below:

WF-1 A variable cannot be defined in a class if there is already a variable with the same name in the inheritance hierarchy of this class (i.e., in the class itself or in an ancestor class or descendant class).

WF-2 A method with the same signature cannot be implemented twice in the same class.

WF-3 A method in a class cannot refer to variables that are defined in its descendant classes.

These constraints can be expressed by the forbidden subgraphs of Figure 4.[3] The forbidden subgraph for **WF-3** uses two path expressions. The expression $B \xrightarrow[?^*\{a|u\}]{} V$ denotes the set of all nonempty edge paths from a B-node to a V-node, where the last edge must have either type a or type u. It specifies the existence of a variable update or access from within the given method body.

[2] This is a simplified use of the path expressions provided by PROGRES [11].

[3] Nodes identified by a different number in the forbidden subgraph are mapped onto different nodes in the graph.

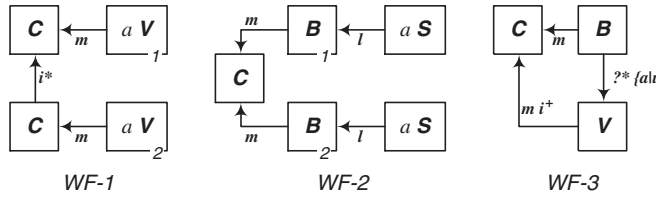
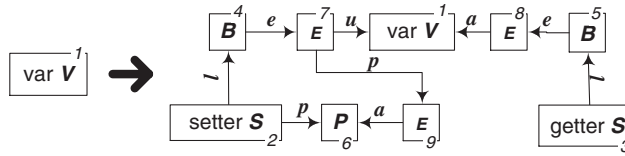


Fig. 4. Well-formedness constraints expressed as forbidden subgraphs

3.3 Graph Rewriting Productions

A *graph rewriting* is a transformation that takes an initial graph as input and transforms it into a result graph. This transformation occurs according to some predefined rules that are specified in a so-called *graph production*. Such a graph production is specified by means of a *left-hand side* (LHS) and a *right-hand side* (RHS). The LHS is used to specify which parts of the initial graph should be transformed, while the RHS specifies the result after the transformation. Often, a graph production can be applied to different parts of a graph, leading to different occurrences (or matches) of the graph production's LHS. In this paper, we use *parameterised graph productions* that contain variables for labels. Such parameterised productions can be instantiated by assigning concrete values to the variables.



incoming edges	outgoing edges
$(u,1) \rightarrow (d,2)$	$(m,1) \rightarrow (m,1),(m,4),(m,5)$
$(a,1) \rightarrow (d,3)$	$(t,1) \rightarrow (t,1),(t,3),(t,6)$

Fig. 5. Parameterised graph production *EncapsulateField(var,accessor,updater)* with embedding table

Figure 5 shows a production where *var*, *accessor* and *updater* are such variables. This production represents the refactoring *EncapsulateField*. LHS and RHS are separated by means of an arrow symbol. All nodes are numbered. Nodes that have a number occurring in both the LHS and the RHS are preserved by the rewriting (e.g., node 1). Nodes with numbers that only occur in the LHS are removed, and nodes with numbers that only occur in the RHS (e.g., nodes 2 and 3) are newly created.

In order to take into account the "context" in which the production is applied, consisting of the (sub)expressions referring to the variable that is encapsulated, the production is equipped with an *embedding mechanism* similar to the one of [12].[4] This embedding mechanism specifies how edges are redirected. *Incoming edges*, i.e., edges that have their target node in the LHS but not their source node, are redirected according to the incoming edges specification in Figure 5. For example, $(u, 1) \rightarrow (d, 2)$ means that each update of the variable *var* (represented by an incoming u-edge to node 1) is replaced by a dynamic method call to the updater method (represented by an incoming d-edge to node 2). *Outgoing edges* are treated similarly, using the outgoing edges specification in Figure 5. For example, $(m, 1) \rightarrow (m, 1), (m, 4), (m, 5)$ means that the method bodies (nodes 4 and 5) that correspond to the *accessor* and *updater* signature must be implemented in the same class as the one in which the variable *var* (node 1) was defined.

The parameterised production of Figure 5 may be viewed as a specification of an infinite set of productions in the algebraic approach to graph rewriting [14,15]. A concrete production can be obtained by filling in the variables of the parameterised graph production with concrete values and extending the LHS and RHS of the embedding-based production with a concrete context. Figure 6 shows the production instance *EncapsulateField(name,getName,setName)* that is applied in the context of the LAN example of Figure 2. The two gray E-nodes in Figure 6 are matched with the gray E-nodes of Figure 2.

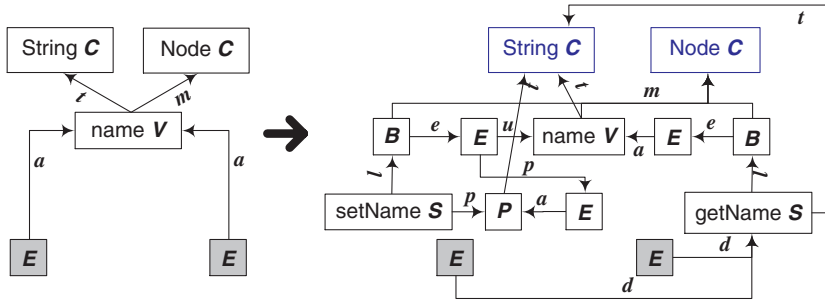


Fig. 6. Graph production instance *EncapsulateField(name,getName,setName)* obtained from the parameterised production of Figure 5

The second refactoring we want to express, *PullUpMethod(parent,child,name)*, moves the implementation of a method *name* in some *child* class to its *parent* class, and removes the implementations of the method *name* in all other children

[4] A similar but more visual mechanism is available in PROGRES [11] and Fujaba [13].

of *parent*. Expressing this refactoring by a single production –even a parameterised one with embedding mechanism– is problematic: changes may have to be made in *all* subclasses of *parent*, and the number of such subclasses is not a priori bounded. A way to cope with the problem is to control the order in which productions are applied. Mechanisms for *controlled graph rewriting* (also known as programmed or regulated graph rewriting) have been studied in, e.g., [16,17,18]. Using these mechanisms, *PullUpMethod* can be expressed by two parameterised productions P_1 and P_2. P_1 moves the method *name* one level higher in the inheritance hierarchy (i.e., from *child* to *parent*), and can only be applied if it is immediately followed by an application of P_2. This second production removes the implementation of method *name* from another subclass of *parent*, and has to be applied until there are no more occurrences of its LHS present. Both productions P_1 and P_2 are equipped with an identity embedding, i.e., all incoming and outgoing edges are preserved.

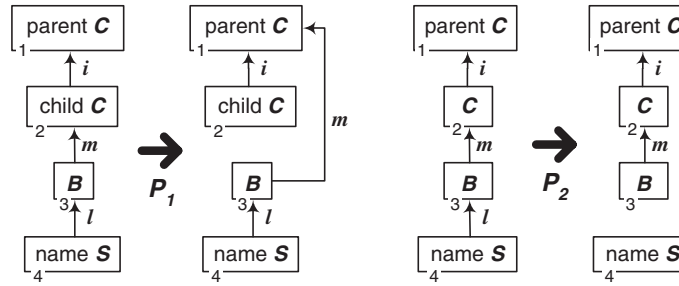


Fig. 7. Productions P_1 and P_2 for controlled graph rewriting *PullUpMethod(parent, child,name)*

4 Preservation of Refactoring Properties

In this section we combine the formalisation of refactorings of Subsection 3.3 with another graph rewriting technique, negative application conditions, to guarantee certain properties of the graphs that are derived. In particular, we consider certain types of behaviour preservation, refactoring preconditions and well-formedness constraints.

4.1 Preserving Behaviour

The types of behaviour preservation discussed in Subsection 2.3 can be expressed formally using the path expression notation of Subsection 3.2.

The path expression $B \xrightarrow[?^* a]{} V$ can be used to express the property of *access preservation*. It specifies all possible access paths from a method body (B-node) to a variable (V-node). The constraints imposed by the type graph of Figure 3 guarantee that there is only a single a-edge on such a path, and this edge is the

last one in the path. Access preservation means that, for each occurrence of $B \xrightarrow[?^*a]{} V$ in the initial graph to be rewritten, there is a corresponding occurrence of this path expression in the resulting graph, connecting the same B-node and V-node. Thus, the nodes that match B and V should not be removed or added by the graph production. In a similar way, we can express *update preservation* by means of the expression $B \xrightarrow[?^*u]{} V$.

Path expression $B \xrightarrow[?^*d]{} S \xrightarrow[l]{} B$ formalises the property of *call preservation*. For each method body (B-node) that performs a dynamic call (d-edge) to some signature (S-node) in the initial graph, there should still be a method lookup of the same method body in the resulting graph. In general, this requirement is not sufficient, since we also need to ensure that this method body has not been overridden by another one via late binding. Unfortunately, our current notation of path expressions cannot specify this constraint.

EncapsulateField. To show *update preservation* for *EncapsulateField*, it suffices to show that the preservation property expressed by $B \xrightarrow[?^*u]{} V$ is satisfied for each method body B that updates the variable *var* that is being encapsulated. It follows from the form of the graph production of Figure 5 that this is the case. This is illustrated in Figure 8, that shows how a direct update of *var* is replaced by a slightly longer path that still satisfies the property $B \xrightarrow[?^*u]{} V$. *Access preservation* can be shown in a similar way. *Call preservation* is trivial since the refactoring does not change any dynamic method calls or method bodies. (It does add new method signatures and method bodies, but this does not affect existing method calls.)



Fig. 8. Update preservation property of *EncapsulateField(var,accessor,updater)*



Fig. 9. Call preservation property of *PullUpMethod(parent,child,name)*

PullUpMethod. To show *call preservation* for *PullUpMethod*, we have to check whether the property $B \xrightarrow[?^*d]{} S \xrightarrow[l]{} B$ is preserved by the refactoring. Subexpression $B \xrightarrow[?^*d]{} S$ is trivially fulfilled. Subexpression $S \xrightarrow[l]{} B$ is illustrated in Figure 9: all implementations of the signature *name* by some method body are preserved, even if the class in which this body resides changes to *parent*. *Access preservation* and *update preservation* are trivial except for the case where an implementation of the signature *name* in some child class of *parent* accesses or updates a variable.

Since this implementation is removed (pulled up) by the refactoring, it is possible that variable accesses or updates in this method implementation are not preserved. Hence, the *PullUpMethod* refactoring is not necessarily access preserving or update preserving! This can be solved by adding extra preconditions for the refactoring, as shown in the next subsection.

4.2 Preserving Constraints

In general, the graph obtained by the application of a refactoring production has to satisfy several constraints. On the one hand, it has to satisfy well-formedness constraints, and on the other hand, refactorings are often subject to more specific constraints. For example, *EncapsulateField* does not cause accidental method overriding, i.e.,
the refactoring may not introduce new methods in a class if these methods are already defined in one of its descendants or ancestors (**RC-1**)

All these constraints can be expressed in a natural way as postconditions for the graph production. However, for efficiency reasons, it is desirable to transform these postconditions into *preconditions*. This avoids having to undo the refactoring if it turns out that the constraints are not met. Formally, preconditions can be defined by using graph rewriting with negative application conditions [19,20]. Formal proofs are available that specify how postconditions can be transformed into equivalent preconditions for the graph production [20].



Fig. 10. Negative preconditions for the *EncapsulateField* refactoring

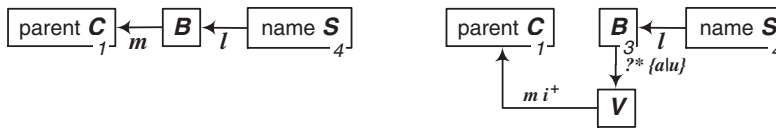


Fig. 11. Negative preconditions for the *PullUpMethod* refactoring

EncapsulateField. Figure 10 presents the negative preconditions needed in order for *EncapsulateField* to satisfy refactoring constraint **RC-1**. The conditions specify that no ancestor or descendant of the class containing *var* should have implemented a method with signature *updater*. Two similar negative application conditions are needed for the *accessor* method. Well-formedness constraint **WF-1** is satisfied since *EncapsulateField* does not introduce or move any variables, or change anything to the class hierarchy. Constraint **WF-2** is satisfied thanks to the preconditions of Figure 10, in the special case where i^* is the empty word.

Constraint **WF-3** is satisfied because *EncapsulateField* only introduces a new variable access and update to a variable that is defined by the class itself.

PullUpMethod. Figure 11 presents two negative preconditions for *PullUpMethod*, or more specifically, for subproduction P_1 of Figure 7. The condition on the left specifies that the method *name* to be pulled up should not yet be implemented in *parent*, and the condition on the right specifies that the implementation of the method to be pulled up should not refer to (i.e., access or update) variables outside the scope of the *parent*. *PullUpMethod* satisfies well-formedness constraint **WF-1** since it does not introduce or redirect any variables, or change anything to the class hierarchy. Constraint **WF-2** is satisfied thanks to the precondition on the left of Figure 11. Constraint **WF-3** is satisfied thanks to the precondition on the right of Figure 11.

5 Conclusion and Future Work

This paper presented a feasibility study concerning the use of graph rewriting as a formal specification for refactoring. Based on the specification of two refactorings ("encapsulate field" and "pull up method") we conclude that graph rewriting is a suitable formalism for specifying the effect of refactorings, because (i) graphs can be used as a language-independent representation of the source code; (ii) rewriting rules are a concise and precise way to specify the source-code transformations implied by a refactoring; (iii) the formalism allows us to prove that refactorings indeed preserve the behaviour that can be inferred directly from the source code.

In order to achieve our goal, we had to combine a number of existing graph rewriting mechanisms and techniques. Type graphs and forbidden subgraphs made it possible to express well-formedness constraints in a natural way. The specification of infinite sets of productions was facilitated by using parameterisation and an embedding mechanism. The application of graph productions was restricted by using negative application conditions and controlled graph rewriting. All these techniques are provided by state-of-the-art graph rewriting tools such as PROGRES [11] and Fujaba [13].

The two refactorings as well as the types of behaviour preservation we studied, are realistic and well documented. Because there are many other types of refactorings and behaviour preservation, further research is needed in order to find out whether the used graph representation needs to be modified, or whether other graph rewriting techniques should be used. Initial attempts to specify refactorings such as "move method" and "push down method" showed that it is difficult to manipulate nested structures in method bodies. Therefore, we need to resort to techniques (such as hierarchical graphs [10] and garbage collection) that tackle the inevitable complexity of large graphs.

A central topic in future work will be the investigation of methods to detect, for a given graph property and graph transformation, whether or not the property is preserved by the transformation. This requires further research into

formalisms to express such properties, and to use these in an automated refactoring tool.

Similar to what has been described in [6], we will also study the impact of language specific features (e.g., Java interfaces and exceptions) to verify whether it is possible to express refactorings independently of the programming language being used.

Because tool support is essential to cope with the complexity of refactoring productions, and to automate the checking of behaviour preservation, we are currently implementing our work in a graph rewriting tool. We have also implemented a translator to convert Java source code into our underlying graph representation.

In the longer run, we want to investigate combinations of refactorings. Roberts [8] has argued that primitive refactorings can be chained in sequences, where the preconditions of one refactoring are guaranteed by the postconditions of the previous ones. Moreover, in some refactoring sequences it is possible to change the order without changing the global effect. Such properties can be expressed using graph rewriting formalisms like the one in [20]. Refactoring tools may exploit these properties to optimise the number of program transformations, in much the same way as database tools perform query optimisations. This reduces the amount of analysis that must be performed by a tool, which is crucial for the performance and usability of refactoring tools.

References

1. Fowler, M.: Refactoring: Improving the Design of Existing Programs. Addison-Wesley (1999)
2. Opdyke, W.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign (1992)
3. Opdyke, W., Johnson, R.: Creating abstract superclasses by refactoring. In: Proc. ACM Computer Science Conference, ACM Press (1993) 66–73
4. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for Smalltalk. Theory and Practice of Object Systems **3** (1997) 253–263
5. Casais, E.: Automatic reorganization of object-oriented hierarchies: a case study. Object Oriented Systems **1** (1994) 95–115
6. Tichelaar, S.: Modeling Object-Oriented Software for Reverse Engineering and Refactoring. PhD thesis, University of Bern (2001)
7. Sunyé, G., Pollet, D., LeTraon, Y., Jézéquel, J.M.: Refactoring UML models. In: Proc. UML 2001. Volume 2185 of Lecture Notes in Computer Science., Springer-Verlag (2001) 134–138
8. Roberts, D.: Practical Analysis for Refactoring. PhD thesis, University of Illinois at Urbana-Champaign (1999)
9. Corradini, A., Ehrig, H., Löwe, M., Montanari, U., Padberg, J.: The category of typed graph grammars and their adjunction with categories of derivations. In: Proceedings 5th International Workshop on Graph Grammars and their Application to Computer Science. Volume 1073 of Lecture Notes in Computer Science., Springer-Verlag (1996) 56–74
10. Engels, G., Schürr, A.: Encapsulated hierarchical graphs, graph types and meta types. Electronic Notes in Theoretical Computer Science **2** (1995)

11. Schürr, A., Winter, A.J., Zündorf, A.: Graph grammar engineering with PROGRES. In Schäfer, W., Botella, P., eds.: Proc. European Conf. Software Engineering. Volume 989 of Lecture Notes in Computer Science., Springer-Verlag (1995) 219–234
12. Janssens, D., Mens, T.: Abstract semantics for ESM systems. Fundamenta Informaticae **26** (1996) 315–339
13. Niere, J., Zündorf, A.: Using Fujaba for the development of production control systems. In Nagl, M., Schürr, A., Münch, M., eds.: Proc. Int. Workshop Agtive 99. Volume 1779 of Lecture Notes in Computer Science., Springer-Verlag (2000) 181–191
14. Ehrig, H.: Introduction to the algebraic theory of graph grammars. In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 1–69
15. Löwe, M.: Algebraic approach to single-pushout graph transformation. Theoretical Computer Science **109** (1993) 181–224
16. Bunke, H.: Programmed graph grammars. In Claus, V., Ehrig, H., Rozenberg, G., eds.: Graph Grammars and Their Application to Computer Science and Biology. Volume 73 of Lecture Notes in Computer Science., Springer-Verlag (1979) 155–166
17. Kreowski, H.J., Kuske, S.: Graph transformation units and modules. Handbook of Graph Grammars and Computing by Graph Transformation **2** (1999) 607–638
18. Schürr, A.: Logic based programmed structure rewriting systems. Fundamenta Informaticae **26** (1996) 363–385
19. Habel, A., Heckel, R., Taentzer, G.: Graph grammars with negative application conditions. Fundamenta Informaticae **26** (1996) 287–313
20. Heckel, R.: Algebraic graph transformations with application conditions. Master's thesis, TU Berlin (1995)

Unparsing of Diagrams with DiaGen

Mark Minas and Frank Strüber

Lehrstuhl für Programmiersprachen, Universität Erlangen
Martensstr. 3, 91058 Erlangen, Germany
Mark.Minas@informatik.uni-erlangen.de

Abstract. When diagram editors are used as components of larger software systems they are not only used as an input facility for diagrams, but also for visualizing results or other data structures which are externally represented. Examples are UML tools which are used for re-engineering. This paper presents a generic unparsing approach which allows for creating diagrams from XML-coded external data structures. This approach has been integrated into the diagram editor generator *DiaGen*. It is based on an XSLT specification of the rule-based unparsing process which creates a hypergraph model of the resulting diagram.

1 Introduction

Diagrams are widely used in engineering and, in particular, in computer science. UML has demonstrated the success of diagrammatic languages, which is largely based on the availability of graphical tools, e.g., Rational Rose, Together etc. Their user interfaces mainly consist of *diagram editors* for such graphical languages. The user, e.g., may draw his class diagram or statechart with this diagram editor, and the tool then creates program code from such diagrams. Diagrams and, therefore, the diagram editor are used as an input facility for the UML tool. But they can be used for communication in the opposite direction, too: When used for re-engineering, class diagrams are created from existing code, i.e., diagrams are not created by the user interactively, but from some external data structure which has to be represented by the diagram with an equivalent meaning. In this example, the external data structure just consists of the existing code that has to be visualized by a class diagram.

The editor activity of creating diagrams from abstract representations is frequently termed *unparsing*, in contrast to *parsing* as a term for the more familiar task of analyzing diagrams that have been drawn by the user and creating external representations from those diagrams. While *unparsing* in the context of textual languages is generally used for (re-) creating text from its abstract syntax tree that has been created by *parsing* before (and similarly for visual languages; cf. Section 5), the term *unparsing* will be used in a more extended way in this paper: *Unparsing* describes any process that creates a diagram from any external data structure such that the resulting diagram's meaning is equivalent to the meaning of the original data structure.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 302–316, 2002.
© Springer-Verlag Berlin Heidelberg 2002

This paper describes an approach for adding diagram unparsing support to *DiaGen* which is a graph-transformation-based framework together with a generator for generating diagram editors from a formal specification of the diagram language. While previous work on *DiaGen* has mainly focused on generating diagram editors which support free-hand editing as well as syntax-directed editing and, therefore, translating diagrams into some external representation ("parsing" direction), there was no generator support for translating some external data structure into diagrams ("unparsing" direction). This gap has been closed by the approach that is presented in this paper.

In order to seamlessly fit into the existing environment, *DiaGen* has to provide an abstract way to specify external data structures and the process of translating them into diagrams. The approach of this paper tries to find a balance between a fixed translation process with well-defined specification capabilities for the diagram editor creator and generality of the approach. Therefore, an XML[1]-based approach has been chosen: External data structures have to be XML-coded, and the translation process had to be specified by XSLT, the *Extensible Stylesheet Language for Transformation*. XML-coding does not impose a strong restriction on data structures that can be used for external restrictions due to the generality of XML; XML-based exchange formats have been published and proposed already for many data structures (cf. [18]). On the other hand, XSLT is an expressive and yet a simple transformation language for specifying processing of XML-based information. Since there is already a standard on XSLT which makes it well-known to a large number of users and XSLT can be easily extended by so-called *extension functions* being implemented in Java, XSLT appears to be best suited for specifying unparsing diagrams from XML-based external data structures.

With this approach, *DiaGen* is able to translate any external data structure into a diagram with equivalent meaning as long data structures are XML-coded and there is an XSLT-based specification of the correspondence between the data structure and a diagram with an equivalent meaning. This has been proved. e.g., for statecharts, ladder diagrams and other application specific diagrams [16] that can be created from external XML data.

The next section gives a brief survey of *DiaGen* and how unparsing fits into the existing architecture of editors that have been generated with *DiaGen*. Section 3 then briefly introduces XSLT which is used in Section 4 for specifying the translation process of XML-based data into diagrams. Section 5 gives an overview of related work, and Section 6 concludes the paper.

2 *DiaGen*

DiaGen provides an environment for the rapid development of diagram editors [9]. This section outlines this environment, and how it is used for creating a diagram editor that is tailored to a specific diagram language. *DiaGen* can be used to create editors for a wide variety of diagram languages, e.g., finite automata, control flow diagrams, Nassi-Shneiderman diagrams, message sequence

[1] Extensible Markup Language

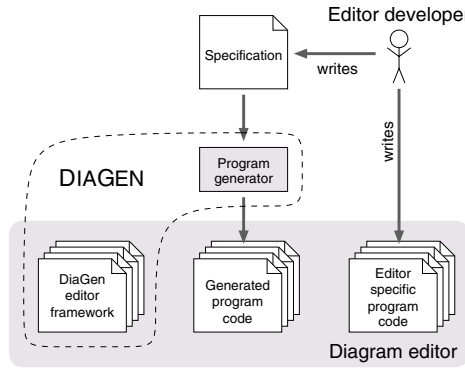


Fig. 1. Generating diagram editors with *DiaGen*.

charts, visual expression diagrams, sequential function charts, and ladder diagrams [12,7,8,6,9]. Actually we are not aware of a diagram language that cannot be specified so that it can be processed with *DiaGen*.

2.1 The *DiaGen* Structure

DiaGen is completely implemented in Java and consists of an *editor framework* and a *program generator*[2]. Fig. 1 illustrates the structure of *DiaGen*, and how it is used for developing diagram editors. The *DiaGen* editor framework, as a collection of Java classes, provides the generic functionality needed for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer supplies a textual *specification* for the syntax and semantics of a diagram language. Additional program code which is written "manually" can be supplied too. This may be necessary for the visual representation of special diagram components on the screen, and for processing objects of the problem domain, e.g., for semantic processing when the editor is used as a component in another software system. The specification is then translated into Java classes by the program generator.

The generated classes, together with the editor framework and the manually written code, implement an editor for the specified diagram language. This editor can be used as a stand-alone program, but also as a software component since the editor framework and the generated program code conform with the *JavaBeans* standard, the software component model for Java.

Diagram editors that have been developed using *DiaGen* (we call them "*DiaGen* editors" below) provide the following features:

- *DiaGen* editors always support *free-hand editing* so that the editor user can arbitrarily create, delete, and modify diagram components (e.g., circles and arrows which represent nodes and edges in trees), as with an off-the-shelf drawing tool. After each editing operation, the editor analyzes the "drawing"

[2] *DiaGen* is free software, available from `http://www2.cs.fau.de/DiaGen/`.

according to the syntax of the diagram language, and informs the user about syntax errors.
- Well-formed diagrams can be translated into a semantic or external representation, e.g., an XML document representing its structure. This process is driven by the syntactic analysis and makes use of the editor specific program code appearing in Fig. 1.
- The developer of a *DiaGen* editor may also specify compound operations for *syntax-directed editing*. Each of these operations is geared to modify the meaning of the diagram (e.g. for trees, a node could be deleted, together with its incoming edge and its whole sub-tree).
- *Automatic layout* is another optional feature of *DiaGen* editors. It is obligatory when syntax-directed editing operations are specified. The automatic layout mechanism adjusts the layout of a diagram (i.e., position, size etc. of its components) after any modification. Automatic layout also assists free-hand editing: After each layout modification by the user, the layout mechanism updates the diagram so that its structure remains unchanged. *DiaGen* offers constraints for specifying the layout mechanism in a declarative way [6], and a programming interface for plugging in other layout mechanisms.

2.2 The *DiaGen* Editor Architecture

Fig. 2 shows the structure that is common to all *DiaGen* editors and will be described below. Ovals are data structures, and rectangles represent functional components. Gray rectangles are parts of the editor framework which have been adjusted by the *DiaGen* program generator based on the specification of the diagram language provided by the developer. Arrows represent flow of information. If not labeled, information flow means reading resp. creating the corresponding data structures.

The editor supports free-hand editing by means of the included drawing tool which is part of the editor framework, but which has been adjusted by the program generator. With this drawing tool, the editor user can create, arrange and modify diagram components which are specific to the diagram language. Editor specific program code which has been supplied by the editor developer is responsible for the visual representation of these language specific components. Examples are circles or arrows in trees. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (position, size, etc.).

The sequence of processing steps which starts with the *modeler* and ends with *attribute evaluation* (cf. Fig. 2) realizes diagram analysis which is necessary for free-hand editing: The modeler first transforms the diagram into an internal model, the *hypergraph model* which is described in some detail below.

Diagram components (e.g., circles and arrows in trees) have *attachment areas*, i.e., the parts of the components that are allowed to connect to other components (e.g., start and end of an arrow). The most general and yet simple formal description of such a component is a hyperedge which connects to the nodes which represent the *attachment areas* of the diagram components.

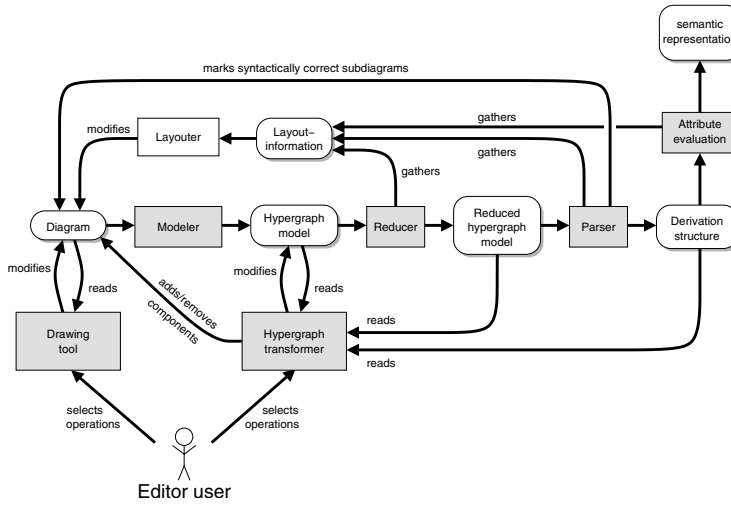


Fig. 2. Architecture of a diagram editor based on *DiaGen*.

These nodes and *component hyperedges* first make up an unconnected hypergraph. The *modeler* connects nodes by additional *relation edges* if the corresponding attachment areas are related in a specified way, which is described in the specification. The result of this scanning step is the *hypergraph model* (HGM) of the diagram. Fig. 4 shows the HGM of the tree shown in Fig. 3. (Hypergraph) nodes are represented by black dots, hyperedges either by gray arrows (relationships between attachment areas) or by rectangles (diagram components) that are connected to their nodes ("attachment areas") by thin lines. Small numbers are used to distinguish the connections of hyperedges. Unary `circle`-hyperedges represent tree nodes. Arrows are represented by `arrow`-hyperedges which visit two nodes, i.e., the ends of the arrow. Relation edges of type `inside` indicate that an arrow is connected to a (tree) node[3]. Special `leftOf`-edges are explained later; they are actually not required for free-hand editing.

The task of analyzing this hypergraph model is quite similar to familiar compiler techniques: The *reducer* – which corresponds to the *scanner* of a compiler – lexically analyzes the hypergraph model and creates a *reduced hypergraph model* which is then syntactically analyzed by the hypergraph *parser*. This processing step identifies maximal parts of diagram that are (syntactically) correct and provides visual feedback to the user by coloring each correct subdiagram with a different color. A correct diagram is thus entirely colored with just a single color, and errors are indicated by missing colors. Driven by the syntactic structure of each subdiagram and similar to the semantic analysis step of compilers, *attribute*

[3] Arrows are actually not required to start or end exactly at the border of a circle; their ends may rather be located inside of a circle in order to support convenient editing.

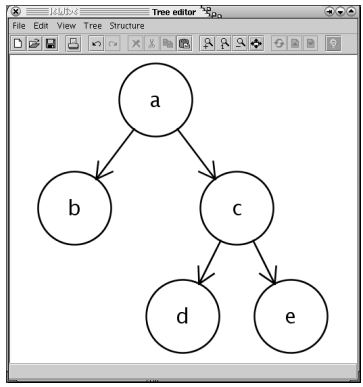


Fig. 3. A sample tree, drawn with an editor which has been generated with *DiaGen*.

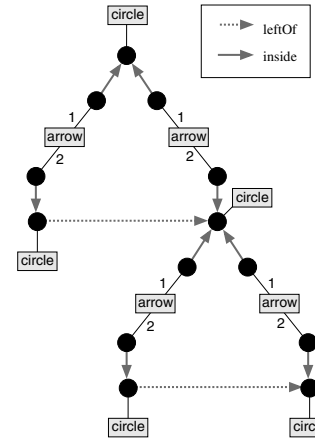


Fig. 4. The hypergraph model of the tree of Fig. 3. The hypergraph representation is explained in the text.

evaluation is then used to create a semantic (or just an external) representation for each of these subdiagrams.

The *layouter* modifies attributes of diagram components and thus the diagram layout by using information which has been gathered by the *reducer* and the *parser* or by *attribute evaluation*. The layouter is responsible for diagram beautification [2] and for maintaining the diagram structure after user interaction. The latter is necessary if, e.g., the user changes the size of a diagram component which forces other components to change their size, too. Of course, the layouter can be switched off if the diagram structure is changed deliberately.

The layouter is required for *syntax-directed editing*, too: Syntax-directed editing operations are specified by programmed hypergraph transformations (cf. [6,9]) which get selected by the user. Each editing operation is executed by the hypergraph transformer which directly modifies the hypergraph model primarily (see Fig. 2). Each operation may make use of information from the reduced hypergraph model or the derivation structure which has been generated during the previous diagram analysis step. The operation may affect the diagram, too. However, it is restricted to add or remove diagram components whose component hyperedges are added resp. removed by transformations to the HGM[4]. As illustrated in Fig. 2, hypergraph transformations do not affect the diagram directly. Instead, regular diagram analysis is started after the HGM has been processed by reducer, parser, and attribute evaluation. It is the layouter's task to adapt the diagram based on the information collected during this process. Please note that any component hyperedge that has been added or removed by

[4] Please note that this does not provide extra functionality; adding resp. removing diagram components is rather required such that the following diagram analysis step does not remove resp. add the component hyperedges again.

transformation of the HGM has been accompanied by adding resp. removing the corresponding diagram component.

Primarily modifying the HGM and depending on diagram analysis for syntax-directed editing may appear much too complicated to the reader. Is modifying the HGM by (programmed) hypergraph transformation and layouting the diagram directly without an additional diagram analysis step not better suited for that purpose? However, this would require additional layout rules which would define diagram layout by its HGM only. Additional structural information which can be created by syntactic analysis only would not be available for layout. Moreover, changing the diagram by editing operations without analyzing the resulting diagram with respect to the same criteria as they are used for free-hand editing might allow to create diagrams which would be detected as incorrect when trying to create them by free-hand editing. Instead, the described procedure allows the layouter to make use of syntactic information and avoids the diagram language to be extended by syntax-directed editing.

2.3 Unparsing of Diagrams

Syntax-directed editing has been described in some detail since the same procedure can be used to unparse diagrams, i.e., to create a diagram from an external data structure: Instead of creating a new diagram from scratch with a correct layout, we create just a hypergraph model, analyze this HGM as described in Section 2.2 and then use the regular layouter which then comes up with the corresponding diagram. Any diagram editor which supports syntax-directed editing has to provide automatic layout anyway. Unparsing of diagrams, therefore, can confine itself to creating the more abstract hypergraph model and leave the diagram layout to the existing layouter that relies on syntactic analysis of the hypergraph model.

However, this approach does have its shortcomings: If a diagram is translated to some external representation by diagram analysis and, later on, a diagram is unparsed from this external representation, the unparsed diagram is expected to be equivalent to the original diagram. However, such an equivalence is not guaranteed by the proposed approach. It rather depends on the way how hypergraph models are constructed from external data structures. If this process results in non-equivalent diagrams, this has to be considered as a specification bug or programming error. A well-known technique to avoid such errors or, at least, to reduce them is to use abstract specifications instead of explicit programming. The following sections present such a pattern-based approach to create hypergraphs from external data structures. In order to increase generality, existing techniques are used: External data structures are required to be based on XML, and pattern-based creating of hypergraphs is based on XSLT which is presented in the following section.

3 XSLT

XSLT, the *Extensible Stylesheet Language for Transformation*, which has originally been designed and which is mainly used for transforming XML documents

```
<?xml version="1.0"?>
<nodes>
<node text="a">
    <node text="b"/>
    <node text="c">
        <node text="d"/>
        <node text="e"/>
    </node>
</node>
</nodes>
```

Fig. 5. An XML representation of the tree of Fig. 3.

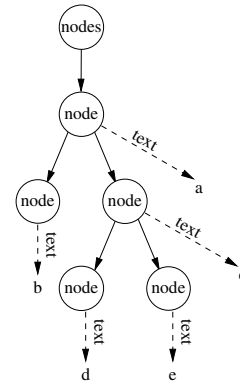


Fig. 6. DOM tree of the XML representation of Fig. 5.

into another representation, presumably another XML document or an HTML document, is actually a specification language for programmed tree transformations and, therefore, very well-suited for specifying transformations from external data structures into diagrams. Before presenting some details of XSLT, this section briefly introduces DOM, the tree data structure for XML documents which is subject to tree transformations, and XPath, a sub-language of XSLT for describing path expressions in DOM trees and thus patterns in tree transformations.

3.1 XML, DOM and XPath

The Extensible Markup Language (XML) is the universal format for structured documents and data on the Web. With XML being a subset of SGML, XML has some similarities to HTML. However in contrast to HTML, XML does not define a fixed set of markup tags nor does it give tags a certain meaning. This is left to specific applications and application domains. XML documents are text documents where their markup tags define a tree-like structure on them. As an example see Figure 5 and its tree structure in Figure 6. This tree contains special nodes which are linked by labeled edges: They represent attributes of tags together with their values[5]. This tree structure is termed *DOM tree* where DOM (Document Object Model) is a platform-independent interface to access and change documents providing a set of objects for representing XML (and HTML) documents. XML implementations like the JDK version 1.4 of Java 2 come with a so-called *DOM parser* which reads an XML document and automatically creates its DOM tree as a data structure in memory. In order to provide means for identifying patterns and for conveniently navigating in such trees, XSLT contains

[5] Because Figure 5 is an XML representation of the tree of Figure 3, Figure 6 is quite similar to the one in Figure 3. However, there is no such close similarity between the DOM tree and its represented data structure in general.

XPath as a sub-language. XPath allows to describe, starting at a certain DOM node, paths or sets of paths, e.g., by walking from a node to all of its children or to its following sibling. XPath will be considered in more detail in Section 4.

3.2 XSLT Processing

XSLT has been designed as a language that transforms a source XML document into a result stream, usually an XML document. The transformation is specified by an XSLT stylesheet being a well-formed XML document as well. A sample stylesheet is shown in Figure 7 which will be explained in Section 4. The stylesheet consists of several *template rules* that consist of a condition and an action. Conditions are XPath expressions which check whether a template rule may be applied to a sub-tree of the DOM tree. Given the XSLT stylesheet and the source DOM tree, the XSLT processor traverses the source DOM tree depth-first (in so-called "document order") and checks whether the condition of one of the template rules is satisfied for the current DOM tree node. If so, the action part of this rule is executed. If several rules do match, the first rule is selected. After execution of the rule action, the XSLT processor continues traversing the DOM tree without the sub-tree which just has been subject to the template rule application, i.e., the XSLT processor does not walk down this sub-tree. XSLT processing of this sub-tree has to be specified by the action part of the template rule which just has been applied. XSLT is thus a specification language for programmed tree-transformations, however without modifying the source DOM tree, but creating a new document by the actions of the template rules which are applied to the source tree. Actions are described next. There are yet more implicit rules which apply for "regular" XSLT processing. In the context of diagram unparsing, however, they may be neglected.

Actions are XML sub-documents which may contain XSLT markup elements, extension elements, and elements without special meaning to the XSLT processor. Elements are distinguished by their *XML namespace* (see Section 4). XSLT markup elements are used to control the XSLT processor, e.g., how the current sub-tree is transformed, to define or read variable values. Extension elements are handled outside the XSLT processor. The Java implementation of the XSLT processor (e.g., JDK version 1.4 of Java 2) allows for calling any Java method by such elements. Finally, elements without special meaning to the XSLT processor are just written to the result stream.

4 Unparsing Diagrams with XSLT

As Section 2.3 has described, unparsing diagrams in *DiaGen* editors consists of creating a hypergraph model from the external data structure. External data structures have to be XML-coded, and the transformation process is specified by an XSLT stylesheet. This approach is described in some detail for a fairly simple example – unparsing trees from an XML document – which hardly justifies this approach; programming unparsing process manually would be comparably

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
  xmlns:make="diagen.editor.sample.cltree.XSLaux"
  xmlns:diagen="diagen.editor.graph.XSLTrafo">

<xsl:template match="node">
  <!-- Create circle component for this node -->
  <xsl:value-of select="make:circle( diagen:attach(., 'c'),
                                     @text )"/>

  <!-- Add arrow from parent node -->
  <xsl:if test="name(..)='node'">
    <xsl:value-of
         select="make:arrow( diagen:attach(., 'from'),
                             diagen:attach(., 'to'))"/>
    <xsl:value-of
         select="diagen:addRelation( 'inside',
                                     diagen:attach(., 'from'),
                                     diagen:attach(.., 'c') )"/>
    <xsl:value-of
         select="diagen:addRelation( 'inside',
                                     diagen:attach(., 'to'),
                                     diagen:attach(., 'c') )"/>
  </xsl:if>

  <!-- Add ordering edge to following sibling node -->
  <xsl:variable name="next" select="following-sibling::*"/>
  <xsl:if test="$next">
    <xsl:value-of
         select="diagen:addTmpLink( 'leftOf',
                                    diagen:attach(., 'c'),
                                    diagen:attach($next, 'c') )"/>
  </xsl:if>
  <xsl:apply-templates/>
</xsl:template>

</xsl:stylesheet>
```

Fig. 7. XSLT stylesheet transforming the semantic representation of a diagram into its hypergraph model of DiaGen.

simple. Due to its simplicity, however, the complete unparsing specification fits on a single page. At the end of this section, more demanding examples which demonstrate the usability of the approach are presented.

Figure 3 shows a tree diagram which has been drawn with a *DiaGen* editor for trees. Figure 5 shows a generated external representation of this tree. The unparsing task consists of constructing an equivalent tree like the one of Figure 3 from this XML representation by creating its hypergraph model which is shown

in Figure 4. For the unparsing task, the hypergraph model has to be created with the `leftOf` hyperedges which have not been explained in Section 2.2: The plain hypergraph model without those edges does not represent the ordering of children nodes, e.g., that node b lies left of node c. When editing the diagram by free-hand editing, this fact can be deduced from the position attributes of the nodes which are accessible by the parser (cf. [11]). When creating the hypergraph model, however, by unparsing, i.e., without a diagram representation with its attribute values[6], this information is not available. Special `leftOf` edges, which cannot be created by the modeler, have to be created by the unparsing process in order to explicitly represent child ordering. Diagram analysis has to take them into account.

The unparsing task, based on an XML representation like the one of Figure 5, can be informally described as follows (cf. Figure 4):

> For each *node* element, create a `circle` hyperedge which visits a node c. If this *node* element has a *node* parent element, i.e., if it is not the tree root, create an `arrow` hyperedge which visits two new nodes *from* resp. *to*. Connect the *to* node to the new c node with an `inside` relation edge, and add a similar edge between the *from* node and the c node of the parent. Furthermore, if the current *node* element has a following sibling on its "right", create a `leftOf` edge between the current c node and the one of its sibling[7].

Figure 7 shows the specification of this process by an XSLT stylesheet. Since we are not interested in a result stream which has to be created by the XSLT processor from the XML-coded external data structure, the stylesheet does not contain any output actions, but XSLT markup elements and extension elements only. They are distinguished by their namespaces which are declared in lines 3–5: Namespace prefix `xsl` is declared for the XSLT namespace as usual, whereas prefix `make` resp. `diagen` determine extensions which refer to the specified Java classes. Prefix `diagen` with its `XSLTrafo` class is a generic extension which is used by every *DiaGen* editor and its unparsing functionality. It is used to create hypergraph nodes as well as relation edges and special (so-called "link") edges, whose implementation does not depend on the specific diagram language. Prefix `make` with its generated class, however, refers to functionality which is specific to the specified diagram language: The generated class offers methods which create language specific diagram components together with their diagram hyperedges[8].

[6] See Section 2.3: Attribute values will be defined by the layouter which, however, runs after the parser.

[7] Please note that this creates the sibling's c node if the XML document is traversed in document order.

[8] In the current state of implementation, this class is actually generated as a class skeleton. Since graphical representations of diagram components still have to be programmed manually and the number as well as types of parameters which have to be passed to their constructors are not part of the specification, those methods cannot be completely generated yet.

The stylesheet specifies just a single template rule which matches any `node` element (line 7). The XSLT processor would not traverse the whole tree of the XML document since this `node` pattern matches the tree root already (cf. Section 3.2). However, the XSLT element `<xsl:apply-templates/>` (line 35) at the end of the action part of the template rule tells the XSLT processor to continue traversing the XML document. This element specifies that this rule has to be applied to each `node` element of the XML document in document order.

The other elements of the action part (lines 9–34) specify the action for each `node` element as it has been presented informally:

- Line 9–10 create a new `circle` hyperedge by calling the extension method `circle` of the generated class `XSLaux` (cf.line 4). Parameters to this method are the *c* node for this `node` element and the value of its `text` attribute. Please note that the dot (`.`) is an XPath expression which refers to the current node, i.e., the root of XML sub-tree which has been matched by the rule pattern, and `@text` is an XPath expression for selecting the `text` attribute of the current node, whereas `diagen:attach` is an extension function which is implemented in the generic `XSLaux` class (cf. line 4). This method creates resp. refers to a hypergraph node by means of two parameters: The specified DOM tree node (`.` for the current DOM tree node) and a name (`'c'` for *c* as it has been used in the informal description).
- Lines 14–24 are executed only if the parent element is also a `node` element (line 13), i.e., if the current node does not represent the tree root. Please note that `..` is an XPath expression which selects the parent node of the current DOM tree node and that `name` is a built-in function which returns the tag name of the passed DOM tree node. Lines 14–24 then create the `arrow` hyperedge together with the *inside* hyperedges and the necessary hypergraph nodes. Please note that `diagen:attach(.., 'c')` refers to the *c* node of the parent element, whereas `diagen:attach(., 'c')` refers to the one of the current DOM tree node.
- Line 28 declares a new variable `next` whose value is defined by the following sibling DOM tree node which is selected by the XPath expression `following-sibling::*`. If such a node exist (line 29), a corresponding special `leftOf` edge is added to the hypergraph model (lines 30–33).

While unparsing trees from a tree-structured XML document is admittedly simple, unparsing functionality has also been added [16] to editors for *ladder diagrams* [8], *Nassi-Shneiderman diagrams* [7], and *Statecharts* [10]. The latter is based on an external Statechart representation which uses the XML-based *Graph Exchange Language* GXL [4,17]. The most interesting aspect of externally representing statecharts and unparsing statecharts from such representations is the way how hierarchy is being represented. GXL supports representation of hierarchy by directly nesting the corresponding XML elements or by special edges which make containment explicit. The statechart editor specification has been extended such that a GXL representation of the statechart is created and written to a file. The user may choose how hierarchy is presented. Additionally, unparsing has been specified by an XSLT stylesheet (130 lines of XSLT code). This

application has proven the generality of the XSLT-based unparsing approach of *DiaGen*: A single XSLT stylesheet is capable of dealing with different kinds of hierarchy representations. It is even possible to use mixed style in a single GXL representation.

5 Related Work

Unparsing is generally used as a term for the reverse process of parsing of textual languages. While parsing creates an abstract syntax tree for a given text, unparsing creates a text from such an abstract syntax tree. Sample systems that support unparsing of texts are the IDEM tool of the ELI tool suite [5] and PROGRES [15]. IDEM allows to specify the unparsing process by special unparsing rules which are used when traversing the abstract syntax tree. PROGRES uses unparsing (also termed *pretty-printing*) of mixed text and graphic elements from visualizing an internal logical document which is manipulated by syntax-directed editing. However, in both examples, the abstract representation as source of the unparsing process has a fixed format with respect to the syntax, i.e., it cannot be chosen as an external representation which has to be adapted to its context.

However, there are approaches which support unparsing of diagrams, too. The approaches of Read and Marlin as well as Cross and Hendrix create diagrammatic representations from abstract syntax trees [13,3]. However, their diagrammatic representation is just a visualization of regular text, i.e., their visual representations do not present more information than the corresponding textual ones. The approach by Andries and others is not limited by this restriction [1,14]. A visual sentence with its physical layout is represented by a spatial relationship graph (SRG). An additional abstract syntax graph (ASG) gives an abstract view of the SRG. Coupled graph-grammars are used to define the syntax of both graphs. When modifying the ASG by syntax-directed editing, the SRG and the physical layout have to be unparsed. This is done by parsing the modified ASG with respect to its grammar. Because of their coupled graph grammars, this defines a corresponding SRG. However, the ASG is not easily adapted to external requirements since coupled graph grammars for SRG and ASG do have to exist, and there have to exist efficient parsers for both grammars, too. However, in contrast to the approach which has been presented in this paper, this approach does not require an additional specification of the unparsing process.

6 Conclusions

This paper has presented a generic approach for unparsing of diagrams, i.e., to create a diagram from some external data structure that describes the diagram's meaning. Such a functionality, e.g., is required for any diagram editor as part of a larger software system that is not only used as an input facility, but also for visualization purposes. The approach depends on an XML-coded external diagram representation and an unparsing specification by an XSLT stylesheet. It has been used to create statecharts, ladder diagrams and other application

specific diagrams [16]. In this paper, however, it has been described in detail for a tree editor which is a sufficiently simple example in order to be completely presented in a paper. The described approach is now part of *DiaGen*, a system for rapid prototyping of diagram editors from formal specifications.

While the presented approach has shown its usability, it has its shortcomings, too: The approach requires an additional specification, i.e., the XSLT stylesheet, which is used for the unparsing process. Unfortunately, there is no way to check whether this stylesheet always creates diagrams whose meaning is equivalent to the external data structure from which they have been created. The specification of the diagram language which is used by *DiaGen* in terms of diagram analysis, i.e., translating a diagram to its meaning, cannot be used in "reversed" direction for unparsing purposes as their is no restriction on data structures which are used to represent a diagram's meaning.

References

1. M. Andries, G. Engels, and J. Rekers. How to represent a visual specification. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, chapter 8, pages 245–260.
2. S. S. Chok, K. Marriott, and T. Paton. Constraint-based diagram beautification. In *Proc. 1999 IEEE Symp. on Visual Languages, Tokyo, Japan*, 1999.
3. J. H. Cross and T. D. Hendrix. Language independent program visualisation. In P. Eades and K. Zhang, editors, *Software Visualisation*, pages 27–45, Singapore, 1996. World Scientific.
4. R. Holt, A. Winter, and A. Schürr. GXL: Towards a standard exchange format. In *Proc. 7th Working Conference on Reverse Engineering (WCRE 2000)*, pages 162–171, Los Alamitos, 2000. IEEE Computer Society.
5. U. Kastens, P. Pfahler, and M. Jung. The Eli system. In K. Koskimies, editor, *Proceedings 7th International Conference on Compiler Construction (CC'98)*, volume 1383 of *Lecture Notes in Computer Science*, pages 294–297. Springer Verlag, 1998.
6. O. Köth and M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In *Proc. International Workshop on Graph Transformation (*GraTra *2000), Berlin*, Mar. 2000.
7. M. Minas. Diagram editing with hypergraph parser support. In *Proc. 1997 IEEE Symp. on Visual Languages, Capri, Italy*, pages 230–237, 1997.
8. M. Minas. Creating semantic representations of diagrams. In M. Nagl and A. Schürr, editors, *Int. Workshop on Applications of Graph Transformations with Industrial Relevance (*Agtive*'99), Selected Papers*, volume 1779 of *Lecture Notes in Computer Science*, pages 209–224. Springer, Mar. 2000.
9. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
10. M. Minas and B. Hoffmann. Specifying and implementing visual process modeling languages with diagen. In C. E. Hartmut Ehrig and J. Padberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
11. M. Minas and O. Köth. Generating diagram editors with diagen. In M. Nagl and A. Schürr, editors, *Int. Workshop on Applications of Graph Transformations with*

*Industrial Relevance (*AGTIVE*'99), Selected Papers*, volume 1779 of *Lecture Notes in Computer Science*. Springer, Mar. 2000.
12. M. Minas and G. Viehstaedt. DiaGen: A generator for diagram editors providing direct manipulation and execution of diagrams. In *Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, pages 203–210, 1995.
13. M. Read and C. Marlin. Specifying and generating program editors with novel visual editing mechanisms. In *Proc. of the 10th Conference on Software Engineering and Knowledge Engineering*, pages 418–425, San Francisco, USA, 1998. KSI Press.
14. J. Rekers and A. Schürr. A graph based framework for the implementation of visual environments. In *Proc. 1996 IEEE Symp. on Visual Languages, Boulder, Colorado*, pages 148–155, 1996.
15. A. Schürr, A. Winter, and A. Zündorf. Visual programming with graph rewriting systems. In *Proc. 1995 IEEE Symp. on Visual Languages, Darmstadt, Germany*, pages 195–202, 1995.
16. F. Strüber. Unparsing von Diagrammen in DiaGen [Unparsing of diagrams with DiaGen]. Diploma thesis, Computer Science, University of Erlangen, Erlangen, Germany, July 2002. In German.
17. G. Taentzer. Towards common exchange formats for graphs and graph transformation systems. In C. E. Hartmut Ehrig and J. Padberg, editors, *Electronic Notes in Theoretical Computer Science*, volume 44. Elsevier Science Publishers, 2001.
18. List of XML applications. http://www.oasis-open.org/cover/siteIndex.html#toc-applications.

Linear Ordered Graph Grammars and Their Algebraic Foundations*

Ugo Montanari[1] and Leila Ribeiro[2]

[1] Dipartimento di Informática, Universitá di Pisa
Pisa, Italy
ugo@di.unipi.it
[2] Instituto de Informática, Universidade Federal do Rio Grande do Sul
Porto Alegre, Brazil
leila@inf.ufrgs.br

Abstract. In this paper we define a special kind of graph grammars, called linear ordered graph grammars, that can be used to describe distributed systems with mobility and object-based systems. Then we show how to model such grammars and their semantics in terms of tiles making explicit the aspects of interactivity and compositionality, that are of great importance in distributed systems.

1 Introduction

Wide area network programming is important and difficult to cope with using ad hoc techniques. In particular, testing in such environments has shown to be very inefficient due to the high dynamicity and possibility of failures of the environment. Thus, formal techniques to assure the correct behavior of the system are even more needed in this framework.

Graph transformation systems [18,7] have been used in a wide variety of applications, and can provide suggestive and technically adequate models of computation, semantic foundations and verification methods. There have been many approaches to model distributed systems using graph transformation systems [19]. Considering that our aim is to be able to model mobility of processes within a system, a natural way of representing distributed systems graphically is to model communication channels as nodes (abstracting out the physical network), agents/processes/ambients as (hyper)arcs and ports as tentacles (for example, the modeling of π-calculus agents using graph transformations defined in [16]). If we want to have a more object-oriented view, we could see objects as nodes and messages as (hyper)arcs, as it was done in [6]. What is important to notice is that in both modelings, mobility involves the creation and passing of nodes, while the process/message interpretation of arcs requires resource consciousness ("linearity" in Girard sense). In the graph transformation modelings discussed above, this means that neither rules nor matches for rules may identify

* This work was partially supported by the projects AGILE (FET project), COMETA (MIUR project), IQ-Mobile (CNPq and CNR) and ForMOS (CNPq and Fapergs).

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 317–333, 2002.
© Springer-Verlag Berlin Heidelberg 2002

arcs (because they model resources). Actually, if we do not have this linearity, or resource consciousness, property, the construction of a suitable compositional true concurrency semantics (based on concatenable processes, for example) is not possible (this will be further explained in Sect. 4).

In this paper we propose to specialize the existing single pushout (SPO) approach to graph transformations [14,8] to a kind of grammars that can be used to model both distributed systems with mobility and object-based systems. These grammars will be called *linear ordered graph grammars*. The linearity property is achieved by ordering nodes and arcs, and allowing only injective morphisms on items that represent resources (arcs).

Interactivity and abstract semantics via observations are key features of communicating systems. Compositionality is also very important for large distributed systems. Tiles have these as their main aims, and previous work has shown their feasibility [10,9]. In addition, they exhibit an algebraic flavor (tiles have a straightforward axiomatization as monoidal double categories) which may allow for universal constructions, compositionality and refinement in the classical style of algebraic semantics [15]. We show that the representation of SPO graph transformations of linear ordered graph grammars within tile logic is simple and natural. In this way, we provide a basis for an algebraic theory of graph grammars where observations and interactions are in the foreground, and this is not usually emphasized in the DPO/SPO literature, although it is a very important aspect of the kind of systems we are interested in modeling

This paper is organized as follows: Sect. 2 reviews the basic concepts of SPO graph rewriting using typed hypergraphs, Sect. 3 presents the main concepts of tile systems, introducing the kind of arrows needed for modeling graphs and (partial) graph morphisms as tiles, Sect. 4 introduces linear ordered graph grammars and shows how their syntax and semantics can be represented as tiles/tile rewrite systems. Section 5 brings a summary of our main results.

2 Graph Rewriting

In this section we will introduce the basic concepts of graph rewriting in the single pushout (SPO) approach [14,8]. However, instead of the standard definitions of morphisms based on total morphisms from subgraphs, we will explicitly use partial functions and a weak commutativity requirement (instead of the usual commutativity). In [13] it was shown that the two definitions are equivalent. Moreover, the definition based on partial functions has two advantages: it allows us to define typed hypergraph categories in an easy way (as instances of generalized graph structure categories), and it makes clearer the correspondence to the tile rewriting model that will be defined in Sect. 4.

Definition 1 (Weak Commutativity). *For a (partial) function $f : A \to B$ with domain $dom(f)$, let $f? : A \leftarrow dom(f)$ and $f! : dom(f) \to B$ denote the corresponding domain inclusion and the domain restriction. Given functions $a : A \to A'$, $b : B \to B'$ and $f' : A' \to B'$, where a and b are total, we write $f' \circ a \geq b \circ f$ and say that the diagram* **commutes weakly** *iff $f' \circ a \circ f? = b \circ f!$.*

Remark 1. If f and f' are total, weak commutativity coincides with commutativity. Note that $((f?)^{-1}, f!)$ is a ***factorization*** of f.

The compatibility condition defined above means that everything that is mapped by the function must be compatible. The term "weak" is used because the compatibility is just required on preserved (mapped) items, not on all items.

Now we will define hypergraphs and partial morphisms. In this paper we will use undirected hypergraphs, and therefore we will only define a *source* function connecting each arc with the list of nodes it is connected to. We use the notation S^* to denote the set of all (finite) lists of elements of S.

Definition 2 ((Hyper)Graph, (Hyper)Graph Morphism). *A **(hyper) graph** $G = (N_G, A_G, source^G)$ consists of a set of nodes N_G, a set of arcs A_G and a total function $source^G : A_G \to N_G^*$, assigning to each arc a list of nodes.*

*A **(partial) graph morphism** $g : G \to H$ from a graph G to a graph H is a pair of partial functions $g_N : N_G \to N_H$ and $g_A : A_G \to A_H$ which are weakly homomorphic, i.e., $g_N^* \circ source^G \geq source^H \circ g_A$ (g_N^* is the extension of g_N to lists of nodes). A morphism is called total if both components are total. The category of hypergraphs and partial hypergraph morphisms is denoted by* **HGraphP** *(identities and composition are defined componentwise).*

To distinguish different kinds of nodes and arcs, we will use a notion of typed hypergraphs, analogous to typed graphs [5,12][1].

Definition 3 (Typed Hypergraphs). *A typed hypergraph is a tuple $HG^{TG} = (HG, type^{HG}, TG)$ where HG and TG are hypergraphs, called instance and type graph, respectively, and $type^{HG} : HG \to THG$ is a total hypergraph morphism, called typing morphism.*

A morphism between typed hypergraphs $HG1^{TG}$ and $HG2^{TG}$ is a partial function $f : HG1 \to HG2$ such that $type1^{HG1} \geq type2^{HG2} \circ f$. The category of all hypergraphs typed over a type graph TG, denoted by **THGraphP**(**TG**)*, consists of all hypergraphs over TG as objects and all morphisms between typed hypergraphs (identities and composition are the identities and composition of partial functions).*

For the well-definedness of the categories above and the proof that these categories have pushouts we refer to [13] (these categories can be constructed as generalized graph structures categories). The construction of pushouts in such categories is done componentwise, followed by a free construction to make the *source* and *type* components total (actually, as types are never changed by morphisms, this free construction is not needed in the case of typed graphs). In the rest of the paper, we will usually denote typed hypergraphs just by graphs.

[1] Note that, due to the use of partial morphisms, this is not just a comma category construction: the morphism *type* is total whereas morphisms among graphs are partial, and we need weak commutativity instead of commutativity. In [13] a way to construct such categories was defined.

Definition 4 (Graph Grammar). *Given a (hyper)graph TG, a* ***rule*** *is a morphism in* $\mathbf{THGraphP}(\mathbf{TG})$. *A* ***graph grammar*** *is a tuple $GG = (TG, IG, Rules)$ where TG is a (hyper)graph, called type graph, IG is a (hyper)graph typed over TG, called initial or start graph, and $Rules$ is a set of rules over TG.*

Definition 5 (Derivation Step, Sequential Semantics of a Graph Grammar). *Given a graph grammar $GG = (TG, IG, Rules)$, a rule $r : L \to R \in Rules$, and a graph $G1$ typed over TG, a* ***match*** *$m : L \to G1$ for rule r in $G1$ is a total morphism in* $\mathbf{THGraphP}(\mathbf{TG})$. *A* ***derivation step*** *$G1 \overset{r,m}{\Longrightarrow} G2$ using rule r and match m is a pushout in the category* $\mathbf{THGraphP}(\mathbf{TG})$. *The morphism r' is called* ***co-rule****.*

$$\begin{array}{ccc} L & \xrightarrow{r} & R \\ {\scriptstyle m}\downarrow & (PO) & \downarrow{\scriptstyle m'} \\ G1 & \xrightarrow[r']{} & G2 \end{array}$$

A derivation sequence of GG is a sequence of derivation steps $G_i \overset{r_i,m_i}{\Longrightarrow} G_{i+1}$, $i \in \{0..n\}$, $n \in \mathbb{N}$, where $G_0 = IG$, $r_i \in Rules$. The composition of all co-rules r'_i is called ***derived production*** *of a sequential derivation. The* ***sequential semantics*** *of GG is the set of all sequential derivations of GG.*

An example of a rule and derivation using typed hypergraphs is shown in Figure 1. Arcs are drawn as squares and nodes as bullets. The tentacles of the arcs describe the *source* function and are numbered to indicate their order. Indices on nodes and arcs here just indicate different instances of items with same types. Intuitively, arcs represent resources, for example agents or ambient names, and nodes represent ambients. The rule may describe a kind of *open* operation of ambients and the creation of a new ambient: agent B is preserved, the ambient represented by $\bullet_1$ (that is inside ambient $\bullet_2$ with name A) is opened, having the effect that all agents and ambients that were inside $\bullet_2$ before the application of the rule are now inside ambient $\bullet_1$, and a new ambient inside $\bullet_1$ is created (ambient $\bullet_2$ of R, named C). Note that this interpretation does not correspond exactly to a rule of the ambient calculus, since the agent that triggers the execution (B) is preserved, instead of being substituted by its continuation. This rule was chosen to illustrate using one example that our framework is expressive enough to model name fusion, a feature that is needed in some calculi (for example, in the fusion calculus by Parrow and Victor, as well as in the ambient calculus) and preservation of resources, needed to model, for example, replication (! operator in the *pi*-calculus).

3 Tiles and Algebraic Semantics

In this section we will define the basic notions of tile rewrite systems. Each tile can be seen as a square consisting of four arrows: the horizontal ones describe states, and the vertical ones observations. First we define some monoidal theories that will be used to build suitable arrows of tiles that correspond to graphs and graph morphisms. Then we define a tile rewrite system based on such arrows.

The following definition is a slight restriction of the definition in [17] to consider only undirected hypergraphs.

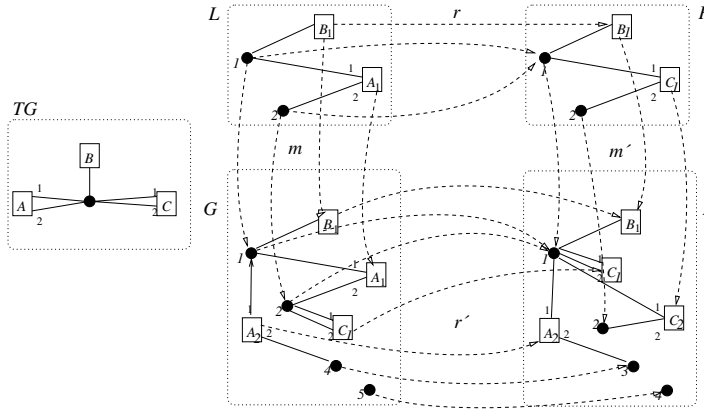


Fig. 1. Derivation using rule r at match m

Definition 6 (One-sorted (hyper-)signatures). *A* ***hyper-signature*** $\Sigma = (S, OP)$ *consists of a singleton* $S = \{s\}$ *and a family* $OP = \{op_h\}_{h \in \mathbb{N}}$ *of sets of* operators *with* h *arguments and no result (no target sort), where* h *is a natural number. Given an operation* $op_h \in OP$, h *is the arity of* op_h, *and the domain of* op_h *is denoted by* s^h. Σ_n *denotes the set of operators of* OP *with arity* n.

Now we define an extension of a signature Σ that will add as sorts all operation names in Σ and as target of each operation the corresponding sort.

Definition 7 (Extended (hyper-)signatures). *Given a signature* $\Sigma = (S, OP)$, *its* ***extension*** *is a signature* $\Sigma^E = (S_E, OP_E)$, *where* $S_E = \{\underline{s}\} \cup \{\underline{op} | op : n \to 0 \in OP\}$ *and* $OP_E = \{op_h^E : \underline{s^h} \to \underline{op_h} | op_h : s^h \to 0 \in OP\}$.

3.1 Monoidal Theories

We will define algebraic theories to describe graphs and graph morphisms. In particular, the gs-monoidal theory defined here is an extension of the one defined in [4] because we consider different arrows for sorts of an extended hyper signature: for the only sort coming from the underlying hyper signature, we use the usual definition of gs-monoidal theories, but for the other ones, we do not allow the duplicator and discharger operators. In addition, we define a new theory: the partial graph morphism (pgm) theory, allowing discharger and new operators for sorts that are not in the underlying signature only (the rest is the same as in the gs-monoidal theory). The reason for this will become clear in the next section, where we will use gs-monoidal theories to model (hyper)graphs and pgm-monoidal theories to model special (hyper)graph morphisms.

Definition 8 (Connection diagrams). *A* connection diagram G *is a 4-tuple* $\langle O_G, A_G, \delta_0, \delta_1 \rangle$: O_G, A_G *are sets whose elements are called respectively* objects *and* arrows, *and* $\delta_0, \delta_1 : A_G \to O_G$ *are functions, called respectively* source *and*

target. *A connection diagram G is* reflexive *if there exists an* identity *function $id_- : O_G \to A_G$ such that $\delta_0(id_a) = \delta_1(id_a) = a$ for all $a \in O_G$; it is* with pairing *if its class O_G of objects forms a monoid; it is* monoidal *if it is reflexive with pairing and also its class of arrows forms a monoid, such that id, δ_0 and δ_1 respect the neutral element and the monoidal operation.*

In the following we will use the free commutative monoid construction over a set S, denoted by $(S^{\otimes}, \otimes, \underline{0})$. The elements of $S^{\otimes}$ can be seen as lists of elements of S, and will be referred to using underlined variables. If S is a singleton set, we will denote elements of $S^{\otimes}$ by underlined natural numbers (indicating the number of times the only element of S appears in the list, for example $\underline{s} \otimes \underline{s} \otimes \underline{s}$ will be denoted by $\underline{3}$).

Definition 9 (Graph theories). *Given a (hyper)signature $\Sigma = (S, OP)$ and its extension $\Sigma^E = (S_E, OP_E)$, the associated* graph theory *$G(\Sigma)$ is the monoidal connection diagram with objects the elements of the free commutative monoid over S_E $((S_E)^{\otimes}, \otimes, \underline{0})$ and arrows generated by the inference rules* generators, sum *and* identities *shown in Table 1 satisfying the* monoidality *axiom $id_{\underline{x} \otimes \underline{y}} = id_{\underline{x}} \otimes id_{\underline{y}}$ for all $\underline{x}, \underline{y} \in S_E^{\otimes}$.*

Table 1. Inference Rules

$$(\textit{generators})\ \frac{f \in \Sigma_n}{f : \underline{n} \to \underline{f} \in G(\Sigma)} \qquad (\textit{sum})\ \frac{s : \underline{x} \to \underline{y}, t : \underline{x'} \to \underline{y'}}{s \otimes t : \underline{x} \otimes \underline{x'} \to \underline{y} \otimes \underline{y'} \in G(\Sigma)}$$

$$(\textit{identities})\ \frac{\underline{x} \in S_E^{\otimes}}{id_{\underline{x}} : \underline{x} \to \underline{x} \in G(\Sigma)}$$

$$(\textit{composition})\ \frac{s : \underline{x} \to \underline{y}, t : \underline{y} \to \underline{z}}{s; t : \underline{x} \to \underline{z} \in \mathbf{M}(\Sigma)}$$

$$(\textit{duplicators})\ \frac{\underline{n} \in S^{\otimes}}{\nabla_{\underline{n}} : \underline{n} \to \underline{n} \otimes \underline{n} \in \mathbf{GS}(\Sigma)} \qquad (\textit{dischargers})\ \frac{\underline{n} \in S^{\otimes}}{!_{\underline{n}} : \underline{n} \to \underline{0} \in \mathbf{GS}(\Sigma)}$$

$$(\textit{permutations})\ \frac{\underline{x}, \underline{y} \in S_E^{\otimes}}{\rho_{\underline{x},\underline{y}} : \underline{x} \otimes \underline{y} \to \underline{y} \otimes \underline{x} \in \mathbf{GS}(\Sigma)}$$

Definition 10 (Monoidal theories). *Given a (hyper)signature $\Sigma = (S, OP)$ and its extension $\Sigma^E = (S_E, OP_E)$, the associated* monoidal theory *$\mathbf{M}(\Sigma)$ is the monoidal connection diagram with objects the elements of the free commutative monoid over S_E $((S_E)^{\otimes}, \otimes, \underline{0})$ and arrows generated by the following inference rules:* generators, sum *and* identities *(analogous to the rules in Def. 9), and the rule* composition *in Table 1. Moreover, the composition operator ; is associative, and the monoid of arrows satisfies the* functoriality *axiom*

$$(s \otimes t); (s' \otimes t') = (s; s') \otimes (t; t')$$

(whenever both sides are defined) and the identity *axiom $id_{\underline{x}}; s = s = s; id_{\underline{y}}$ for all $s : \underline{x} \to \underline{y}$.*

Definition 11 (gs-monoidal theories). *Given a (hyper)signature* $\Sigma = (S, OP)$ *and its extension* $\Sigma^E = (S_E, OP_E)$*, the associated* ***gs-monoidal theory*** $\mathbf{GS}(\Sigma)$ *is the monoidal connection diagram with objects the elements of the free commutative monoid over* S_E $((S_E)^{\otimes}, \otimes, \underline{0})$ *and arrows generated by the following inference rules:* generators, sum, identities *and* composition *(analogous to the rules in Def. 10), and the rules* duplicators, dischargers *and* permutations *in Table 1. Moreover, the composition operator* ; *is associative, and the monoid of arrows satisfies the functoriality axiom (see Def. 10); the identity axiom (see Def. 10); the monoidality axioms: for all* $\underline{n}, \underline{m} \in S^{\otimes}$ *and* $\underline{x}, \underline{y}, \underline{z} \in S_E^{\otimes}$

$$\rho_{\underline{x}\otimes\underline{y},\underline{z}} = (id_{\underline{x}} \otimes \rho_{\underline{y},\underline{z}}); (\rho_{\underline{x},\underline{z}} \otimes id_{\underline{y}}) \qquad !_{\underline{0}} = \nabla_{\underline{0}} = \rho_{\underline{0},\underline{0}} = id_{\underline{0}} \qquad \rho_{\underline{0},\underline{x}} = \rho_{\underline{x},\underline{0}} = id_{\underline{x}}$$

$$!_{\underline{x}\otimes\underline{y}} = !_{\underline{x}} \otimes !_{\underline{y}} \qquad \nabla_{\underline{n}\otimes\underline{m}} = (\nabla_{\underline{n}} \otimes \nabla_{\underline{m}}); (id_{\underline{n}} \otimes \rho_{\underline{n},\underline{m}} \otimes id_{\underline{m}})$$

the coherence *axioms: for all* $\underline{n} \in S^{\otimes}$ *and* $\underline{x}, \underline{y} \in S_E^{\otimes}$

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes \nabla_{\underline{n}}) = \nabla_{\underline{n}}; (\nabla_{\underline{n}} \otimes id_{\underline{n}}) \qquad \nabla_{\underline{n}}; \rho_{\underline{n},\underline{n}} = \nabla_{\underline{n}}$$

$$\nabla_{\underline{n}}; (id_{\underline{n}} \otimes !_{\underline{n}}) = id_{\underline{n}} \qquad \rho_{\underline{x},\underline{y}}; \rho_{\underline{x},\underline{y}} = id_{\underline{x}} \otimes id_{\underline{y}}$$

and the naturality *axiom: for all* $s : \underline{x} \to \underline{y}, t : \underline{z} \to \underline{w}$

$$(s \otimes t); \rho_{\underline{y},\underline{w}} = \rho_{\underline{x},\underline{z}}; (t \otimes s)$$

Arrows of gs-monoidal theories that are constructed without the generators *axiom are called* ***basic gs-monoidal arrows****. The theory obtained only with basic arrows is denoted by* $\mathbf{bGS}(\Sigma)$*.*

In [4] it was shown that term graphs (a particular kind of hierachical hypergraphs) could be suitably represented as arrows of a gs-monoidal theory. Here we are rather interested in non-hierachical graphs, and the restriction is achieved by selecting signatures where operators have output types which are disjoint from input types, i.e. operators cannot be nested. Objects represent lists of variables (nodes) and an arrow $\underline{n} \to \underline{m}$ represents the application of a (multi)set of operators (arcs) to the variables $\underline{n}$ generating the results (roots) $\underline{m}$. For example, assuming that the sort of nodes of L is $\{s\}$, the graph L in Figure 1 represents the union of term-graphs $A(\bullet_2, \bullet_1)$ and $B(\bullet_1)$ sharing the variable $\bullet_1$. This term-graph can be modeled as the following gs-monoidal arrow: $\rho_{\underline{s},\underline{s}}; (id_{\underline{s}} \otimes \nabla_1); (A \otimes B) : \underline{s} \otimes \underline{s} \to \underline{A} \otimes \underline{B}$. Graphically, this arrow is represented in Figure 2 (1 and 2 are the first and second occurrences of sort s in $\underline{s} \otimes \underline{s}$).

Notation: In the pictures, the operator $\otimes$ and the underlines in the domain and codomain of the arrows will be omitted. Moreover, we will draw indexed bullets ($\bullet$) in the graphical representation to describe the instances of sorts.

The theory defined below will be used to model a special kind of graph morphism: morphisms that are total on nodes (and may identify nodes), and are partial on arcs (and may not identify arcs). Identification of nodes is achieved by the ∇ operator, as in gs-monoidal theories. The possibility of deleting arcs is achieved by the ¡ operator, yielding partial functions (only for arcs).

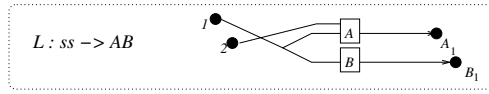


Fig. 2. Arrow of a gs-monoidal theory

Definition 12 (pgm-monoidal theories). *Given a (hyper)signature* $\Sigma = (S, OP)$ *and its extension* $\Sigma^E = (S_E, OP_E)$*, the associated* ***pgm-monoidal theory*** $\mathbf{PGM}(\Sigma)$ *is the monoidal connection diagram with objects the elements of the free commutative monoid over* S_E $((S_E)^{\otimes}, \otimes, \underline{0})$ *and arrows generated by the following inference rules:* sum, identities, composition *and* permutations *(analogous to the rules in Def. 11), and*

$$\text{(new)}\ \frac{\underline{x} \in (S_E - S)^{\otimes}}{¡_{\underline{x}} : \underline{0} \to \underline{x} \in \mathbf{PGM}(\Sigma)}$$

$$\text{(duplicators)}\ \frac{\underline{n} \in S^{\otimes}}{\nabla_{\underline{n}} : \underline{n} \to \underline{n} \otimes \underline{n} \in \mathbf{PGM}(\Sigma)} \qquad \text{(dischargers)}\ \frac{\underline{x} \in S_E^{\otimes}}{!_{\underline{x}} : \underline{x} \to \underline{0} \in \mathbf{PGM}(\Sigma)}$$

The composition operator ; *is associative and the monoid of arrows satisfies the* functoriality *axiom (Def. 10); the* identity *axiom (Def. 10); the* monoidality *axioms (all of Def. 11 plus* $!_{\underline{0}} = ¡_{\underline{0}} = \rho_{\underline{0},\underline{0}} = id_{\underline{0}}$ *and* $¡_{\underline{x}\otimes\underline{y}} = ¡_{\underline{x}} \otimes ¡_{\underline{y}}$*, for all* $\underline{x}$, $\underline{y}$, $\underline{z} \in S_E^{\otimes}$*); the* coherence *axiom (Def. 11); and the* naturality *axiom (Def. 11). The theory obtained using the same objects and arrows generated by the rules above, except the* composition *is called* ***short pgm-graph theory****, denoted by* $\mathbf{sPGM}(\Sigma)$*. The theory obtained using all rules except* new *and with* discharger *only for* $\underline{x} \in S^{\otimes}$ *is called* ***basic pgm-graph theory****, denoted by* $\mathbf{bPGM}(\Sigma)$*.*

Note that in pgm-monoidal theories we did not use the *generators* axiom. This has the effect that arrows of these theories do not correspond to graphs as gs-monoidal theories (because arcs are not allowed), they rather describe relationships between the objects (that are lists of nodes and arc labels). Moreover, an arrow of $\mathbf{bPGM}(\Sigma)$ is also an arrow of $\mathbf{bGS}(\Sigma)$.

In the following definition, we use the fact that basic gs-monoidal arrows of one sorted signatures correspond to total functions in the inverse direction, that is, each gs-monoidal arrow $\underline{n} \to \underline{m}$ corresponds to a total function $\underline{m} \to \underline{n}$ (see [4] for the proof). As basic pgm-monoidal arrows are also basic gs-monoidal arrows, they also correspond to total functions. This will be used to construct pullback squares of such arrows based on pushouts of functions. These pushouts will model the node-component morphisms of rule applications.

Definition 13 (Derivation Pair). *Given one-sorted signatures* Σ_h *and* Σ_v*, and arrows* $s : \underline{n} \to \underline{m} \in \mathbf{bGS}(\Sigma_h)$ *and* $t : \underline{q} \to \underline{m} \in \mathbf{bPGM}(\Sigma_v)$*. The* ***derivation pair*** *of* s *and* t*, denoted by* $deriPair(s, t)$*, is a pair of a basic gs-monoidal and a basic pgm-monoidal arrows* $(s' : \underline{p} \to \underline{q}, t' : \underline{p} \to \underline{n})$ *such that the inverse square is a pushout in the category of sets and total functions.*

3.2 Tile Rewriting Systems

Definition 14 (Tile sequent, Tile rewrite system). *Let Σ_h and Σ_v be two (one sorted) signatures, called the* horizontal *and the* vertical *signature respectively. A Σ_h-Σ_v* tile sequent *is a quadruple $l \overset{a}{\underset{b}{\rightarrow}} r$, where $l : \underline{x} \to \underline{y}$ and $r : \underline{p} \to \underline{q}$ are arrows of $\mathbf{GS}(\Sigma_h)$ while $a : \underline{x} \to \underline{p}$ and $b : \underline{y} \to \underline{q}$ are arrows of $\mathbf{PGM}(\Sigma_v)$. Arrows l, r, a and b are called respectively the* initial configuration*, the* final configuration*, the* trigger *and the* effect *of the tile. Trigger and effect are called* observations*. Underlined strings $\underline{x}$, $\underline{y}$, $\underline{p}$ and $\underline{q}$ are called respectively the* initial input interface*, the* initial output interface*, the* final input interface *and the* final output interface.

A short *tile sequent is a tile $l \overset{a}{\underset{b}{\rightarrow}} r$ where observations a and b are arrows of the short pgm-graph theory $\mathbf{sPGM}(\Sigma_v)$ (i.e. no sequential composition is allowed to build them).*

A tile rewrite system (TRS) *$\mathcal{R}$ is a triple $\langle \Sigma_h, \Sigma_v, R \rangle$, where R is a set of Σ_h-Σ_v short tile sequents called* rewrite rules*.*

A TRS $\mathcal{R}$ can be considered as a logical theory, and new sequents can be derived from it via certain inference rules.

Definition 15 (pgm-monoidal tile logic). *Let $\mathcal{R} = \langle \Sigma_h, \Sigma_v, R \rangle$ be a* TRS*. Then we say that $\mathcal{R}$* entails *the class $\mathbf{R}$ of the* tile sequents *$s \overset{a}{\underset{b}{\rightarrow}} t$ obtained by finitely many applications of the inference rules depicted in Table 2.*

4 Linear Ordered Graph Grammars

Now we will define a restricted type of graph grammar: the nodes and arcs of each graph are ordered, the rules do not delete nodes and do not collapse arcs, and the matches do not collapse arcs. With this kind of restriction, the derivation steps can be obtained componentwise as a pushout of nodes and pushout of arcs (because no nodes are deleted).

The motivation for the definition of this kind of graph grammar is mainly based on the applications we have in mind: mobile code and object-based systems. The idea is to model static components (places, communication channels, objects) as nodes and resources (processes, agents, ambients, messages) as arcs. In this view, it is natural to require that we have a different way to handle these two kinds of entities, and that is why we have a distinct treatment of nodes and arcs in a linear ordered graph grammar. The fact that nodes and arcs are ordered was imposed to achieve a suitable true concurrency semantics. In the unordered case, the description of concurrency is analogous to the collective view of tokens in a Petri net, that leads to a different semantical model as the individual token philosophy. The latter is the standard way of considering the semantics of graph rewriting. A good comparison of the approaches for the case of Petri nets can be found in [3]. To illustrate this situation, consider the grammar depicted in Figure 3. There is only one rule, that deletes and creates arcs of type B. The start

Table 2. pgm-monoidal Tile Logic

Basic rules:

$$(generators)\ \frac{s \xrightarrow[b]{a} t \in \mathcal{R}}{s \xrightarrow[b]{a} t \in \mathbf{R}} \qquad (h\text{-}refl)\ \frac{s : \underline{x} \to \underline{y} \in \mathbf{GS}(\Sigma_h)}{id_s = s \xrightarrow[id_{\underline{y}}]{id_{\underline{x}}} s \in \mathbf{R}}$$

$$(v\text{-}refl)\ \frac{a : \underline{x} \to \underline{y} \in \mathbf{PGM}(\Sigma_v)}{id_a = id_{\underline{x}} \xrightarrow[a]{a} id_{\underline{y}} \in \mathbf{R}};$$

Composition rules:

$$(p\text{-}comp)\ \frac{\alpha = s \xrightarrow[b]{a} t, \alpha' = s' \xrightarrow[b']{a'} t' \in \mathbf{R}}{\alpha \otimes \alpha' = s \otimes s' \xrightarrow[b \otimes b']{a \otimes a'} t \otimes t' \in \mathbf{R}}$$

$$(h\text{-}comp)\ \frac{\alpha = s \xrightarrow[c]{a} t, \alpha' = s' \xrightarrow[b]{c} t' \in \mathbf{R}}{\alpha * \alpha' = s; s' \xrightarrow[b]{a} t; t' \in \mathbf{R}} \qquad (v\text{-}comp)\ \frac{\alpha = s \xrightarrow[b]{a} u, \alpha' = u \xrightarrow[b']{a'} t \in \mathbf{R}}{\alpha \cdot \alpha' = s \xrightarrow[b;b']{a;a'} t \in \mathbf{R}};$$

Auxiliary rules:

$$(perm)\ \frac{a : \underline{x} \to \underline{y}, b : \underline{x'} \to \underline{y'} \in \mathbf{PGM}(\Sigma_v)}{\rho_{a,b} = \rho_{\underline{x},\underline{x'}} \xrightarrow[b \otimes a]{a \otimes b} \rho_{\underline{y},\underline{y'}} \in \mathbf{R}}$$

$$(PBnodes)\ \frac{s : \underline{n} \to \underline{m} \in \mathbf{bGS}(\Sigma_h), a : \underline{q} \to \underline{m} \in \mathbf{bPGM}(\Sigma_v)}{\underline{s} \xrightarrow[a]{a'} \underline{s'} \in \mathbf{R}} (s', a') = deriPair(s, a)$$

graph of the grammar has two B-arcs. Some possible behaviors of this grammar are processes $P1$ and $P2$, that replace, using rule r, the B_1 and B_2-arcs of IG respectively by other B-arcs. If we consider that arcs are not ordered, these processes are isomorphic. Let process $P3$ be the concatenation of $P1$ and $P2$. This process has two applications of rule r, but we can not distinguish whether these applications are dependent or independent, that is, the process in which the second application of r depends on the first is equivalent to the one in which they are independent. This contradicts the basic idea of a true concurrency semantics, in which dependencies between actions are essential.

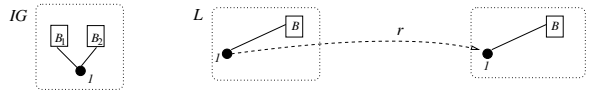


Fig. 3. Start Graph IG and Rule r

Definition 16 (Linear Ordered Graph Grammars). *A **linear ordered (hyper)graph** over a type graph $TG = (\{s\}, A_{TG}, source^{TG})$ is a typed hypergraph HT^{TG} with $HG = (N, A_{HG}, source^{HG})$ where $N = \{0..n\}$, with $n \in \mathbb{N}$, and $A_{HG} = \bigcup_{a \in A_{TG}} A^a_{HG}$, with $A^a_{HG} = \{a_i | i \in \{0..m\}\}$, where m is the car-*

dinality of $\{ha \in A_{HG} | type^{HG}(ha) = a\}$. *A morphism between linear ordered hypergraphs is simply any morphism typed hypergraph morphism.*

A ***linear ordered graph grammar*** *is a graph grammar* $GG = (TG, IG, Rules)$ *where* TG *has only one type of nodes,* IG *and all graphs in the rules of* $Rules$ *are linear ordered graphs and for each rule* $r = (r_N, r_A) : L \to R$ *of* $Rules$, r_A *is injective and* r_N *is total. A* ***linear ordered match*** $m = (m_N, m_A)$ *is a total graph morphism where the* m_A *component is injective.*

The semantics of a linear ordered graph grammar is defined as for a usual graph grammar, taking into account that only linear ordered matches are allowed, that is, a match is resource conscious on arcs, but may identify nodes.

In the following sections, we will show how to describe the syntax and semantics of such a graph grammar using tiles and tile rewriting systems.

4.1 Syntax: Graphs and Rules

Consider the linear ordered derivation step shown in Figure 1. The graphs used in this derivation are hypergraphs having only one type of node and three types of (hyper)arcs, having the following arities (consider the sort of nodes of TG to be s): $A : s \times s, B : s, C : s \times s$. Our idea of representing graphs and rules as tiles is as follows:

Graphs: Graphs will be modeled as the horizontal components of the tiles. Let $TG = (\{\mathrm{s}\}, A, source^{TG})$ be a type graph. Based on this graph we can build a siganture $\Sigma_h = (\{s\}, A_\Sigma)$, where A_Σ contains all arcs of A as operations (the information about the arity is given by the $source^{TG}$ function). A graph will be then an arrow $\underline{x} \to \underline{y}$ of the corresponding gs-monoidal theory, where $\underline{x}, \underline{y} \in (\{s\} \uplus A)^\otimes$. This means that the mapping from $\underline{x}$ to $\underline{y}$ is constructed by using, besides the arcs of the graph, the operations of identity and permutation for all sorts, and for elements of sort s (nodes), we additionally allow duplication and discharging, to allow that the same node may be used as source of many hyperarcs, and that it may not be used by any hyperarc. For the hyperarcs of the graph we include a target function that assigns to each arc an occurrence of its type (indexed by a natural number). For example, the graph L can be described by the gs-monoidal arrow illustrated in Figure 2. Usual (closed) graphs correspond to the special case of $G : \underline{x} \to \underline{y}$ when we have $\underline{x} \in \{s\}^\otimes$ and $\underline{y} \in A^\otimes$. Other graphs are called *open graphs*, and will be used as auxiliary components to allow the modeling of the direct derivation.

Rules: A rule $r : L \to R$ is a (partial) graph morphism $r = (r^N, r^A)$. Such a rule can be represented as a tile having as horizontal arrows the graphs L and R, and as vertical arrows mappings that allow to glue nodes, delete arcs, preserve and create arcs and nodes. These vertical arrows are arrows of the pgm-monoidal theory for the signature $\Sigma_v = \Sigma_h$. Rule r of Figure 1 corresponds to the tile of Figure 4 (vertical mapping is shown as dashed arrows, creation - new - is denoted by the ¡, and deletion - bang - is denoted by !). Note that at the west side of the tile r we have modeled the component r^N of the rule, whereas in the east side we model the component r^A.

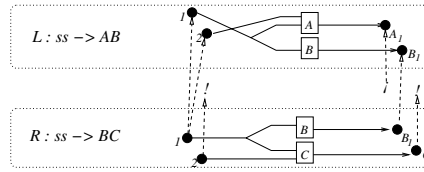


Fig. 4. Modeling a rule as a tile

Definition 17 (Signature of a Type Graph). *Given a type graph* $TG = (N, A, source^{TG})$*, the corresponding hypergraph signature is* $\Sigma_{TG} = (N, A_{TG})$*, where* $A_{TG} = \{a : source^{TG}(a) \to \underline{0} | a \in A\}$*.*

The characterization below describes which arrows of a gs-monoidal theory correspond to graphs. Other arrows will be called *open graphs*. The proof was omitted due to space limitations (similar proofs can be found in [4] and [17]).

Proposition 1 (Characterization of graphs). *The graphs typed over* $TG = (N, A, source^{TG})$ *with* N *being a singleton are the arrows of kind* $\underline{n} \to \underline{a}$ *of a gs-monoidal theory with signature* Σ_{TG}*, where* $\underline{n} \in N^{\otimes}$ *and* $\underline{a} \in (N^E - N)^{\otimes}$*, with* $\Sigma_{TG}^E = (N^E, A^E)$*.*

The characterization below describes a kind of graph morphism that may identify nodes but not arcs, and may be partial on arcs, but not on nodes. The rules of a linear ordered graph grammar are exactly this kind of morphisms.

Proposition 2 (Characterization of graph morphisms). *Graph morphisms typed over* TG *are tiles having as horizontal and vertical signature* Σ_{TG} *where the north and south sides are the arrows corresponding to (closed) graphs (the left- and right-hand sides of the rule, respectively), and the vertical arrows are arrows of* $\mathbf{PGM}(\Sigma_{TG})$*.*

Proof. An SPO-rule consists of a left-hand side L, a right-hand side R and a partial graph morphism $r = (r_N, r_A) : L \to R$. Consider the gs-monoidal arrows $s_L : \underline{n} \to \underline{a}$ and $s_R : \underline{n'} \to \underline{a'}$ corresponding to graphs L and R, respectively. As r_N is a total function, there is a corresponding gs-monoidal arrow $b_N : \underline{n'} \to \underline{n}$ mapping the nodes of R into the nodes of L. Arrow $b_N \in \mathbf{PGM}(\Sigma_{TG})$ because r_N is just a mapping of nodes, and this can be modeled just by using identities, duplicators, discharger and permutation operators for the sort of nodes, and the corresponding rules may be used to build $\mathbf{PGM}(\Sigma_{TG})$. The function that maps arcs of L into arcs of R does not allow the identification of arcs, but allows to delete and create arcs. Therefore, we need identities, dischargers, new and permutation operators for the sorts of arcs, and corresponding rules are allowed to build $\mathbf{PGM}(\Sigma_{TG})$. The other direction can be proven analogously.

4.2 Semantics

Now we will show how a direct derivation of a grammar GG can be modeled by a suitable composition of tiles of the pgm-monoidal tile logic $\mathbf{R}$ obtained by the

TRS $\langle \Sigma_h, \Sigma_v, R \rangle$, where $\Sigma_h = \Sigma_v = \Sigma_{TG}$ (as discussed above), and R is the set of tiles representing the rules of GG. A derivation $G \overset{r,m}{\Longrightarrow} H$ using rule $r : L \to R$ at match $m : L \to G$ can be obtained as a composition of tiles that will give raise to a tile $r' : G \to H$. This can be done in 4 steps:

1. **Context Graph**: Construct the (context) graph G', that contains all nodes of G and all arcs that are in G and are not in the image of the match m (that is, G' is G after removing the deleted and preserved arcs). Note that G' must be a graph because we do not allow rules that delete nodes (and therefore G' can not contain dangling edges). For the example derivation in Figure 1 we would have the graphs corresponding to G and G' in Figure 5 (note that the arc A_1 of G' corresponds to the arc A_2 of G – that name in G' must have the index 1 because it is the first occurrence of sort A in AC).

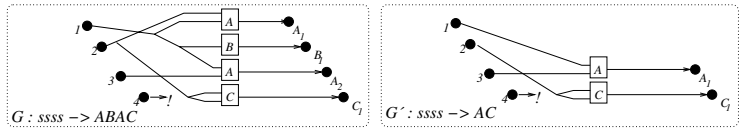


Fig. 5. Graph G and context graph G'

2. **Tile 1**: Construct the tile $r \otimes id_{G'}$, that is the parallel composition of the tile corresponding to rule r and the identity tile of graph G'. Note that, as L and G are graphs, the west side of this tile has only nodes, whereas the east side has only arcs. This tile belongs to $\mathbf{R}$ because rule $r \in \mathbf{R}$ (rule *generators*), $id'_G \in \mathbf{R}$ (rule *h-refl*) and thus the composed tile also belongs to $\mathbf{R}$ (rule *p-comp*). In the example we will get the tile (a) in Figure 6.

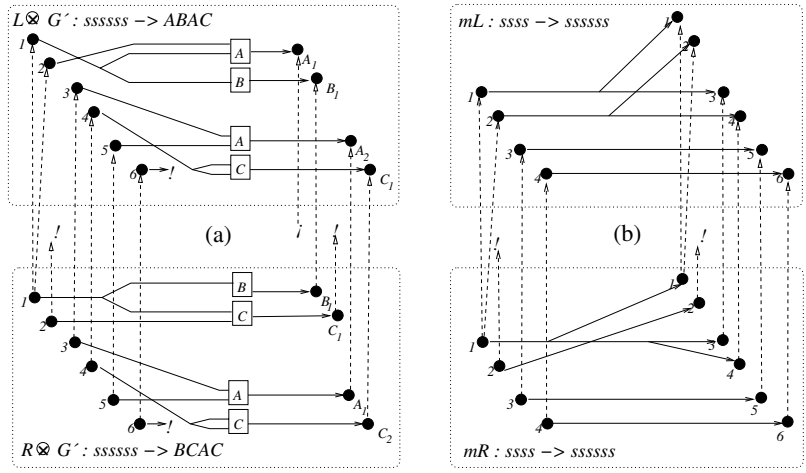


Fig. 6. (a) Tile 1 (b) Tile 2

3. **Tile 2**: Construct the tile corresponding to the match and derivation on the node component: the north side of this tile corresponds to m^N (the component of the match morphism that maps nodes), the east side is the west side of the tile obtained in step 2, and the remaining sides will be mappings of nodes such the the resulting square commutes and has no nodes that are not in the north or east sides already. Considering that the algebraic structures used to build these mappings are functions in the inverse direction, this square will be a pushout in the category of sets and partial functions (in this case, all functions are actually total because deleting of nodes is not allowed in our setting). This tile belongs to **R** due to rule *PBnodes*, that says that all such pushouts (considered as pullbacks in the opposite direction) are auxiliary tiles of **R**. In our example, the tile obtained by this construction is shown in Figure 6 (b).
4. **Resulting Tile**: The result of the application of rule r at match m is then given by the sequential composition of the tiles obtained in the last 2 steps: $Tile\ 1 * Tile\ 2$ (obtained by the application of rule *h-comp* of **R**). In the example, this is shown in Figure 7.

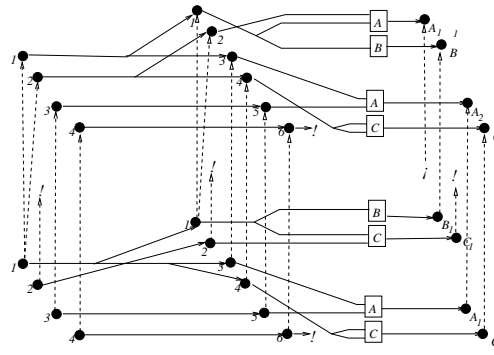


Fig. 7. Resulting tile

Note that we did not need to construct an auxiliary tile that corresponds to the pushout of arcs, as we did for the nodes (in tile 2). This is due to the fact that we do not allow to identify arcs neither in the rule nor in the match, and therefore the east-south component of tile 1 gives us already the arcs of the resulting graph of the derivation. In the classical SPO-setting, the corresponding pushout of arcs would have exactly the same effect: it would put all arcs that are in the right-hand side together all arcs that are in G and not in the image of the left-hand side of the rule (because the rule and the match are injective on arcs and the match is total). In case we would allow identification of arcs we would need a step for arcs corresponding to step 2.

Definition 18 (Algebraic Semantics of a Linear Ordered Graph Grammar). *Given a graph grammar $GG = (TG, IG, Rules)$, its* ***algebraic semantics****, denoted by $Alg(GG)$, is given by the tile rewrite system $\mathcal{R} = \langle \Sigma_{TG}, \Sigma_{TG}, R \rangle$, R is the set of tiles corresponding to the rules of GG.*

Proposition 3. *Let* $\sigma = G_0 \overset{r_1,m_1}{\Longrightarrow} G1 \Rightarrow \cdots \overset{r_n,r_m}{\Longrightarrow} G_n$ *be a sequential derivation of a graph grammar GG. Then there is a tile* $s \overset{a}{\underset{b}{\rightarrow}} t$ *in Alg(GG) such that s and t are the graphs corresponding to* G_0 *and* G_n*, respectively, and* (a, b) *corresponds to the derived rule of the sequential derivation* σ*.*

Proof sketch. As discussed above, all derivation steps of a grammar can be constructed using the tile corresponding to the rule, identity tiles and pullback tiles of nodes, composed using parallel and sequential composition. According to the definition of tile rewrite system, these tiles and operations can be used to build composed tiles, and therefore a tile corresponding to the derived rule must be in this tile rewrite system. Moreover, vertical composition of tiles is allowed, yielding the composition of co-rules. Note that the vertical composition of tiles is done by composing the west and east sides, and these sides correspond to the mapping of nodes and arcs of the involved graphs, respectively. The derived rule obtained by the composition $rn' \circ \cdots \circ r1'$ is also constructed componentwise (composition of partial graph morphisms).

Proposition 4. *Given a graph grammar and a tile rewrite system Alg(GG), the horizontal tiles of Alg(GG) that use one rule of GG (one generator tile) and have graphs as north and south sides correspond to derivation rules of GG.*

Remark: Horizontal tiles are the ones that are built without using the rule for vertical composition. If we allow more than one rule, the resulting tile would correspond to a parallel application of rules.

Proof sketch. Assume that a tile $s \overset{a}{\underset{b}{\rightarrow}} t$ uses one rule of GG and has graphs as north and south sides. If this tile is not obtained by composition of other tiles, it must be a generator, that is, a rule of GG. Generators correspond to rules of GG and thus such tiles correspond to derivations of kind $L \overset{r,id_L}{\Longrightarrow} R$, with $r : L \rightarrow R$. All tiles using tile Tr corresponding to rule r can obtained by composition can be rearranged (using the axioms) to the form $T1; (T2 \otimes Tr \otimes T3); T4$, where $T1$ has only nodes, $T2$ and $T3$ have graphs as north and south sides, and $T4$ has only arcs (elements of sorts Ai). Thus, $T1$ must be a PBnode tile because it must be the composition of identities and PBnode tiles (due to compositionality of colimits, the resulting tile must be a PBnode). $T2 \otimes T3$ can be seen as the context graph to which the rule is applied to. As $T2$ and $T3$ must be closed graphs, they can only be built by parallel composition of other closed graphs, and therefore must be compositions of identities (they can not be rules because Tr is the only rule in this tile, and can not be PBnodes or permutations of nodes/arcs because all these tiles have been rearranged to the component $T1/T4$). If $T4$ is not empty, it is just a permutation of arcs, depending on which permutation we take, we get all different isomorphic results of a direct derivation (on the arc component, on the node component this correspond to using permutation tiles to build $T1$).

5 Conclusion and Future Work

In this paper we defined a special kind of grammars, called linear ordered graph grammars, that can be used to describe distributed systems with mobility and

object-based systems. We showed how to model such grammars and their semantics in terms of tiles. In [11] graph grammars in the DPO approach have been modeled using tiles. The idea was that horizontal arrows correspond to graph morphisms, vertical arrows are pairs of an identity (on the gluing graph) and the name of a production and tiles correspond to DPO rules. In this way, the observations (vertical components of a tile) are not put into foreground (they are just names of rules). In our approach, we have modeled a graph morphism as vertical component of a tile. We could easily extend our approach to consider different kinds of arcs (observable and non-observable) corresponding to different horizontal and vertical signatures. This way we could model, for example, transactions, analogously to what have been done for Petri nets in [1]. The main aspects of our approach are:

• Types in graphs correspond to signatures in tiles: typed graphs correspond to gs-monoidal arrows of a restricted type on a signature. Graph isomorphism is directly represented by gs-monoidal axioms. Moreover, the set of all arrows has a natural meaning in terms of open graphs (due to space limitations, this topic was not further detailed here, see [17]).

• Graph morphisms of the special kind we need are obtained by choosing the vertical structure of tiles adequately. It is immediately apparent the correspondence between the vertical "wire structure" (again a variation of symmetric monoidal categories) and the properties of morphisms.

• A SPO can be represented by a horizontal tile construction. Again, the pushout property becomes concrete in terms of certain tiles which can be constructed starting from a finite number of auxiliary tiles [2].

• The effect of a derivation can be represented by a (flat) tile. We expect that canonical derivations are exactly represented by tiles equipped with their proof terms (in the Curry-Howard style). Note that we did not restrict to injective rules (as it is done in almost all true concurrency models for graph grammars).

• The observable effect of a derivation (which is not usually emphasized in the SPO/DPO literature) is very important in the interactive interpretation of our models of computations [2]. It represents the creation/fusions of channels and creation/termination of processes which took place in the computation. Typically, a useful way of connecting systems is via composition of their states, that can be done through shared channels. Therefore it is important to have information about these channels in interfaces (that is, make them observable) to obtain a compositional semantics.

• The model would be more satisfactory introducing also restriction/hiding of channels and processes, which would be straightforward (as shown by other papers [9]) in the tile model. This would correspond to new and interesting classes of graphs, making even more explicit the advantage of establishing a connection between graph transformations and tiles.

References

1. R. Bruni and U. Montanari, *Zero-Safe Nets: Comparing the Collective and Individual Token Approaches*, Information and Computation, Vol. 156, 2000, pp. 46–89.
2. R. Bruni, U. Montanari and F. Rossi, *An Interactive Semantics of Logic Programming*, TLP 6 (1): 647-690, Nov. 2001.
3. R. Bruni, J. Meseguer, U. Montanari, and V. Sassone, *Functorial models for Petri nets*, Information and Computation, Vol. 170, 2001, pp. 207–236.
4. A. Corradini and F. Gadducci, *An Algebraic Presentation of Term Graphs via Gs-Monoidal Categories*, Applied Categorical Structures, Vol. 7, 1999, pp. 299–331.
5. A. Corradini, U. Montanari and F. Rossi, *Graph processes*, Fundamentae Informatica, Vol. 26, no. 3-4, 1996, pp. 241–265.
6. F. Dotti and L. Ribeiro, *Specification of mobile code systems using graph grammars*, Formal Methods for Open Object-based Systems IV, Kluwer Academic Publishers, 2000, pp.45–64.
7. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*, World Scientific, 1999.
8. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner and A. Corradini, *Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach*, in [18], pp. 247–312.
9. G. Ferrari and U. Montanari, *Tile Formats for Located and Mobile Systems*, Information and Computation, Vol. 156, no. 1/2, 2000, pp. 173-235.
10. F. Gadducci and U. Montanari, *Comparing Logics for Rewriting: Rewriting Logic, Action Calculi and Tile Logic*, TCS, to appear. Available at http://www.di.unipi.it/ũgo/ABSTRACT.html#TCS-Asilomar.
11. R. Heckel, *Open Graph Transformation Systems: A New Approach to the Compositional Modelling of Concurrent and Reactive Systems*, Ph.D. thesis, Technical University of Berlin, 1998.
12. M. Korff, *True concurrency semantics for single pushout graph transformations with applications to actor systems*, Information Systems - Correctness and Reusability, World Scientific, 1995, pp. 33–50.
13. M. Korff, *Generalized graph structures with application to concurrent object-oriented systems*, Ph.D. thesis, Technical University of Berlin, 1995.
14. M. Löwe, *Algebraic approach to single-pushout graph transformation*, Theoretical Computer Science, Vol. 109, 1993, 181-224.
15. J. Meseguer and U. Montanari. *Mapping Tile Logic into Rewriting Logic*, Springer, LNCS 1376, 1998, pp. 62–91.
16. U. Montanari, M. Pistore and F. Rossi, *Modeling concurrent, mobile and coordinated systems via graph transformations*, The Handbook of Graph Grammars, vol. 3: Concurrency, Parallelism and Distribution, World Scientific, 1999, pp. 189–268.
17. U. Montanari and F. Rossi, *Graph Rewriting, Constraint Solving and Tiles for Coordinating Distributed Systems*, Applied Categorical Structures 7(4): 333-370; Dec 1999.
18. G. Rozenberg (editor), *The Handbook of Graph Grammars, vol. 1: Foundations*, World Scientific, 1997.
19. G. Taentzer, *Parallel and distributed graph transformation: Formal description and application to communication-based systems*, Ph.D. thesis, Technical University of Berlin, 1996.

Rule Invariants in Graph Transformation Systems for Analyzing Safety-Critical Systems

Julia Padberg and Bettina E. Enders

Technical University of Berlin, University of Essen
padberg@cs.tu-berlin.de, enders@informatik.uni-essen.de

Abstract. Automating software engineering activities for developing safety-critical systems reliably and correctly brings along the challenge to combine expressive specification methods with powerful analysis techniques. The focus of this paper lies in the analysis of graph transformation systems by analysis techniques transferred from Petri nets. Since Petri nets are famous for their powerful analysis techniques we have started to transfer several notions from Petri nets to graph transformation systems. Especially, invariants provide vast possibilities for analysing a model. Hence, we have transferred transition invariants to rule invariants in graph transformation systems. This allows a suitable analysis of graph transformation systems as is exemplified by a safety-critical system specification in the area of human-computer interaction.

1 Introduction

Automating software engineering activities requires a deep understanding of the software systems to be developed. Such a comprehension can be achieved suitably by investigating the respective software specifications which means formalizing and subsequently analyzing the systems. This does not only lead to more reliable and correct systems but, moreover, allows for using tool support in order to develop the software systems. Generally, various specification methods exist each having different advantages and disadvantages. Moreover, the specification methods provide different analysis techniques. It is a challenge to combine expressive specification methods with powerful analysis techniques. The idea of this paper is to combine the highly expressive specification methods of graph transformation with the powerful analysis techniques of Petri nets. Such a combination provides excellent investigation possibilities for dynamic systems, especially for the highly important safety-critical aspects of safety-critical systems.

In this contribution we undertake the transfer of Petri net notions to graph transformation systems in order to obtain similar possibilities of analyzing the underlying model. As mentioned above analysis of the model is especially crucial in the area of safety-critical systems due to their high quality demands for the software. An important example provides the entire complex aspect of awareness by the pilot interacting with the flight management system on the flight-deck of an aircraft during the flight. We express the crucial notions for invariant analysis in terms of graphs, graph rewrite rules and transformations. The difficulty of this

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 334–350, 2002.
© Springer-Verlag Berlin Heidelberg 2002

transfer stems from the higher expressiveness and complexity of graph transformation systems in comparison to Petri nets. Among the notions transferred from Petri nets to graph transformation systems in this contribution are reachability, liveness, boundedness, and especially, the notion of transition invariant.

The paper is organized as follows. In chapter 2 a graph transformation system specifying a suitable part of a safety-critical system concerning the interaction between the pilot and the flight management system on the flight deck of an aircraft is presented. This example system is used in order to demonstrate the particular notions in the subsequent two chapters. There we present *remarks* to demonstrate the notions in terms of the given example. The transfer of the relevant Petri net notions to graph transformation systems is reviewed in chapter 3. In chapter 4 we introduce rule invariants and give a characterization based on the construction of the minimal graphs of change of the left hand and right hand sides of the graph rewrite rules. There we also give the main result, namely that a finite, bounded and live graph transformation system is covered by rule invariants. Because of space limitations we have omitted the proofs in this paper, but they can be found in [PE01]. Finally, this contribution closes with a short summary.

2 Graph Transformation System Specifying a Safety-Critical System

Concerning the goal of transferring notions of Petri nets to graph transformation systems an accompanying example is presented in this chapter: a graph transformation system specifying a part of a safety-critical system, namely the interaction between the pilot or the crew and the flight management system (FMS) on the flight deck of an aircraft. Although several notions of Petri nets can be transferred to graph transformation systems comprising directed attributed graphs in this contribution as a first step graphs without labels at nodes and edges are considered for clarity. Dealing with attributed graphs will then be the focus of future work.

In section 2.1 the complex attributed start graph G_{Start} specifying the start state of the interaction between pilot and FMS will be presented and explained for motivation. The graph rewrite rules of the underlying graph transformation system are only briefly mentioned with the aim of completeness. A detailed description of the entire system can be found in [ES01] where the aim is to describe a continuous collaborative specification process of human-computer interaction.

Transferring Petri net notions to graph transformation systems can be demonstrated clearly by using a suitable small part of the complex interaction between the pilot and the FMS. Therefore, a particular part of this system will be considered in a detailed manner in section 2.2. It will be specified by very simple graphs and corresponding transformation rules. All graphs playing a role in this context will be entirely unlabeled. The considered part of the interaction between pilot and FMS will concern, especially, the highly important mode changes during the flight.

2.1 Interaction between the Pilot and the FMS

As pointed out already in [ES01] and [Suc97] interaction between human and machine can nicely be understood by modelling it using the concept of information resources (cf. [WFH99], [WFH96], [FWH96]) and by specifying it formally by the notation of graph transformation systems (cf. [CMR+97]).

The specification of the interaction between the pilot and the flight management system on the flight deck of an aircraft is given here at an abstract level. This specification captures the entire interaction between the pilot and the FMS required to tackle the stages beginning with the take-off and ending up with the touch-down of the aircraft. The start graph G_{Start} of the graph transformation system specifying this interaction is presented in figure 1. It appears in an intuitive manner with the aim of comprehensibility. However, the graph consists of nodes, edges and attributes for names and for types of graph elements, as well as attributes for describing certain features of particular graph elements.

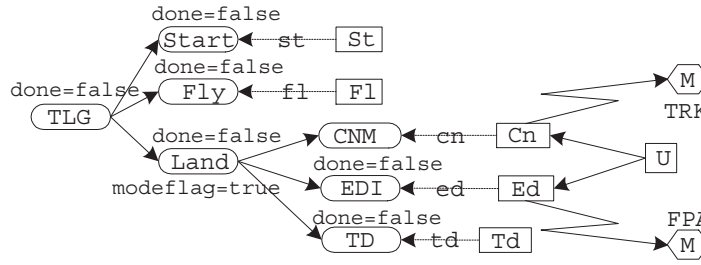


Fig. 1. Start graph G_{Start} of the interaction specification between the pilot and the flight management system (FMS) on the flight deck of an aircraft

The root node on the left hand side of graph G_{Start} specifies the top level goal `TLG` with the meaning of a specific flight. This goal is decomposed into three subgoals `Start`, `Fly` and `Land`, landing in turn is decomposed into the subgoals `CNM` (Changing Navigation Mode), `EDI` (Entering Descent Input) and `TD` (Touch Down). All goals not further decomposed are associated to subparts of the system's user interface specified by the right hand side of the graph. Action-effect edges indicate that actions of a specific type could be performed within the respective parts of the system's user interface in order to approach or, even, to complete the respective goals. The goal `Start`, e.g., can be completed by performing the action `st` within the user interface part `St`. Whether a goal has been completed already or not is indicated by the boolean attribute `done` attached to the respective node. Solely, the goal `CNM` does not have such an attribute because this goal exists within a certain context of switching modes.

The navigation and the descent mode constitute an important safety-critical aspect within this system. A mode change during the flight by the pilot is sometimes necessary in order to comply with radar guidance. The safety-critical aspect in this context is, that these modes are automatically coupled in the autopilot system so that one mode change enforces the change of the other one

immediately ([KJ96]). In fact, the pilot has always to be aware of the actual mode values, especially of the automatically coupled mode change she or he did **not** intend. For reason of this tight connection the parts of the system's user interface corresponding to the goals `CNM` and `EDI` are grouped together by specifying them as subwidgets `Cn` and `Ed`, respectively, with a common superwidget `U`. Because of the importance of the actual modes in every situation during the flight a mode node is added attributed by the current mode value of its associated system part and connected to it by a special edge. The navigation mode has the actual mode value `TRK` (*Track*) and the descent mode has the actual mode value `FPA` (*Flight Path Angle*). Changing the modes would lead to the respective values `HDG` (*Heading*) and `V/S` (*Vertical Speed*).

Graph rewrite rules specify the interactions. The system discussed here contains six rules specifying its dynamics. One rule describes the start of the flight, a second one the opening of the flying stage. During the flying stage the pilot can change the modes for navigation and descent arbitrarily often: if she or he changes one mode which is specified by one rule, the other mode changes automatically specified by an enforced second rule. The safety-critical aspect at this point concerns the awareness of the automatically coupled mode change by the pilot: she or he **has to** be aware of the *new* value and **must not** rely upon the *old* one. A mode specification of this kind together with special *modal subdialogs* (cf. [ES01], [Suc97]) allows to ensure that the pilot is always aware of the current mode situation. This is a very important issue in order to avoid mode errors (due to confusion of the pilot caused by the modes) which already have led to accidents claiming casualties in the past (for a detailed information see [KJ96] and [Mon92]). At the end, the pilot can enter the descent rate and finally perform the touch-down. Based on changes of the modes for navigation and descent the transfer of the notion *transition invariant* concerning Petri nets to the so-called *rule invariant* for graph transformation systems is demonstrated comprehensibly in chapter 4.

After this motivation concerning the whole complex safety-critical system (cf. [ES01]) a small part of it is considered in section 2.2 where it is specified by a suitable simple graph transformation system based on unlabeled graphs. This subsystem is introduced with the goal of demonstrating the transfer of several notions of Petri nets to graph transformation systems in chapters 3 and 4.

2.2 Subsystem of the Interaction between Pilot and FMS

The interaction between the pilot and the FMS on the flight deck of an aircraft presented in section 2.1 will now be simplified as follows. The focus lies upon the mode changes only, that is being in the flying stage and switching between the two possible mode value combinations concerning navigation and descent mode during the flying stage. The *combined* change of mode values is chosen here with the aim of comprehensibility. Therefore, the safety-critical aspect of the awareness by the pilot is not yet considered in a very detailed manner in this simple system. However, based on it and, especially, on the notion *rule invariant* for graph transformation systems according to *transition invariant*

concerning Petri nets this will become possible within an extended and more complex system. Further stages like opening the flight, entering the descent input and performing the touch-down are omitted with the aim of simplicity. Thus, the graph transformation system specifying this simple example system consists of an unlabeled start graph and two graph rewrite rules (cf. figures 2, 3 and 4). The start graph G_0 (figure 2) characterizes the flying stage (nodes v_1, v_2 and edges e_1, e_2 inbetween). G_0 further characterizes the existence of the two kinds of mode: the navigation mode (specified by node v_3 and edge e_3) and the descent mode (specified by node v_4 and edge e_4). The single edge e_0 attached to node v_2 specifies the actual mode value combination `TRK` – `FPA` concerning navigation and descent mode respectively. A single edge belonging to every state in this system specifies the actual mode value situation during the flight: the pilot can arbitrarily often change one of the two modes implying thus the automatically coupled change of the other mode. Thus, a state in which a single edge is attached to the other node v_1 specifies the second possible mode value combination `HDG` – `V/S` concerning navigation and descent mode respectively.

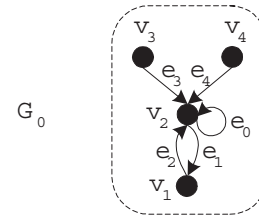


Fig. 2. Start graph G_0 of the simple subsystem specifying the interaction between pilot and FMS

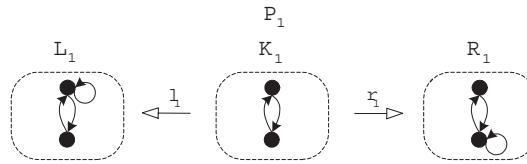


Fig. 3. Graph rewrite rule P_1 specifying a change of the actual mode value combination concerning navigation and descent mode

Such combined mode value changes from the mode value combination `TRK` – `FPA` to `HDG` – `V/S` and vice versa are specified by the two graph rewrite rules P_1 (figure 3) and P_2 (figure 4). **Both** rules change the actual mode value combination into the respective other one. Thus, in all states a single edge at node v_1 *or* v_2 specifies the actual mode combination `HDG` – `V/S` *or* `TRK` – `FPA` respectively.

The graph transformation system specifying this simple subsystem concerning the interaction between pilot and FMS is defined by

$$GG_{Pilot-FMS} = \{G_0, P\} \text{ where}$$
$$P = \{P_1, P_2\} \text{ and } \mathrm{G_0} = \{\mathrm{V}, \mathrm{E}, \mathrm{s}, \mathrm{t}\} \text{ with}$$
$$V = \{v_1, v_2, v_3, v_4\},\ E = \{e_0, e_1, e_2, e_3, e_4\},$$
$$s(e_1) = v_2,\ t(e_1) = v_1,\ s(e_2) = v_1,\ t(e_2) = v_2,$$

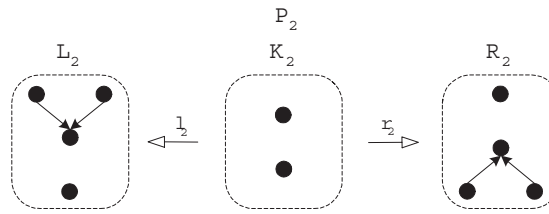


Fig. 4. Graph rewrite rule P_2 specifying a change of the actual mode value combination concerning navigation and descent mode

$$s(e_0) = t(e_0) = v_2,$$

$$s(e_3) = v_3,\ t(e_3) = v_2,\ s(e_4) = v_4,\ t(e_4) = v_2.$$

Because of the simplicity of this system the dynamics of its graph transformation system specification $GG_{Pilot-FMS}$ can be described as follows (as can easily be seen):

$$G_0 \overset{(P_1, m_1)}{\underset{(P_2, m_2)}{\Longleftrightarrow}} G_1$$

Here, the morphisms m_1 and m_2 are the matches at which the respective graph rewrite rules P_1 and P_2 are applied to the graphs G_0 and G_1.

The graph transformation system $GG_{Pilot-FMS}$ specifying the simple subsystem of the interaction between pilot and FMS will serve for demonstration of transferring notions concerning Petri nets like *marking*, *enabling*, *firing*, *reachability*, *liveness*, *boundedness* and, especially, *transition invariant*, to graph transformation systems in chapter 4. Based on these results many important aspects in the area of safety-critical systems like the entire complex aspect of the awareness by the pilot during the flight may be formulated more concretely and more comprehensibly. Moreover, such a formulation allows a suitable investigation of the corresponding model. And it is especially the safety-critical features which have to be considered deeply while developing systems in this area.

The reason for these suitable investigation and analysis possibilities lies in the high expressiveness of graph transformation. Additionally, the transfer of notions from Petri nets which are famous for their powerful analysis techniques helps much in analyzing models expressed in terms of graph transformation systems. For example, the computation of all reachable graphs within the simple safety-critical subsystem introduced above provides all possible states specifying all possible situations during the flight. These situations can then be further analyzed with the goal of characterizing the safety-critical ones the pilot by all means has to be aware of during the flight. This shows the importance and advantage of transferring the notion *reachability* from Petri nets to graph transformation systems.

Usually, an adequate model of a system should be bounded, finite and live. So, if the characterization of rule invariants presented in this paper shows that the

graph transformation system is not covered by rule invariants this gives a strong suspicion that the model is severely wrong. Suppose, for example, the simple safety-critical subsystem introduced above is not covered by rule invariants and especially, it is not live. This implies that at least one of its two graph rewrite rules is not live. Thus, there must exist a reachable state graph such that the respective rule can not be applied anymore according to the dynamics of the system. This, in turn, shows that one important mode change which is highly necessary for a safe flight can not be performed in the corresponding model. Therefore, the model has to be corrected by all means with respect to this safety-critical aspect of mode change! On the other hand, let an already complex system specification be covered by a rule invariant. This rule invariant then contains all graph rewrite rules concerning changes of modes necessary in order to comply with radar guidance. The fact that the corresponding transformation sequence is cyclic allows to make constructive statements about the different mode changes and, especially, about how they influence each other. Thus, the corresponding model helps much in detecting and characterizing especially the safety-critical aspects in the system.

In order to allow a powerful analysis of safety-critical systems as described above several notions concerning Petri nets are transferred to graph transformation systems in chapter 3, and rule invariants are introduced in chapter 4.

3 Review of Graph Transformations and the Relation to Petri Nets

In this chapter we give a short review of the relevant notions in graph transformation systems. For limitations of the paper‘s length we have to assume the reader is familiar with the basic notions of Petri nets. The relation between graph transformation systems and Petri nets is well-known, so we merely sketch it. But we explicitly give the corresponding notions for graph transformation systems in order to provide a sound mathematical foundation of invariants in the next chapter.

Obviously, the paper is based on results in Petri nets, as e.g. in [Rei85]. Related work concerns on the one hand different approaches to verifying graph transformation systems using temporal logic as in [GHK98,Koc99]. On the other hand, a first attempt to analyze cycles in graph transformation systems has been presented in [Mül98]. But no transfer of the notion of invariants from Petri nets has been undertaken. Many different ideas to express Petri nets as special cases of graph transformation systems [Kre81,KR95,Cor00] have already been proposed for various reasons, but for our approach we transfer the following notions directly to graph transformations. Because the result of graph transformation is defined up to isomorphism throughout this paper we have to assume that there is some suitable standard representation of graph and graph transformations [CEL$^+$94].

– The **initial marking** of a Petri net denotes the initial state of the system from which the process develops. Clearly this corresponds to the initial graph in a graph transformation system.

- A **marking** of a net denotes some state of the system, this intuitively corresponds to a graph. The higher complexity of graph transformation systems can be seen clearly. A state of a Petri net is given by a (multi-)set, namely of tokens, whereas a state of a graph transformation system is a graph. Thus the description of a single state is much more complex.
- The **firing** of a transition describes a local and deterministic system change. Analogously, a graph rewrite rule together with a given match describes a local and deterministic system change.
- A marking is **reachable** if it can be obtained from the initial marking by firing several transitions. And a graph is reachable if it is deduced from the initial graph via some transformation steps.
- **Liveness** of a net means that from any reachable marking every transition can fire eventually after firing some others. So, reachability of a graph transformation system means that from any reachable graph every graph rewrite rule can be applied after some more transformation steps.
- A net is **bounded** if the amount of tokens on each place in any reachable marking has some upper bound. Analogously, if the amount of nodes and edges of each reachable graph has an upper bound the graph transformation system is bounded.

We first reconsider the usual notions of unlabeled graphs and rules and transformations in the *algebraic double-pushout (DPO) approach* (cf. [Ehr79, CMR+97]). Unlabeled graphs are given in the algebraic style by the sets of nodes and edges together with the functions that map each edge to its source and target node.

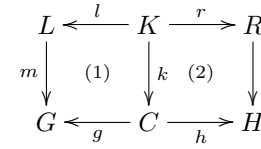
Definition 1. *A* graph $G = (E, V, s, t : E \to V)$ *is given by* E *the set of edges and* V *the set of nodes. The function* $s : E \to V$ *maps each edge to its source node and the function* $t : E \to V$ *maps each edge to its target node.*

A graph rewrite rule in the DPO approach is given by a span of morphisms, where the interface K denotes the positive application condition in the sense that K has to be present in the target Graph G. $K \xrightarrow{l} L$ describes by $L \backslash K$ what is deleted in the sense that the pushout complement C is the context that is not changed. $K \xrightarrow{r} R$ describes by $R \backslash K$ what is added since the resulting graph H is pushout of C and R.

Definition 2.

1. *A* rule $L \xleftarrow{l} K \xrightarrow{r} R$ *with name* p *in* **Graph** *consists of the graphs* L, K *and* R, *called the left hand side, the interface (or gluing object) and the right hand side respectively, and two morphisms* $K \xrightarrow{l} L$ *and* $K \xrightarrow{r} R$ *with both morphisms* l, r **injective**.
   We have a rule without empty interface *if* K *consists at least of a node.*

2. *Given a rule* $(L \xleftarrow{l} K \xrightarrow{r} R)$ *with name* p, *a* direct transformation $G \overset{(p,m)}{\Longrightarrow} H$ *from a graph* G *to a graph* H *using the rule* p *is given by the following two pushout diagrams* **(1)** *and* **(2)** *in the category* **Graph**.
The morphism $L \xrightarrow{m} G$ *is called match of* L *in* G *and* m *is also* **injective**.



Note, l, r, m injective implies that all morphisms in the diagram are injective. Based on this we define graph transformation systems where the rules have explicit names (see also [HCEL96]). Moreover, we have to assume that the interfaces of the rules are not empty.

Definition 3. *A* graph transformation system $GTS = (G_S, \mathbf{Rules}, \pi)$ *consists of a start graph* $G_S \in Obj_{\mathbf{Graph}}$, *the class of rules* $\mathbf{Rules} = \{(L \xleftarrow{l} K \xrightarrow{r} R) | l, r \in \mathcal{MOR}_{\mathbf{Graph}}$ *and* l, r *injective*$\}$, *and the naming function* $\pi : P \to \mathbf{Rules}$.
A graph transformation system GTS *is finite if* P *is finite.*

Petri nets[1] are given in their classical presentation with $N = (S, T, F, W, M_0)$ where we have the flow relation $F \subseteq S \times T \cup T \times S$, the weight $W : F \to \mathbb{N}^+$, and the initial marking $M_0 : S \to \mathbb{N}$. They correspond to graph transformation systems based on labeled graphs without any edges (see [Kre81]). Here we consider in a first step only unlabeled graphs. Nevertheless, the intuition can be transferred from Petri nets to these graph transformation systems as well.

In this analogy the initial marking $M_0 : S \to \mathbb{N}$ of a Petri net corresponds to the start graph $G_S \in Obj_{\mathbf{Graph}}$ of a graph transformation system $GTS = (G_S, \mathbf{Rules}, \pi)$. The enabling of a transition is given by $M(s) \geq W(s,t)$ for all $s \in {}^\bullet t$. A rule can be applied if there is $m : L \to G$ such that the gluing condition (see [CMR+97]) is satisfied and m is **injective**. The firing of a transition $M[t > M'$ corresponds to a direct transformation $G \overset{(p,m)}{\Longrightarrow} H$. This transfer of basic notions from Petri nets to graph transformation systems can now be demonstrated using the simple example system introduced in section 2.2.
Remark 1. $GG_{Pilot-FMS}$ is a $GTS = (G_S, \mathbf{Rules}, \pi)$ (**Rules** and π as in definition 3) with $G_S = G_0$ and $P = \{P_1, P_2\}$ (cf. figures 2, 3 and 4). The GTS $GG_{Pilot-FMS}$ is finite since P is finite. The GTS $GG_{Pilot-FMS}$ thus corresponds to a finite Petri net.

Remark 2. The start graph $G_0 \in Obj_{\mathbf{Graph}}$ of $GG_{Pilot-FMS}$ corresponds to the initial marking of a Petri net.

Remark 3. The match $m_1 : L_1 \longrightarrow G_0$ of the graph rewrite rule $P_1 \in P$ corresponds to an enabling within a Petri net as m_1 is injective and the gluing condition for applying P_1 to G_0 at this match m_1 is satisfied: $l_1^{-1}(m_1^{-1}(v_2)) \in K_1$ for $\mathrm{v}_2 \in \mathrm{G}_0$ (cf. figure 5).

[1] More precisely, we here have Place/Transition nets without capacities.

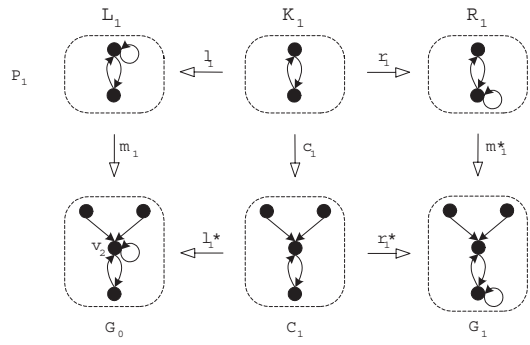


Fig. 5. Application of graph rewrite rule P_1 to graph G_0 at match m_1 (yielding graph G_1)

Remark 4. The direct transformation $G_0 \overset{(P_2,m_2)}{\Longrightarrow} G_1$ corresponds to the firing of a transition belonging to a Petri net.

Now we can transfer reachability in Petri nets, computed by $M_0 \in [M_0 >$ and $M' \in [M_0 > \wedge M'[t > M'' \implies M'' \in [M_0 >$, to graph transformation systems in the following way, if we assume some standard representation as in [CEL$^+$94].

Definition 4. *The set of reachable graphs* $[G_S >$ *of a graph transformation system* $GTS = (G_S, \mathbf{Rules}, \pi)$ *in some standard representation [CEL$^+$94] is computed by:*
$G_S \in [G_S >$
$G' \in [G_S >$ *and* $G' \overset{(p,m)}{\Longrightarrow} G''$ *implies* $G'' \in [G_S >$

Remark 5. The set of reachable graphs of the GTS $GG_{Pilot-FMS}$ is the set $\{G_0, G_1\}$ as (cf. definition 4):

$$G_0 \overset{(P_1,m_1)}{\Longrightarrow} G_1 \overset{(P_2,m_2)}{\Longrightarrow} G_0.$$

Liveness of Petri nets is defined by $t \in T$ is live if and only if $\forall M \in [M_0 >$: $\exists M' \in [M >$: $M'[t > M''$. Moreover, N is live iff $\forall t \in T : t$ is live.
The analogous definition for graph transformation systems is the following.

Definition 5. *Given a graph transformation system* $GTS = (G_S, \mathbf{Rules}, \pi)$, *then* $p \in P$ *is live if and only if*
$\forall G \in [G_S >$: $\exists G' \in [G >$: $G' \overset{(p,m)}{\Longrightarrow} G''$
GTS *is live if and only if* p *is live for all* $p \in P$.

Remark 6. The graph rewrite rule P_1 is live: $G_1 \overset{(P_1,m_1)}{\Longrightarrow} G_0$ and $G_0 \overset{(P_1,m_1)}{\Longrightarrow} G_1$ (cf. definition 5). The graph rewrite rule P_2 is live as well: $G_1 \overset{(P_2,m_2)}{\Longrightarrow} G_0$ and $G_0 \overset{(P_2,m_2)}{\Longrightarrow} G_1$.

Remark 7. The *GTS* $GG_{Pilot-FMS}$ is live since both of its graph rewrite rules P_1 and P_2 are live (cf. definition 5).

Similarly, we can transfer the concept of boundedness. Boundedness of Petri nets is defined by
N is bounded if and only if $\exists n \in \mathbb{N} : \forall s \in S : \forall M \in [M_0 >: M(s) \leq n$
Boundedness of graph transformation systems is defined in the following way.

Definition 6. *A graph transformation system GTS is bounded if and only if:* $\exists n \in \mathbb{N} : \forall G \in [G_S >: |V \cup E| \leq n$

Corollary 1. *Boundedness of GTS implies that the set of reachable graphs* $[G_S >$ *is finite.*

Remark 8. The *GTS* $GG_{Pilot-FMS}$ is bounded because $\forall G \in [G_0 >: |V \cup E| \leq n \in \mathbb{N}$. Of course, the boundedness of the *GTS* $GG_{Pilot-FMS}$ implies that the set of reachable graphs $[G_0 >$ is finite: $[G_0 >= \{G_0, G_1\}$ (cf. definition 6 and corollary 1).

4 Rule Invariants in Graph Transformation Systems

Transition invariants describe in Petri nets possible cycles in a net. Correspondingly, rule invariants describe possible cyclic transformations. The main advantage of transition invariants is their computation using linear algebra. Since graph transformations are far more expressive than Petri nets this computation cannot be transferred. Instead we have developed a more complex characterization of rule invariants. Transition invariants can be computed from the static net structure and need not the investigation of possibly infinite runs. This makes them very attractive for analysis and verification. Similarly, rule invariants can be computed from the rules of a graph transformation system and need not the investigation of all possibly infinite computations.

Transition invariants in Petri nets are transition vectors (or multisets, linear sums, or elements of the free commutative monoid over T) such that the sum of the predomain of the involved transitions equals the sum of the postdomain. More precisely, we can characterize a transition invariant by $\sum_{t \in T} \lambda_t * t$ so that for all reachable markings we have:

$$m[\textstyle\sum_{t \in T} \lambda_t * t > m$$

There the linear sum $\sum_{t \in T} \lambda_t * t$ denotes the firing of each transition $t \in T$. And the coefficient $\lambda_t \in \mathbb{N}$ states how often that transition is fired. For Petri nets transition invariants $\sum_{t \in T} \lambda_t * t$ can be achieved just by adding places. In case of graph transformation we suggest to use the following constructions.

The basic intuition of this construction of rule invariants is to glue together all left hand sides resulting in a graph L. This construction glues everything that is within the interfaces and keeps anything else distinct. The same construction for the right hand sides yields R. Hence, we make nodes and edges that are

really deleted (respectively added) explicit. So, if there is a cyclic derivation (up to isomorphism) the resulting graph is isomorphic to the source graph. Hence, for each subgraph that is deleted at some step during the derivation this subgraph has to be added in some other steps of the derivation and vice versa. So we have a rule invariant if L and R are isomorphic.

The subsequent definition of the minimal graphs of change for the left and right hand sides yields the gluing of the left hand sides and the gluing of the right hand sides respectively.

Definition 7. *Given rules without empty interfaces* $\pi(p_i) = L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i$ *for* $i = 1, .., n$ *then we can construct the* minimal graph of change *of the left (respectively right) hand side by:*
the product K *of all interfaces* $K = \prod_{i=1}^{n} K_i$ *and the multiple pushout* $(L, \widehat{l_i})$ *of all left hand sides* $K \to K_i \xrightarrow{l_i} L_i$ *(respectively the multiple pushout* $(R, \widehat{r_i})$ *of all right hand sides* $K \to K_i \xrightarrow{r_i} R_i$*) (cf. figure 7).*

The next lemma states the fact that everything from the interfaces is glued together in Items 1 and 2. Items 3 and 4 state that everything else is kept distinct.

Lemma 1. *Given the minimal graph of change* L *of the left hand sides, then we have*

1. *for all* $v \in K_{iV}$ *there is exactly one* $v_L \in L_V$*, so that* $\widehat{l_i} \circ l_i \circ k_i(v) = v_L$
2. *for all* $e \in K_{iE}$ *there is at most one* $e_L \in L_E$*, so that* $\widehat{l_i} \circ l_i \circ k_i(e) = e_L$
3. *for all* $v_i \in L_i \backslash l_i(K_i)$ *and* $v_j \in L_j \backslash l_j(K_j)$
   we have $\widehat{l_i}(v_i) = \widehat{l_j}(v_j)$ *implies* $v_i = v_j$
4. *for all* $e_i \in L_i \backslash l_i(K_i)$ *and* $e_j \in L_j \backslash l_j(K_j)$
   we have $\widehat{l_i}(e_i) = \widehat{l_j}(e_j)$ *implies* $e_i = e_j$

Analogously for the minimal graph of change R *of the right hand sides.*

Note that for the minimal graph of change L (and R respectively) all interfaces are glued together, namely to one node and at most one edge, and all other nodes and edges are kept distinct.

Definition 8. *Given rules without empty interfaces* $\pi(p_i) = (L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i)$ *for* $i = 1, .., n$ *then there is the rule invariant, written* $\bigsqcup_{i=1}^{n} p_i$ *or* $p_1 \sqcup p_2 ... \sqcup p_n$*, if we have for the minimal graphs of change* L *and* R *(as in Definition 7):*

$$L \cong R$$

A graph transformation system is covered by rule invariants if there is some rule invariant $\bigsqcup_{p \in P} \lambda_p * p$ *with* $\lambda_p \in \mathbb{N}$ *and all* $\lambda_p \geq 1$*.*

Similar to transition invariants in Petri nets the rule invariant $\bigsqcup_{p \in P} \lambda_p * p$ denotes which graph rewrite rules are applied. λ_p states how often the rule p is

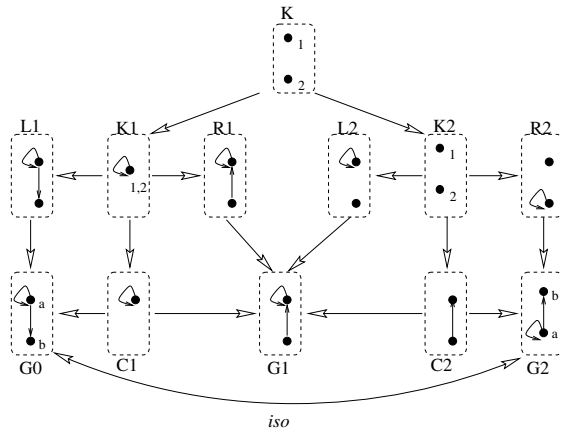


Fig. 6. Not a cyclic transformation

applied. Hence, a graph transformation system is covered by rule invariants if all rules are applied at least one time.

In order to express the main characteristics of invariants we need the notion of cyclic transformation sequence. A cycle of transformations is a sequence of rules starting and ending with the same graph up to isomorphism. In order to define precisely a cycle of transformations we have to exclude isomorphisms that are not compatible with the transformation sequence (see also example 1).

Definition 9. *A transformation sequence* $G_0 \overset{(p_1,m_1)}{\Longrightarrow} .. \overset{(p_n,m_n)}{\Longrightarrow} G_n$ *is called* cyclic *if the following two conditions hold:*

1. *The start and end graphs are isomorphic:*
   there exists an isomorphism $iso : G_0 \to G_n$
2. *this isomorphism is compatible with the transformation sequence:*
   $K \to L_1 \to G_0 \xrightarrow{iso} G_n = K \to R_n \to G_n$

The above conditions require not only isomorphic start and end graphs, but that the arrows from the product K of all interfaces commute through this isomorphism. This means, that the origin of mapped elements of the start graph are respected.

Example 1. In figure 6 we have a transformation of two steps with isomorphic start and end graphs, $G_0 \cong G_2$. Nevertheless, we have $K \to L_1 \to G_0 \xrightarrow{iso} G_2 \not\models K \to R_2 \to G_2$, so the second condition is violated.
The mapping of the nodes is indicated by the labeling with 1 and 2, respectively with a and b.
The morphism $K \to L_1 \to G_0 \xrightarrow{iso} G_2$ identifies the nodes 1 and 2, where as the morphism $K \to R_2 \to G_2$ keeps them seperate. So, $K \to L_1 \to G_0 \xrightarrow{iso} G_2$ **does not commute** with $K \to R_2 \to G_2$.

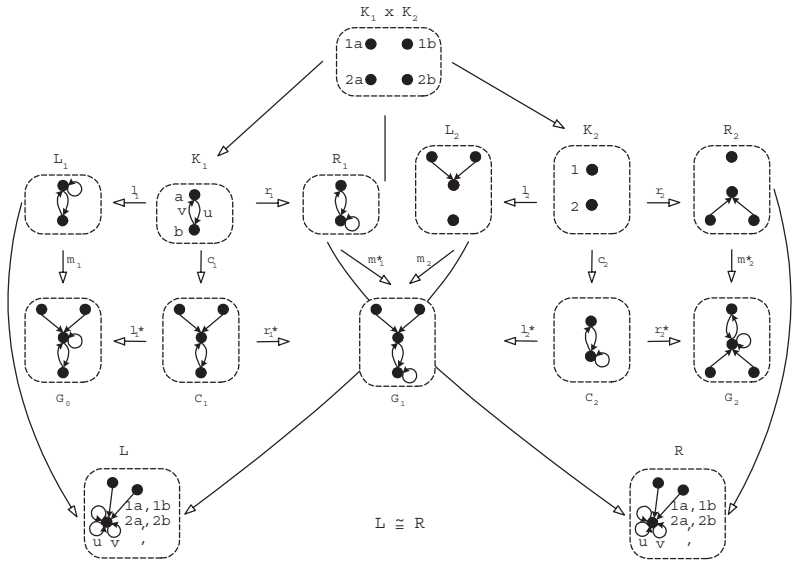


Fig. 7. Presentation of the rule invariant $\bigsqcup_{i=1}^{2} P_i = P_1 \sqcup P_2$ ($P = \{P_1, P_2\}$) of the GTS $GG_{Pilot-FMS}$

The next theorem states that cyclic transformations imply rule invariants. Hence, this notion corresponds to transition invariants in Petri nets.

Theorem 1. *Rule Invariants*
Given a cyclic transformation sequence $G_0 \overset{(p_1,m_1)}{\Longrightarrow} .. \overset{(p_n,m_n)}{\Longrightarrow} G_n$ *and* $\pi(p_i) = L_i \xleftarrow{l_i} K_i \xrightarrow{r_i} R_i$ *rules without empty interfaces and all morphisms* l_i, r_i, m_i *injective, then we have the rule invariant* $\bigsqcup_{i=1}^{n} p_i$.

Remark 9. The graph rewrite rules P_1 and P_2 of the GTS $GG_{Pilot-FMS}$ constitute the rule invariant $\bigsqcup_{i=1}^{2} P_i = P_1 \sqcup P_2$ since for the minimal graphs of change L and R (as in definition 7): $L \cong R$ (cf. figure 7 and definition 8. The annotations `a`, `b`, `1`, `2`, `u` and `v` at the graph elements are given here with the aim of comprehensibility.).

Remark 10. The rule invariant $\bigsqcup_{i=1}^{2} P_i$ is a rule invariant for the GTS $GG_{Pilot-FMS}$ (cf. figure 7 and theorem 1) since $G_0 \cong G_2$, the transformation sequence $G_0 \Longrightarrow G_1 \Longrightarrow G_2$ is cyclic and for $1 \leq i \leq 2$ the morphisms l_i, r_i and m_i (and therefore l_i^*, r_i^* and m_i^* as well) are all injective.

Now we can transfer one of the main results concerning invariants from Petri nets to graph transformation systems. A net that is finite, live and bounded is covered by transition invariants ([Rei85]).

Theorem 2. *Covered by Rule Invariants*
A finite, bounded and live graph transformation system is covered with rule invariants, that is there is an $\bigsqcup_{i=1}^{n} p_i$ *so that* $\{p_i | i = 1, ..., n\} = P$.

Remark 11. The GTS $GG_{Pilot-FMS}$ is covered by the rule invariant $\bigsqcup_{i=1}^{2} P_i = P_1 \sqcup P_2$ since it is finite, bounded and live, and since $P = \{P_1, P_2\}$ (cf. theorem 2).

5 Conclusion

For the goal of automating software engineering activities wrt the development of safety-critical systems this contribution suggests a suitable combination of expressive specification methods and powerful analysis techniques. We have presented a first step of transferring analysis techniques of Petri nets to graph transformation systems and conclude with a short summary in the following. Graph transformation systems become more and more an adequate modeling technique in the area of safety critical systems. But this application area has strong demands on analysis and verification methods of the modeling technique. In contrast to graph transformation systems Petri nets are well investigated with respect to analysis and verification. Nevertheless, as graph transformation systems are much more powerful in their expressiveness, it is often advisable to use graph transformation systems.

In this contribution we have undertaken a first step to transfer the possibilities of analysis known for Petri nets to graph transformation systems. We gave a review of how some basic Petri net notions (namely liveness, boundedness, and reachability) can directly be expressed in terms of graph transformation systems. These correspondences are obvious and well-known.

We then have transferred the very prominent notion *transition invariant* from Petri nets to *rule invariant* in graph transformation systems. Analogously to a transition invariant a rule invariant denotes a possible cycle. The main result of this contribution is the characterization of rule invariants independent of concrete transformation sequences. This characterization corresponds to the computation of transition invariants in Petri nets based on the solution of the linear equation system induced by the net's incidence matrix. The computation of rule invariants uses a new construction of the minimal graphs of change of the left and right hand sides of the graph rewrite rules. So, only the changes induced by the rules are kept distinct. If these changes of the left hand sides are the same as the changes of he right hand sides this implies possible cycles of transformations. Given these rule invariants we could transfer a main result from Petri nets concerning liveness, boundedness and transition invariants to graph transformation systems.

The results of this paper provide a first step in the direction of a methodology for automating software engineering activities for the development of safety-critical systems by using analysis techniques of graph transformation systems which have been transferred from Petri nets.

References

CEL+94. A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Note on standard representation of graphs and graph derivations. In *Proc. Graph Grammar Workshop Dagstuhl 93*, pages 104–118, 1994. Lecture Notes in Computer Science 776.

CMR+97. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. *Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach*, volume 1, chapter 3, pages 163–245. World Scientific, 1997. G. Rozenberg (ed.).

Cor00. Andrea Corradini. Concurrent computing: from Petri nets to graph grammars. In A. Corradini and U. Montanari, editors, *Electronic Notes in Theoretical Computer Science*, volume 2. Elsevier Science Publishers, 2000.

Ehr79. H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars (A Survey). In *Graph Grammars and their Application to Computer Science and Biology*. Springer LNCS 73, 1979.

ES01. B. E. Enders-Sucrow. Describing a Continuous Collaborative Specification Process of Human-Computer Interaction by Graph Rewriting. In *a Special Issue of the Transactions of the SDPS: Journal of Integrated Design and Process Science*, volume 5 of *1*. Society for Design and Process Science, March 2001.

FWH96. B. Fields, P. Wright, and M. Harrison. Designing Human-System Interaction Using The Resource Model. In *Proceedings of APCHI'96: Asia Pacific Conference on Human-Computer Interaction*, Singapore, June 1996.

GHK98. F. Gadducci, R. Heckel, and M. Koch. Model checking graph-interpreted temporal formulas. In G. Engels and G. Rozenberg, editors, *Proc. 6th International Workshop on Theory and Applications of Graph Transformation (TAGT'98)*, number tr–ri–98–201 in Reihe Informatik, pages 292–299. Universität–Gesamthochschule Paderborn, 1998.

HCEL96. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.

KJ96. V. De Keyser and D. Javaux. *Human Factors in Aeronautics*, pages 28–45. Springer, Vienna, 1996. F. Bodart and J. Vanderdonckt (eds.).

Koc99. M. Koch. *Integration of Graph Transformation and Temporal Logic for the Specification of Distributed Systems*. PhD thesis, Technische Universität Berlin, FB 13, 1999.

KR95. M. Korff and L. Ribeiro. Formal relationship between graph grammars and Petri nets. In J. Cuny, H. Ehrig, G. Engels, and G. Rozenberg, editors, *Graph Grammars and their Applications to Computer Science*, pages 288 – 303. Springer, LNCS 1073, 1995. 5th International Workshop , Williamsburg, USA, November 1994, Selected Papers.

Kre81. H.-J. Kreowski. A comparison between Petri-nets and graph grammars. In *Lecture Notes in Computer Science 100*, pages 1–19. Springer, 1981.

Mon92. A. Monnier. Rapport préliminaire de la Commission d'enquête administrative sur l'accident du Mont Saint-Odile du 20 janvier 1992. Technical report, Ministère de l'Equipement, du Logement, des Transports et de l'Espace, Paris, France, 1992.

Mül98. J. Müller. Terminating and cyclic graph rewriting. In G. Engels and G. Rozenberg, editors, *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98)*, number tr–ri–98–201 in Reihe Informatik, pages 316–323. Universität–Gesamthochschule Paderborn, 1998.

PE01. J. Padberg and B. E. Enders. Rule Invariants in Graph Transformation Systems for Analyzing Safety-Critical Systems. Internal report, `http://www.informatik.uni-essen.de/enders/Papers/textframe.html`, 2001.

Rei85. W. Reisig. *Petri Nets*, volume 4 of *EATCS Monographs on Theoretical Computer Science*. Springer, 1985.

Suc97. B. E. Sucrow. Formal Specification of Human-Computer Interaction by Graph Grammars under Consideration of Information Resources. In *Proceedings of the 1997 Automated Software Engineering Conference (ASE'97)*, pages 28–35. IEEE Computer Society, November 1-5 1997.

WFH96. P. Wright, B. Fields, and M. Harrison. Distributed Information Resources: A New Approach to Interaction Modelling. In T. Green and J. Canas and C. Warran, editor, *EACE*, pages 5–10, 1996.

WFH99. P. Wright, B. Fields, and M. Harrison. Analysing Human-Computer Interaction as Distributed Cognition. Draft–Revised, August 1999.

Incremental Transformation of Lattices: A Key to Effective Knowledge Discovery*

John L. Pfaltz

Dept. of Computer Science, Univ. of Virginia
{jlp,cmt5n}@virginia.edu

Abstract. We show that a formal concept lattice $\mathcal{L}$, with explicit generators, constitutes a viable medium for discrete, deterministic, data mining. All implications of interest can be found and extracted from $\mathcal{L}$ independent of the frequency of their occurance.
More importantly, we show that these lattices can be grown from a binary relation R one row, or observation, at a time using a transformation that is based on the mathematical properties of the generators and faces of closed sets. Incremental growth is orders of magnitude faster than existing methods which rely on global closure operators.

1 Introduction

In [17], we first presented the ideas: (a) that concept lattices represent a viable way of representing deterministic logical implications based on a set of observations, and (b) that the concept lattices should evolve through a sequence of "local" lattice transformations. In that paper we could illustrate that such transformations were indeed local in nature, but we had no effective algorithm for performing it. Since then, we have developed the necessary mechanisms to make graph transformation (or in this case lattice transformation) an important new approach to knowledge discovery, which we call discrete, deterministic data mining, or DDDM. We believe that DDDM is closely analogous to classical scientific empiricism.

We begin by contrasting our concept of DDDM with more traditional data mining approaches. The basic goal of "data mining" has been the discovery of frequent, or common, attribute associations in very large data sets such as point-of-sale data [2,22,24]. These associations are statistical in nature; we have for example the old favorite "people who buy hot dogs often buy catsup". They may also be evanescent. A frequent association this month may not be frequent next month. The association may be indicative of a causal relationship, but most often it is not. Surely the purchase of hot dogs does not "cause" a purchase of catsup. Frequent associations, whether temporary or not, can be very important to our understanding of the world around us. But, we would argue it is not for the most part "scientific empiricism". Classical science seeks the discovery

* Research supported in part by DOE grant DE-FG05-95ER25254.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 351–362, 2002.
© Springer-Verlag Berlin Heidelberg 2002

and understanding of causal relationships among world phenomena that can be expressed with logical precision.

Like all data mining algorithms, the raw data of DDDM is a binary relation $R : O \rightarrow A$. We denote the domain (rows) of this relation as O, which we regard as "observations", or observations of "objects"[1]. By A, the codomain (columns) of R, we mean those attributes, or properties, that may be observed, *i.e.* that an object may exhibit.

Now, let O denote any universe of interest. Our paradigm of scientific knowledge is an assertion of the form $(\forall o \in O)[P(o) \rightarrow Q(o)]$ where $P(o)$ and $Q(o)$ denote predicate expressions over the bound variable o. Readily, there are other forms of scientific knowledge; but a focus on logical implication subsumes most causal assertions.

This concentration on deterministic causal implications appears to be new, even though other authors have suggested many of its elements. The lattice transformation paradigm [19] which treats its input as a possibly unbounded stream of observations appears to be completely new.

2 Closure, Concepts and Implication

DDDM is based on formal concept analysis [7] and closure theory [16]. These tools have been used before in data mining. For example, Zaki uses closure concepts in [24] to minimize the occurance of redundant associations; and both Godin and Missaoui [8] and Pasquier *et al.* [15] use concept lattices to optimize an *a priori* like search.

As developed in [7], formal concept analysis relies on visual display to reveal structure. But this becomes problematic with more than 30 concepts. One can no longer "see" the structure. Lindig and Snelting [10] discovered this when they applied formal concept analysis to code re-engineering. We make no use of visual display techniques, except for illustrative examples.

Where our approach differs from these earlier works is our emphasis on the "generators" of closed sets. Let φ denote a closure operator, so φ satisfies the 3 standard closure axioms for all X, Y,

$$X \subseteq X.\varphi,$$
$$X \subseteq Y \text{ implies } X.\varphi \subseteq Y.\varphi, \text{ and}$$
$$X.\varphi.\varphi = X.\varphi.$$

(We use a suffix notation, so read $X.\varphi$ as "X closure".) A set Z is closed if $Z.\varphi = Z$. Let Z be a closed set; a set $X \subseteq Z$ is said to be a generator if $X.\varphi = Z = Z.\varphi$. We are normally only interested in the minimal ($w.r.t$ inclusion) generators, which we denote by $Z.\gamma$. A closed set Z can have several minimal generators, so we subscript them if necessary, as in $Z.\gamma_1 \ldots Z.\gamma_n$, and let $Z.\Gamma$ denote the set of all minimal generators.

Formal concept analysis yields a lattice of pairs of closed sets, $k : (O_k, A_k)$ where $O_k \subseteq O, A_k \subseteq A$. Every closed set A_k has a set, $A_k.\Gamma = \{A_k.\gamma_1, \ldots, A_k.\gamma_i,$

[1] These rows are often regarded as "transactions" in the data mining literature.

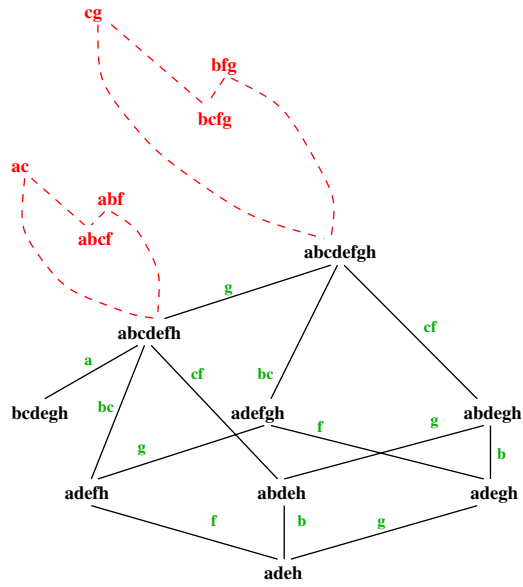


Fig. 1. A small portion of a concept lattice $\mathcal{L}$.

$\ldots, A_k.\gamma_n\}$ of minimal generators. That is, the closure φ of $A_k.\gamma_i$ (the i-th minimal generator of A_k) is $A_k.\gamma_i.\varphi = A_k$ for all i. Figure 1 shows a small portion of a concept lattice. (For this paper, we illustrate only the closed sets of attributes and their generators, but in practice we also retain O_k which constitutes the support of the concept.) The solid lines connect closed concepts. The dashed lines suggest generators. The set cg and bfg are minimal generators of the closed set $abcdefgh$. (Later, using Theorem 1 we will prove that these are its minimal generators. For now, we only observe that $cg.\varphi \subseteq abcdefgh$, but looking at the rest of the concept lattice consisting of closed subsets of $abcdefgh$, we see none containing cg.) All subsets Y such that $cg \subseteq Y \subseteq abcdefgh$, such as $bcfg$ have the same closure; $Y.\varphi = abcdefgh$. The minimal generators, $abcdefh.\Gamma = \{ac, abf\}$. We have illustrated the generating sets for only these two closed sets.

Unlike other research, we keep an explicit list of the minimal generators with each closed concept. Creation of a concept lattice *with generators* is quite a bit slower than, say the algorithm of Godin and Missaoui [8]. But in compensation, one can directly read off all possible implications. For each concept-k, we can assert that $(\forall o \in O)[A_k.\Gamma(o) \rightarrow A_k.\varphi]$. That is, because of the way the concept lattice closure operator is defined, the generators of a closed set logically imply all of the attributes in the closed set[2]. For example, from Figure 1 we immediately see that $(\forall o \in O)[[c(o) \wedge g(o)] \vee [b(o) \wedge f(o) \wedge g(o)] \rightarrow a(o) \wedge b(o) \wedge c(o) \wedge d(o) \wedge$

[2] This φ closure is a Galois closure. See [7] for its precise definition. Also observe that there is a strong similarity to the concept of attribute closure, X^+, with respect to functional dependencies in relational database theory [11].

$e(o) \wedge f(o) \wedge g(o) \wedge h(o)]$. Or more compactly we say, $(cg \vee bfg) \rightarrow abcdefgh$ and $(ac \vee abf) \rightarrow abcdefh$. And, as Zaki has observed [24], because the left side of these implications is minimal while the right side is maximal, we can expect several orders of magnitude fewer implications than obtained with frequent set data mining. In Section 4, we describe in some detail the results of applying our method to a $8,124 \times 54$ relational database describing mushroom characteristics. The closed set DDDM method yields 2,640 concepts (or implications) versus 15,552,210 associations derived by frequent set mining[3]. In a smaller 76×141 relational database listing the properties of marine sponges, frequent set mining yields 22,029,172 associations to 77,762 concepts with DDDM.

3 Incremental Growth

Since a concept lattice $\mathcal{L}$ captures all the logical attribute implications one can make about a collection of objects; it is natural to ask "suppose we observe one more object and its attributes. The set of attributes characterizing some new observation o' must either (a) be identical to an existing concept, in which case only the support of that concept need be incremented, or (b) the attribute set constitutes a new closed concept whose only support is itself. How will this latter case transform the lattice $\mathcal{L}$?" This is the essence of discrete, empirical induction. Our understanding of world phenomena normally occurs incrementally. We assimilate new observations, or rows of R if you choose, into an existing concept structure. Some new observations reinforce existing understanding; others lead to new understanding.

Godin and Missaoui [8] have shown one can easily insert a new concept into $\mathcal{L}$ using the intersection property of closed sets. Since the intersection of closed sets must be closed, if a new closed set, call it Y is introduced into the lattice so that it is covered by an existing set Z. Then at most we need only concern ourselves with the substructure below Z. This nature of closure systems ensures that the resulting changes will be, in some sense, "local". Figure 2, in which the closed set $Y = acdegh$ has been inserted, illustrates this principle. In addition to $acdegh$, a second closed set $acdeh$ also had to be added because the intersection of closed sets $abcdefh \cap acdegh$ must be closed. See the dot-dashed lines in Figure 2.

But, this ignores the real problem. How does such an addition change the generators of existing closed sets? Or equivalently, given a system of universally quantified implications, $(\forall o \in O)[P(o) \rightarrow Q(o)]$, how does a single existential assertion of the form $(\exists o' \in O)[P'(o')]$ necessarily change this logical system?

In Figure 1 (and Figure 2) we have labeled the edges with those elements which represent the difference between the closed sets. Each difference we call a face F of the larger closed set. For example, the faces of $abcdefgh$ are $\{g, bc, cf\}$. A close relationship between the faces and the generators of a closed set was proven, in [18], that is:

[3] In all comparisons, we set $\sigma = 3\%$ and $\gamma = 95\%$ in the *apriori* procedure.

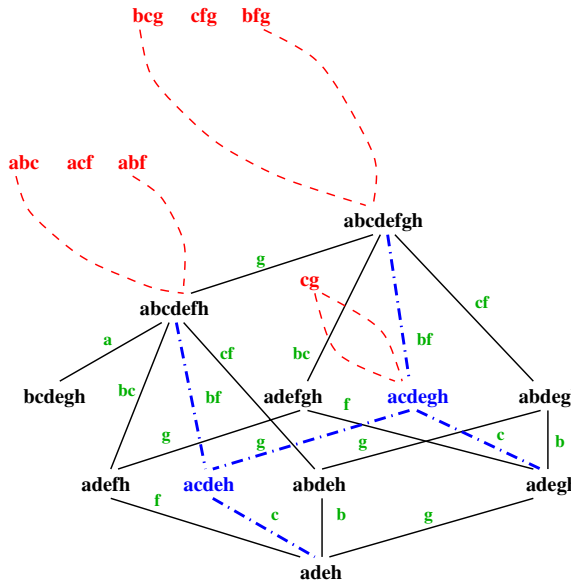


Fig. 2. The concept lattice of Figure 1 after *acdegh* has been incrementally added.

Theorem 1. *If Z is closed and $Z.\Gamma = \{Z.\gamma_i\}$ is its family of minimal generators then Z covers X in $\mathcal{L}$ iff $Z - X$ is a minimal blocker of $Z.\Gamma$.*

A set $F = Z - X$ blocks the family of generators $Z.\Gamma$ if for each $Z.\gamma_i \in Z.\Gamma$, $F \cap Z.\gamma_i \not= \emptyset$. Similarly each generator $Z.\gamma_i$ is a minimal blocker of the set of faces.

A new closed concept Y (*e.g. acdegh*) will be covered by a single closed concept Z [4]. Consequently, Y creates a new face F (*e.g.* bf) of Z. If an existing generator $Z.\gamma_i \subseteq Y$, then it can no longer be a generator of Z. If $Z.\gamma_i \cap Y = \emptyset$, it must be augmented by elements of F so that each becomes a blocker of F. In this case, $bfg \cap bf \not= \emptyset$, so bfg remains a generator with no changes. However, $cg \cap bf = \emptyset$, so $cg \cup b$ becomes a new generator of Z, as does $cg \cup f$. In the following pseudocode, a COLLECTION denotes a set of SETS.

```
void update_generators( CONCEPT cov_c, CONCEPT new_c )
     // "cov_c" will cover the new concept "new_c". update cov_c.gens to
     // reflect the face cov_c.atts - new_c.atts
     {
     ELEMENT     elem;
     SET         face, gsup;
     COLLECTION  new_GENS, keep_GENS, diff_GENS;
     BOOLEAN     minimal;
```

[4] Z covers Y if $Y \subset Z$, but there is no closed set Y' such that $Y \subset Y' \subset Z$. That there can be only one such Z is easily shown using the intersection property of closed sets.

```
face = cov_c.atts - new_c.atts;
keep_GENS = EMPTY_COL;
// these generators will be unchanged by the insertion
for each gen_set in cov_c.GAMMA
     if( not_EMPTY (gen_set INTERSECT face) )
          add_to_collection (gen_set, keep_GENS);
new_GENS = keep_GENS;
diff_GENS = cov_c.GAMMA - keep_GENS;
for each gen_set in diff_GENS
     {                    // gen_set does not intersect the new face
     for each elem in face
          {
          minimal = true;
          gsup = gen_set UNION {elem};
          for each keep_set in keep_GENS
               if( keep_set IS_SUBSET_OF gsup )
                    {             // gsup is not a minimal generator
                    minimal = false; break;
                    break;
                    }
          if( minimal )
               add_to_collection (gsup, new_GENS);
          }
     }          // replace cov_c.GAMMA with the new generating sets
cov_c.GAMMA = new_GENS;
}
```

Next the intersections of Y with each of its "siblings" (other concepts covered by Z) must be calculated to determine the faces, and generators, of Y itself. If one of these intersections, such as *acdeh*, yields a new closed concept not already in $\mathcal{L}$, it is recursively added in the same manner as Y.

```
insert_closed_set (SET Z, SET Y, LATTICE L)
       // Insert the closed set Y into L so that it is covered
       // by Z
       {
       SET    Y[];

       update_generators (Z, Y);
       for_each Y[i] covered by Z do
              {             // Y[i] will be a sibling of Y
              if ( not_EMPTY (Y INTERSECT Y[i]) )
                     {
                     update_generators (Y, Y INTERSECT Y[i]);
                     if (Y INTERSECT Y[i] in L)
                            continue;
```

```
                    X = new SET (Y INTERSECT Y[i]);
                    insert_closed_set (Y[i], X, L);
                    }
                }
            }
```

Entry of a new closed set Y into an existing concept lattice $\mathcal{L}$ can require a variable number of additional forced insertions, through recursive calls to INSERT_CLOSED_SET in the code above. As expected, the local nature of each insertion limits the extent of this recursion[5]. In Figure 3, we plot the *average* relative increase per insertion (dashed line) represented by a sequence of up-

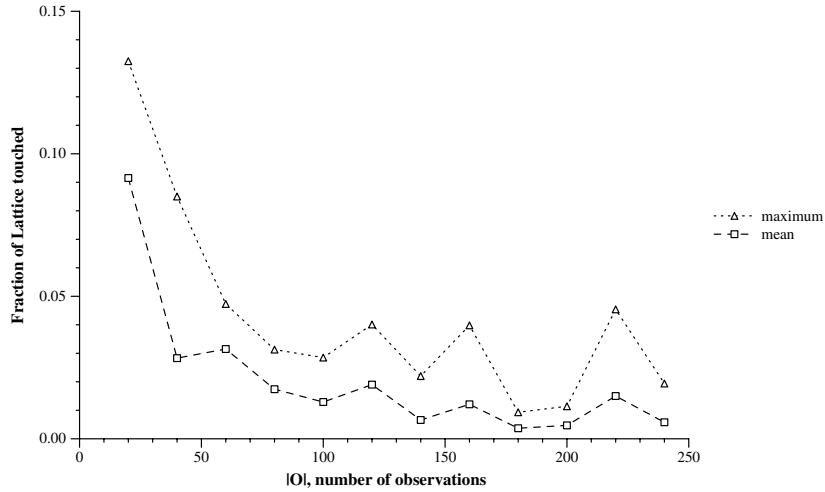


Fig. 3. Relative increase in $\mathcal{L}$ per insertion.

dates to a random 20 attribute relation together with the *maximum* relative increase (dotted line). This graph, although generated synthetically, appears to be quite representative. Initial entries into the lattice affect nearly the entire lattice, $\mathcal{L}$. But gradually the structure stabilizes, and successive entries tend to be much more local. A process which normally affects approximately 2% of the total lattice structure can reasonably be called "local".

We have discovered that, compared to a batch process which analyzes the entire relation R, as required by *a priori* [1], incremental update decreases processing time upto 3 orders of magnitude. Incremental lattice transformation makes concept lattices with generators a practical knowledge discovery method.

[5] It was this locality that Godin and Missaoui [8] exploited. But, although their paper gives performance results in terms of times, it does not detail an actual counts of their recursive step.

4 Representative Output

Interesting data sets are large. One cannot visualize them; we can only select representative concepts from the lattice and display them[6]. To get a feeling for DDDM we consider the well-known MUSHROOM data set, which we obtained from the UCI Machine Learning Repository at

`http://www1.ics.uci.edu/ mlearn/MLRepository.html`.

Many data mining experiments, using this data set, have been reported in the literature. Most have been concerned with the edibility of various mushrooms. The data set R consists of 8124 observations of 22 nominal attributes. Attribute-0 has values "edible" and "poisonous", denoted `e0` and `p0` respectively. For illustrative brevity, we ran the process using just the first 9 attributes listed below:

Attr-0 **edibility**: e=edible, p=poisonous;
attr-1 **cap shape**: b=bell, c=conical, f=flat, k=knobbed, s=sunken, x=convex;
attr-2 **cap surface**: f=fibrous, g=grooved, s=smooth, y=scaly;
attr-3 **cap color**: b=buff, c=cinnamon, e=red, g=gray, n=brown, p=pink, r=green, u=purple, w=white, y=yellow;
attr-4 **bruises?**: t=bruises, f=doesn't bruise;
attr-5 **odor**: a=almond, c=creosote, f=foul, l=anise, m=musty, n=none, p=pungent, s=spicy, y=fishy;
attr-6 **gill attachment**: a=attached, d=descending, f=free, n=notched;
attr-7 **gill spacing**: c=close, d=distant, w=crowded;
attr-8 **gill size**: b=broad, n=narrow.

Because of multiple attribute values, these correspond to a binary array of 54 boolean attributes. The concept lattice generated by this $8,124 \times 54$ binary relation, R, consists of 2,640 concepts.

Implications with a single antecedent are often the most important. They are the easiest to apply in practice. Scanning the concept lattice for singleton generators yields the 22 implications of Figure 4. We see that 12 of the implications have to do with edibility, `e0` or `p0`.

We list the support for each rule to the right. The reader can decide whether the σ-frequency they use would have discovered the implication. For example, to discover that mushrooms with "sunken" caps are edible (#313) would require $\sigma < 0.004$. Such a low σ-value would suggest that the number of frequent sets would approach $2^{54-\epsilon}$, or possibly as many as $2^{50} = 1.125 \times 10^{15}$, a number that can exhaust main memory.

Virtually any data mining process would discover that "odor" is a crucial determinant. In particular, a "creosote"(#668), "foul"(#924), "musty"(#2022), "spicy"(#1597) or "fishy"(#1687) odor betokens "poisonous". Since "almond" (#117) and "anise"(#144) indicate "edible", only "no odor" is ambiguous. Such a mushroom can be "edible"(#313, #1081, #1553) or "poisonous"(#1401,

[6] It is interesting to "surf" the lattice following links to higher and lower conclepts; but demonstration of this process in a paper is impractical.

CONCEPT	IMPLICATION			SUPPORT
60	w3	->	f6	1040
105	t4	->	f6	3376
109	n8	->	f6	2512
117	a5	->	e0, t4, f6	400
144	l5	->	e0, t4, f6	400
313	s1	->	e0, f2, f4, n5, f6, c7, n8	32
556	f2	->	f6	2320
604	w7	->	f6	1312
668	c5	->	p0, x1, f4, f6, n8	192
720	y3	->	f6	1072
898	b3	->	t4, f6, c7, b8	168
924	f5	->	p0, f6, c7	2160
1007	p3	->	f6	144
1081	u3	->	e0, y2, f4, n5, f6, c7, n8	16
1401	g2	->	p0, w3, t4, n5, f6, w7, n8	4
1553	r3	->	e0, y2, f4, n5, f6, c7, n8	16
1597	s5	->	p0, f4, f6, c7, n8	576
1687	y5	->	p0, f4, f6, c7, n8	576
2019	a6	->	f4, c7, b8	210
2022	m5	->	p0, y2, f4, c7, b8	36
2162	e3	->	c7	1500
2562	c1	->	p0, n5, f6, w7, n8	4

Fig. 4. Implications in the MUSHROOM data set that have singleton generators (or antecedents).

#2562). There are only 4 conically capped instances and only 4 with grooved cap surfaces; but, although not frequent, eating any might be unpleasant.

Because "poisonous" is thought to be an important characteristic of mushrooms, we scanned the concept lattice for those concepts which (a) had `p0` in the closed (consequent) set, and (b) had a two element generator *not* containing `p0`. There are 64 such implications. We then passed this list through a filter, eliminating those whose generators included a poisonous odor, *viz.* `c5, f5, m5, s5` or `y5`. The resulting 15 implications are shown in Figure 5.

We observe that 7 of these instances could also be determined by odor, either `c5` or `m5`. But, 7 have "no odor" (`n5`) and would thereby be ambiguous in any case. In none of these extractions has the support played a role. DDDM implications are found independently of their frequency — which might be desirable if one is considering tasting one of the 876 instances of "red" mushrooms that "don't bruise" easily. Figure 6 illustrates the same kinds of logical criteria for edibility. Figures 5 and 6 both illustrate implications used to "classify" data, into either "edible" or "poisonous". It is worth comparing the DDDM approach with the nice approach found in [13].

Because DDDM yields implications that are universally quantified over the data set, one can perform logical transformations. One can reason. But, we must consider to data errors. Since it is not statistical, DDDM is not forgiving of erroneous input. If a new observation o' would change the generators of a concept above a specified threshold, our system can flag the observation and defer the insertion. The observation is then carefully examined for validity, and either discarded or reentered.

This approach to error control has an unexpected benefit. Suppose for example in a study of animal species that $a_1 \equiv$ "nurses its young" and $a_2 \equiv$ "gives live

CONCEPT	IMPLICATION			SUPPORT
666	p3, w7	->	p0, x1, f4, c5, f6	32
667	p3, f4	->	p0, x1, c5, f6, n8	64
667	p3, n8	->	p0, x1, f4, c5, f6	64
696	f2, p3	->	p0, x1, f4, c5, f6, n8	32
1184	b1, n8	->	p0, n5, f6	12
1495	b1, b3	->	p0, t4, n5, f6, c7, b8	12
1567	b1, p3	->	p0, t4, n5, f6, c7, b8	12
2081	y3, n5	->	p0, f4, f6, n8	24
2177	e3, f4	->	p0, c7	876
2181	y2, a6	->	p0, f4, m5, c7, b8	18
2372	c3, a6	->	p0, y2, f4, m5, c7, b8	6
2470	e3, a6	->	p0, y2, f4, m5, c7, b8	6
2561	c1, y3	->	p0, y2, f4, n5, f6, w7, n8	2
2561	c1, f4	->	p0, y2, y3, n5, f6, w7, n8	2
2563	c1, y2	->	p0, n5, f6, w7, n8	3

Fig. 5. Implications of "poisonous" in the MUSHROOM data set with two generating attributes (or antecedents).

CONCEPT	IMPLICATION			SUPPORT
61	w7, b8	->	e0, f4, n5, f6	1056
119	y3, t4	->	e0, f6	400
131	s2, y3	->	e0, t4, f6	152
452	f2, t4	->	e0, f6	912
586	f2, e3	->	e0, t4, n5, f6, c7	288
1015	x1, r3	->	e0, y2, f4, n5, f6, c7, n8	8
1261	k1, b3	->	e0, t4, n5, f6, c7, b8	16
1347	f2, c3	->	e0, f4, n5, f6, w7, n8	12
1944	b1, g3	->	e0, f4, n5, f6, w7, b8	48
1966	k1, g3	->	e0, f4, m5, f6, w7, b8	48
2342	s2, c3	->	e0, t4, m5, f6, c7, b8	4
2343	c3, t4	->	e0, n5, f6, c7, b8	8

Fig. 6. Implications of "edible" in the MUSHROOM data set with two generating attributes (or antecedents).

birth". The implication $a_1 \rightarrow a_2$ is supported by thousands of observations — until we encounter a "duck-billed platypus". If convinced of its validity, we still flag the occurance as being "unusual", and hence of possible importance. Surely the duck-billed platypus is a relative unimportant mammal, *except* for this deviant behavior. Because DDDM works with deterministic, logical assertions, we can uncover this kind of "outlying" occurance.

A great deal of the phenomena around us is non-deterministic and statistical in nature. As a tool, DDDM is inappropriate to analyze this kind of data. But, when we are looking for deterministic, causal relationships, incremental DDDM becomes an effective knowledge discovery process. Moreover, it mimics to a large extent the procedures that classical science has always used to understand its world.

5 Conclusions

In the past graphs have been primarily used as a static display of the relationships between objects. Only fairly recently has the transformation of graphs really been regarded as an area of interest. There have been many formal studies of relatively abstract graph transformation, *e.g.* [3,9,14]. The transformation of program structure was an early target of the field [20,6] which is naturally extended to type structures [5]. Having working systems such as [4,21] has been a boon. Since UML diagrams are already in graphical form, their transformation becomes interesting [23].

Data mining has not been a traditional target of graph transformation (in this case lattice transformation); but it ought to be. This paper and [19] have demonstrated that an iterative, transformational approach to data mining can be far more effective that frequent set *apriori* mining given the right circumstances. One can alternatively regard this as empirical learning where a knowledge structure is transformed by new observations [17]. The empirical accquisition of knowledge [12] and concept formation are ripe for further exploration.

References

1. Jean-Mark Adamo. *Data Mining for Association Rules and Sequential Patterns.* Springer Verlag, New York, 2000.
2. Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining Association Rules between Sets of Items in Large Databases . In *Proc. 1993 ACM SIGMOD Conf.*, pages 207–216, Washington, DC, May 1993.
3. Stefan Arnborg, Bruno Courcelle, Andrzej Proskurowski, and Detlef Seese. An Algebraic Theory of Graph Reduction. In Hartmut Ehrig, Hans-Jorg Kreowsik, and Gregor Rozenberg, editors, *Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, #532, pages 70–83, Bremen, Germany, 1990. Springer-Verlag.
4. R. Bardohl, T. Schultzke, and G. Taentzer. Visual Language Parsing in GenGEd. In *2nd International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 291–296, Crete, Greece, July 2001. Satellite Workshop of ICALP 2001.
5. H. Ehrig and F. Orejas. Integration Paradigm for Data Type and Process Specification Techniques. In G. Paun, G. Rozengerg, and A. Salomaa, editors, *Current Trends in Theoretical Computer Science: Entering the 21st Century*, pages 192–201. World Scientific, Singapore, 2001.
6. Gregor Engels, Claus Lewerentz, and Wilhelm Schafer. Graph Grammar Engineering: A Software Specification Method. In Hartmut Ehrig, Manfred Nagl, Gregor Rozenberg, and Azriel Rosenfeld, editors, *Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, #291, pages 186–201, Warrenton, VA, 1986. Springer-Verlag.
7. Bernard Ganter and Rudolf Wille. *Formal Concept Analysis - Mathematical Foundations.* Springer Verlag, Heidelberg, 1999.
8. Robert Godin and Rokia Missaoui. An incremental concept formation approach for learning from databases. In *Theoretical Comp. Sci.*, volume 133, pages 387–419, 1994.

9. Reiko Heckel, Hartmut Ehrig, Gregor Engels, and Gabriele Taentzer. Classification and Comparison of Modularity Concepts for Graph Transformation Systems, 1999.
10. Christian Lindig and Gregor Snelting. Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis. In *Proc of 1997 International Conf. on Software Engineering*, pages 349–359, Boston, MA, May 1997.
11. David Maier. *The Theory of Relational Databases*. Computer Science Press, Rockville, MD, 1983.
12. Brian Mayoh. Graph Grammars for Knowledge Representation. In Hartmut Ehrig, Hans-Jorg Kreowsik, and Gregor Rozenberg, editors, *Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, #532, page 53, Bremen, Germany, 1990. Springer-Verlag.
13. Carlos Ordonez, Edward Omiecinski, Levien de Braal, Cesar A. Santana, Norberto Ezquerra, Jose A. Taboada, David Cooke, Elizabeth Krawczynska, and Ernst V. Garcia. Mining Constrained Association Rules to Predict Heart Disease. In *IEEE International Conf. on Data Mining, ICDM*, pages 433–440, 2001.
14. J. Padberg, H. Ehrig, and L. Riberio. Algebraic High-Level Net Transformation Systems. *Math. Structures in Computer Science*, 5:217–256, 1995.
15. Nicolas Pasquier, Yves Bastide, Rafik Taouil, and Lofti Lakhal. Discovering Frequent Closed Itemsets for Association Rules. In *Proc. 7th International Conf. on Database Theory (ICDT)*, pages 398–416, Jan. 1999.
16. John L. Pfaltz. Closure Lattices. *Discrete Mathematics*, 154:217–236, 1996.
17. John L. Pfaltz. Transformations of Concept Graphs: An Approach to Empirical Induction . In *2nd International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 320–326, Crete, Greece, July 2001. Satellite Workshop of ICALP 2001.
18. John L. Pfaltz and Robert E. Jamison. Closure Systems and their Structure. *Information Sciences*, 139:275–286, 2001.
19. John L. Pfaltz and Christopher M. Taylor. Concept Lattices as a Scientific Knowledge Discovery Technique. In *2nd SIAM International Conference on Data Mining*, pages 65–74, Arlington, VA, Apr. 2002.
20. Terrence W. Pratt. Pair grammars, graph languages, string to graph translations. *JCSS*, 5(6):760–795, Dec. 1971.
21. Andy Schurr. PROGRESS, A VHL-Language Based on Graph Grammars. In Hartmut Ehrig, Hans-Jorg Kreowsik, and Gregor Rozenberg, editors, *Graph Grammars and their Application to Computer Science*, Lecture Notes in Computer Science, #532, pages 641–659, Bremen, Germany, 1990. Springer-Verlag.
22. Ramakrishnan Srikant and Rakesh Agrawal. Mining Quantitative Association Rules in Large Relational Tables . In *Proc. of 1996 SIGMOD Conference on Management of Data*, pages 1–12, Montreal, Quebec, June 1996.
23. Aliki Tsiolakis. Integrating Model Information in UML Sequence Diagrams. In *2nd International Workshop on Graph Transformation and Visual Modeling Techniques*, pages 268–276, Crete, Greece, July 2001. Satellite Workshop of ICALP 2001.
24. Mohammed J. Zaki. Generating Non-Redundant Association Rules. In *6th ACM SIGKDD Intern'l Conf. on Knowledge Discovery and Data Mining*, pages 34–43, Boston, MA, Aug. 2000.

GraCAD – Graph-Based Tool for Conceptual Design

Janusz Szuba[1], Andy Schürr[2], and Adam Borkowski[1]

[1] Institute of Fundamental Technological Research, Polish Academy of Sciences
ul. Swietokrzyska 21, 00-049 Warsaw, Poland
{jszuba,abork}@ippt.gov.pl
[2] Institute for Software Technology, University of the German Armed Forces Munich
D-85577 Neubiberg, Germany
Andy.Schuerr@unibw-muenchen.de

Abstract. This paper proposes a new methodology for computer-aided design of layouts of buildings. The presented methodology addresses the very early phase of designing that is called Conceptual Design. In this methodology the architect does not pay attention to layout details, but operates on functional requirements, constraints, and high-level architectural concepts such as functional areas and rooms. The GraCAD system realizes this methodology by adding a high-level design tool to the well-known commercial program ArchiCAD. This design tool has been realized using graph-based knowledge representation techniques and graph grammars as offered by the system PROGRES. Preliminary results show that the developed ArchiCAD add-on and the formalism of graph grammars seems to be useful in aiding architectural design.

1 Introduction

The main aim of this project is to create a system based on graph transformations which supports architects in the *early phase of architectural design*. Contrary to conventional expert systems proposed previously in this domain (compare, e.g.[8]) our system can be seen as a knowledge-based add-on for a standard CAD-tool. We restrict our considerations to a rather simple example of designing a single-store family house, but the methodology remains valid for designing other types of buildings. The system, called GraCAD, helps the architect to create a prototype design that fulfils a given set of high-level requirements. GraCAD is integrated with the commercial program for architects ArchiCAD [1]. The graph-based knowledge representation of family house requirements and corresponding layouts form the theoretical background of our system. The graph rewrite tool PROGRES developed at the RWTH Aachen [15] is used to manipulate these graph representations and to translate high-level requirements graphs into low-level graphs reflecting the internal data structures of ArchiCAD.

Pioneered by N. Chomsky [6], the linguistic approach to world modelling found applications in many areas. The core idea in this methodology is to treat certain primitives as letters of an alphabet and to interpret more complex objects

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 363–377, 2002.
© Springer-Verlag Berlin Heidelberg 2002

and assemblies as words or sentences of a language based upon the alphabet. Rules governing generation of words and sentences define a grammar of the regarded language. In terms of the world, modelling such a grammar generates a class of objects that are considered plausible. Thus, grammars provide a very natural knowledge representation formalism for computer-based tools that should aid the design.

Since G. Stiny [16] has developed shape grammars many researchers showed how such grammars allow the architect to capture essential features of a certain style of buildings. However, the primitives of shape grammars are purely geometrical, which restricts their descriptive power. Substantial progress was achieved after graph grammars were introduced and developed (cf. e.g. [14]). Graphs are capable to encode much more information than linear strings or shapes. Hence, their applicability for CAD-systems was immediately appreciated [9].
A special form of graph-based representation has been developed by E. Grabska [10]. In 1996-98 Grabska's model served as the basic knowledge representation scheme in the research reported by the third author at the conferences in Stanford [11], Ascona [3], and Wierzba [4]. It turned out that by introducing an additional kind of *functionality graphs* into Grabska's model one can conveniently reason about conceptual solutions for the designed object. The functionality analysis of houses as the starting point of the conceptual design has been proposed by several researchers (compare, e.g. [5], [7]). Such a methodology allows the designer to distract himself from details and to consider the functionality of the designed object, the constraints and the requirements to be met, and the possible ways of selecting optimum alternatives more clearly.

The results described in Sections 2 to 4 can be viewed as the further development of the research reported previously in [17], [18], and [19]. The novelty is twofold. First, an intermediate functionality graph has been inserted between the initial specification of the project and the decomposition graph (former CP-graph). Second, the PROGRES transactions allowed us to implement rule checkers that were missing in the previous versions of software.

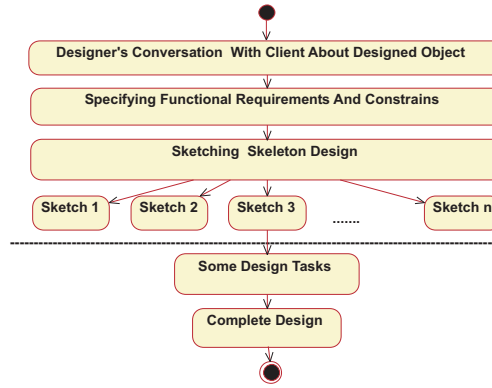
2 Early Phase of Designing - Conceptual Design

This section concerns the early phase of designing called conceptual design. This is the phase when the architect, knowing functional requirements and constraints from the conversation with the client, sketches the layout of the building (compare the part of Fig. 1a above the horizontal line and Fig. 1b). These sketches, prototype designs, are presented to the client and discussed with him. Finally, one of the created prototype designs becomes the basis for the complete design (compare, e.g. [12] or [13]).

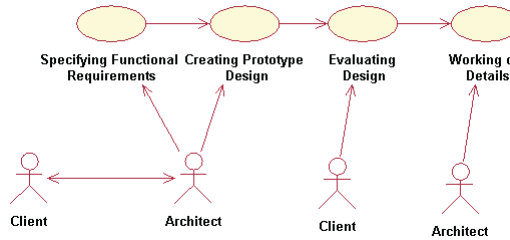
Rapid prototyping is important in many areas and architecture is no exception. It enables the designer to present the draft of the design to the client in short time and to achieve approval or disapproval of the presented proposition.

The scheme given in Fig. 1 is deliberately simplified. The real design process includes repetitive loops, abandoning partial solutions and backtracking.

The aim of this scheme is merely to emphasize the role of sketching conceptual alternatives at the initial stage.



a) Activity Diagram



b) Use Case Diagram

Fig. 1. The design process

3 GraCAD - Aiding Conceptual Design

GraCAD aids the phase of designing described in the previous section. Our system helps architects rapidly creating prototype designs for given functional requirements and constraints. Using GraCAD, the architect works on the level of functional areas and rooms. He does not operate on the level of detailed geometrical descriptions of the regarded object, a house: exact coordinates and dimensions would distract him from conceptual thinking. The architect avoids details not important in the conceptual phase of designing.

GraCAD is equipped with a set of functions aiding in rapid prototyping:

- checking if specified functional requirements are coherent, sensible
- checking if the created design fulfils specified functional requirements
- checking if basic architectural norms are not violated in the design
- automatically inserting windows and doors
- adjusting the rooms to the chosen contour of the house
- generating standard solution layouts

In the following subsections of this chapter details of using the system are presented.

3.1 Basic Parameters

The first step is to *specify the basic functional requirements* for a designed object. As mentioned before, we restrict our consideration to designing a single-store family house. The functional requirements for such a house are straightforward:

- number of inhabitants
- total area of the house
- functional areas of the house

The user specifies the basic parameters by filling the form shown in Fig. 2a.

3.2 Functionality Graph

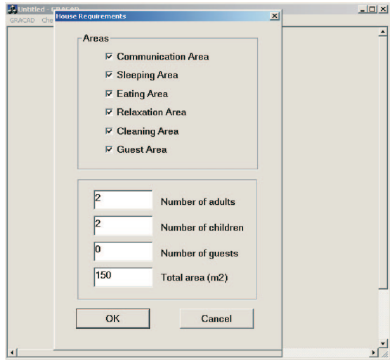
The next step is to *specify the functionality of the house*. This is done by drawing a functionality graph (Fig. 2b). The nodes of this graph represent areas that are supposed to carry out specific functions. The edges describe relations between such areas. For example, an edge *AreaAccess* (all edges shown in Fig. 2b are *AreaAccess* type) between two areas indicates that one area is accessible directly from the other. In this way the user specifies the accessibility of the areas.

3.3 Decomposition Graph

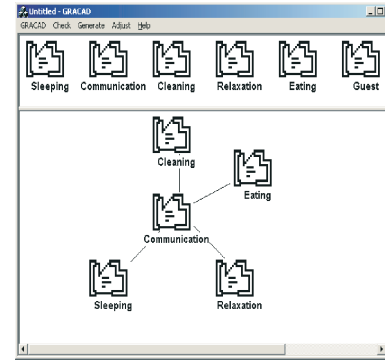
The next step is to *map functional requirements* to rooms. Fig. 2c shows GraCAD in the mode of drawing the decomposition graph of the building. Each node of this graph represents a room of the designed house, an edge *RoomAccess* (all edges shown in Fig. 2c are *RoomAccess* type) between two rooms indicates that one room is accessible directly from the other. Above each node a list of the room functions is displayed. Thus, creating the decomposition graph, the user specifies simultaneously the accessibility and the function of each room.

3.4 Room Editor

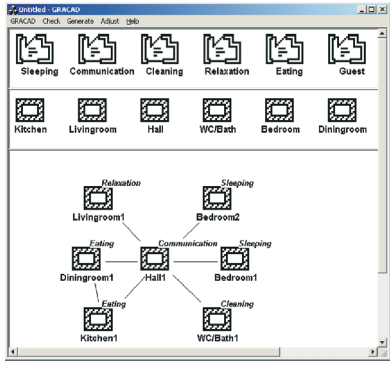
After specifying the functional requirements (steps 1-3) the architect starts to work on the prototype design that matches defined functional requirements. Fig. 2d shows the module *Room Editor* that is used to create a *prototype floor layout*.



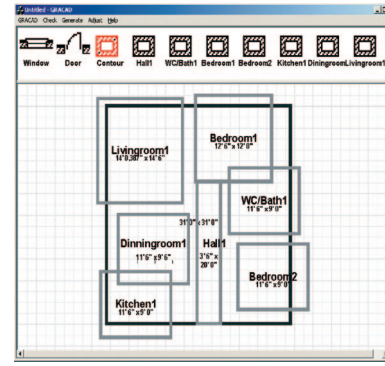
a) Input form for basic parameters



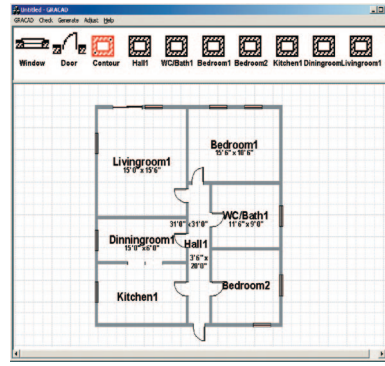
b) Functionality graph



c) Decomposition graph



d) Rooms before adjustment



e) Rooms after adjustment

Fig. 2. Aiding conceptual design with GraCAD

The top window displays the elements that should be assembled like windows, doors, house contour and the rooms from the decomposition graph. The bottom window shows the top view of the floor layout.

How the prototype design is created? First, the architect chooses the house contour element from the top window, drags it and drops in the bottom window. The square contour of the house (dark walls) appears in the bottom window. The total area of this contour is specified by the basic parameters entered in the first step of the design process. Then the architect can modify the shape of the contour. In the present version of GraCAD only rectangular rooms and contours are allowed.

In the next step the rooms are created. The user chooses the room from the top window, drags it and drops in the bottom one. The place in the bottom window, which the user pointed selected, becomes the center of the square that appears. That square/room is associated with the node/room chosen from decomposition graph and has the minimal surface suggested in architectural norms (compare Fig. 3).This way the user positions approximately all the rooms of the building. After that the architect can either adjust the rooms manually or call a *Room Adjuster* described in the next section.

3.5 Room Adjuster

The *Room Adjuster* module is able to *move and resize rooms* automatically. As shown in Figs. 2d and 2e, the rooms retain their relative positions but fill the entire contour of the building and obey constraints on their area and on the ratio between their width and length.

Moreover, the *Room Adjuster* may insert automatically the doors and windows. It accomplishes these tasks because the module knows functional requirements of the designed object as well as architectural norms and rules.

After using the *Room Adjuster* the architect usually returns to the manual design mode.

3.6 Rule Checkers

Each design created by the architect may be checked upon request whether it fulfils a given set of rules (constraints). In the present version of GraCAD three types of checkers are available:

1. checking if requirements imposed by the basic parameters are coherent
2. checking if the floor layout matches specified functional requirements
3. checking if architectural norms are not violated

The checker of the first type checks the following rules:

- if there is an edge *AreaAccess* between the areas a_1 and a_2 in the functionality graph, then either a room r_1 belonging to the area a_1 and a room r_2 belonging to the area a_2 connected by a *RoomAccess* edge should exist or a room r belonging simultaneously to both areas a_1 and a_2 should exist;

- if a it guest area is present in the basic requirements, then the room intended for guests should appear in the decomposition graph (analogous rules apply for other functional areas);
- whether the decomposition graph contains sufficient number of sleeping rooms for the given number of house inhabitants;
- whether the bathroom and the WC are separated if the house is intended for more than five inhabitants;
- if the sum of minimal areas for all rooms does not exceed the total living area specified in the basic parameters.

The checker of the second type takes into account:

- whether there exists a room corresponding to each node of the decomposition graph;
- whether nodes connected by the *RoomAccess* edge in the decomposition graph correspond in the floor layout to rooms that are adjacent and accessible via properly placed doors;
- whether the area inside the contour coincides with the total area specified in the basic parameters.

No	Name of the room	Minimal surface of the room (in m^2)	
		4-5 users	6-7 users
1	Living-room with eating place	22-24	24-26
2	Living-room without eating place	18	20
3	Dining-room	8	9
4	Kitchen without eating place	8	8
5	Kitchen with eating place	13-14	15-16
6	Pantry (near the kitchen)	2	2
7	Bedroom for parents	13-15	13-15
8	Bedroom (2 people)	11-13	11-13
9	Bedroom (1 person)	8-9	8-9
10	Bathroom/WC	4	4
11	WC	1,5	1,5
12	Laundry	6	6
13	Drying-shed	10	10
14	Garage	18	18

Element of the house	The best location in the house
Living-room	S, SW
Kitchen	NW, NE
Dinning-room	W
WC	N
Bathroom	N
Balcony, terrace	S

Fig. 3. Architectural norms a) minimal surface for the room of given type b) the best location of the room in the house with regard to world directions

Finally, the third type checker takes care of the following architectural norms (Fig. 3):

- whether windows provide rooms with enough light;
- whether the ratio width to length of each room is appropriate;
- whether each room has adequate spacing;
- whether the location of each room with regard to the world directions is appropriate.

3.7 Generators for Standard Solutions

The next feature of GraCAD aiding conceptual design is the mechanism of generators that play the role of providers of standard solutions.

After basic parameters of the house were specified, the user may request templates of functionality graphs and decomposition graphs. Having chosen some of them, the architect asks the system to generate the corresponding floor layouts. These layouts are then evaluated and possibly adjusted manually for the needs of the current design goal.

The option of using generators is complementary to adjusting and checking solutions developed from scratch manually.

3.8 Transforming Rooms into Walls

After the prototype design accepted by the client is finished, the architect starts working on detailed design. In contrast to the conceptual phase, where the architect works with rooms and functional areas, in the detailed design phase he operates on the walls. The ArchiCAD environment is more suitable for the detailed design than GraCAD. Therefore, a special module called *Rooms To Walls Transformer* has been developed. This module transforms the GraCAD project into the format accepted by ArchiCAD.

As the result of this transformation

- some walls from the GraCAD model (every room contains four walls in GraCAD representation) are not present in ArchiCAD model (like the wall between WC and Bath in Fig. 4)
- some walls from the GraCAD model are merged in one in the ArchiCAD model (compare contour walls in Fig. 4).

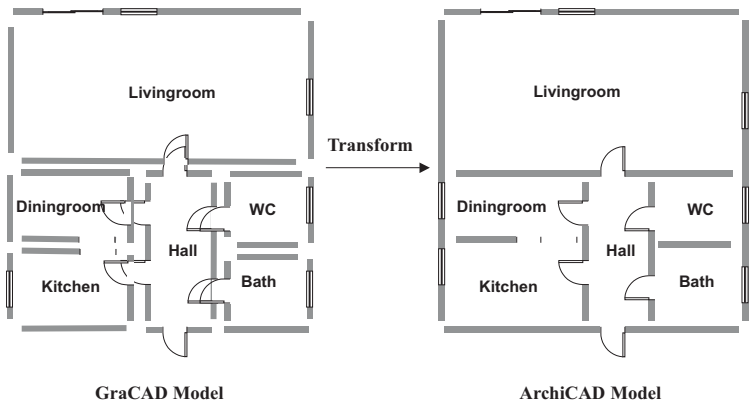


Fig. 4. Transformation from GraCAD model to ArchiCAD model

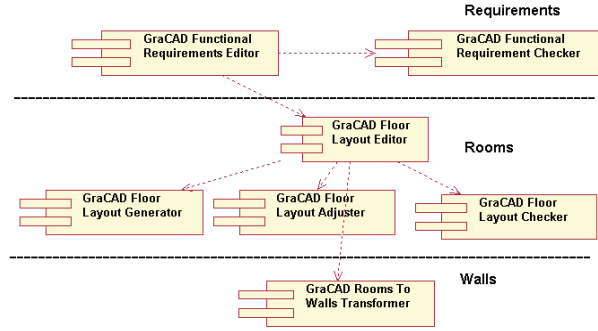


Fig. 5. Components of GraCAD System

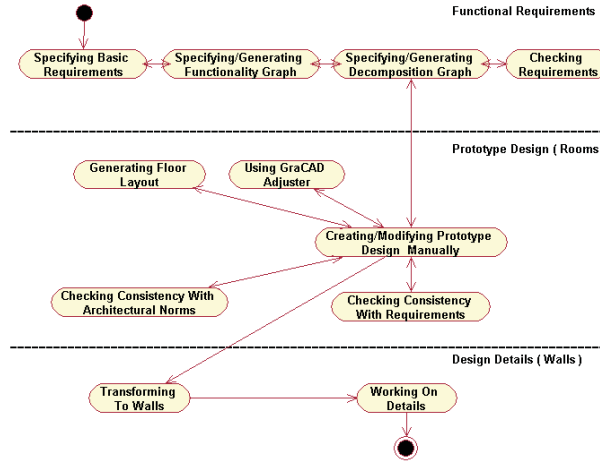


Fig. 6. Activities of the architect while using the GraCAD system

3.9 Integrated Environment

The MS-Windows version of the popular commercial system ArchiCAD serves as the basis for our prototype. GraCAD is implemented using the Microsoft Visual C++ 6.0 as an add-on of ArchiCAD according to the requirements specified in the ArchiCAD API Development Kit 2.2.

The graph-oriented part of our system uses the Linux version of the graph rewrite tool PROGRES created at the RWTH Aachen. The add-on communicates with PROGRES using a CORBA protocol (version Mico Corba 2.3.3).

The UML diagrams shown in Figs. 5 and 6 supplement the description of GraCAD presented in this chapter. The first diagram shows the component structure of the system, the second one depicts the architect's activities when designing with GraCAD.

Attributed graphs and graph grammars are chosen as the main knowledge representation techniques in GraCAD. In our representation nodes represent concepts that are used by the architect during the conversation with the investor, specifying functional requirements and creating the prototype design. Examples of such nodes are a house, a functional area, a wall, a window, etc. The nodes have attributes that describe specific properties. In the case of a house, these attributes would be the total area of the house, the number of inhabitants (adults, children, guests) and the functions of the house. In the case of a wall, we have attributes describing the location of the wall in the system of co-ordinates, the height, the width, and the texture of the wall. The edges represent relations between objects (for instance, relations of accessibility, adjacency or containment).

In each system aiding design it is necessary to represent certain rules. In our system we have the following types of rules:

- checking whether the created design fulfils functional requirements
- checking whether architectural norms are not violated
- automatically inserting doors, windows or adjusting the prototype design

We represent these rules using the formalism of graph grammars or, more precisely, the formalism of PROgrammed Graph Rewrite Systems. As mentioned before, we use the graph-rewrite system PROGRES for this purpose. It allows us to represent all necessary rules in terms of graph transformations. In the next section, more details about our graph-based knowledge representation approach can be found.

4 House Specification in PROGRES

The graph rewrite system PROGRES serves in our prototype the following purposes:

- It provides access to the domain knowledge stored in the form of a graph grammar. At present this grammar encompasses single-family houses, but the knowledge base can be easily extended to cover other domains.
- It provides means of constructing graphs that describe the currently designed object.
- It allows the user to trigger rules that check whether the current design fulfills architectural norms and other constraints.

4.1 Class Diagram

The knowledge about a specific domain is represented in our system in an object-oriented manner. Fig. 7 shows the collection of classes used in the description of single-family houses.

The most important class in the specification is the concrete class *House* (concrete classes with instances are called "node types" in PROGRES and depicted as rectangles with round edges) that represents the entire building. The

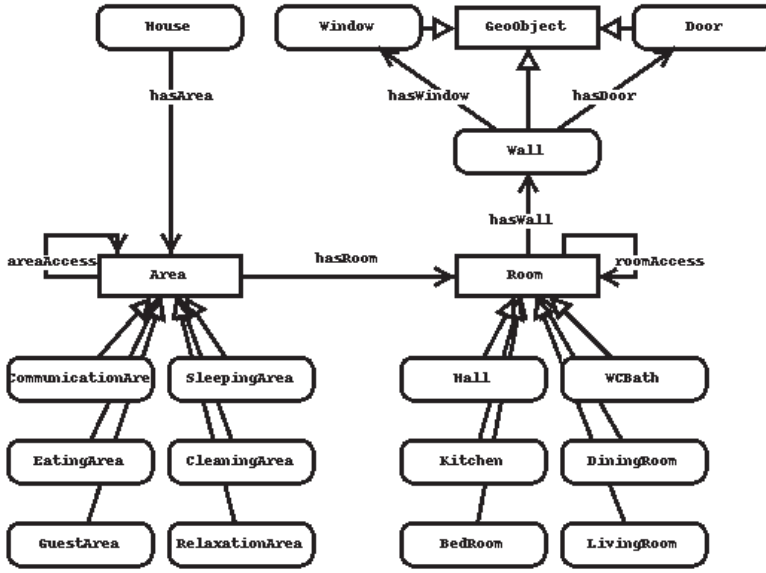


Fig. 7. Class diagram for single-family house

basic parameters concerning the designed house (see section 3.1) are stored in the attributes of this class. The abstract class *Area* (depicted as a regular rectangle) is the base class for the subclasses *SleepingArea, CommunicationArea, RelaxationArea, EatingArea, GuestArea , CleaningArea*. These subclasses represent the main functions of the house.

The abstract class *Room* is the base class for subclasses *BedRoom, WCBath, Kitchen, DinningRoom, Hall, LivingRoom* that represent a physical decomposition of the building. *Wall, Window* and *Door* are subclasses of *GeoObject* corresponding to the class of geometrical objects available in ArchiCAD. The abstract class *Object* not shown in the diagram is the base class for *House, Area, Room* and *GeoObject*.

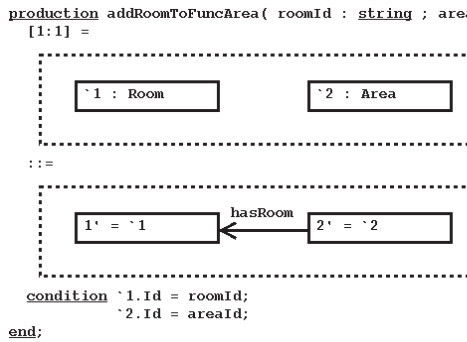
The following relations are defined between classes:

- *hasArea* between *House* and *Area*;
- *hasRoom* between *Area* and *Room*;
- *hasWall* between *Room* and *Wall*;
- *hasWindow* between *Wall* and *Window*;
- *hasDoor* between *Wall* and *Door*;
- *roomAccess* - between two *Areas*;
- *areaAccess* - between two *Rooms*.

4.2 Creating Internal Structure of the House

In GraCAD every operation that is used to specify functional requirements or to create a prototype design is invoking a PROGRES graph transformation that modifies the graph of the house structure.

For example, assigning a room to a functional area is equivalent to invoking the following PROGRES rule (production):



This rule matches with its left-hand side (above "::=" a node with identifier *roomId* belonging to the class *Room* and an *Area* node with identifier *areaId*. Its right-hand side (below "::=") creates a *hasRoom* edge between the two given nodes.

Note that in ArchiCAD the layout of the building is coded in terms of walls, whereas the main building block in GraCAD is a room. Therefore, seamless transformation between both formats of knowledge representation must be granted for the user. Each time when the user inserts a room into the prototype design window the system creates four instances of the class *Wall*. Their attributes contain data defining the location of each wall in the global co-ordinate system. The relation between the room and its walls is established by connecting *Wall* nodes with the *Room* node by means of the *hasWall* edges.

The graph of the house structure created this way is passed to the *Room Adjuster* and to the rule checkers.

4.3 Implementation of Rule Checkers

In this subsection a simple example of an architectural norm checker is presented. The PROGRES transaction *findTooSmallRooms* finds the rooms that are too small according to architectural norms. The class *Room*

```
node class Room is a Object
  meta
    minSurface : integer := 0;
  derived
    surface : integer
      = ABS ( (self.wall ( "N" ).x2 - self.wall ( "N" ).x1)
              * (self.wall ( "E" ).y2 - self.wall ( "E" ).y1) );
end;
```

has two attributes: *minSurface* and *surface*. The first one is a meta-attribute (attribute with constant value for all nodes of the same class/type) that is redefined in the classes inheriting the *Room* class. An example is the class *LivingRoom*

```
node type LivingRoom : Room
  redef meta
    minSurface := 18 ;
end;
```

The *minSurface* attribute indicates the minimal surface area for the given type of room. The attribute *surface* is derived on the basis of the known attributes of walls, namely, their co-ordinates. It uses a path expression *wall* with a direction parameter to determine the north and west walls of a room (not shown in the paper) and uses their coordinate attributes (x1,y1) and (x2,y2) to compute a room's actual surface. The surface area of the room is automatically re-evaluated whenever the positions of walls change.

The PROGRESS restriction *tooSmall*

```
restriction tooSmall : Room =
  valid (self.minSurface > self.surface)
end;
```

is used in the query *findTooSmallRooms*:

```
query findTooSmallRooms( out rooms : Room [0:n]) =
  rooms := (instance of Room).tooSmall
end;
```

This query first determines the set of all instances of class *Room* of the regarded graph and selects then all those rooms which fulfil the restriction *tooSmall*. This set of nodes is returned in the out parameter *rooms* whose multiplicity *[0:n]* indicates that it is a set-valued parameter.

In the future version of the GraCAD PROGRES specification all kinds of constraints/norms will be translated into incrementally checked constraints of the PROGRES language (or derived boolean attributes). Some of them will have useful automatic repair actions, others not. GraCAD tool equipped with such a specification may be used in several modes, thereby supporting different styles of interaction of an architect with our tool:

- *constraint checking deactivated*
- *constraint checking activated*: violated constraints are displayed, a user is responsible for eliminating violations
- *repair actions activated*: existing repair actions eliminate violated constraints if possible; remaining inconsistencies are displayed/highlighted
- *constraints enforcement activated*: any transaction/action leading to a violation of a constraint without repair action results in aborting the intended editing process

5 Summary

The interviews with architects confirm that an add-on to a standard CAD-tool aiding the conceptual phase of design is worth developing. It may increase the architect's productivity, shorten the way from the talk with the client to the prototype design, help the designer to operate on the abstract conceptual level, and facilitate the communication with the investor.

GraCAD is a suggestion of such a system in which graph-based knowledge representation techniques are used. The proposed methodology seems to be appropriate for architects because they often use graphs - sometimes being unaware of it. The experience gained so far suggests that it is possible to hide the formalism of a graph-oriented language from the user allowing him or her to work comfortably at the level of a graphical interface.

The GraCAD add-on and the PROGRES specification are still in a process of constructing. The development of GraCAD will be continued in a co-operation of architects from the Computer Modelling Department at Cracow Technical University. In collaboration with them we will evaluate and improve the already realized constraint, checkers and adjustment generation strategies. The other interesting task to consider is a reverse transformation from an ArchiCAD representation into a graph one, however such reverse engineering seems to be quite difficult (or even impossible) because of the lack of semantic information about the rooms' purpose in standard ArchiCAD model. As a result of the research we intend to create a tool useful in praxis that aids designing objects more complex then a single-store family house.

Acknowledgement

This research was partly supported by the Federal Ministry of Education and Research (BMBF) under the joint Polish-German research project "Graph-based tools for conceptual design in Civil Engineering" coordinated by M. Nagl (RWTH, Aachen).

References

1. ArchiCAD 6.5 Reference guide, Graphisoft, Budapest, 2000
2. Booch, G., Rumbaugh, J., Jacobson, I.: The Unified ModelingLanguage User Guide. Addison Wesley Longman, Reading(1999)
3. Borkowski, A., Grabska, E.: Converting function into object. In: I. Smith, ed., Proc. 5th EG-SEA-AI Workshop on Structural Engineering Applications of Artificial Intelligence, LNCS 1454, Springer-Verlag, Berlin (1998), 434-439
4. Borkowski, A. (ed.): Artificial Intelligence in Structural Engineering, WNT, Warszawa (1999)
5. Borkowski, A., Grabska, E., Hliniak, G.: Function-structure computer-aided design model, Machine GRAPHICS & VISION, 9, Warszawa (1999), 367-383
6. Chomsky, N.: Aspects of Theory of Syntax, MIT Press, Cambridge (1965)

7. Cole Jr., E.L.: Functional analysis: a system conceptual design tool, IEEE Trans. on Aerospace & Electronic Systems, 34 (2), 1998, 354-365
8. Flemming, U., Coyone, R., Gavin, T., Rychter, M.: A generative expert system for the design of building layouts - version 2, In: B. Topping, ed., Artificial Intelligence in Engineering Design, Computational Mechanics Publications, Southampton (1999), 445-464
9. Göttler, H., Günther, J., Nieskens, G.: Use graph grammars to design CAD-systems! 4th International Workshop on Graph Grammars and Their Applications to Computer Science, LNCS 532, Springer-Verlag, Berlin (1991), 396-410
10. Grabska E.: Graphs and designing. In: H. J. Schneider and H. Ehrig, eds., Graph Transformations in Computer Science, LNCS 776, Springer-Verlag, Berlin (1994), 188-203
11. Grabska, E., Borkowski, A.: Assisting creativity by composite representation, In: J. S. Gero and F. Sudweeks eds., Artificial Intelligence in Design'96, Kluwer Academic Publishers, Dordrecht (1996), 743-760
12. Korzeniewski, W.: Apartment Housing - Designers Guide (in Polish), Arkady, Warszawa (1989)
13. Neufert, E.: Bauentwurfslehre, Vieweg & Sohn, Braunschweig-Wiesbaden (1992)
14. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, World Science, Singapore (1997)
15. Schürr, A., Winter, A., Zündorf, A.: Graph grammar engineering with PROGRESS. Proc. 5th European Software Engineering Conference (ESEC'95), W. Schäfer, P. Botella (Eds.), LNCS 989, Springer-Verlag, Berlin (1995), 219-234
16. Stiny, G.: Introduction to shape and shape grammars, Environment and Planning B: Planning and Design, 7, 1980, 343-351
17. Szuba. J.: Applying generative systems in design (in Polish). Master Thesis, Jagiellonian University, Kraków (1999)
18. Szuba, J., Grabska, E., Borkowski, A.: Graph visualisation in ArchiCAD. In: M. Nagl, A. Schürr, M. Münch, eds., Application of Graph Transformations with Industrial Relevance, LNCS 1779, Springer-Verlag, Berlin (2000), 241-246
19. Szuba, J., Borkowski, A.: Graph transformation in architectural design, Computer Assisted Mechanics and Engineering Science, to appear

A Formal Semantics of UML Statecharts by Model Transition Systems*

Dániel Varró

Budapest University of Technology and Economics
Department of Measurement and Information Systems
H-1521, Budapest, Magyar tudósok körútja 2
varro@mit.bme.hu

Abstract. UML Statecharts are well-known visual means to capture the dynamic behavior of reactive systems in the object-oriented design methodology. Since the UML standard only contains an informal description on how to execute such statemachines mathematically precise semantic frameworks are required for an automated analysis. The current paper presents a formal semantics for UML statecharts based on a combination of metamodeling and graph transformation that is (i) simultaneously visual and precise, and (ii) clearly separates derived static concepts (like priorities, conflicts, etc.) from their dynamic interpretation thus scaling up well for different statechart variants (with, e.g., various priority strategies) and potential future changes in the standard.

1 Introduction

For the recent years, the Unified Modeling Language (UML) [18] has become the de facto standard modeling language in the design process of object–oriented systems including a variety of complex applications ranging from object–oriented software to embedded real-time systems. Both static and dynamic aspects of such systems are captured visually, by a series of diagrams. The dynamic behavior of system objects are described by *UML Statecharts*, which is a statemachine variant having its origins in the well–known formalism introduced for the first time by Harel in [8].

However, the growing complexity of IT systems revealed several shortcomings of the language stemming from the fact that UML lacks a precise dynamic semantics. Whereas the static semantics is described by metamodels and a constraint language up to a certain level of preciseness, its execution semantics is only given informally in a natural language.

Unfortunately, there is a huge abstraction gap between the "graphical" world of UML and all the mathematical models of describing dynamic semantics (such

* This work was carried out during my visit to Computer Science Laboratory at SRI International (333 Ravenswood Ave., Menlo Park, CA, U.S.A.) and supported by the National Science Foundation Grant (CCR-00-86096), and the Hungarian Information and Communication Technologies and Applications Grant (IKTA 00065/2000)

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 378–392, 2002.
© Springer-Verlag Berlin Heidelberg 2002

as transition systems, Petri nets, abstract state machines, process algebras, etc.). Consequently, systems engineers will require the back–annotation of analysis results into the original UML formalism, as well as an easy–to–understand, visual specification of the dynamic semantics.

Graph transformation (see e.g. [17]) provides a mathematically precise and visual specification technique by combining the advantages of graphs and rules into a single computational paradigm. Its potential domains of application include e.g. pattern recognition, functional programming languages, database systems, distributed systems, and, recently, transformations within and between UML diagrams (cf. [6, 9, 23]).

In the current paper, we present a *rule-based, visual specification of statecharts semantics by* means of *model transition systems* (a combination of metamodeling and graph transformation with explicit control structures) that provides a better understanding for systems engineers by separating derived static concepts (conflicts, priorities, etc.) and their dynamic interpretation (enabledness, fireability).

In fact, the framework presented here was implemented to form the semantic front-end for several analysis methods within our general, transformation-based formal verification and validation framework of UML models based on the VIATRA environment [3]. For instance, in [21], we propose an automated encoding of model transition systems into the SAL (Symbolic Analysis Laboratory [1]) intermediate language to provide access to *wide range of verification methods* provided by the SAL environment. The UML statechart semantics of the current paper served as the benchmark application for evaluating this encoding, however, a detailed discussion of this approach is out of the scope of the current paper.

1.1 Related Statecharts Semantics

Since the original formalism of Harel [8], the theory of statecharts has been under an extensive research and many different semantic approaches evolved from the academic world (a comparison of different approaches can be found in e.g. [24]). However, as the industrial interest is rather limited to the Statemate and UML variants, therefore, the majority of recent approaches for statecharts semantics have typically focused on the formalization of those variants. In this section, we restrict our attention to compare only proposals for the UML dialect with a stress on the support of formal verification.

Extended Hierarchical Automata, which form the structural basis of our statechart semantics, were introduced in [14] for Statemate and in [13] for UML. In a second phase, both approaches transform their models into Promela code and verify them by the model checker SPIN [10]. A major stress is put on formal verification in [20] where UML statecharts are encoded into a PVS [15] specification enabling the access to automated theorem proving of UML design, while in [12], the model checking of UML statecharts is aimed.

An entire verification round-trip is reported in [2] and [16] where the results of model checking are represented visually in the original UML models. The

semantic core of statecharts is formalized in those papers by means of abstract state machines and state term graphs, respectively.

Despite their success from a verification point of view, the use of precise, formal mathematics is also the common weakness of all these approaches: they fail to provide a high level of abstraction that can be properly understood (and implemented) by systems engineers. Previous proposals in the field of graph transformation (e.g., [7, 11]) have tried to tackle this problem by providing a *visual specification* of statechart semantics.

Even though these proposals derive their internal graph representation for UML models directly or indirectly from the standard UML metamodel, semantic concepts are typically hard coded into the semantic rules, which does not scale up well for different statechart languages or the future evolution of the UML language itself. For instance, to implement the inverse priority concepts of Statemate semantics would require a major revision in all these approaches.

In the current paper, we define the dynamic behavior of UML statecharts by combining metamodeling and graph transformation techniques. However, our main contribution is to simultaneously *include the purely syntactic* (states, transitions, events) *and derived static semantic concepts of statecharts* (like conflicts, priorities, etc.) *in the metamodel, but separate them from their dynamic operational semantics,* which is specified by graph transformation rules. This philosophy keeps the metamodel and the graph transformation rules easy to be understood and maintained for statechart variants. Additionally, our methodology provides direct access to the formal verification of UML statecharts by applying the techniques investigated in [21].

The rest of the paper is structured as follows. Section 2 gives a brief introduction to UML Statecharts and Extended Hierarchical Automata [13], which latter one will serve as the underlying mathematical structure. In Sec. 3.1, an theoretical overview is provided on model transition systems, while Sec. 3.2 formalizes the semantics of statecharts. Finally, Sec. 4 concludes our paper.

2 An Informal Introduction to UML Statecharts

UML statecharts (see the example in Fig. 1) are an object–oriented variant of classical Harel statecharts [8] that describe behavioral aspects of the system under design. In fact, the statechart formalism itself is an extension of traditional state transition diagrams.

UML **Statemachines** are basically constructed from *states* (including the top state of the hierarchy) and *transitions.*

States. As one of the main concepts of statecharts is state refinement, states can be either *simple*, *composite* or *concurrent* (disregarding from pseudo states like *initial* and *final* states).

Simple states (like `s2` or `s3` in Fig. 1) are at the final level of refinement. State `s1`, on the contrary, is refined into two distinct regions (represented as a state in UML), `s4` and `s5`, each of them is refined in turn into an automaton

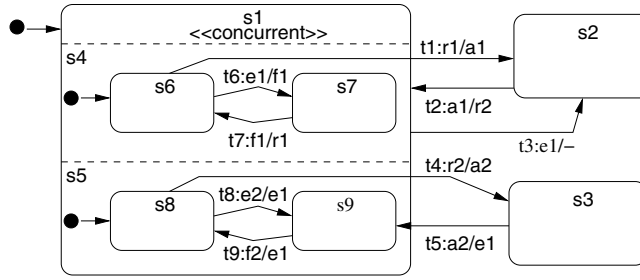


Fig. 1. A sample UML statemachine

consisting of further substates (e.g. `s6`, `s7`). States refined to sub–states are denoted as **composite**, additionally, `s4` and `s5` are called **concurrent** regions of the concurrent state `s1` as each of them has an active substate.

At one point in time, the set of all active states forms the active **configurations**. For instance, our sample system can be any of the following configurations: {`s1,s6,s8`}, {`s1,s6,s9`}, {`s1,s7,s8`}, {`s1,s7,s9`}, {`s2`}, {`s3`}.

Transitions. A **Transition** connects a **source** state to a **target** state. A transition is labeled by a trigger **event**, a boolean **guard** and a sequence of **actions**.

A transition is enabled and can fire if and only if its source state is in the current configuration, its trigger is offered by the external environment and the guard is satisfied. In this case, the source state is left, the actions are executed, and the target state is entered.

In our example, if event `a1` is offered by the environment and the current configuration is {`s2`}, then state `s2` is left and state `s1` is entered. In particular, as `s1` is composite, we also have to define which are the *substates* that are reached. In the case at hand, they are the default ones specified by the **initial** states of `s4` and `s5`, namely, `s6` and `s8`. In a general case, the source and target state of a transition may be at a different level of the state hierarchy. Such a transition is denoted then as **interlevel**.

Event Dispatching. In general, more than one event can be available in the environment. The UML semantics assumes a **dispatcher**, which selects one event at a time from the environment and offers it to the state machine. As a result, more than one transition can be enabled, which may cause a **conflict** to be resolved if the intersection of the states left by the enabled transitions is not empty. Conflicting transitions are tried to be resolved by using priorities: a *transition has higher priority* than another transition if its source state is a substate of the other transition's source state. If conflicts cannot be resolved by priorities, any of the enabled transitions can be fired, moreover, according the the run-to-completion step, all transitions from the non–conflicting subset of enabled transitions are fired at a time.

Non–standard UML Extensions. In the current version of our statechart semantics, several concepts of the UML standard (such as history states, deferred

events) have been omitted for space limitations. We hope that our formalization concepts in Section 2.1 will demonstrate that the "neglected" parts can easily be integrated into a future version.

On the other hand, we also had to extend the original UML standard due to the lack of a proper specification for event queues. Currently, one queue is attached to one object and stores a *sets* of events, however, our approach can easily be extended to alternate queue models (FIFO, multiset, etc).

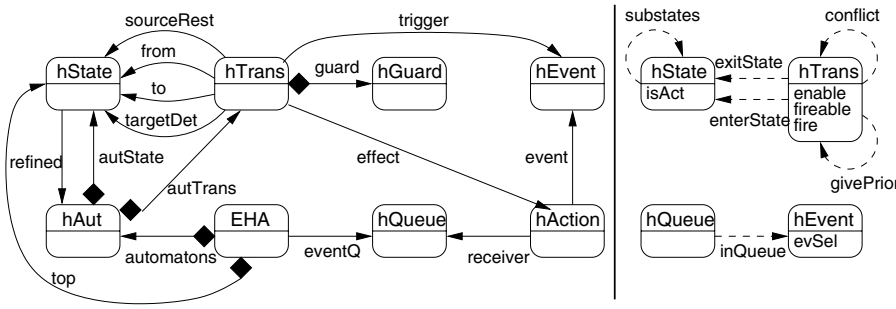
2.1 Extended Hierarchical Automaton

The UML version of Extended Hierarchical Automaton (EHA) were introduced in [13] to provide an alternate representation and a formal operational semantics for Statechart diagrams by a small number of complex transition rules. In this paper, the EHA model is considered to be only an *alternate structural representation of statecharts*, while the original semantic domain is replaced by a set of dynamic attributes and relations manipulated by graph productions.

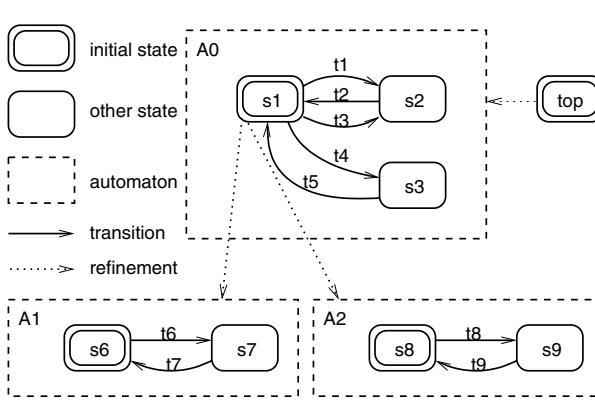
Note that the formalization method to be presented below could be applied straightly to the language of UML statecharts. We believe that the intermediate EHA representation provides greater flexibility when further statechart variants (e.g. the Statemate or Matlab dialects) are considered in the future. In fact, the EHA notation can be derived automatically from the original UML statechart notation by a rather syntactic graph transformation process (see a detailed discussion in [22]).

Metamodel and Model of Extended Hierarchical Automaton. The structural basis of Extended Hierarchical Automaton are defined by its metamodel in Fig. 2(a), while the EHA encoding of our sample statechart is shown in Fig. 2(b). The classes of the EHA metamodel are prefixed with the letter *'h'* in order to avoid name clashes with the original notions of UML statecharts.

- An Extended Hierarchical Automaton (EHA) is composed of a top hState, and sequential automatons of class hAut. Additionally, an EHA is attached to an arbitrary number eventQueues. Each instance of a UML class that is associated with a statemachine is projected into a distinct EHA instance.
- A sequential automaton hAut is generated for (i) each non-concurrent composite state and (ii) for each regions of a concurrent composite state in a UML statechart. It is composed of hStates (referred by autStates) and hTransitions (accessed by autTrans).
- Each UML state that is not a region of a concurrent state is transformed into a hState. The state refinement relation is preserved by the refined association linking states to their subautomatons.
- A transition in UML statecharts has a corresponding transition hTrans in its EHA equivalent linking the from hState and the to hState. All EHA transitions are non-interlevel thus connecting states that belong to the same sequential automaton, which is the EHA equivalent of the *least common*



(a) An EHA metamodel



t	SR	EV	AC	TD
t1	s6	r1	a1	s2
t2	s2	a1	r2	s6 s8
t3	s1	e1	–	s2
t4	s8	r2	a2	s3
t5	s3	a2	e1	s6 s9
t6	s6	e1	f1	s7
t7	s7	f1	r1	s6
t8	s8	e2	e1	s9
t9	s9	f2	e1	s8

(b) A sample EHA model

Fig. 2. Extended Hierarchical Automaton

ancestor[1] state in the UML statemachine. The original source state(s) of transition is denoted by the source restriction relation (sourceRest). The target determinator (targetDet) of a transition relates the set of hStates that (i) have a simple state equivalent in the original UML model, and (ii) they must be entered implicitly when the transition is fired (thus more than a single target determinator state may be connected to e.g., transition t2).

- A hTransition is triggered by a related hEvent, and its effect is defined by its corresponding hAction. In the paper, we restrict our attention to send actions, i.e., all the actions are considered as sending an event to the specified eventQueue. The guard condition is a boolean expression that must hold to allow the transition to be enabled.
- An EHA object is automatically associated to one event queue where the messages sent to the object arrive from.

[1] The *least common ancestor* of a UML transition is the lowest level state that is a superstate of both the source and the target states

In order to improve the clarity of Fig. 2(b), we represented the events (EV), actions (AC), source restrictions (SR) and target determinators (TD) of a transition in a table. Note that grey areas in the table represent the corresponding values for interlevel UML transitions.

In the metamodel, static parts of the metamodel (left from the vertical line) were kept separated from dynamic (and derived) relations and attributes (right from the vertical line). All the previous concepts are regarded as *static parameters*, since they are derived from the original UML model at compile time. For describing the *dynamic behavior* of statecharts in an easy-to-understand way, additional attributes and relations are required.

- The boolean attribute isAct of a hState will be set to true whenever the hState is active (i.e. member of the current configuration). The attribute evSel of a hEvent is true when this hEvent is selected by the dispatcher (as the dispatcher belongs to the environment, this attribute will be handled non–deterministically).
- The relation substates connects a hState to its descendent hStates in the EHA state hierarchy. When a hQueue is related to a hEvent by an inQueue edge, this fact denotes that the hEvent is a member of the hQueue set.
- The attributes enabled, fireable and fire will denote, respectively, (i) when a transition is triggered by the selected event and its origin is an active state, (ii) it is enabled and has sufficient priority to be fired, and (iii) it is selected to be fired.
- There are four additional relations for hTransitions, which are, in fact, static but not part of the EHA model introduced in [13]. The relation exitState explicitly enumerates all the states that have to be exited when the transition is fired. Similarly, the enterState relation lists all the states to be entered when a transition is fired. Two hTransitions are in a conflict relation (i.e. they might be in conflict when firing them) if their exitState set is not disjoint. While givePrior specifies the priority relation between hTransitions.

The major distinction between static and dynamic concepts is that static parts are not modified while an EHA is being operated by the upcoming graph transformation rules. Therefore, the dynamic attributes (and relations) are *initialized* and not *compiled*. With this respect, the semantic formalization of EHA will be divided into two subsequent phases, namely, (i) a *preprocessing* phase for generating derived properties and initializing dynamic constructs, and (ii) the *execution* of core EHA semantics itself.

3 A Rule-Based Visual Semantics for UML Statecharts

3.1 Theoretical Background: Model Transition Systems

Up to this point, the only the *syntactic domain* of statechart models and metamodels were discussed. A traditional approach of defining *static semantics* for models and metamodels is provided by a mapping to directed, typed and attributed graphs. In this sense, a model will conform to its metamodel, if its

model graph conforms to the corresponding type graph with respect to a typing homomorphism.

Type Graphs and Model Graphs. All classes are mapped into a graph node and all associations are projected into a graph edge in the **type graph**. The inheritance hierarchy of metamodels can be preserved by an appropriate *subtyping relation* on nodes (and possibly, on edges). Class attributes are derived into graph attributes where the latter may be treated mathematically as (possibly partial) functions from nodes to their domains.

Objects and links between them are mapped into nodes and edges, respectively, in the **model (instance) graph**. Each node and edge in the model graph is related to a corresponding graph object in the type graph by a corresponding *typing homomorphism.*

The operational dynamic semantics of Hierarchical Automaton will be formalized by *model transition systems* (introduced in [23]), which is a variant of graph transformation systems with a predefined set of control structures.

Definition 1. *A **model transformation rule** $r = (L, N, R)$ is a special graph transformation rule, where all graphs L, N and R are model graphs.*

The **application** of r to a **host graph** G (according to the single pushout approach [5]) replaces an occurrence of L (left-hand side, LHS) in G by an image of R (right-hand side, RHS) yielding the derived graph H. This is performed by

1. *finding an occurrence* of L in G, which is either an isomorphic or a non–isomorphic image according to M
2. *checking the negative application condition* N, which prohibit the presence of certain nodes and edges (negative application conditions are denoted by shaded grey/red areas labeled with the NEG keyword).
3. *removing* those nodes and edges of the graph G that are present in L but not in R yielding the **context graph** D (all dangling edges are removed at this point)
4. *adding* those nodes and edges of the graph G that are present in R but not in L attaining the **derived** graph H.

The entire model transformation process is defined by an initial graph manipulated by a set of model transformation rules (micro steps) executed in a specific mode in accordance with the semantics (macro steps) of a hierarchical control flow graph.

Definition 2. *A **model transition system** $MTS = (Init, R, CFG)$ with respect to (one or more) type graph TG is a triple, where $Init$ defines the **initial graph**, R is a set of **model transformation rules** (both compatible with TG), and CFG is a set of a **control flow graphs** defined as follows.*

- *There are six types of nodes of the CFG: **Start**, **End**, **Try**, **Forall**, **Loop** and **Call**.*
- *There are two types of edges: **succeed** and **fail**.*

The control flow graph is evaluated by a virtual machine which traverses the graph according to the edges and applies the rules associated to each node.

1. The execution starts in the **Start** and finishes in the **End** node. Neither types of nodes have rules associated to them.
2. When a **Try** node is reached, its associated rule is tried to be executed. If the rule was applied successfully then the next node is determined by the **succeed** edge, while in case the execution failed, the **fail** edge is followed.
3. At a **Loop** node, the associated rule is applied as long as possible (which may cause non-termination in the macro step).
4. When a **Forall** node is reached, the related rule is executed parallelly for all distinct (possible none) occurrences in the current host graph.
5. At a **Call** node (which has an associated CFG and not a rule) the state of the CFG machine is saved and the execution of the associated CFG is started (in analogy with function calls in programming languages). When the sub CFG machine is terminated, the saved state is restored, and the execution is continued in accordance with the outgoing edge (**succeed** or **fail**).

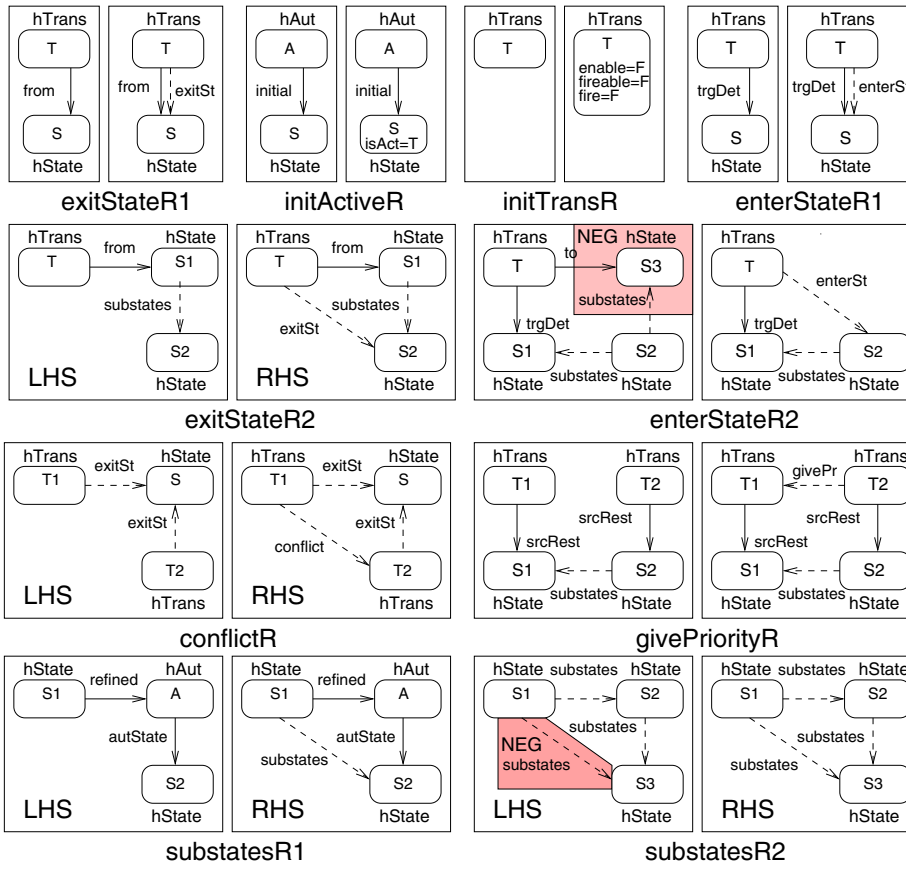
Note that this CFG model follows the control flow concepts of the VIATRA tool. However, the use of "as long as possible" kind of control conditions (and additional negative application conditions) instead of forall nodes would almost directly yield the appropriate control conditions, for instance, for PROGRES [19].

3.2 An Operational Semantics of Extended Hierarchical Automaton

The semantics of Extended Hierarchical Automaton is defined by a model transition system. The **initial graph** is the *static model* generated by the SC2EHA transformation. The initialization of the dynamic aspects and the derivation of extensional attributes are separated into a *preprocessing phase*, while the execution of the semantic rules of EHA form the *operational phase*. The top–level *ehaSemantics* module thus consists of two **Call** nodes — one for the specification of each phases.

The Preprocessing Phase. The *initDynamics* module is mainly responsible for *deriving those static relations* that are required for an easy-to-understand formalization of dynamic behavior. In addition, the initialization of dynamic attributes also takes place at this stage. The preprocessing phase consists of 10 rules (see Fig. 3) that are executed in according to the control flow graph in the following order. An alternate solution for this preprocessing phase is to use path expressions for such derived relationships.

1. *substatesR1*: The rules *substatesR1* and *substatesR2* build up the substate relationship in two steps. At first, if a hState S_1 is refined to a hAutomaton A, all the hStates S_2 of this automaton are substates of S_1 (a **Forall** execution).



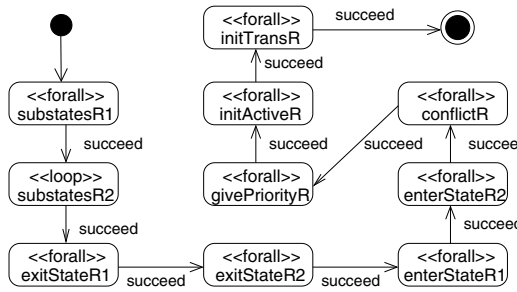


Fig. 3. Static relations in the EHA model

2. *substatesR2*: Secondly, the transitive closure of the substates relation is calculated by looping rule *substatesR2* as long as possible. This means to add **substates** edges between hStates S_1 and S_3, if S_2 is substate of S_1, S_3 is a substate of S_2 but no substate edges are leading yet between S_1 and S_3

3. *exitStateR1*: Rules *exitStateR1* and *exitStateR2* explicitly connect the hStates that are exited by a hTransition when the transition is fired to the hTransition. At first, the **from** hState S of a hTransition T must be exited.
4. *exitStateR2*: Afterwards, all the **substates** S_2 of the **from** state S_1 of the hTransition T must also be exited.
5. *enterStateR1*: All the states target determinator hStates S of a firing hTransition T are should be entered when firing this transition.
6. *enterStateR2*: Additionally, all the states S_2 that are *superstates* of the target determinator state S_1 of a hTransition T but *not superstates* of the **to** hState S_3 must also be entered. Note that the states to be exited and entered when firing a transition are static information.
7. *conflictR*: The rule *conflictR* connects two hTransitions T_1 and T_2, if there exists a hState S that is linked to both of them by an **exitState** edge, thus it is exited by both transitions causing a *conflict*.
8. *givePriorityR*: According to this rule, a hTransition T_2 has lower priority than hTransition T_1 if for the corresponding source restriction hStates (S_2 and S_1, respectively), S_1 is a substate of S_2.
9. *initTransR* and *initActiveR*: Initially, all the dynamic attributes of hTransitions are set to false.
10. *initActiveR*: A state S becomes active if it is an initial state of some sequential hAutomaton A.

Example. By the end of the preprocessing phase of the EHA model in Fig. 2(b), we derived, for instance, that

- s_6, s_7, s_8 and s_9 are substates of s_1;
- the states to be exited when firing transition t_1 are s_1 (the **from** state), s_6, s_7, s_8 and s_9 (the substates of the **from** state);
- the states to be entered when firing transition t_2 are s_6, s_8 (the target determinators) and s_1 (which is the **to** state);
- transitions t_1, t_3 and t_6 are in conflict with each other since state s_6 is exited by all of them;
- however, t_3 gives priority to t_1 and t_6 as the source restriction state s_1 of t_1 is a superstate of s_6 (the source restriction of t_1 and t_6).

Operational Phase. Now we continue with the discussion of the "more semantical" operational phase (depicted in Fig. 4), where the run-to-completion step of statecharts are refined into a *sequence of more elementary operations*.

1. *selectEventR*: At first, an event E is non-deterministically selected from an event queue Q. If no such events are available, then the execution of EHA terminates. Note at this point, that different (more complex) event handling mechanisms can be selected at this point, however, this non-deterministic selection overapproximates the semantics of such mechanisms.
2. *enableR*: A hTransition T is *enabled* if its source restriction hState S is active, and its trigger hEvent E is selected by the dispatcher.

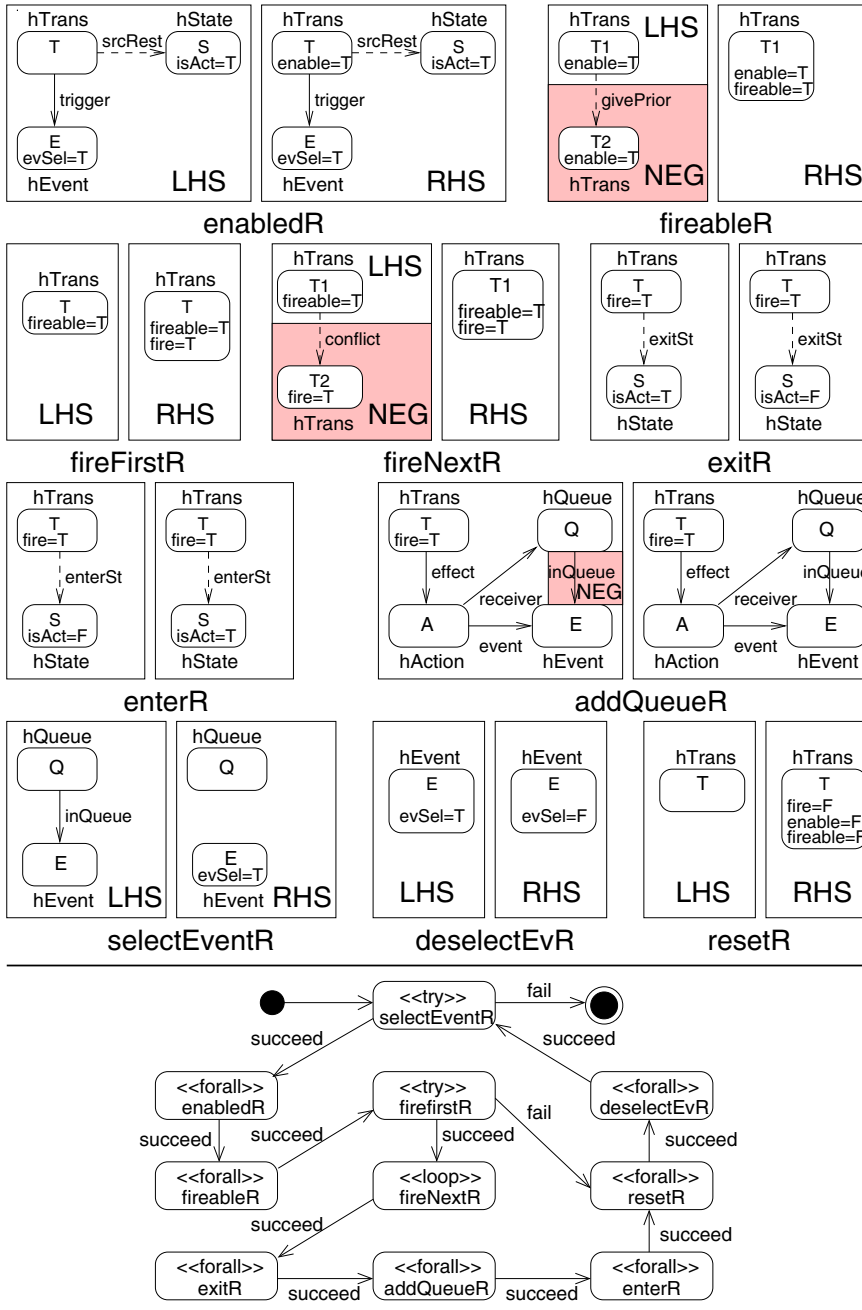


Fig. 4. The model transition system specifying the EHA semantics

3. *fireableR*: An enabled hTransition T_1 becomes *fireable* if there are no enabled hTransitions T_2 of higher priority (see the negative condition).
4. *fireFirstR*: The first (fireable) hTransition T is selected to be fired non–deterministically by setting its **fire** attribute to true. If no such transitions found then the execution continues by resetting dynamic attributes of transitions (see the final step).
5. *fireNextR*: After the success of *fireFirstR*, the set of transitions to be fired fired is extended one by one (by looping `fireNextR`) until all the remaining enabled hTransitions are in conflict with at least one element in the fire set.
6. *exitR*: All the states S marked by an **exitState** edge leading from a firing hTransition T are exited.
7. *addQueueR*: As the effect of firing a hTransition T, the hEvent E associated to the (send) hAction A is added to the corresponding hQueue Q (if the negative condition is removed, then the event queue is modeled as a bag and not a set).
8. *enterR*: All the states S marked by an **enterState** edge leading from a firing hTransition T are entered. This step results in a valid configuration as (i) the origins of target determinator states were simple states, hence no states need to be entered at a lower level than the target determinators and (ii) all EHA transitions are non–interlevel thus no states had been exited at a higher level than the **from** and **to** hStates.
9. *resetR* and *deselectEvR*: All the dynamic attributes of a transition T and an event E are set to false, and a new step of the EHA commences.

Example. Let us assume that the active states of our sample EHA model are the initial states (thus s_1, s_6, and s_8) and the event queue only contains the single event of e_1. According to the previous rules, a run-to-completion step of our hierarchical automaton proceeds as follows.

1. Transitions t_3 and t_6 are enabled by applying *enableR* as the source restriction states (s_1 and s_6, respectively) of both transitions are active.
2. From this enabled set of transitions, the application of *fireableR* eliminates t_3 since t_3 gives priority to t_6.
3. The set of transitions to be fired will consist of the single transition t_6.
4. By applying *exitR*, states s_1, s_6 and s_8 become inactive, while the application of *enterR* results in the activation of state s_2. Meanwhile, event f_1 is added to the event queue by rule *addQueueR*.
5. Finally, all dynamic attributes of all transitions are set to false, and a new run-to-completion step commences.

4 Conclusions

In the current paper, we proposed a visual operational semantics for UML statecharts based on metamodeling techniques (Extended Hierarchical Automaton as the underlying static structure) and model transition systems (for defining operational semantics). The main contribution of the paper is to partition the

complex (but rather informal) semantic rules of statecharts in the UML standard into elementary operations separating derived static concepts (conflicts, priorities) from their dynamic interpretation (enabledness, fireability). In this respect, our approach can be easily adapted to different statechart variants (e.g., with different priority concepts) and upcoming changes in the UML standard.

The presented framework was tested within the VIATRA tool [23]. Moreover, following the guidelines of [21], we directly transformed our UML statechart semantics to SAL specifications [1] in order to provide access to a combination of symbolic verification techniques.

Acknowledgments

I would like to thank András Pataricza, István Majzik (Budapest University of Technology), John Rushby and many of his colleagues (at SRI International) the fruitful discussions, and their encouragement and support. Moreover, I am also very grateful to the anonymous reviewers for their valuable comments.

References

1. S. Bensalem, V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman, and A. Tiwari. An overview of SAL. In C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.
2. K. Compton, Y. Gurevich, J. Huggins, and W. Shen. An Automatic Verification Tool for UML. Technical Report CSE-TR-423-00, 2000.
3. G. Csertán, G. Huszerl, I. Majzik, Z. Pap, A. Pataricza, and D. Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In *Proc. ASE 2002: International Conference on Automated Software Engineering*. Submitted paper.
4. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation*, volume 2: Applications, Languages and Tools. World Scientific, 1999.
5. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. *In [17]*, chapter Algebraic Approaches to Graph Transformation — Part II: Single pushout approach and comparison with double pushout approach, pages 247–312. World Scientific, 1997.
6. G. Engels, J. H. Hausmann, R. Heckel, and S. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard*, volume 1939 of *LNCS*, pages 323–337. Springer, 2000.
7. M. Gogolla and F. Parisi-Presicce. State diagrams in UML: A formal semantics using graph transformations. In M. Broy, D. Coleman, T. S. E. Maibaum, and B. Rumpe, editors, *Proceedings PSMT'98 Workshop on Precise Semantics for Modeling Techniques*. Technische Universität München, TUM-I9803, 1998.
8. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

9. R. Heckel, J. Küster, and G. Taentzer. Towards automatic translation of UML models into semantic domains. In *Proc. AGT 2002: Workshop on Applied Graph Transformation*, pages 11–21, Grenoble, France, April 2002.
10. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
11. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.
12. G. Kwon. Rewrite rules and operational semantics for model checking UML statecharts. In A. Evans, S. Kent, and B. Selic, editors, *UML 2000 - The Unified Modeling Language. Advancing the Standard. Third International Conference, York, UK, October 2000, Proceedings*, volume 1939 of *LNCS*, pages 528–540. Springer, 2000.
13. D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.
14. E. Mikk, Y. Lakhnech, and M. Siegel. Hierarchical automata as model for statecharts. In R. Shyamasundar and K. Euda, editors, *ASIAN'97 Third Asian Computing Conference. Advances in Computer Science*, volume 1345 of *LNCS*, pages 181–196. Springer-Verlag, 1997.
15. S. Owre, N. Shankar, J. Rushby, and D. Stringer-Calvert. The PVS language reference, Version 2.3. Technical report, SRI International, September 1999.
16. I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In R. J. Hall and E. Tyugu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.
17. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations: Foundations*. World Scientific, 1997.
18. J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.
19. A. Schürr, A. J. Winter, and A. Zündorf. *In [4]*, chapter The PROGRES Approach: Language and Environment, pages 487–550. World Scientific, 1999.
20. I. Traoré. An outline of PVS semantics for UML Statecharts. *Journal of Universal Computer Science*, 6(11):1088–1108, Nov. 2000.
21. D. Varró. Towards symbolic analysis of visual modelling languages. In *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*. Submitted paper.
22. D. Varró and A. Pataricza. Mathematical model transformations for system verification. Technical report, Budapest University of Technology and Economics, May 2001.
23. D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*, 44(2), August 2002. To appear.
24. M. von der Beeck. A comparison of statecharts variants. In *Proc. Formal Techniques in Real Time and Fault Tolerant Systems*, volume 863 of *LNCS*, pages 128–148, 1994.

Hierarchical Vertex Ordering

Sung-Ho Woo and Sung-Bong Yang

Department of Computer Science
Yonsei University
Seoul, Korea, 120-749
{shwoo,yang}@mythos.yonsei.ac.kr

Abstract. The k-way graph partitioning problem has been solved well through vertex ordering and dynamic programming which splits a vertex order into k clusters [2,12]. In order to obtain "*good clusters*" in terms of the partitioning objective, tightly connected vertices in a given graph should be closely placed on the vertex order. In this paper we present a simple vertex ordering method called *hierarchical vertex ordering* (HVO). Given a weighted undirected graph, HVO generates a series of graphs through graph matching to construct a tree. A vertex order is then obtained by visiting each nonleaf node in the tree and by ordering its children properly. In the experiments, dynamic programming [2] is applied to the vertex orders generated by HVO as well as various vertex ordering methods [1,6,9,10,11] in order to solve the k-way graph partitioning problem. The solutions derived from the vertex orders are then comapred. Our experimental results show that HVO outperforms other methods for almost all cases in terms of the partitioning objective. HVO is also very simple and straightforward.

1 Introduction

Given a weighted undirected graph, vertex ordering is to find a linear mapping of the vertices in the graph into one dimensional space. Vertex ordering can be found in various areas such as classical graph traversals, sparse matrix ordering, and VLSI circuit layout. In this paper we present a simple, yet effective vertex ordering method for linear clustering which splits a vertex order into k clusters, that is, the vertex order is divided simply into k contiguous subsets. One of the most crucial properties for a vertex order is that tightly connected vertices should be closely placed on the order so that the vertices in any contiguous subset of the vertex order should form a "good" cluster while preserving the connectivity of the graph.

The k-way graph partitioning problem has been solved successfully using linear clustering under the condition that a given vertex order captures the clustering structure of a given graph [1,2,12]. The goal of k-way graph partitioning is to divide the vertices in a given graph into k clusters subject to size constraints or balancing factors while minimizing the sum of the weights between the vertices in different clusters. Note that the k-way graph partitioning problem is known to be *NP-complete* [4]. A representative method to find linear clustering

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 393–401, 2002.
© Springer-Verlag Berlin Heidelberg 2002

is the *dynamic programming for restricted partitioning* (DPRP) [2] which splits optimally a given vertex order into k clusters. Alpert and Kahng [1] proposed a general framework for vertex ordering and evaluated various ordering methods with DPRP.

Besides linear clustering, vertex ordering can also be applied to database systems such as geographical information systems and CAD systems. Both types of systems require a proper mapping of the related data items into an one-dimensional disk space so that the I/O cost is minimized in accessing them [3,13]. Vertex ordering can also facilitate a fast remapping in parallel and distributed systems. When the system environment changes such as machine utilization, use pattern, and machine failures, remapping should be performed to minimize the overall communication cost [7] while maintaining load balancing among the processors.

In the literature of graph theory, the vertex ordering problem is known as the *minimum linear arrangement problem* (MINLA) or the graph layout problem which is known to be *NP-hard* [4]. The MINLA is to find a vertex order that minimizes the total distance in the vertex order among all adjacent vertices. Although the minimization of the total distance and the *clustering properties* of the vertex orders are closely related, it is not clear if contiguous vertices on the vertex order obtained by the MINLA form a "good" cluster in terms of the clustering objective.

In this paper, we present a new vertex ordering method named the *hierarchical vertex ordering* (HVO) method. Given a weighted undirected graph G, HVO constructs a vertex order for linear clustering. First, HVO builds a series of graphs G^0, G^1, $\cdots$, and G^h sequentially. G^0 is set to G and G^l is constructed through graph matching in G^{l-1}, where $1 \leq l \leq h$. Building such graphs stops when the constructed graph has two vertices. A tree called the *matching tree* is also constructed along with building the series of graphs. Second, HVO achieves a vertex order of G by traversing the tree top-down. When a node in the tree is visited, the children of the visited node is ordered according to the weights of some associated nodes.

In our experiments, HVO is compared with other ordering methods designed for various applications such as linear clustering [1], VLSI circuit layout [1,6], MINLA problem [11] and sparse matrix ordering [9]. DPRP is used to evaluate the qualities of the clusterings generated by these vertex ordering methods. Our experimental results show that HVO outperforms other methods for almost all cases in terms of the qualities of clusters. HVO is also very simple and straightforward.

The rest of this paper is organized as follows. In the next section we define formally the k-way graph partitioning problem and the linear clustering. Section 3 describes HVO and its analysis in detail. Section 4 presents our experimental results and the performance comparisons of HVO and other methods. Section 5 provides the conclusions.

2 Graph Partitioning and Linear Clustering

Given a weighted undirected graph $G = (V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$ and $E = \{(v_i, v_j) |\ v_i \text{ and } v_j \text{ are adjacent}\}$, vertex ordering is a one-to-one mapping $\varphi : V \rightarrow \Pi$ such that $\Pi = [\pi_1, \pi_2, \cdots, \pi_n]$. $\varphi(v_i) = \pi_j$ means that v_i is the jth vertex in the order and π_j represents v_i. $w_{v_i v_j}$ represents the connectivity between v_i and v_j. If $(v_i, v_j) \in E$, $w_{v_i v_j}$ is positive and its value means how tightly v_i and v_j are connected each other. Otherwise, $w_{v_i v_j}$ is zero.

The k-way graph partitioning problem is defined as follows [1] : Given a weighted undirected graph $G = (V, E)$, a number k of clusters, $2 \leq k \leq |V|$, and the cluster size bounds L and U which are the lower and upper bounds, respectively, construct $P^k = \{C_1, C_2, \cdots, C_k\}$ with $L \leq |C_u| \leq U$, $1 \leq u \leq k$, that minimizes a given objective function $f(P^k)$. We define $f(P^k)$ as the sum of the weights between the vertices in different clusters:

$$f(P^k) = \sum_{u=1}^{k} W(C_u), \quad \text{where } W(C_u) = \sum_{\{(v_i, v_j) \in E | v_i \in C_u \text{ and } v_j \notin C_u\}} w_{v_i v_j}.$$

Given G and its vertex order Π, linear clustering solves the k-way graph partitioning problem by splitting Π into k clusters each of which is a contiguous subset of Π. Let $C_{[i,j]}$ be a cluster which consists of the contiguous vertices from π_i to π_j on Π, where $1 \leq i < j \leq n$. Since each $C_{[i,j]}$ is uniquely determined by π_i and π_j, only $(U - L + 1) \cdot n$ clusters can be part of any solution of linear clustering [1]. Among the $(U - L + 1) \cdot n$ clusters, DPRP finds the optimal k clusters with the time complexity of $O(k(U - L)n^2)$.

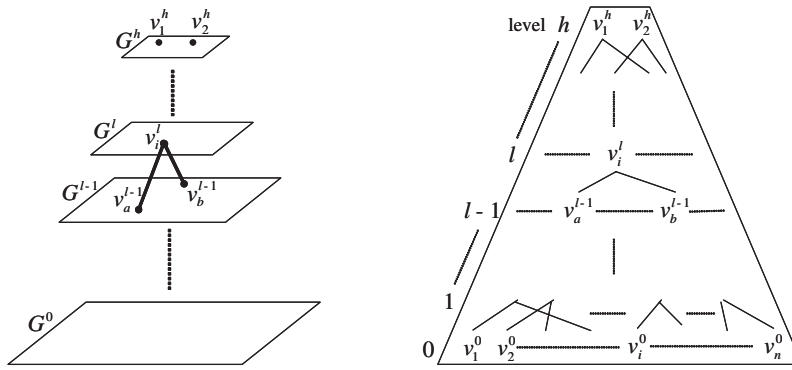
3 Hierarchical Vertex Ordering

HVO has two steps: 1) the construction of a *matching tree* T and 2) hierarchical ordering. In the first step, a graph matching is used to construct a matching tree. In the second step, a vertex order is achieved by visiting each nonleaf node from the highest level and by ordering the children of the visited node.

3.1 Construction of a Matching Tree

The algorithm for constructing T is described in Figure 2. *ConstrHierarchy()* accepts $G = (V, E)$ as input and returns a series of graphs $G^0, G^1, \cdots$, and G^h, where $G^0 = G$, and a matchinng tree T. For $1 \leq l \leq h$, $G^l = (V^l, E^l)$ is constructed from $G^{l-1} = (V^{l-1}, E^{l-1})$. We call G^l the *supergraph* of G^{l-1}. The vertex set of G^l is represented by $V^l = \{v_1^l, v_2^l, \cdots, v_{n_l}^l\}$, where $n_l = |V^l|$. For vertex v_i^l which is merged by the vertices v_a^{l-1} and v_b^{l-1} (or possibly only one vertex v_a^{l-1}) in G^{l-1} as shown in Figure 1 (a), we call v_i^l the *supervertex* of v_a^{l-1} and v_b^{l-1}.

To generate G^l from G^{l-1}, all the vertices in G^{l-1} are intialized to *unmatched* (line 4). In the inner **while** loop (lines 6-22), an *unmatched* vertex v_a^{l-1} is selected



(a) The series of graphs (b) The structure of a matching tree

Fig. 1. The series of graphs and its corresponding matching tree

ConstrHierarchy(G)
Input : $G = (V, E)$
Output : $G^0, G^1, \cdots, G^h$ and T

```
[1]  l = 1;
[2]  While |V^{l-1}| > 2 do
[3]      Generate a supergraph G^l = (V^l, E^l) with V^l = ∅ and E^l = ∅;
[4]      Each vertex in V^{l-1} is marked with unmatched;
[5]      i = 1;
[6]      While there exists an unmatched vertex in V^{l-1} do
[7]          Randomly select an unmatched vertex v_a^{l-1};
[8]          Generate a new supervertex v_i^l;   i = i + 1;
[9]          Add v_i^l to V^l;
[10]         If there exist unmatched vertices among the neighbors of v_a^{l-1} then
[11]             v_a^{l-1} is paired with unmatched vertex v_b^{l-1} according to the matching policy;
[12]             v_a^{l-1} and v_b^{l-1} are marked with matched;
[13]             v_i^l.left = v_a^{l-1}; v_i^l.right = v_b^{l-1};
[14]         else
[15]             v_a^{l-1} is marked with matched;
[16]             v_i^l.left = v_a^{l-1}; v_i^l.right = NIL;
[17]         endif
[18]         For each v_j^l ∈ V^l which has a child (or children) adjacent to either v_i^l.left or v_i^l.right
[19]             Add (v_i^l, v_j^l) to E^l;
[20]             w_{v_i^l v_j^l} = w_{v_i^l.left v_j^l.left} + w_{v_i^l.left v_j^l.right} + w_{v_i^l.right v_j^l.left} + w_{v_i^l.right v_j^l.right};
[21]         endfor
[22]     endwhile
[23]     l = l + 1;
[24] endwhile
```

Fig. 2. Pseudo-codes for constructing a matching tree T

randomly (line 7) and one of its *unmatched* adjacent vertex v_b^{l-1} is selected according to a matching policy (line 11) which will be described later in this section. v_a^{l-1} and v_b^{l-1} become the children of a supervertex v_i^l in T (line 13) as shown in Figure 1 (b). Note that HVO does not care who becomes the left or the right child of v_i^l. If there is no *unmatched* vertex among the adjacent vertices (line 14), v_i^l will have a single child in T (line 16). To construct the

connectivity between the supervertices, suppose $v_j^l \in V^l$ which has a child (or children) adjacent to either $v_i^l.left$ or $v_i^l.right$ (line 18). Then, (v_i^l, v_j^l) is added to E^l (line 19) and $w_{v_i^l v_j^l}$ is the sum of the weights between each child of v_i^l and each child of v_j^l (line 20). For the sake of simplicity, we assume that the weight value associated with *NIL* pointer is zero.

We now discuss about the matching policy (line 11). Among several heuristics appeared in [9], we adopt the maximum weighted matching. We also propose a new matching policy, called *relative weighted matching*.

• Maximum weighted matching (MWM) : v_a^{l-1} is matched with v_b^{l-1}, where $w_{v_a^{l-1} v_b^{l-1}}$ is the largest among v_a^{l-1}'s unmatched adjacent vertices.

• Relative weighted matching (RWM) : Let v_b^{l-1} be a vertex adjacent to v_a^{l-1}. Then v_a^{l-1} is matched with v_b^{l-1} whose value of $\widetilde{w}_{v_a^{l-1} v_b^{l-1}}$ is the largest among v_a^{l-1}'s unmatched adjacent vertices, where

$$\widetilde{w}_{v_a^{l-1} v_b^{l-1}} = \frac{\textit{the weight between } v_a^{l-1} \text{ and } v_b^{l-1}}{\textit{the sum of the weights between } v_b^{l-1} \text{ and its adjacent vertices}} = \frac{w_{v_a^{l-1} v_b^{l-1}}}{\sum_{(v_b^{l-1}, v_c^{l-1}) \in E^{l-1}} w_{v_b^{l-1} v_c^{l-1}}}.$$

3.2 Hierarchical Ordering

In the second step of HVO, a vertex order Π^l of G^l, from $l = h$ down to 1, is determined. Let Π^l be $[\pi_1^l, \pi_2^l, \cdots, \pi_{n^l}^l]$. As shown in Figure 3, Π^h is decided first (line 1). Since G^h has only two vertices, the order between these two vertices may be determined arbitrarily.

Once Π^h is achieved, Π^{h-1} is determined by visiting π_1^h in T first and then by visiting π_2^h. When π_i^l is visited, its children on level $l-1$ are ordered according to the connectivities between the left or right child of π_i^l and each of the previous node π_{i-1}^l and the next node π_{i+1}^l of π_i^l in Π^l. We call these four connectivities LPW, LNW, RPW, and RNW as shown in Figure 4. If the value of (LPW-LNW) is positive, the left child $\pi_i^l.left$ is more tightly connected to the children of π_{i-1}^l than those of π_{i+1}^l. Similarly, if (RPW-RNW) is positive, the right child $\pi_i^l.right$ is more tightly connected to the children of π_{i-1}^l than those of π_{i+1}^l. In order to determine the order between the two children, $\pi_i^l.left$ and $\pi_i^l.right$, we compare the value of (LPW-LNW) with that of (RPW-RNW) (line 9). If (LPW-LNW)>(RPW-RNW), then $\pi_i^l.left$ becomes π_j^{l-1} and $\pi_i^l.right$ is placed at the next position, that is, π_{j+1}^{l-1}, in Π^{l-1} (line 10-11). Otherwise $\pi_i^l.right$ becomes π_j^{l-1} and $\pi_i^l.left$ becomes π_{j+1}^{l-1} in Π^{l-1} (lines 13 and 14). Note that when (LPW-LNW) = (RPW-RNW), it does not matter whether $\pi_i^l.left$ positions in front of $\pi_i^l.right$ in Π^{l-1} or not.

After all the nodes on level l are visited according to the sequence of Π^l, Π^{l-1} is completed. In this way, visiting each nonleaf node of T and ordering its children are iterated level by level. Then Π^0 can be achieved finally as a vertex order of the given graph G.

HierarchicalOrdering()
Input : T and $G^0, G^1, \cdots, G^h$
Output : $\Pi : [\pi_1, \pi_2, \cdots, \pi_n]$

```
[1]   π_1^h = v_1^h;  π_2^h = v_2^h;
[2]   For l = h down to 1
[3]       i = 1;  j = 1;      /* Note that i is used for π_i^l and j is used for π_j^{l-1}. */
[4]       For i = 1 to n_l
[5]           If π_i^l.right = NIL  then /* π_i^l has only one child. */
[6]               π_j^{l-1} = π_i^l.left; j = j + 1;
[7]           else
[8]               LPW = w_{π_i^l.left π_{i-1}^l.left} + w_{π_i^l.left π_{i-1}^l.right};
                  LNW = w_{π_i^l.left π_{i+1}^l.left} + w_{π_i^l.left π_{i+1}^l.right};
                  RPW = w_{π_i^l.right π_{i-1}^l.left} + w_{π_i^l.right π_{i-1}^l.right};
                  RNW = w_{π_i^l.right π_{i+1}^l.left} + w_{π_i^l.right π_{i+1}^l.right};
[9]               If (RPW − RNW) < (LPW − LNW) then
[10]                  π_j^{l-1} = π_i^l.left;  j = j + 1;
[11]                  π_j^{l-1} = π_i^l.right;
[12]              else
[13]                  π_j^{l-1} = π_i^l.right;  j = j + 1;
[14]                  π_j^{l-1} = π_i^l.left;
[15]              endif
[16]              j = j + 1;
[17]          endif
[18]      endfor
[19]  endfor
```

Fig. 3. Pseudo-codes for hierarchical ordering

3.3 Time Complexity

The number of iterations of the inner loop in *ConstrHierarchy()* is $O(|E^l|+|V^l|)$, where $O(|E^l|)$ is the time required for the matching in G^{l-1} and $O(|V^l|)$ for constructing a matching tree T. Hence *ConstrHierarchy()* requires $O((|E^0| + |V^0|) \cdot h)$ time because the height of T is $h + 1$. Observe that $|V^l|$ is $|V^{l-1}| - 1$ in the worst case, so h is at most $|V^0| - 1$. Therefore, the total time required for *ConstrHierarchy()* is $O((|E^0| + |V^0|) \cdot |V^0|)$.

The time required for *HierarchicalOrdering()* depends on the number of nodes in T. The number of nodes in T is the total number of vertices in all the graphs of $G^0, G^1, \cdots$, and G^h, since each node on level l was made for each vertex in G^l. The total number of vertices in all the graphs is at most $\frac{1}{2}|V^0|(|V^0| + 1) - 1 = O(|V^0|^2)$, since $|V^l|$ is $|V^{l-1}| - 1$ in the worst case, where $1 \le l \le h$. Therefore, the overall time complexity of HVO is $O((|E^0| + |V^0|) \cdot |V^0|) + O(|V^0|^2) = O(|E| \cdot |V| + |V|^2)$.

4 Experimental Results

In this section, we compare HVO with other ordering methods from various areas. These methods are max-adjacency ordering (MAO) [10], window method (WIN) [1], spectral ordering (SO) [6], hillclimbing with MINLA cost function (HC) [11], and multi-level nested dissection algorithm (MLND) [9]. We also tested BFS, DFS and space-filling curves. But the clusterings derived by them

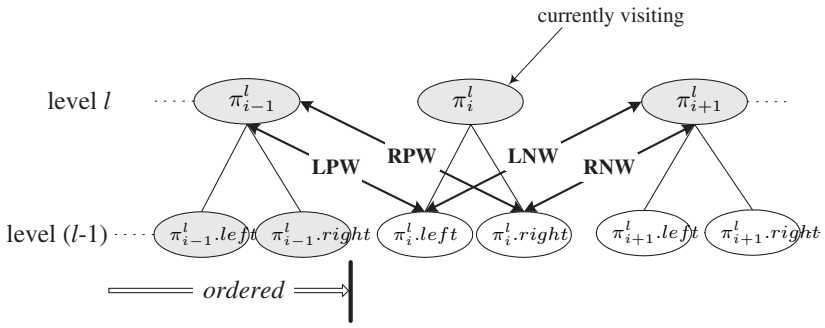


Fig. 4. Criteria for the ordering policy

Table 1. Graphs used in experiments

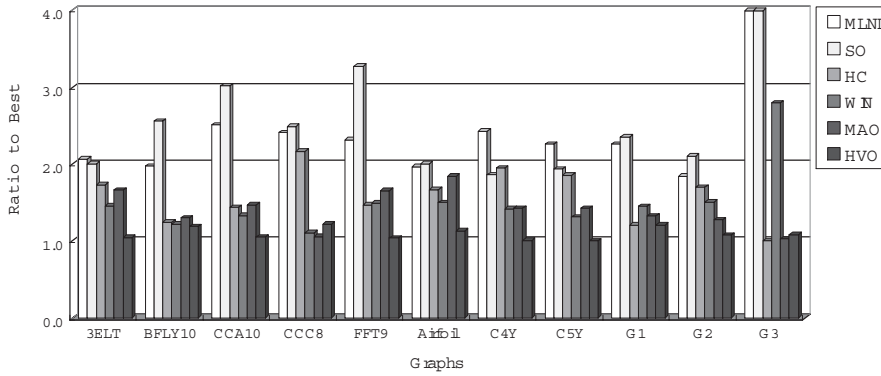
Graphs	$\|V\|$	$\|E\|$	Degree (Min./Avg./Max.)	Description
CCC8	2,048	3,072	3 / 3 / 3	Finite Element Discretizations
Airfoil	4,253	12,289	3 / 5.7 / 9	Finite Element Discretizations
3ELT	4,720	13,722	3 / 5.8 / 9	Finite Element Discretizations
FFT9	5,120	9,216	2 / 3.6 / 4	Finite Element Discretizations
CCA10	10,240	14,336	2 / 2.8 / 3	Finite Element Discretizations
BFLY10	10,240	20,480	4 / 4 / 4	Finite Element Discretizations
C4Y	1,366	2,915	1 / 4.3 / 309	VLSI Design
C5Y	1,202	2,577	1 / 4.2 / 323	VLSI Design
G1	9,000	37,522	2 / 8.3 / 18	Geometric Graph
G2	9,000	23,268	1 / 5.2 /15	Random and Geometric Graph
G3	9,000	28,727	2 / 6.4 /17	Random Graph

were very poor when compared with other methods, so we do not include their results in this paper.

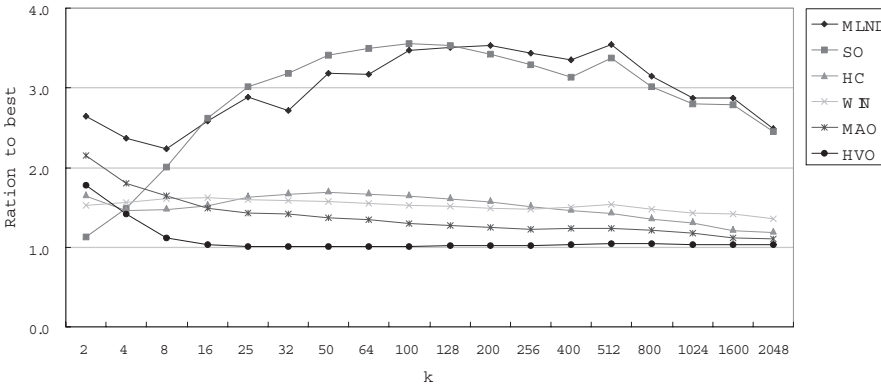
Vertex orders obtained by HVO and other methods are applied to k-way graph partitioning using DPRP [2]. k-way graph partitioning has been performed by changing k over a wide range. In our experiments the size constraints of clusters, U and L, are selected to permit a load imbalance upto 10%. The input graphs used in our experiments are described in Table 1. We used three classes of input graphs from Finite Element Discretizations [8], VLSI design [11], and randomly generated graphs.

All the experiments have been conducted on a SGI workstation with a MIPS R12000 processor. We tested the matching policies, MWM and RWM, as described in Section 3.1. But we adopted RWM as the matching policy in the following experiments because MWM has given worse results for all the inputs.

Figure 5 compares the DPRP costs on the vertex orders obtained by various methods. All the values in Figure 5 represent the normalized values, that is, each cost is divided by the best cost. Therefore, if the normalized value is 1.0, the ordering method and DPRP achieve the best clustering. The experiments have been performed with various k values between 2 and 2048 on each input graph. Figure 5 (a) presents the average DPRP cost for all the input graphs as k varies. Observe that HVO outperformed other methods for almost all k values, except



(a) DPRP costs per k



(b) DPRP costs per graph

Fig. 5. Comparison of HVO with other methods

2 and 4 of k. Figure 5 (b) presents the average DPRP cost for all k values for different input graphs. Except for CCC8 and G3, HVO is a clear winner and the gaps are not so large even on CCC8 and G3. Table 2 presents the execution times for constructing vertex order by each method. HVO runs faster than other methods exceept MLND.

5 Conclusion

In this paper, we have presented the hierarchical vertex ordering (HVO) method and evaluated it by applying the obtained vertex orders to k-way graph partitioning. HVO constructs a tree by merging two vertices level-by-level and traverse the nodes in the tree top-down to obtain vertex orders. The experimental results showed that vertex order obtained by HVO is superior to those by existing ordering methods in terms of clustering objective. For a future study, we plan to apply HVO to a practical problem like the mapping of a graph dataset on a linear disk-space.

Table 2. Execution times in constructing vertex orders (seconds)

Graphs	HC	MAO	WIN	SO	MLND	**HVO**
CCC8	5.620	0.260	0.670	0.296	0.110	0.100
Airfoil	2.100	0.390	3.315	0.818	0.230	0.260
3ELT	28.840	0.460	4.095	1.032	0.250	0.300
FFT9	45.850	0.650	4.075	0.627	0.280	0.280
CCA10	66.310	1.730	14.885	2.442	0.080	0.470
BFLY10	136.530	1.810	17.830	1.764	0.780	0.720
C4Y	3.880	0.140	0.300	0.141	0.060	0.080
C5Y	2.320	0.130	0.230	0.091	0.050	0.070
G1	15.800	1.300	15.015	5.240	0.710	0.860
G2	65.900	1.310	12.305	4.090	0.570	0.600
G3	47.000	1.150	16.860	6.319	0.600	0.830

References

1. Alpert, C., Kahng, A.: A general framework for vertex orderings, with applications to netlist clustering. IEEE Trans. Very Large Scale Integrations Systems. **4(2)** (1996)
2. Alpert, C., Kahng, A.: Multiway partitioning via geometric embeddings, orderings, and dynamic programming. IEEE Trans. Computer-Aided Design of Integrated Circuits and Systems. **14(11)** (1995)
3. Banerjee, J., Kim, W., Kim, S., Garza, J.: Clustering a DAG for CAD databases. IEEE Trans. Software Engineering. **14(11)** (1988) 1684-1699
4. Garey, M., Johnson, D.: Computers and intractability : A guide to the theory of NP-completeness. Freeman and Company (1979)
5. Hendrickson B., Leland R.: The Chaco user's guide Version 2.0. Tech. Rep. Sandia National Laboratories (1995)
6. Juvan, M., Mohar, B.: Optimal linear labelings and eigenvalues of graphs. Discrete Applied Mathematics. **36** (1992) 153-168
7. Kaddoura, M., Ou, C., Ranka, S.: Partitioning unstructured computational graphs for nonuniform and adaptive environments. IEEE Parallel and Distributd Technology. **3(3)** (1995) 63-69
8. Karypis, G., Kumar, V.: METIS, a software package for partitioning graphs. Available on WWW at URL: http://www.cs.umn.edu/ karypis/metis/.
9. Karypis, G., Kumar, V.: A fast and high quality scheme for partitioning irregular graphs. SIAM Journal on Scientific Computing. **20(1)** (1999) 359–392
10. Nagamochi, H., Ibaraki, T.: Computing edge-connectivity in multi-graphs and capacitated graphs. SIAM Journal on Discerete Mathematics. **5(1)** (1992) 54-66
11. Petit, J.: Approximation heuristics and benchmarkings for the MINLA problem. Algorithms and Experiments. (1998) 112–128
12. Riess, B., Doll, K., Johannes, F.: Partitioning very large circuits using analytical placement techniques. Proc. ACM/IEEE Design Automation Conf. (1994) 645-651
13. Shekhar, S., Liu, D.: CCAM: A connectivity-clustered access method for networks and network computations. IEEE Trans. Knowledge and Data Engineering. **9(1)** (1997)

Tutorial Introduction to Graph Transformation: A Software Engineering Perspective

Luciano Baresi[1] and Reiko Heckel[2]

[1] Politecnico di Milano, Italy
baresi@elet.polimi.it
[2] University of Paderborn, Germany
reiko@upb.de

Abstract. We give an introduction to graph transformation, not only for researchers in software engineering, but based on applications of graph transformation in this domain. In particular, we demonstrate the use of graph transformation to model object- and component-based systems and to specify syntax and semantics of diagram languages. Along the way we introduce the basic concepts, discuss different approaches, and mention relevant theory and tools.

1 Introduction

Graphs and diagrams are a very useful means to describe complex structures and systems and to model concepts and ideas in a direct and intuitive way. In particular, they provide a simple and powerful approach to a variety of problems that are typical to software engineering [41]. For example, bubbles and arrows are often the first means to reason on a new project, but also the structure of an object-oriented system or the execution flow of a program can be seen as a *graph*.

The artefacts produced in order to conceptualize a system are called *models*, and *diagrams* are used to visualize their complex structures in a natural and intuitive way. In fact, for almost each activity of the software process, a variety of visual diagram notations has been proposed. We can mention, for example, state diagrams, UML, Structured Analysis, control flow graphs, architectural languages, function block diagrams, and several others. Besides having plenty of general-purpose notations, we should also take into account the many domain-specific customizations that provide both dedicated notation elements and special-purpose interpretations.

If graphs define the structure of these models, graph transformation can be exploited to specify both how they should be built and how they can evolve. Although applications of graphs and graph transformations abound and established foundations are available, the knowledge of the formal and conceptual underpinnings is not widely spread among software engineers. Frequently, this leads to ad-hoc solutions to problems that are already well understood in a more general context.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 402–429, 2002.
© Springer-Verlag Berlin Heidelberg 2002

The definition and implementation of visual modeling techniques poses new problems, when compared to programming or formal specification languages: As a first problem, most established techniques for language definition, like the denotational [88] or the operational [77] approach are intrinsically based on terms (abstract syntax trees) as representation of the structure of the language. This results from the use of context-free grammars (e.g., in Backus-Naur form [3]) for defining their textual syntax. However, as most diagram languages have a *graph-like structure*, such techniques are not readily applicable.

A second problem is the number and diversity of modeling languages and dialects that are currently in use, and the rate in which new ones are proposed to fit the particular needs of certain problem areas. To provide language definitions and implementations for these notations within reasonable resource constraints, meta-level techniques and tools are required which allow, e.g., to generate implementations of languages, like model editors, compilers, or analysis tools, from high-level specifications.

The third problem is the consistency of models consisting of interrelated submodels for different aspects and at different levels of abstractions [38]. Generally speaking, an inconsistent model does not have any correct implementation because the requirements expressed by different submodels are in conflict. To deal with such problems in a systematic way, it is essential to understand the relationships of submodels at a semantic level. Besides techniques and tools to define semantics, this requires semantic domains which are able to capture the essentials of today's systems including aspects like object-orientation, software architecture, concurrency, distribution, mobility, etc. Also here, a general tendency is the shift from hierarchical, tree-like to graph-like structures which are, moreover, dynamic to reflect, for example, the behavior of mobile systems or architectural reconfiguration.

In this tutorial paper, we demonstrate how graph transformation techniques can contribute to solve the above problems. We start (Section 2) with an informal introduction to the basic concepts of graph transformation (like graph, rule, transformation, etc.), we discuss semantic choices (like which notion of graph to use; how to put labels, attributes, or types; or what to do with dangling links during rewriting, etc.) and mention the different ways for formalizing the basic concepts. After introducing the theory, we exemplify (Section 3) what can be done through some examples. To this end, we distinguish between the use of graph transformation as:

- *semantic domain* to supply a specification language and semantic model for reasoning on particular problems: for example, the consistency between functional requirements and software architecture models in concurrent and distributed systems.
- *meta language* to supply a means to formally specify the syntax, semantics, and manipulation rules of visual diagrammatic languages.

We then continue (Section 4) with the review of the main branches of the theory of graph transformation relevant to the applications discussed. The last

step, to let possible users really exploit graph transformation to reason on and solve their problems, is a brief summary (Section 5) of available tools to clearly explain the support offered to (fully) automate proposed and foreseen solutions. We conclude the paper (Section 6) by identifying future research directions with the hope that the graph transformation community can more and more serve as technology provider to other communities that need this formal basis to improve their current practice.

2 Foundations of Graph Transformation

This section is intended as a first introduction to graph transformation, its basic notions, different approaches, and more advances concepts. Graph transformation has evolved in reaction to shortcomings in expressiveness of classical approaches to rewriting, like Chomsky grammars and term rewriting, to deal with non-linear structures. The first proposals appeared in the late sixties and early seventies [76,70,83,79,95]. They were concerned with rule-based image recognition, translation of diagram languages, or efficient implementation of λ-reduction, based on graph-like structures.

Fundamental approaches that are still popular today include the algebraic or double-pushout (DPO) approach [31,17], the node-label controlled (NLC) [57,32], the monadic second-order (MSO) logic of graphs [19,20], and the PROGRES approach [86,87] which represents the first major application of graph transformation to software engineering [33,73].

Below we introduce a simple form of graph transformation which shall serve as a basis for further discussion, i.e., a set-theoretic presentation of the double-pushout approach [31]. Then, we will discuss alternatives and extensions to this approach.

2.1 A Basic Formalism

Graphs. A *graph* consists of a set of vertices V and a set of edges E such that each edge e in E has a source and a target vertex $s(e)$ and $t(e)$ in V, respectively. In object-oriented modeling graphs occur at two levels: the type level (given by a class diagram) and the instance level (given by all valid object diagrams). This idea can be described more generally by the concept of *typed graphs* [16], where a fixed *type graph* TG serves as abstract representation of the class diagram. Its object diagrams are graphs equipped with a structure-preserving mapping to the type graph, formally expressed as a *graph homomorphism*.

Figure 1 shows examples of an object and a class diagram in UML notation [75] modeling some data objects of a banking application. The (instance graph representing the) object diagram on the left can be mapped to the (type graph representing the) class diagram by defining $type(o) = C$ for each instance $o : C$ in the diagram. Extending this to links, preservation of structure means that, for example, a link between objects o_1 and o_2 must be mapped to an association in the class diagram between $type(o_1)$ and $type(o_2)$. By the same

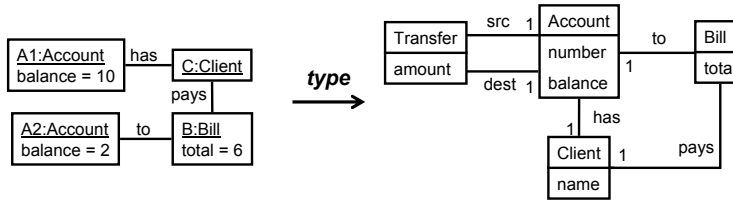


Fig. 1. Object diagram (left) typed over class diagram (right)

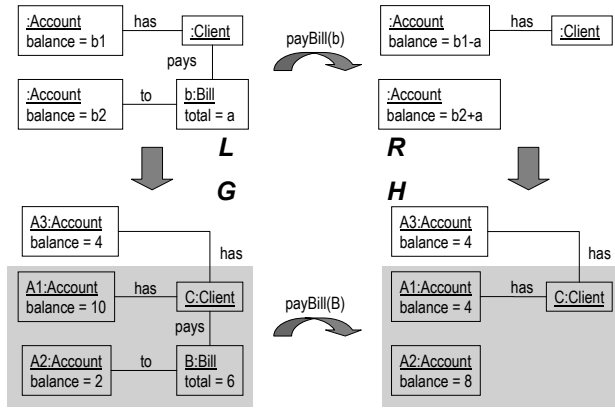


Fig. 2. A sample transformation step using rule payBill

mechanism of structural compatibility we ensure that an attribute of an object is declared in the corresponding class, etc.

Rules and Transformations. A *graph transformation rule* $p : L \to R$ consists of a pair of TG-typed instance graphs L, R such that the union $L \cup R$ is defined. (This means that, e.g., edges that appear in both L and R are connected to the same vertices in both graphs, or that vertices with the same name have to have the same type, etc.) The left-hand side L represents the pre-conditions of the rule while the right-hand side R describes the post-conditions.

A *graph transformation* from a pre-state G to a post-state H, denoted by $G \overset{p(o)}{\Longrightarrow} H$, is given by a graph homomorphism $o : L \cup R \to G \cup H$, called *occurrence*, such that

- $o(L) \subseteq G$ and $o(R) \subseteq H$, i.e., the left-hand side of the rule is embedded into the pre-state and the right-hand side into the post-state, and
- $o(L \setminus R) = G \setminus H$ and $o(R \setminus L) = H \setminus G$, i.e., precisely that part of G is deleted which is matched by elements of L not belonging to R and, symmetrically, that part of H is added which is matched by elements new in R.

Figure 2 shows an application of the graph transformation rule payBill modeling the payment of a bill by transferring the required amount from the account

of the client. Operationally, the application is performed in three steps. First, find an occurrence $o|_L$ of the left-hand side L in the current object graph G. Second, remove all the vertices and edges from G which are matched by $L \setminus R$. Make sure that the remaining structure $D := G \setminus o(L \setminus R)$ is still a legal graph, i.e., that no edges are left dangling because of the deletion of their source or target vertices. This is made sure by the *dangling condition* [31][1] which is checked for a given occurrence before the application of the rule. If the condition is violated, the application is prohibited. Third, glue D with $R \setminus L$ to obtain the derived graph H. In Fig. 2 the occurrence of the rule is designated by the shaded objects and links in the transformation.

2.2 Variants and Extensions

Graphs. The graphs introduced above are often referred to as *multi-graphs* because they allow for multiple parallel edges of the same type. An (untyped) multi-graph can be regarded as a two-sorted algebraic structure. Therefore, formalisms dealing with such graphs are known as *algebraic approaches* [17,28].

The more common alternative is to consider graphs as *relational structures* $G = (V, E)$ with $E \subseteq V \times V$. In this case, there exists at most one edge between a given pair of vertices, which represents a restriction of the algebraic notion. Other variations include, for example, *undirected graphs* which are formalized as directed graphs closed under symmetry of edges, or hypergraphs, where each edge may have a sequence of source and/or target vertices. *Hypergraphs* are sometimes encoded as bipartite graphs.

Typed graphs as above combine the association of labels to nodes and edges with structural constraints expressed by the graph structure on the label sets. In *labelled graphs*, the label sets do not have any additional structure. For example, if vertices and edges are labelled over separate label alphabets L_V, L_E, the relational variant is given by (V, E, lv) with $E \subseteq V \times L_E \times V$ and $lv : V \to L_V$. Attributed graphs are graphs labelled over pre-defined abstract data types, like strings or natural numbers [65]. An example is given in Fig. 1 above where, e.g., the vertex B:Bill has an attribute total = 6.

It is interesting to note that many of the notions and constructions in the theory of graph transformation can be (and have been) described for a wide range of the above structures. Formally, this is reflected in approaches based on category theory like [27,30,10], which can be instantiated to a variety of different graph models. In the following we stick to our simple model of typed and attributed graphs.

Rules and Transformations. Co-related with the two notions of graphs, the algebraic and the relational one, are two fundamentally different approaches to

[1] Indeed, the notion of graph transformation introduced here is a set-theoretic presentation of the categorical double-pushout (DPO) approach [31], which owes its name to the fact that a transformation step may be characterized as a pair of gluing diagrams (pushouts) of graphs.

rewriting which have been referred to as the *gluing* and the *connecting* approach. They differ for the mechanism used to embed the right-hand side R of the rule in the context D (i.e., the structure left over from G after deletion). In a gluing approach like [31,64], graph H is formed by gluing R with the context along common vertices. In a connecting approach like NLC [57], the embedding is realized by a disjoint union, with additional edges connecting graph R with the context. To establish these connections, *embedding rules* are specified as part of the transformation rule to determine the connections of the right-hand side from those of the left-hand side in the given graph.

Most practical approaches use combinations of gluing and connection [86,39]. The first is more efficient in implementations since it allows items in the left-hand side to be preserved by the rule. Otherwise, this option, which is obviously desirable, has to be simulated by deletion and re-generation of the items concerned. On the other hand, the connection approach based on embedding rules is more expressive because it allows to deal with unknown context. In fact, operations like "turning all outgoing edges of a certain vertex into ingoing ones" can only be specified in a gluing approach if the number of such edges is known in advance or specific control structures are used. Instead, such operations are typical to connecting approaches like node-label controlled (NLC) graph grammars [57]. A well-known example of the limitation of gluing approaches has been discussed above: Deletion of a vertex is prohibited by the dangling condition if an edge in the context is attached to it. In the algebraic approach, this problem has been solved in [64] by implicitly deleting all edges whose source or target vertex is deleted.

The most common restriction imposed on graph transformation rules is *context-freeness* in the context of graph grammars: A rule is context-free if it has only one vertex or edge in its left-hand side. Similarly to context-free Chomsky grammars, context-free graph grammars are interesting because of their simplicity, combined with sufficient expressive power to describe many interesting graph languages [22,32].

Control Structures and Constraints. Besides approaches that increase (or restrict) the expressiveness of individual rules, a major concern of a rule-based approach is to control the application of rules. This includes programming language-like control structures [85,61] and application conditions [90,26,45] restricting the admissible occurrences for a given rule.

Moreover, different kinds of constraints have been proposed, which restrict the class of graphs and can therefore be used to control the transformation process implicitly by ruling out transformations leaving this class. In their simplest form, such constraints can be compared to cardinality restrictions in class diagrams like in Fig. 1 where, e.g., the association to connects every Bill object to exactly one Account object. More complex constraints deal with the (non-) existence of certain patterns, including paths, cycles, etc. Constraints on graphs are formalized in terms of first- or higher-order logic [85,20], by means of linear inequations [40], or as graphical constraints [51,11].

3 Modeling and Meta-modeling

As evident from the title, we distinguish between the application of graph transformation to the modeling of individual systems, and to the specification of syntax and semantics of visual modeling languages. However, in both cases, similar notations and tools may be employed, and we refer to the latter activity as meta modeling in order to stress the fact that a language may be defined by the same techniques that are also used to model a system.

3.1 Modeling with Graph Transformation

This section illustrates the use of graph transformation for modeling functional requirements and dynamic change of software architectures. Then, these two aspects of a model are related by means of a common meta model. Along the way, some more advanced concepts and constructions based on graph transformation systems are introduced.

Object Dynamics. Functional requirements are often presented in terms of use cases, i.e., services a system shall provide to its users. Every such use case provides a more detailed description of its pre- and post-conditions and of the interactions required to perform the respective service. In [47] typed graph transformation systems have been proposed as a way to specify use cases in a visual, yet formal way with the additional benefit of an executable specification. An example for such a specification has been given in Fig. 2 with the rule payBill and its application.

The idea of a rule-based modeling of operations by means of graphical pre- and post-conditions can be traced back to several sources. PROGRES [87] provides a database-oriented programming language and environment based on graph transformation. In the object-oriented FUSION method [14] actions are specified by snapshots of the object configuration before and after the operation. CATALYSIS [23] advocates the use of UML collaboration diagrams for this purpose, an approach which has also been adopted and implemented in the FUJABA method and tool [59]. A formal relation between collaboration diagrams and graph transformation has been established in [50].

Graph transformation rules like payBill provide a high-level specification of functional requirements in terms of pre and post conditions, abstracting from intermediate actions and states. If the more fine-grained structure of an operation is of interest, a rule may be decomposed into more elementary steps. For example, the rule payBill(b) may be decomposed sequentially as payBill(b) = createTransfer(b,t); executeTransfer(t), where the relation between the two elementary rules is specified by the abstract parameters b: Bill and t: Transfer. This decomposition is shown in Fig. 3 where createTransfer(b,t) is applied to the left-hand side of payBill(b) and executeTransfer(t) is applied to the resulting graph, the result being the right-hand side of the original rule.

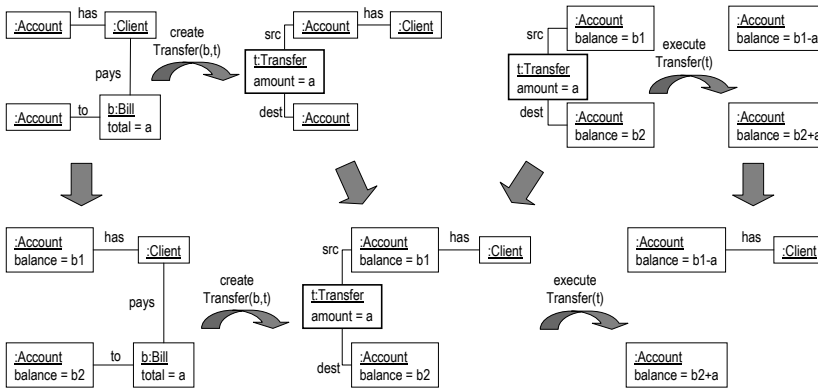


Fig. 3. Rule payBill(b) derived from createTransfer(b,t); executeTransfer(t)

The (obvious) semantic consistency condition for this kind of decomposition requires that, for any given graph G, there exists a transformation

$$G \overset{\mathsf{payBill(B)}}{\Longrightarrow} H$$

if and only if there are a graph X and transformations

$$G \overset{\mathsf{createTransfer(B,T)}}{\Longrightarrow} X \overset{\mathsf{executeTransfer(T)}}{\Longrightarrow} H.$$

Thus, we may think of the composed rule payBill as a two-step transaction. The desired semantic condition can be checked by composing the elementary rules as it is shown in Fig. 3. First, createTransfer(b,t) is applied to the left-hand side of payBill(b) at the occurrence determined by the parameter b: Bill. Then, executeTransfer(t) is applied to the resulting graph, this time determining the occurrence through the parameter t:Transfer. The result of this second step has to be isomorphic to the right-hand side of the original rule. In this case, it can be shown that the desired consistency condition holds [17].

Architectural Change. A quite different interpretation of nodes and edges in a graph is in terms of components and connectors of a software architecture model, see e.g., [89,60]. Here, instance and type graphs model, respectively, configurations and architectural styles, while graph transformation rules specify the reconfiguration of architectures.

An example of a type graph representing an architectural style for distributed banking applications is given in Fig. 4 on the left. The style consists of components like electronic cashboxes, smartcards, and banking servers—all communicating through the Internet, telephone lines, or card readers. A sample configuration of this style is shown in the lower left of Fig. 4. It consists of two instances of the banking server component, and one instance each of cashbox and smart card, where only the two banking servers are currently connected.

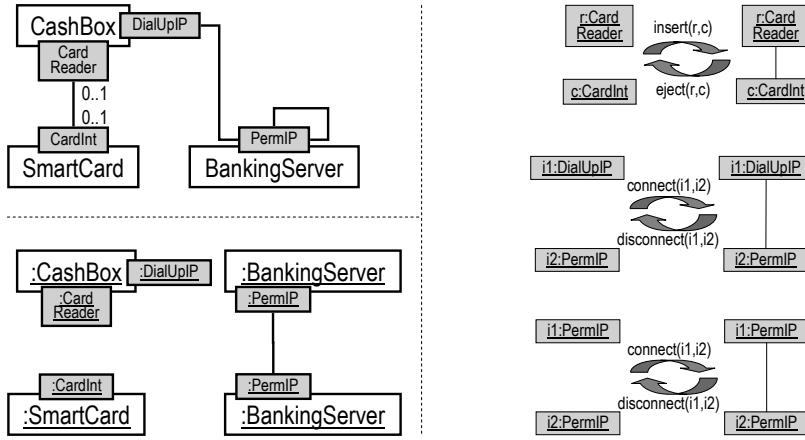


Fig. 4. Architecture of a distributed banking system: a configuration (lower left) typed over an architectural style (upper left) and reconfiguration rules (right).

Beside this declarative approach to the specification of architectural styles, constructive approaches based on graph grammars have been proposed. Here the idea is to generate the set of all eligible configurations from a given initial one by means of (often context-free) production rules [63,52]. Context-freeness, i.e., the restriction to one component (vertex or (hyper)-edge) in the left-hand side of rules, corresponds to the recursive refinement of components by configurations of sub-components.

The main benefit of graph transformation for describing software architectures is the ability to model dynamic reconfigurations in an abstract and visual way. Some approaches [42,63,96,93] assume a global point of view when describing reconfiguration steps which, in a real system, would correspond to the perspective of a centralized configuration management. In a distributed system, the existence of such centralized services cannot be taken for granted. Therefore, [53,52] model reconfiguration from the point of view of individual components which synchronize to achieve non-local effects. Here, locality corresponds to context-freeness, that is, a rule is local if it accesses only one component (or connector) and their immediate neighborhood. Synchronization of rules is expressed in the style of process calculi like CCS [67] or CSP [54], see [53] for a more detailed discussion.

The rules in Fig. 4 do not require any synchronization. In the upper left, the transformation from left to right establishes a connection between the interface of a smart card and a card reader, modeling the insertion of the card into the reader. The inverse transformation models the disconnection (ejection) of the card from the reader. In a similar way, the two lower rules model the establishment and release of Internet connections.

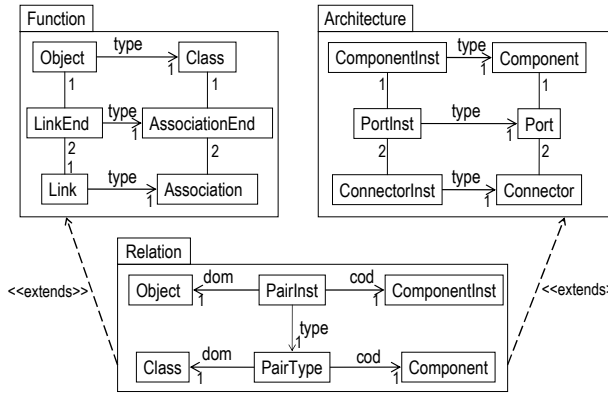


Fig. 5. Meta model relating the functional and architectural view

Relating Object Dynamics and Architectural Change. In the previous section, we have shown how to model both functional requirements and architectures by separate graph transformation systems. This section is devoted to relate these two views both statically by relating classes to components and dynamically by interleaving computations on objects with reconfiguration steps and communication between components.

The static/syntactic integration is achieved by defining the abstract syntax of both the functional and the architectural view by means of a *meta model*. Generally speaking, a meta model is a model for a modeling language expressed within a simple subset of the language itself. This technique has become popular with the UML whose abstract syntax is specified by a subset of UML class diagrams [75]. This subset is determined by the meta object facility (MOF) specification [74] which is also referred to as a meta-meta model because it has meta models as instances whose instances, in turn, are models.

Formally, meta models are type graphs whose instance graphs represent models. That means, the type-instance mapping of typed graphs, which has so far been used to model the relation of objects to their classes and component instances to their components, shall now be reserved for the mapping between a model and its meta model. Therefore, the object-class and component instance-component mappings are defined in the meta model itself.

Consider, for example, the meta model in Fig. 5. It consists of three packages, whose top left one specifies the functional view of the system by means of meta classes Class and Association as well as Object and Link, etc. whose relation is given by the meta association type. Therefore, every instance of this meta model represents a pair of a class diagram and an associated object diagram.

A similar structure is present in the package for the architectural view whose instances represent both a declarative definition of an architectural style (meta classes Component, Connector, and Port with their meta associations) as well as an individual configuration (meta classes ComponentInst, ConnectorInst, and

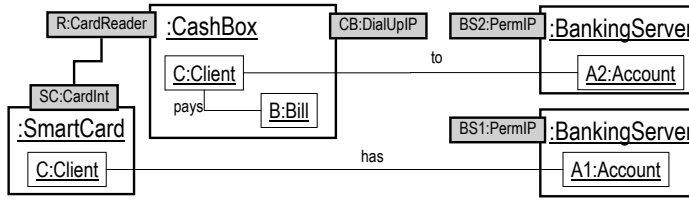


Fig. 6. Sample configuration after transmission of client data

PortInst with their meta associations) and their interrelation by the meta association type.

The meta model allows a uniform representation where elements of different submodels are represented as vertices of the same *abstract syntax graph* [2], i.e., an instance of the meta model. Based on this uniform representation, the different submodels can be related by extending the meta model in Fig. 5 with a package extending the other two packages to define a relation between the functional and the architectural view [58]. Such a relation consists of pairs of, respectively, objects and component instances or classes and components, where instance-level pairs are associated to type-level pairs by a meta association type: To relate an object to a component instance, a corresponding relation between the respective class and component is required.

In our example, the relation between classes and components shall be given by { (SmartCard, Client), (CashBox,Transfer), (CashBox,Bill), (CashBox,Client), (BankingServer,Account), (BankingServer,Transfer) }. This information could be given in diagrammatic form, like in UML component diagrams where the relation is expressed by containment or dependencies. Here we simply list it as a set of pairs. A diagrammatic presentation of the object-component instance relation is given in Fig. 6 by means of containment, combining information from all three meta model packages.

Based on the uniform representation of object structures and architectures as meta model instances we may now present the rules specifying functional requirements and architectural reconfigurations as graph transformation rules typed over the corresponding packages of the meta model. However, this is nothing more than a disjoint union of models, turning two distinct models into yet unrelated submodels of the same overall model. To map the functional requirements on a given architecture, we assign responsibilities for operations to components (or sets of components) by specifying the location of the objects in the corresponding rules. Figure 7 shows this step for the rule createTransfer which is associated with the CashBox component.

An example of a communication rule is shown in the bottom of Fig. 7. It models the transmission of a message orderTransfer(t) from a cashbox holding an object t:Transfer to the banking server holding the corresponding source account. The effect of this transmission is the replication of the object. We denote by using the same object identity t that the object is logically shared between the

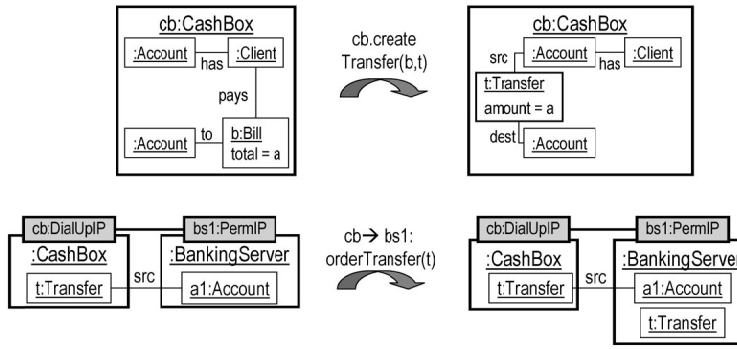


Fig. 7. Operation createTransfer executed on a CashBox component (top) and the effect of transmitting a Transfer object (bottom)

two components, that is, the banking server has access to all references and attributes.

Related Work. The integration of functional and architectural models is implicit in many works on graph transformation for specifying software architectures. In [93] an approach based on two-layered distributed graphs [91,92] is presented. The upper layer represents the network graph of a distributed system whose nodes are attributed with object graphs on the lower layer. Two-level rules are then used to manipulate this structure. This approach is conceptually close to ours, but it does not foresee any means for functional decomposition of rules.

Another approach integrating functional and architectural aspects [96] uses the coordination and programming language CommUnity to specify computations and graph transformation rules to model architectural reconfiguration. The use of a programming notation is appropriate at the later design or implementation stages, but for requirements specification and analysis we prefer visual notations.

Recently, the transformation of hierarchical graphs, of which our approach presents a special case, has received much attention, see e.g. [80,36,21,13,68]. In particular [13,68] focus on the separation of *connection* from *hierarchy*. That is, hierarchical graphs can be seen as graphs with two kinds of structures, for expressing links between objects or components, and for modeling hierarchy.

3.2 Meta-modeling with Graph Transformation

In Section 3.1, a meta model has been used to specify the abstract syntax of static diagrams, like class and object diagrams, and graph transformation rules were employed to model, e.g., computations on object structures. More generally, meta modeling techniques may be used to define the concrete syntax, abstract syntax, or semantics of any modeling notation, be it dedicated to static or dynamic aspects [58]. The key idea is that these structures can be specified by means of class diagrams. Then, the crucial question is, how to relate the three levels in

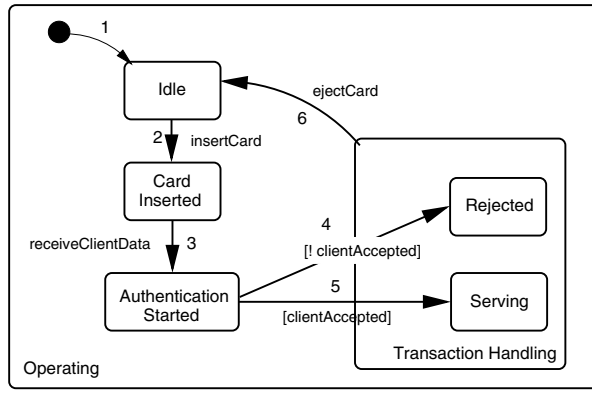


Fig. 8. The CardReader view of the CashBox behavior as a protocol statechart

order to define syntax and semantics of a language. In this section, we discuss the use of graph transformation systems to specify mappings between concrete and abstract syntax and between abstract syntax and semantics, as well as for defining operational semantics based on a direct interpretation of models at the level of abstract syntax.

To make the discussion more concrete, we use a simple example of a protocol statechart[2] that models the behavior of the CashBox component as observed through the CardReader interface, both introduced in Section 3.1. If the CashBox is switched on, from the initial state (the black bullet) we enter state Operating, moving to its default state Idle. As soon as the first card is inserted (i.e., event insertCard occurs), the component enters the state Card Inserted and when it receives the client data from the card (event receiveClientData), it moves to state Authentication Started. Here, a choice occurs depending on the data received. If the client is accepted, the component enters the state Serving otherwise the state Rejected. In both cases – this is why we have the OR-state Transaction Handling comprising the two substates – after processing the transaction, the component returns to state Idle as soon as the card is ejected (event cardEjected).

This diagram, together with its intuitive semantic interpretation is enough to touch all aspects that define a visual notation. More precisely, we can organize it around the three different layers of language definition as shown in Fig. 9. In the remainder of this section, we describe in some detail the transformations on and between these layers represented by the arrows in Fig. 9.

Scanning and Parsing. Moving top-down, each language has a *concrete syntax*, which defines how users perceive the different modeling elements supplied by the notation. Since we are thinking of diagrammatic languages, the concrete syntax predicates in terms of bubbles, rectangles, lines, arrows, etc. Each notation element is concretely rendered in terms of the geometrical elements that define

[2] Readers are referred to [46] for an in-depth presentation of the notation.

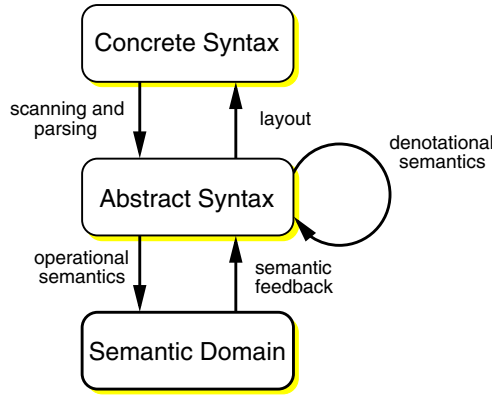


Fig. 9. A layered view of a visual modeling language

its appearance: For example in Fig. 8, states are represented through rectangles with rounded corners, initial states with back bubbles, and state transitions with directed edges. Relationships among these elements can be: a line *connects* two rectangles, a rectangle *contains* other rectangles, or an element *is on the left/right* of another element.

At this level, a graph grammar defines the concrete syntax of the language. It is this grammar that should be employed to *scan* user models. (Notice that we do not yet assign any semantic interpretation to the graphical symbols.) From a practical point of view, the grammar could define also the correct steps that can be done by a user that uses a syntax-directed editor for the supported notation. For example, Minas in [55] proposes a complete hyperedge grammar for *editing* well-formed statechart diagrams, to be fed into the DiaGen tool (see Section 5) for automatically generating a graphical statechart editor. However, the grammar can also be used as a stand-alone definition of the concrete syntax of statecharts.

Formally, the statechart grammar defines all correct Spatial Relationship Hypergraphs (SRHG), that is, hypergraphs with edges like *label*, *rectangle*, *edge*, etc. and nodes representing the points where the hyperedges are connected. SRHGs through a further set of transformation rules become Reduced Hypergraph Model (HGM). These graphs represent the abstract syntax of the example statechart, i.e., the next layer in our hierarchy of Fig. 9. The *abstract syntax* defines the modeling elements supplied by the notation, without the concrete "sugar", and the relationships among them. Models at this level can be *parsed* to check if they are syntactically correct. Tokens at this level are related to the semantic interpretation, that is, we think of the example of Fig. 8 in terms of states (initial, AND-decomposed, and OR-decomposed) transitions, events, and so on.

This level is comparable to the representation of UML models as instances of the meta model [75]. The main difference is the declarative style of specification in the meta model, which defines well-formedness by means of logic constraints, as opposed to the constructive style of using a grammar to generate models.

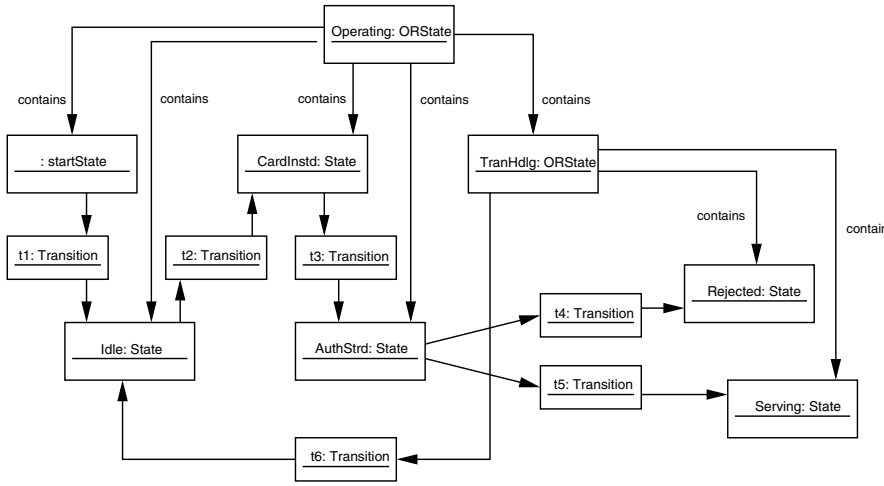


Fig. 10. Abstract syntax graph for the statecharts of Fig. 8

Figure 10 shows a simplified abstract syntax graph for the statechart of Fig. 8. Nodes are instances of a simple metamodel that comprises: startStates, ORStates, States, and Transitions (further details are omitted for the sake of clarity). Edges connect the nodes to render the connections between states and transitions in the statechart diagram. This simple model could easily be constructed through a special-purpose grammar by scanning user-supplied diagrams. In turn, these abstract syntax graphs can be parsed to check if they are syntactically well-formed.

Vice versa, mapping abstract to concrete syntax means that we define the concrete *layout* of models. The grammar in this case defines how abstract concepts should be rendered at the concrete level, but also the correct positioning of each element on the canvas. Special-purpose algorithm for defining the layout of user models can be implemented using graph grammar productions and textual attributes to compute the coordinates of each graphical symbol.

Operational Semantics. Visual models can be given a formal semantics in several different ways. In the case of a class diagram the semantics is given by a set of object diagrams. For behavioral diagrams like statecharts, an operational semantics can be given directly on the abstract syntax of the language, through yet another grammar. In our example of Fig. 8, a formal semantics is desirable to specify precisely the execution of a model. If we think of performing a simple transition, the meaning is clear enough: It moves the *current state* for the source to the target state. But when we think of dealing with multiple events, AND- and OR-decomposed states, histories, etc., then the meaning is not clear anymore. In fact, there are several different statechart dialects, all using essentially the same syntax, but with different rules to execute a model [94].

There are essentially two different ways to define operational semantics by graph transformations. First, graph transformation rules can specify an abstract interpreter for the entire language as proposed, for example, in [34]. Second, each model can be "compiled" into a set of rules. Following the second approach, [62] exploits *structured graph transformation* as means to ascribe UML statecharts with formal dynamic semantics. Roughly, statecharts are translated to structured graph transformation systems that satisfy the well-formedness rules imposed by the notation. Active states are represented as *state configurations* which are (isomorphic to) subgraphs of state hierarchies, that is, tree-like object diagrams that represent instances of the metamodel for the statecharts. Active states change by means of transition firing, specified as graph transformation rules: the left-hand side identifies the current configuration, i.e., the current active state(s), the right-hand side defines the configuration reached by applying the rule (firing the transition), that is, the new set of active states. Transitions for *syntactically correct* statecharts can be generated by applying the rules of the transformation unit *term(S)* presented in [62]: Figure 11 shows the result for some of the transitions of Fig. 11.

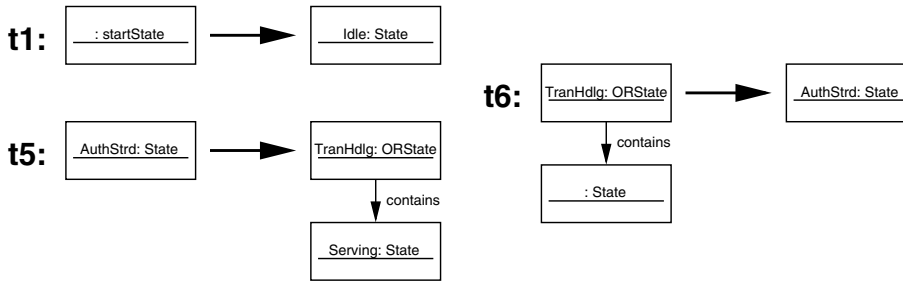


Fig. 11. Some transitions of Fig. 8 as graph transformation rules

Presented rules are self-explaining, but: $t1$ moves the current state from the start state to Idle; $t2$ and $t3$ are similar and are not presented here. $t5$ moves the current state from Authentication Started to the hierarchy Transaction Handling / Serving. Again $t4$ would be similar. $t6$ moves the current state from the hierarchy Transaction Handling / any contained state back to Idle.

Denotational Semantics. The last layer of Fig. 9 introduces semantic domains. Semantics defined as a mapping from the abstract syntax into a semantic domain is referred to as *denotational*. In our example, to define the dynamic semantics of statecharts, the semantic domain itself has to provide an operational model—thus operational and denotational semantics occur in combination.

The overall approach is as follows: We choose a *semantic domain*, i.e., a usually simpler formal method whose execution rules are well-established, and define the "behavior" of each abstract syntax element through a suitable *mapping* onto

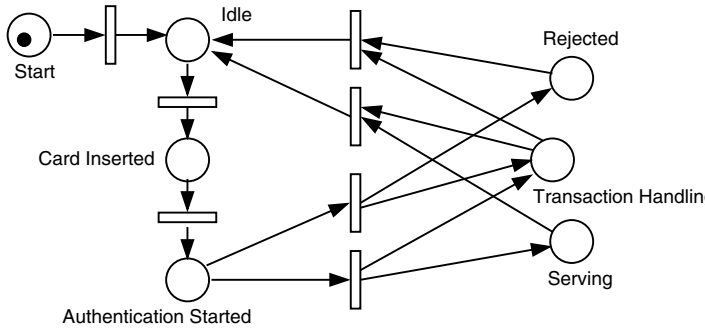


Fig. 12. 8 in terms of Petri nets

the semantic domain. In this case, the role played by graph transformation depends on the chosen formal method. If it is a textual one, the productions of the grammar that defines the abstract syntax can be augmented with textual annotations to build the semantic representation. More generally, the productions can be paired with those of the textual grammar that specify the semantic models, and the application of a production of the abstract syntax grammar would automatically trigger the application of the paired textual production [79].

For example, if we keep thinking of statecharts and want to define its dynamic semantics through CSP (Communicating Sequential Processes [54]), we can mention [35] where the left-hand side of each rule is how UML-like metamodel instances can be built for statecharts (graph grammar productions); the right-hand side part codes how the corresponding CSP specification must be modified accordingly (textual grammar production). A similar approach is described in [9] where high-level timed Petri nets are used as semantic domain and the rules are pairs of graph grammar productions: The first production defines how to modify the abstract syntax representation, while the second production states the corresponding changes on the functionally equivalent Petri nets.

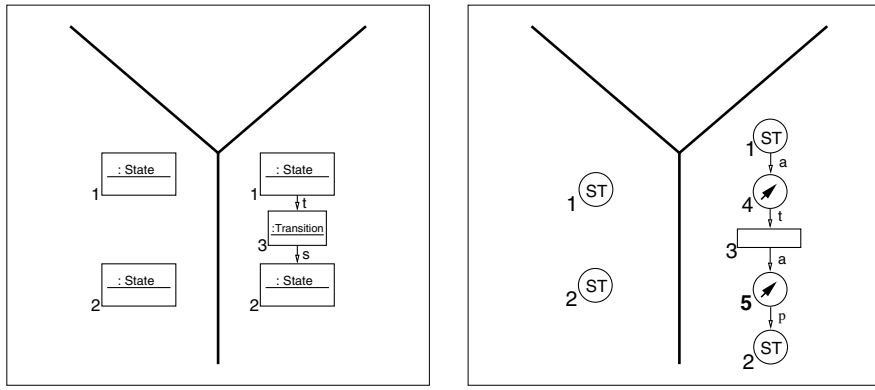
For example, using the *transformation rules* defined in [9], we could ascribe formal dynamic semantics to the statechart of Fig. 8 through the Petri net of Fig. 12[3].

The hypothesis used to transform the statechart into a functionally equivalent Petri net are: States are directly mapped onto places and state transitions into Petri net transitions. Start states are rendered with marked places. OR-decomposed places imply that their transitions be mapped onto a set of Petri net transitions.

A simple transformation rule is presented in Fig. 13. It shows how to connect two SC [4] states through an SC transition. The left production modifies the ab-

[3] Attributes associated with places and transitions are not shown here since the are not needed to understand the approach, but they would simply make the Petri net more complex.

[4] SC and PN are used here to clarify if we are referring to statechart or Petri net elements. The event associated with the SC transition is not considered here for the sake of simplicity.



(a) Abstract Syntax Model (b) Semantic Model

Fig. 13. A simple transformation rule taken from [9]

stract syntax graph by adding the new SG transition, while the right production modifies the Petri net by adding a PN transition, together with two PN arcs between the two places that correspond to the SC states.

Notice that even if Petri nets are formal and simpler than statechart, the resulting net is more complex than the example diagram. Thus, such a transformation is important to let users ascribe formal semantics to their models, but usually produces formal representations that are more complex and difficult for the reader. More sophisticated mappings could be implemented using triple graph grammars [84], that is, besides the two grammars that define the abstract syntax and the corresponding modifications of the semantic domain, a third grammar states the mapping between the two paired productions explicitly.

Moving from the semantic domain back to the abstract syntax layer (Figure 9), the results produced by executing the semantic representations can be mapped onto the abstract syntax elements. For example, the firing of a PN transition that corresponds to an SC transition could be transformed into a suitable animation of the interested states. Also these transformations, mapping of execution results, can be specified through a suitable grammar. Interested readers can refer to [9] for a couple of interesting examples.

4 Theory

As mentioned in Section 2, graph transformation systems generalize other notions of rewriting, like Chomsky grammars, Petri nets, and term rewriting. Such connections have inspired the development of the theory of graph languages, concurrency, and term graph rewriting, respectively. In this section we briefly discuss these developments and give some references to relevant literature.

4.1 Graph Languages

Chomsky grammars have inspired the notion of *graph grammar*, i.e., a graph transformation system with a start graph that is meant to describe graph languages. In particular, a theory of context-free graph grammars [22,32] has been developed, as well as parsing algorithms for graph languages [81,69,12] that are relevant to the application of graph grammars to describe the syntax of visual languages [66].

For example, an important (negative) result of graph grammar theory states that, unlike in the case of programming languages, context-free graph grammars are not sufficient to generate even approximations of many interesting diagram languages. As an example, it can be shown that the simple language of statecharts is not generated by any context-free graph grammar[5]. The same holds for other popular diagram languages used in software development, like object or class diagrams. For this reason, approaches to visual language definitions, like [81,69], have to resort to non-context-free grammars.

4.2 Concurrency

Place-transition Petri nets are essentially rewriting systems on multisets. In this sense, they are like graph transformation systems without edges, and without context [15] if we consider a gluing approach. This view has inspired a variety of developments along the lines of the concurrency theory of nets, like processes and unfoldings, event structures, as well as the notion of typed graphs that we have used in Section 3. These and other concepts in concurrency theory are surveyed in [4] for the gluing approach and [56] for the connecting approach.

The related aim of generalizing to graph transformation systems certain notions of morphisms of Petri nets has influenced the development of structuring and refinement concepts for graph transformation systems. For example, the relation between abstract functional requirements as exemplified by the rule payBill in Fig. 2 and the overall system model, including the architectural aspect, can be described as a refinement relation according to [48,44] combining an extension of type graphs and a (sequential) decomposition of rules.

[5] For hyperedge replacement graph grammars, it is sufficient to observe that the graph language of state diagrams has unbounded connectivity. It follows from the Pumping Lemma (Thm. 2.4.5 in [22]) that such a language cannot be generated by hyperedge replacement. For node replacement grammars, observe that the language includes all finite *square grids*, i.e., all graphs whose nodes can be organized into rows and columns of the same length, and connected by horizontal and vertical edges. Prop. 1.4.8 of [32] states that a language containing infinitely many square grids cannot be generated by a confluent edNCE grammar, the most general form of context-free graph grammar known in the literature. In fact, confluence is a crucial feature of context-free grammars. Approaches like [72,45] which increase the expressive power of (otherwise) context-free rules by path expressions or application conditions do not fall into this category.

4.3 Term Graph Rewriting

To support the efficient implementation of functional languages, term graph rewriting generalizes term rewriting by replacing trees by rooted DAGS [7,18,78,8]. This has given rise to theoretical questions concerning confluence and termination of (term) graph rewriting to ensure functional behavior [78]. In view of the previous section, such questions are relevant to the use of graph transformation for translating models into formal specifications in terms of Petri nets or CSP, which serve as semantic domains. In fact, in order to be well-defined, a semantic mapping should be a total function. This is also the main motivation of [49] studying confluence of attributed graph transformation.

More recently, research on graph transformation approaches has been influenced by the use of term rewriting in the semantics of process calculi [71,68]. Here terms representing processes, e.g., in the π or ambient calculus, are replaced by process graphs which allow a more direct representation of their structure.

5 Tools

This section briefly describes the main tools that are available to "work" with graphs and graph transformation. They all are good research prototypes that, according to the taxonomy already described in Section 3, can be divided in two main groups: general-purpose environments for modeling graph-centric problems and environments for specifying visual languages. The main representatives of these classes of tools are introduced in the next two sections: Interested readers can refer to [6] for more complete descriptions.

5.1 Graph Transformation Tools

The first example of graph-oriented modeling environment is PROGRES (*PROgrammed Graph REwriting Systems* [86,87]). Conceptually, it supplies a graphical/textual language to specify attributed graph structures and graph transformations, parameterized graph queries, graph rewrite rules with complex and negative application conditions, and non-deterministic and imperative programming of composite graph transformations (with built-in backtracking). Pragmatically, the PROGRES environment offers a set of integrated tools to let users models their artifacts. Specifications are produced through a mixed textual/graphical syntax-directed editor (with an incrementally working table-driven pretty printer), but they can also be edited in a fully textual way through an emacs-like text editor. Incrementally, a type-checker detects all inconsistencies with respect to the language's static semantics. Besides being checked, user specifications can be translated into intermediate code and then executed. The intermediate code can be cross-compiled into Modula 2, C, or Java code, to produce "independent" graph manipulation components, and a user interface generator can produce Tcl/Tk code to support rapid prototyping of graph manipulation tools. Tcl/Tk is used also to supply a graph browser to manipulate host graphs and let users apply their PROGRES specifications to particular start graphs.

The tool support offered by AGG [37], which is a rule-based visual language supporting an algebraic approach to graph transformation, is similar. With AGG, the behavior of a system is defined through graph rules, which can have negative conditions, "annotated" with Java code. This means the types of rules' nodes are defined through Java classes and standard Java libraries can be exploited to compute their attributes. The tool environment provides editors for specifying graphs, rules, and the associated Java code, allowing users to play what-if simulations, by specifying the starting graph and the order rules should be applied, but supports also *efficient graph parsing*. This problem is undecidable in general, but it can be solved for restricted classes of graph grammars. For parsing, AGG requires that users define a so-called *parse grammar* that contains reducing parsing rules, that is, roughly rules whose left-hand and right-hand sides are swapped, and a stop graph. The parsing algorithms are based on back-tracking, but since it has exponential time complexity, it is improved by exploiting critical pair analysis [12].

Moreover, being implemented in Java, AGG, can be used as hidden graph transformation engine for all those Java applications that need to manage and transform graphs.

Quite different is the support offered by FUJABA [1], which is an environment for round trip engineering with UML, Java, and design patterns, based on graph transformation. In this case, users are supplied with a standard UML-like CASE tool to design their models, but the tool exploits graph transformation to let them specify the behavioral aspects of their modeling elements. Like other CASE tools, FUJABA generates Java class definitions from UML class diagrams, but it combines UML statechart, activity, and collaboration diagrams to define *story diagrams*, that is, the diagrams that users exploits to model the dynamics of their methods. This way method bodies can automatically be generated and round trip engineering is supported: Users can modify the generated code manually and FUJABA is able to load modified code and to (re)establish the corresponding diagrams.

5.2 Graph Transformation-Enabled Tools

The tools presented in this section provide mainly automatic support to the generation of graphical editors by making users define visual languages through suitable grammars.

For example, DIAGEN supplies two distinct components: a *framework of Java classes*, to provide the generic functionality for editing and analyzing diagrams, and a *generator program*, to produce Java source code for those aspects that depend on the concrete syntax of the language.

Users specify their languages – mainly the concrete syntax, but also the other aspects could be covered – through an hyperedge graph grammar, which allows DIAGEN to recognize the structure and syntactic correctness of diagrams during the editing process. DIAGEN allows both free-hand and syntax-directed editing, with the former directly mapped onto defined grammar productions, and the latter which requires that the user specification be parsed. Thus, hypergraph

transformations and grammars provide a flexible syntactic model and allow for efficient parsing of user specifications (i.e., diagrams produced using the editor).

DiaGen provides also an automatic layout facility, which is based on flexible geometric constraints and relies on an external constraint-solving engine.

Similarly, the GenGED approach (*Generation of Graphical Environments for Design*) and toolset [5] supports the description of visual modeling languages for producing both graphical editors – available in syntax-directed or free-hand editing mode – and simulation environments. To this end, GenGED exploits AGG and graphical constraint solving techniques.

A visual language is defined through a set of grammars: the *syntax grammar* defines the actual syntax of the language and would be enough if the aim is only to implement a syntax-directed editor. To allow for free-hand editing, GenGED requires also a *parse grammar*, which has the same characteristics of the parsing grammars supported by AGG. Moreover, if users want to simulate their models, they must provide a *simulation grammar* to specify how their model behave when fed correctly.

GenGED integrates these grammars, together with defined graphical constraints to properly laying out user models, and supplies an integrated environment for the specified language.

6 Conclusion

It was the aim of this tutorial paper to provide a light-weight introduction to graph transformation, its concepts, applications, theory, and tools, from the point of view of visual modeling techniques in software engineering. To this aim, we have identified in the Introduction three aspects of visual modeling techniques which complicate their definition and implementation: the graphical structure, the number and diversity of different languages, and the view-based nature of models which creates consistency problems.

After introducing a basic graph transformation approach, i.e., a set-theoretic presentation of the double-pushout approach and discussing possible extensions and alternatives, we have presented applications to the modeling of object behavior and architectural dynamics, and we have demonstrated how these two views can be consistently integrated by means of a common meta model. This part shows the adequacy of graph transformation as a semantic domain for today's systems and languages.

Then, applications of graph transformation to the definition of modeling languages have been discussed. In particular, the presentation has focussed on concrete and abstract syntax definitions by means of graph grammars, operational semantics based on rule-based specifications of abstract interpreters, and denotational semantics based on rule-based translations of visual models into semantic domains.

Finally, we have mentioned the most important branches of the theory of graph transformation based on their history in Chomsky grammars, Petri nets, and term rewriting, and we have given a survey of tools supporting, and supported by graph transformation.

For the future, the main short-term goal of the community should be the improvement of the usability of graph transformation for non-experts. Today, such applications require both theoretical background and experience to understand potential benefits and problems in a certain domain, which is not readily accessible outside the community. One key issue, thus, is the transfer of concepts and theory to different application domains. This requires an in-depth study of problems and existing solutions in each domain and a presentation of graph transformation techniques and tools using the domain language and notation. It includes the preparation of learning material for people with different backgrounds and skills, but also the organization of schools and tutorials for different communities, trying to customize the contents on the actual needs of the hosting community.

In return, each application domain will raise new requirements for the development of theory and tools, and this will result in a partial shift of motivations from theory- to application-oriented. Theoretical problems with practical motivation include the verification of graph transformation systems, the transformation of "graphs with semantics" like Petri nets or UML models, and the evolution and modularity of graph transformation systems with corresponding compositionality results, to name only a few examples.

References

1. From UML to Java and Back Again: The Fujaba homepage. `www.fujaba.de`.
2. M. Andries, G. Engels, and J. Rekers. How to represent a visual specification. In K. Marriott and B. Meyer, editors, *Visual Language Theory*, pages 241–255. Springer-Verlag, 1997.
3. J.W. Backus. The syntax and semantics of the proposed international algebraic language of the Zurich ACM-GAMM Conference. In *International Conference on Information Processing, Paris*, pages 125–131, 1959.
4. P. Baldan, A. Corradini, H. Ehrig, M. Löwe, U. Montanari, and F. Rossi. Concurrent semantics of algebraic graph transformation. In Ehrig et al. [29], pages 107 – 188.
5. R. Bardohl and H. Ehrig. Conceptual model of the graphical editor GenGed for the visual definition of visual languages. In Ehrig et al. [25], pages 252 – 266.
6. R. Bardohl, G. Taentzer, M. Minas, and A. Schürr. Application of graph transformation to visual languages. In Engels et al. [24], pages 105–180.
7. H. P. Barendregt, M. C. J. D. van Eekelen, J. R. W. Glauert, J. R. Kennaway, M. J. Plasmeier, and M. R. Sleep. Term graph rewriting. In *PARLE*, volume 259 of *LNCS*, pages 141–158. Springer-Verlag, 1987.
8. E. Barendsen and S. Smeters. Graph rewriting aspects of functional programming. In Engels et al. [24], pages 63 – 102.
9. L. Baresi. *Formal customization of graphical notations.* PhD thesis, Dipartimento di Elettronica e Informazione – Politecnico di Milano, 1997. In Italian.
10. M. Bauderon and H. Jacquet. Categorical product as a generic graph rewriting mechanism. *Applied Categorical Structures*, 9(1), 2001.
11. P. Bottoni, M. Koch, F. Parisi-Presicce, and G. Taentzer. A visualization of OCL using collaborations. In Gogolla and Kobryn [43], pages 257–271.

12. P. Bottoni, A. Schürr, and G. Taentzer. Efficient Parsing of Visual Languages based on Critical Pair Analysis and Contextual Layered Graph Transformation. In *Proc. IEEE Symposium on Visual Languages*, September 2000. Long version available as technical report SI-2000-06, University of Rom.
13. G. Busatto, G. Engels, K. Mehner, and A. Wagner. A framework for adding packages to graph transformation approaches. In Ehrig et al. [25], pages 352–367.
14. D. Coleman, P. Arnold, S. Bodof, C. Dollin, H. Gilchrist, F. Hayes, and P. Jeremes. *Object Oriented Development, The Fusion Method*. Prentice Hall, 1994.
15. A. Corradini and U. Montanari. Specification of Concurrent Systems: from Petri Nets to Graph Grammars. In *Quality of Communication-Based Systems*, pages 35–52. Kluwer Academic Publishers, 1995.
16. A. Corradini, U. Montanari, and F. Rossi. Graph processes. *Fundamenta Informaticae*, 26(3,4):241–266, 1996.
17. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, and M. Löwe. Algebraic approaches to graph transformation, Part I: Basic concepts and double pushout approach. In Rozenberg [82], pages 163–245.
18. A. Corradini and F. Rossi. A new term graph rewriting formalism: Hyperedge replacement jungle rewriting. In Sleep M.R., Plasmeijer M.R., and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 8, pages 101–116. John Wiley & Sons Ltd, 1993.
19. B. Courcelle. The monadic second-order logic of graphs I, recognizable sets of finite graphs. *Information and Computation*, 8521:12–75, 1990.
20. B. Courcelle. The expression of graph properties and graph transformations in monadic second-order logic. In Rozenberg [82].
21. F. Drewes, B. Hoffmann, and D. Plump. Hierarchical graph transformation. In J. Tiuryn, editor, *Foundations of Software Science and Computation Structures (FoSSACS'00), Berlin, Germany*, volume 1784 of *LNCS*. Springer-Verlag, March/April 2000.
22. F. Drewes, H.-J. Kreowski, and A. Habel. Hyperedge replacement graph grammars. In Rozenberg [82], pages 95 – 162.
23. D. D'Souza and A. Wills. *Components and Frameworks with UML: The Catalysis Approach*. Addison-Wesley, 1998.
24. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.
25. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Proc. 6th Int. Workshop on Theory and Application of Graph Transformation (TAGT'98), Paderborn, November 1998*, volume 1764 of *LNCS*. Springer-Verlag, 2000.
26. H. Ehrig and A. Habel. Graph grammars with application conditions. In G. Rozenberg and A. Salomaa, editors, *The Book of L*, pages 87–100. Springer-Verlag, 1985.
27. H. Ehrig, A. Habel, H.-J. Kreowski, and F. Parisi Presicce. Parallelism and concurrency in high-level replacement systems. *Math. Struct. in Comp. Science*, 1:361–404, 1991.
28. H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, and A. Corradini. Algebraic approaches to graph transformation, Part II: Single pushout approach and comparison with double pushout approach. In Rozenberg [82], pages 247–312.
29. H. Ehrig, H.-J. Kreowski, U. Montanari, and G. Rozenberg, editors. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 3: Concurrency and Distribution*. World Scientific, 1999.

30. H. Ehrig and M. Löwe. Categorical principles, techniques and results for high-level replacement systems in computer science. *Applied Categorical Structures*, 1(1):21–50, 1993.
31. H. Ehrig, M. Pfender, and H.J. Schneider. Graph grammars: an algebraic approach. In *14th Annual IEEE Symposium on Switching and Automata Theory*, pages 167–180. IEEE, 1973.
32. J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In Rozenberg [82], pages 1 – 94.
33. G. Engels, R. Gall, M. Nagl, and W. Schäfer. Software specification using graph grammars. *Computing*, 31:317–346, 1983.
34. G. Engels, J.H. Hausmann, R. Heckel, and St. Sauer. Dynamic meta modeling: A graphical approach to the operational semantics of behavioral diagrams in UML. In A. Evans, S. Kent, and B. Selic, editors, *Proc. UML 2000, York, UK*, volume 1939 of *LNCS*, pages 323–337. Springer-Verlag, 2000.
35. G. Engels, R. Heckel, and J.M. Küster. Rule-based specification of behavioral consistency based on the UML meta model. In Gogolla and Kobryn [43].
36. G. Engels and A. Schürr. Hierarchical graphs, graph types and meta types. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.
37. C. Ermel, M. Rudolf, and G. Taentzer. The AGG approach: Language and tool environment. In Engels et al. [24], pages 551 – 601.
38. A. Finkelstein, J. Kramer, B. Nuseibeh, M. Goedicke, and L. Finkelstein. Viewpoints: A framework for integrating multiple perspectives in system development. *Int. Journal of Software Engineering and Knowledge Engineering*, 2(1):31–58, March 1992.
39. T. Fischer, J. Niere, L. Torunski, and A. Zündorf. Story diagrams: A new graph transformation language based on UML and Java. In Ehrig et al. [25].
40. P. Fradet, D. Le Métayer, and M. Périn. Consistency checking for multiple view software architectures. In *Proc. Joint European Software Engineering Conference and Symp. on Foundations of Software Engineering, ESEC/FSE'99*, volume 1687 of *LNCS*, pages 410–428, 1999.
41. C. Ghezzi, M. Jazayeri, and D. Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall Int., 1991.
42. M. Goedicke. Paradigms of modular software development. In R. J. Mitchell, editor, *Managing Complexity in Software Engineering*, volume 17 of *IEE Computing Series*. Peter Peregrinus, 1990.
43. M. Gogolla and C. Kobryn, editors. *Proc. UML 2001 – Modeling Language, Concepts and Tools, Toronto, Kanada*, LNCS. Springer-Verlag, 2001.
44. M. Große-Rhode, F. Parisi-Presicce, and M. Simeoni. Refinement of graph transformation systems via rule expressions. In Ehrig et al. [25], pages 368–382.
45. A. Habel, R. Heckel, and G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3,4):287 – 313, 1996.
46. D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, oct 1996.
47. J.H. Hausmann, R. Heckel, and G. Taentzer. Detecting conflicting functional requirements in a use case driven approach: A static analysis technique based on graph transformation. In *Proc. Int. Conference on Software Engineering (ICSE'2002)*, Orlando, FL, May 2002. ACM/IEEE Computer Society.
48. R. Heckel, A. Corradini, H. Ehrig, and M. Löwe. Horizontal and vertical structuring of typed graph transformation systems. *Math. Struc. in Comp. Science*, 6(6):613–648, 1996.

49. R. Heckel, J. Küster, and G. Taentzer. Confluence of typed attributed graph transformation systems. In A. Corradini and H.-J. Kreowski, editors, *Proc. 1st Int. Conference on Graph Transformation (ICGT 02), Barcelona, Spain*, LNCS. Springer-Verlag, October 2002. To appear.
50. R. Heckel and St. Sauer. Strengthening UML collaboration diagrams by state transformations. In H. Hußmann, editor, *Proc. Fundamental Approaches to Software Engineering (FASE'2001), Genova, Italy*, volume 2185 of *LNCS*. Springer-Verlag, April 2001.
51. R. Heckel and A. Wagner. Ensuring consistency of conditional graph grammars – a constructive approach. In *Proc. of SEGRAGRA'95 "Graph Rewriting and Computation"*, volume 2 of *Electronic Notes in TCS*, 1995.
52. D. Hirsch, P. Inverardi, and U. Montanari. Modeling software architectures and styles with graph grammars and constraint solving. In *Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Texas, E.E.U.U.*, February 1999.
53. D. Hirsch and M. Montanari. Synchronized hyperedge replacement with name mobility. In *Proc. CONCUR 2001, Aarhus, Denmark*, volume 2154 of *LNCS*, pages 121–136. Springer-Verlag, August 2001.
54. C. Hoare. Communicating sequential processes. *Communicat. Associat. Comput. Mach.*, 21(8):666–677, 1978.
55. B. Hoffmann and M. Minas. A generic model for diagram syntax and semantics. In *Proc. ICALP2000 Workshop on Graph Transformation and Visual Modelling Techniques, Geneva, Switzerland*. Carleton Scientific, 2000.
56. D. Janssens. Actor grammars and local actions. In Ehrig et al. [29], pages 57–106.
57. D. Janssens and G. Rozenberg. On the structure of node-label controlled graph grammars. *Information Science*, 20:191–216, 1980.
58. A. Kent and D. Akehurst. A relational approach to defining transformations in a metamodel. In *Proc. UML 2002, Dresden, Germany*, LNCS. Springer-Verlag, 2002. To appear.
59. H.J. Köhler, U. Nickel, J. Niere, and A. Zündorf. Integrating UML diagrams for production control systems. In *Proc. of the* 22th *International Conference on Software Engineering (ICSE), Limerick, Irland*. ACM Press, 2000.
60. J. Kramer and J. Magee. Distributed software architectures. In *Proceedings of the 19th International Conference on Software Engineering (ICSE '97)*, pages 633–634. Springer-Verlag, May 1997.
61. H.-J. Kreowski and S. Kuske. On the interleaving semantics of transformation units - a step into GRACE. In *5th Int. Workshop on Graph Grammars and their Application to Computer Science, Williamsburg '94, LNCS 1073*, pages 89 – 106. Springer-Verlag, 1996.
62. S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In Gogolla and Kobryn [43].
63. Le Métayer, D. Software architecture styles as graph grammars. In *Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering*, volume 216 of *ACM Software Engineering Notes*, pages 15–23, New York, October 16–18 1996. ACM Press.
64. M. Löwe. Algebraic approach to single-pushout graph transformation. *Theoret. Comput. Sci.*, 109:181–224, 1993.
65. M. Löwe, M. Korff, and A. Wagner. An algebraic framework for the transformation of attributed graphs. In M. R. Sleep, M. J. Plasmeijer, and M.C. van Eekelen, editors, *Term Graph Rewriting: Theory and Practice*, chapter 14, pages 185–199. John Wiley & Sons Ltd, 1993.

66. K. Marriott, B. Meyer, and K.B. Wittenburg. A survey of visual language specification and recognition. In B. Meyer K. Marriott, editor, *Visual Language Theory*, chapter 2, pages 5–85. Springer-Verlag, 1998.
67. R. Milner. *Communication and Concurrency*. Prentice-Hall, 1989.
68. R. Milner. Bigraphical reactive systems. In Kim Guldstrand Larsen and Mogens Nielsen, editors, *Proc. 12th Intl. Conference on Concurrency Theory (CONCUR 2002), Aalborg, Denmark*, volume 2154 of *LNCS*, pages 16–35. Springer-Verlag, August 2001.
69. M. Minas. Hypergraphs as a uniform diagram representation model. In Ehrig et al. [25], pages 281 – 295.
70. U. Montanari. Separable graphs, planar graphs and web grammars. *Information and Control 16*, pages 243–267, 1970.
71. U. Montanari, M. Pistore, and F. Rossi. Modeling concurrent, mobile, and coordinated systems via graph transformation. In Ehrig et al. [29], pages 189 – 268.
72. M. Nagl. *Graph-Grammatiken: Theorie, Implementierung, Anwendungen*. Vieweg, 1979.
73. M. Nagl, editor. *Building Tightly Integrated Software Development Environments: The IPSEN Approach, LNCS 1170*. Springer-Verlag, 1996.
74. Object Management Group. Meta object facility (MOF) specification, September 1999. `http://www.omg.org`.
75. Object Management Group. UML specification version 1.4, 2001. `http://www.celigent.com/omg/umlrtf/`.
76. J. L. Pfaltz and A. Rosenfeld. Web grammars. *Int. Joint Conference on Artificial Intelligence*, pages 609–619, 1969.
77. G. Plotkin. A structural approach to operational semantics. Technical Report DAIMI FN-19, Aarhus University, Computer Science Department, 1981.
78. D. Plump. Term graph rewriting. In Engels et al. [24], pages 3 – 62.
79. T. W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
80. T.W. Pratt. Definition of programming language semantics using grammars for hierarchical graphs. In H. Ehrig, V. Claus, and G. Rozenberg, editors, *1st Int. Workshop on Graph Grammars and their Application to Computer Science and Biology, LNCS 73*, volume 73 of *LNCS*. Springer-Verlag, 1979.
81. J. Rekers and A. Schürr. Defining and parsing visual languages with layered graph grammars. *Journal of Visual Languages and Computing*, 8(1):27 –55, 1997.
82. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
83. H.-J. Schneider. Chomsky-Systeme für partielle Ordnungen. Technical Report 3-3, Universität Erlangen, 1970.
84. A. Schürr. Specification of graph translators with triple graph grammars. In Tinhofer, editor, *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*, number 903 in LNCS, pages 151–163. Springer-Verlag, 1994.
85. A. Schürr. Logic based programmed structure rewriting systems. *Fundamenta Informaticae*, 26(3,4):363 – 386, 1996.
86. A. Schürr. Programmed graph replacement systems. In Rozenberg [82], pages 479 – 546.
87. A. Schürr, A.J. Winter, and A. Zündorf. The PROGRES approach: Language and environment. In Engels et al. [24], pages 487–550.
88. D. Scott and C. Strachey. Towards a mathematical semantics for computer languages. In *Computers and Automata*, pages 19–46. Wiley, 1971.

89. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
90. S. H. von Solms. Node-label controlled graph grammars with context conditions. *Intern. J. Computer Math 15*, pages 39–49, 1984.
91. G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996. Shaker Verlag.
92. G. Taentzer, I. Fischer, M. Koch, and V. Volle. Distributed graph transformation with application to visual design of distributed systems. In Ehrig et al. [29].
93. G. Taentzer, M. Goedicke, and T. Meyer. Dynamic change manegement by distributed graph transformation: Towards configurable distributed systems. In *Proceedings TAGT'98*, volume 1764 of *LNCS*, pages 179–193. Springer-Verlag, 2000.
94. M. von der Beek. A comparison of Statecharts variants. In *Formal Techniques in Real-Time and Fault-Tolerant Systems*. Springer LNCS 863, 1994.
95. C.P. Wadsworth. *Semantics and Pragmatics of the Lambda Calculus*. PhD thesis, University of Oxford, 1971.
96. M. Wermelinger and J.L. Fiadero. A graph transformation approach to software architecture reconfiguration. In H. Ehrig and G. Taentzer, editors, *Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra'2000), Berlin, Germany*, March 2000. `http://tfs.cs.tu-berlin.de/gratra2000/`.

Tutorial on DNA Computing and Graph Transformation – Computational Nature of Gene Assembly in Ciliates

Tero Harju[1], Ion Petre[2], and Grzegorz Rozenberg[3]

[1] Department of Mathematics, University of Turku
FIN-20014 Turku, Finland
harju@utu.fi
[2] Department of Computer Science, Åbo Academi University
FIN-20520 Turku, Finland
ion.petre@abo.fi
[3] Leiden Institute for Advanced Computer Science, Leiden University
Niels Bohrweg 1, 2333 CA Leiden, the Netherlands, and
Department of Computer Science, University of Colorado at Boulder
Boulder, Co 80309-0347, USA
rozenber@liacs.nl

1 Introduction

DNA computing, or more generally molecular computing, is a novel exciting area of research at the intersection of mathematics, computer science and molecular biology. The area of DNA computing studies the use of biomolecules for the purpose of computing (*in vitro* or *in vivo*). Molecular biologists assists here computer scientists in their attempt to replace or to complement the silicon-based computers by DNA-based computers. The hardware for such computations consists of biomolecules (*bioware*).

The landmark step towards using DNA for computing was taken by Leonard Adleman [1] in an experiment, where he managed to solve a small instance of an untractable graph theoretic problem, the directed Hamilton problem, using DNA molecules. Since Adleman's experiment many different models have been devised to solve other instances of complex computational problems using molecular techniques. The advantages of DNA computing are in its massive parallelism, huge memory capacity, and energy efficiency.

Research in DNA computing can be roughly divided in two streams: DNA computing *in vitro* and *in vivo*. The former is concerned with building (specialized) DNA-based computers in test tubes, while the latter is concerned with implementing some computational components in living cells, as well as with studying the computational processes taking place in the living cells.

In this tutorial, we shall discuss the computational nature of a very intricate DNA processing taking place in single cell organisms called *ciliates*, in the process of *gene assembly*. We shall discuss the role of graph transformations in modelling and studying gene assembly process. In particular, we demonstrate

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 430–434, 2002.
© Springer-Verlag Berlin Heidelberg 2002

that graph transformations provide the right level of abstraction, and useful technical tools for studying gene assembly. On the other hand, the gene assembly process inspires a new computing paradigm; computing by folding and recombination, which induces novel questions and challenges for research on graph transformation.

2 Nuclear Dualism of Ciliates

Ciliates are an ancient group of single cell organisms that have developed a unique nuclear dualism to store the DNA. They have two kinds of nuclei, the germline *micronucleus* and the somatic *macronucleus*. The genetic information is encoded in very different ways in the two types of nuclei. In the micronucleus, the genes are placed in very long continuous DNA molecules – they can be many millions of base pairs – interrupted by noncoding spacer DNA. As a matter of fact, typically, less than 5% of the micronuclear DNA sequence consists of genetic material. In the macronucleus on the other hand, the DNA is present in short, gene-size molecules, on average two thousand base pairs long. At some stage in the sexual reproduction process, ciliates convert the micronuclear genome into the macronuclear genome, eliminating all the noncoding DNA, and transferring the micronuclear form of each gene into its macronuclear form. Because of the drastically different forms of the macronuclear and the micronuclear genomes, this conversion process is one of the most involved DNA processing known in Nature, producing the shortest known DNA molecules. Moreover, it turns out that this conversion process is fascinating from computational point of view.

Each micronuclear gene is divided in several segments called MDSs (*Macronuclear Destined Segments*), which are interrupted by noncoding segments called *IES*s (*Internal Eliminated Segments*), see Fig. 1, where MDSs are represented by rectangles (with 'pointers' at their ends) and the interspersing IESs are represented by line segments. The MDSs can occur in the micronuclear DNA sequences in a scrambled order, and moreover, some of them may even be inverted. The macronuclear version of the same gene, illustrated in Fig. 2, consists of MDSs from its micronuclear form that are spliced together in the *orthodox* order $M_1 M_2 \dots M_k$, by 'gluing' them on common 'pointers'.

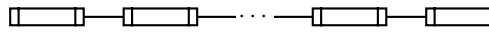


Fig. 1. The structure of a micronuclear gene

Formally, if a gene has k MDSs, then each MDS M_i, for $i = 2, \dots, k-1$, is of the form $M_i = (\pi_i, \mu_i, \pi_{i+1})$ while M_1 has the form $M_1 = (\beta, \mu_1, \pi_2)$, and M_k has the form $M_k = (\pi_k, \mu_k, \varepsilon)$. Thus, the ending nucleotide sequence π_{i+1} of the MDS M_i (the *outgoing pointer* of M_i) is identical to the initial nucleotide sequence of the MDS M_{i+1} (the *incoming pointer* of M_{i+1}), for all $1 \leq i < k$. The segments μ_i are the *bodies*, and the special symbols β in M_1 and ε in M_k

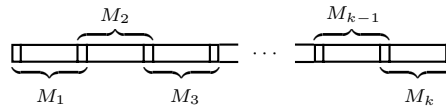


Fig. 2. The structure of a macronuclear gene

are merely *markers* for the beginning and the end of the macronuclear gene. Note that each pointer has two occurrences in two different MDS's: incoming and outgoing.

During gene assembly the micronuclear gene (Fig. 1) is converted to the macronuclear gene (Fig. 2). In this process, all IESs from the micronuclear gene are removed (excised), while the MDSs are spliced together on their common pointers to form the macronuclear gene. Thus, a gene in a macronucleus of a ciliate is described by a string $M_1M_2 \dots M_k$, while its micronuclear predecessor is described by a string $I_1M'_{t_1}I_2M'_{t_2}I_3 \dots I_kM'_{t_k}I_{k+1}$, where each I_i represents an IES, and each M'_{t_i} represents either M_{t_i} or its inversion $\overline{M}_{t_i}$. Here $M_{t_1} \dots M_{t_k}$ is a permutation of the string $M_1M_2 \dots M_k$.

For instance, consider the *actin I gene* of *O. nova* species of ciliates, which has the following micronuclear form: $M_3I_1M_4I_2M_6I_3M_5I_4M_7I_5M_9I_6\overline{M}_2I_7M_1I_8M_8$, where, *e.g.*, $M_2 = (\pi_2, \mu_2, \pi_3)$ has the following nucleotide sequences as pointers

$$\pi_2 = \begin{smallmatrix} ATGTGTAGTAAG \\ TACACATCATTC \end{smallmatrix} \quad \text{and} \quad \pi_3 = \begin{smallmatrix} AATC \\ TTAG \end{smallmatrix}.$$

3 Formalisms: Legal Strings and Overlap Graphs

To simplify the formalism, we introduce the alphabets $\Pi = \{2, 3, \dots, k\}$ and $\overline{\Pi} = \{\overline{2}, \overline{3}, \dots, \overline{k}\}$ for the pointers and their inversions. For $p \in \Pi$, let $\overline{\overline{p}} = p$. Since we are mostly interested in the assembly strategies – how micronuclear genes are translated to macronuclear counterparts, (1) we ignore the IESs in the micronucleus, and (2) we write $i(i+1)$ for the MDS $M_i = (\pi_i, \mu, \pi_{i+1})$ and $\overline{(i+1)}\,\overline{i}$ for the inversion of M_i. (The beginning and the end markers are ignored in the coding.) Thus, for instance, the micronuclear molecule $I_1M_4I_2\overline{M}_2I_3M_1I_4M_5I_5M_3$ is coded into the string $4\,5\,\overline{3}\,\overline{2}\,2\,5\,3\,4$. Strings over $\Pi \cup \overline{\Pi}$ that have exactly two occurrences from $\{p, \overline{p}\}$, for each pointer $p \in \Pi \cup \overline{\Pi}$, are called *legal strings*. A pointer p in a legal string δ is *negative*, if $\delta = \delta_1 p \delta_2 p \delta_3$, and p is *positive*, if $\delta = \delta_1 p \delta_2 \overline{p} \delta_3$.

Instead of strings, we also can use graphs to formalize micronuclear genes. Two pointers $p, q \in \Pi$ are said to *overlap* in a legal string δ, if δ can be written as $\delta = \delta_1 p_1 \delta_2 q_1 \delta_3 p_2 \delta_4 q_2 \delta_5$, where $p_1, p_2 \in \{p, \overline{p}\}$ and $q_1, q_2 \in \{q, \overline{q}\}$), for some strings δ_i. The *signed overlap graph* γ_δ of δ has Π as its set of vertices, and there is an (undirected) edge pq if the pointers p and q overlap in δ. Moreover, each vertex p is labelled by $+$ or $-$ depending whether the pointer p is positive or negative in δ.

4 The Operations for Assembly

It was postulated in [5] and [7] that gene assembly is accomplished through the following three molecular operations: *ld-excision*, *hi-excision/reinsertion*, and *dlad-excision/reinsertion*, see Fig. 3, 4 and 5. These operations are of the *fold and recombine* style: first a molecule is folded, a cut is made (on some pointers), and then recombination takes place. Notice that in each of these operations the pointers that take part in the transformation will cease to be available as pointers.

We shall shortly consider these operations in the formalism of legal strings.
(1) The *ld*-operation, see Fig. 3, is applied to a negative pointer p. The operation excises the part of the molecule that lies between the occurrences of this pointer – this part forms a debris circular molecule, and should not contain any MDSs. In the string notation, the transformation is as follows: $\delta_1 pp\delta_2 \vdash \not{\delta}_1\delta_2$ with $p \in \Pi \cup \overline{\Pi}$.
(2) The *hi*-operation, see Fig 4, is applied to a positive pointer, and it reverses the part of the molecule that lies between the two occurrences of the pointer. In string notation, we have: $\delta_1 p\delta_2\overline{p}\delta_3 \vdash \not{\delta}_1\overline{\delta}_2\delta_3$ with $p \in \Pi \cup \overline{\Pi}$.
(3) The *dlad*-operation, see Fig. 5, is applied to an overlapping pair of negative pointers, and it results in a molecule, where the intermediate parts are transposed. In string notation, we have: $\delta_1 p\delta_2 q\delta_3 p\delta_4 q\delta_5 \vdash \not{\delta}_1\delta_4\delta_3\delta_2\delta_5$ with $p, q \in \Pi \cup \overline{\Pi}$, $p \neq q$.

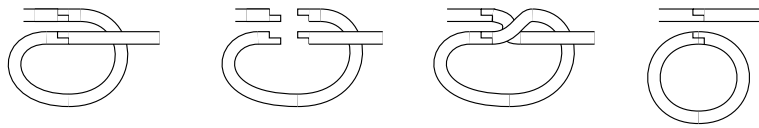


Fig. 3. The ld-operation

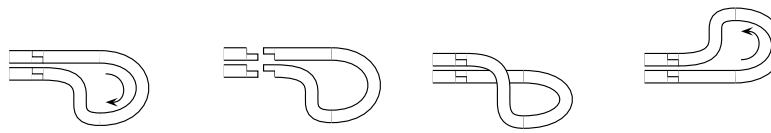


Fig. 4. The hi-operation

We refer to [4] and [2] for more details of these operations.

Importantly, the operations *ld, hi* and *dlad*, provide an *intramolecular* framework for gene assembly: in each operation the molecule involved reacts with itself (and not with another molecule). Note also that folding brings together the pointers that are involved in an operation, and the operations splice to-

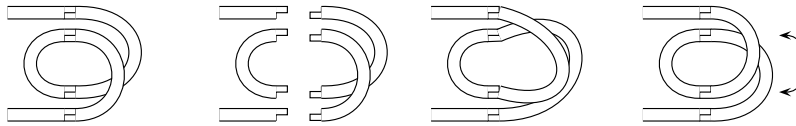


Fig. 5. The dlad-operation

gether smaller MDSs to form composite MDSs – the corresponding IESs are either spliced together forming composite IESs or they are excised.

It can be shown that the three operations *ld*, *hi*, and *dlad* are universal for legal strings, that is, every legal string can be reduced to the empty word Λ by a finite number of these operations, see [3]

One can also express the three molecular operations in terms of the overlap graphs. In this graph theoretic notation: (1) *ld* corresponds to removing a negative isolated vertex p^-; (2) *hi* corresponds to removing a positive vertex p^+, and complementing the neighbourhood of p (also changing the signs of these vertices); (3) *dlad* corresponds to removing two adjacent negative vertices p^- and q^-, and complementing the edge set between the neighbourhoods $N(p)$ and $N(q) \setminus N(p)$, and $N(q)$ and $N(p) \setminus N(q)$. The universality result now states that every overlap graph reduces to the empty graph by a finite number of these three graph theoretic operations, see [2] and [4]. Such a sequence of operations then describes how an micronuclear gene can be transformed to its macronuclear counter part.

References

1. Adleman, L., Molecular Computation of Solutions to Combinatorial Problems, *Science*, Vol. **266**, 11 November 1994, pp. 1021 – 1023.
2. Ehrenfeucht, A., Harju, T., Petre, I., Prescott, D. M., and Rozenberg, G., Formal systems for gene assembly in ciliates, *Theoret. Comput. Sci.*, to appear.
3. Ehrenfeucht, A., Petre, I., Prescott, D. M., and Rozenberg, G., Universal and simple operations for gene assembly in ciliates, to appear in *Words, Sequences, Languages: Where computer science, biology and linguistics come across*, Mitrana, V., Martin-Vide, C. (eds.) (2000).
4. Ehrenfeucht, A., Petre, I., Prescott, D. M., and Rozenberg, G., String and graph reduction systems for gene assembly in ciliates, *Math. Structures Comput. Sci.* **12** (2002), 113 – 134.
5. Ehrenfeucht, A., Prescott, D. M., and Rozenberg, G., Computational aspects of gene (un)scrambling in ciliates. In *Evolution as Computation*, Landwerber, L., Winfree, E. (eds.), 45–86, Springer Verlag, Berlin, Heidelberg, 2000.
6. Prescott, D. M., The evolutionary scrambling and developmental unscabling of germlike genes in hypotrichous ciliates, *Nucl. Acids Res.* **27** (1999), 1243 – 1250.
7. Prescott, D. M., Ehrenfeucht, A., and Rozenberg, G., Molecular operations for DNA processing in hypotrichous ciliates, *European Journal of Protistology*, to appear.

TERMGRAPH 2002 Workshop Survey

Detlef Plump

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

1 Background and Aims

The International Workshop on Term Graph Rewriting (TERMGRAPH 2002) will take place 7 October 2002 as a one-day satellite event of the International Conference on Graph Transformation (ICGT 2002). Term graph rewriting is concerned with the representation of functional expressions as graphs and the evaluation of these expressions by rule-based graph transformation. Using graphs rather than strings or trees allows to share common subexpressions, which improves the efficiency of computations in space and time. Sharing is ubiquitous in implementations of functional and logic programming languages, systems for automated reasoning, and symbolic computation systems.

Research in term graph rewriting ranges from theoretical questions to practical implementation issues. It includes such different lines as the modelling of (finite or infinitary) first-order term rewriting by (acyclic or cyclic) graph rewriting, rewrite rules on so-called sharing graphs for Levy-optimal reduction in the Lambda Calculus, rewrite calculi on cyclic higher-order term graphs for the semantics and analysis of functional programs, graph reduction implementations of functional programming languages, and automated reasoning and symbolic computation systems working on shared structures.

The aim of TERMGRAPH 2002 is to bring together researchers working in these different domains and to foster their interaction, to provide a forum for presenting new ideas and work in progress, and to enable newcomers to learn about current activities in term graph rewriting. The papers submitted to the workshop were refereed by the following program committee:

Zena M. Ariola	University of Oregon, Eugene (USA)
Richard Banach	University of Manchester (United Kingdom)
Rachid Echahed	IMAG, Grenoble (France)
Richard Kennaway	University of East Anglia, Norwich (United Kingdom)
Jan Willem Klop	Vrije Universiteit Amsterdam (The Netherlands)
Rinus Plasmeijer	Katholieke Universiteit Nijmegen (The Netherlands)
Detlef Plump	University of York (United Kingdom), chair

In addition, Adam Bakewell and Olaf Chitil (University of York) refereed papers.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 435–439, 2002.
© Springer-Verlag Berlin Heidelberg 2002

2 Contributions

The contributions summarized below will appear in Volume 72 (Issue 1) of Elsevier's *Electronic Notes in Theoretical Computer Science*.

Conditional Term Graph Rewriting with Indirect Sharing
Enno Ohlebusch

This paper discusses the implementation of various forms of conditional term rewriting systems by term graph rewriting. Technically, term graphs are defined as terms with labels where subterms with equal labels are considered as shared. The main result is that for two classes of orthogonal, conditional term rewriting systems, term graph rewriting is *adequate* in the sense of Kennaway, Klop, Sleep and de Vries. This implies both, soundness in the form that every term graph rewrite sequence corresponds to a term rewrite sequence, and completeness in the form that a term graph can be reduced to an irreducible graph if its represented term can be reduced to an irreducible term.

Inequational Deduction as Term Graph Rewriting
Andrea Corradini, Fabio Gadducci, Wolfram Kahl and Barbara König

The authors consider multi-algebras which model nondeterminism in that operators are functions delivering sets of possible results rather than single results. Multi-rooted, acyclic term graphs are used to represent derived relations in multi-algebras. *Term graph specifications* are sets of inequations of term graphs which are interpreted as relation inclusion in multi-algebras. A deduction system for inequations is defined and shown to be sound. Then a translation from term graph specifications into sets of term graph rewrite rules is presented which is sound and complete for deduction: a graph F rewrites to a graph G if and only if the inequation $F \sqsubseteq G$ can be deduced.

A Duality in Proof Systems for Recursive Type Equality and for Bisimulation Equivalence on Cyclic Term Graphs
Clemens Grabmayer

This paper is concerned with dualities between proof systems for regular cyclic objects. First two proof systems for the equality of recursive types are compared, then the same is done for two proof systems for bisimulation equivalence of cyclic term graphs. In both cases, a duality is established between a coinductive system in the style of Brandt and Henglein, and a "syntactic matching" system in the style of Ariola and Klop. The dualities are given in form of mappings that translate proofs of the compared systems into each other.

Using Term Graph Rewriting Models to Analyse Relative Space Efficiency
Adam Bakewell

Summarizing the author's PhD thesis, an approach is presented to compare the space efficiency of different implementations of functional programming languages. Implementations are modelled as higher-order term term graph rewriting systems (with garbage collectors) and proof methods are described showing that for implementations A and B, every program's evaluation by A is as least as space-efficient as its evaluation by B. Moreover, a method is given for searching for a witness program that with B needs more space than with A, proving that B is "space-leakier" than A. These methods take into account translations, allowing A and B to be specified in different languages.

Lifting Infinite Normal Form Definitions from Term Rewriting to Term Graph Rewriting
Stefan Blom

To give non-terminating higher-order rewrite systems a semantics, the author considers infinite normal forms of terms and term graphs which generalize the Böhm trees of the Lambda Calculus. He defines infinite normal forms of higher-order term graphs by using the infinite normal forms of the underlying (possibly infinite) terms. Term graphs are represented as terms with the `letrec`-construct. A result is established ensuring the uniqueness of infinite normal forms by an infinitary soundness property rather than by confluence as in previous work. Moreover, a class of higher-order term graph rewrite systems is identified for which infinite normal form definitions can be lifted from terms to term graphs.

Packages Duplication in the Interaction Nets and Weak Head Reduction in the Lambda-Calculus
Sylvain Lippi

The author presents a new implementation of the Lambda Calculus by Interaction Nets, aiming at reductions that are optimal in the sense of Lévy. This approach focuses on the duplication of a certain type of nets, so-called *packages*. The novelty—compared, for example, with the work of Abadi, Gonthier and Lévy—lies in the simplicity of the proposed system which does not need certain "house-keeping" constructs like *croissants* and *brackets*. Instead, the implementation gets along with four symbols. The price for this simplicity is that only weak head reductions are considered rather than full reductions.

A Higher-Order Calculus for Graph Transformation
Maribel Fernández, Ian Mackie and Jorge Sousa Pinto

This paper proposes a formalism for defining higher-order term rewriting systems which model various forms of graph rewriting systems. The formalism is inspired by Klop's Combinatory Reduction Systems and uses a notation for graphs

that generalizes the equational presentation of graphs used by Ariola, Klop and Blom. The calculus is flexible enough to define first-order graph and term graph rewriting, interaction nets, interaction systems, and a process calculus, but also hierarchical graph rewriting, proof nets of linear logic and operational semantics of graph-based languages. The richness of the calculus gives rise to many potential applications. For example, the structuring of large graphs by abstracting subgraphs can be defined and studied in this framework.

3 Workshop Program

Monday, October 7, 2002

10.00 – 10.15: Opening

10.15 – 11.00: Conditional Term Graph Rewriting with Indirect Sharing
Enno Ohlebusch

11.00 – 11.45: Inequational Deduction as Term Graph Rewriting
Andrea Corradini, Fabio Gadducci, Wolfram Kahl and Barbara König

11.45 – 12.30: A Duality in Proof Systems for Recursive Type Equality and for Bisimulation Equivalence on Cyclic Term Graphs
Clemens Grabmayer

12.30 – 14.30: Lunch Break

14.30 – 15.15: Using Term Graph Rewriting Models to Analyse Relative Space Efficiency
Adam Bakewell

15.15 – 16.00: Lifting Infinite Normal Form Definitions from Term Rewriting to Term Graph Rewriting
Stefan Blom

16.00 – 16.30: Coffee Break

16.30 – 17.15: Packages Duplication in the Interaction Nets and Weak Head Reduction in the Lambda-calculus
Sylvain Lippi

17.15 – 18.00: A Higher-order Calculus for Graph Transformation
Maribel Fernández, Ian Mackie and Jorge Sousa Pinto

4 Authors' Affiliations

Adam Bakewell	University of York (United Kingdom)
Stefan Blom	CWI, Amsterdam (The Netherlands)
Andrea Corradini	Universitá de Pisa (Italy)
Maribel Fernández	École Normale Supérieure, Paris (France)
Fabio Gadducci	Universitá de Pisa (Italy)
Clemens Grabmayer	Vrije Universiteit Amsterdam (The Netherlands)
Wolfram Kahl	McMaster University, Hamilton (Canada)
Barbara König	Technische Universität München (Germany)
Sylvain Lippi	Institut de Mathématiques de Luminy, Marseille (France)
Ian Mackie	École Polytechnique, Palaiseau Cedex (France)
Enno Ohlebusch	Universität Bielefeld (Germany)
Jorge Sousa Pinto	Universidade do Minho, Braga (Portugal)

Workshop on Graph-Based Tools

Tom Mens[1,⋆], Andy Schürr[2], and Gabriele Taentzer[3]

[1] Vrije Universiteit Brussel, Belgium
tommens@vub.ac.be
[2] Darmstadt University of Technology, Germany
schuerr@rs.e-technik.tu-darmstadt.de
[3] Technische Universität Berlin, Germany
gabi@cs.tu-berlin.de

Abstract. Graphs are well-known, well-understood, and frequently used means to depict networks of related items. They are successfully used as the underlying mathematical concept in various application domains. In all these domains tools are developed that store, retrieve, manipulate and display graphs. It is the purpose of this workshop to summarize the state of the art of graph-based tool development, bring together developers of graph-based tools in different application fields and to encourage new tool development cooperations.

1 Motivation

Graphs are an obvious means to describe structural aspects in various fields of computer science. They have been successfully used in application areas such as compiler compiler toolkits, constraint solving problems, generation of CASE tools, pattern recognition techniques, program analysis, software engineering, software evolution, software visualization and animation, and visual languages [8,14]. In all these areas tools have been developed that use graphs as an important underlying data structure.

Since graphs are a very general structure mechanism, it is a challenge to handle graphs in an effective way. Using graphs inside tools the following topics play an important role: efficient graph algorithms, empirical and experimental results on the scalability of graphs, reusable graph-manipulating software components, software architectures and frameworks for graph-based tools, standard data exchange formats for graphs, more general graph-based tool integration techniques, and meta CASE tools or generators for graph-based tools. The aim of the workshop on graph-based tools [GraBaTs) is to bring together developers of all kinds of graph-based tools in order to exchange their experiences, problems, and solutions concerning the efficient handling of graphs.

The GraBaTs workshop is, therefore, of special relevance for the conference on graph transformation: In many cases the application of graph transformation technology requires the existence of reliable, user-friendly and efficiently working

⋆ Tom Mens is a postdoctoral fellow of the Fund for Scientific Research - Flanders (Belgium)

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 440–444, 2002.
© Springer-Verlag Berlin Heidelberg 2002

graph transformation tools. These tools in turn have to be built on top of basic services or frameworks for graphs, which are the main topic of our workshop. Today, several graph transformation tool implementations have emerged which do not share any basic graph services (e.g. for graph pattern matching or graph layout purposes) and which implement rather different graph concepts and graph transformation approaches. Some of these tools - as a kind of survey of the state of the art - are presented in a special session, which is part of the main conference as well as of this satellite workshop. The presented tools are AGG [9], DiaGen [12], Fujaba [10], GenGED [5], and PROGRES [17].

The GraBaTs workshop is planned for 1 1/2 days. Its schedule contains in addition to the afore-mentioned session on graph transformation tools, an invited talk by Tiziana Margaria (University of Dortmund, Germany) on ETI, an electronic tool integration platform where graph-based tools will play an important role. Apart from four sessions with presentations of 15 accepted papers (out of 19 submissions) on various graph-based tools and tool-relevant topics, a discussion "Towards Standard Exchange Formats for Graph and Graph Transformation" is planned.

2 Workshop Issues

The workshop aims at bringing together tool developers from different fields, dealing with graphs from different perspectives. In the following, we give an overview on the most important perspectives.

2.1 Meta-modeling by Graphs

For a long time the syntax and static semantics of most visual modeling or programming languages was only defined by means of characteristic examples and informal descriptions. To improve this situation the visual language community invented grammar-based formalisms for the definition of the syntax of their languages, such as constraint grammars, graph grammars, relational grammars, etc. For a survey of these approaches the reader is referred to [11]. Unfortunately it turned out that the grammar-based definition of visual languages is rather complicated compared with the meta-modeling approach developed in parallel. The Meta-modeling approach for the definition of visual languages uses a combination of class diagrams (ER-diagrams, etc.) and predicate logic expressions (Z, OCL, etc.) to define the syntax and static semantics of visual languages. It became popular with the standardization of the OO-modeling language UML [18] and is used by various meta-modeling (meta-CASE) tools which are able to generate domain-specific CASE tools. The so-called MOF approach (Meta-Object Facility) [13] is one attempt to come up with a meta-modeling standard. Despite of its limited expressiveness (compared with ER diagrams or UML class diagrams) MOF builds the basis for the formal definition of UML and other visual languages. All meta-modeling approaches used nowadays have one common property: they offer graph-like diagrams for the definition of the structure (syntax) of graph-like diagram languages. Therefore, meta-modeling is in fact the

formal definition of graph languages by means of graphs which are instances of "meta" graph languages. As a consequence, meta-CASE tools are a special class of graph-based tools, which need at least basic services for storing, visualizing, and analyzing graphs.

2.2 Graph Visualization

Facilities for visualizing graphs are needed by all kinds of graph-based tools, independent of the fact whether they are e.g. used for meta-modeling or rule-based programming purposes. Furthermore, graph visualization techniques are the most important means for visualizing various aspects of software architectures, the dynamic behavior of running systems, their evolution history, and so forth [15,7]. Software components developed for these purposes usually have to deal with huge graphs and need services for making these graphs persistent, for introducing abstractions based on hierarchical graph models, for computing reasonable graph layouts (efficiently), and for displaying graphs effectively using "fish-eye-techniques" and the like. And last but not least, graph visualization techniques are often employed for teaching purposes in computer science courses on "data structures and (graph) algorithms". To summarize, almost all kinds of graph-based tools urgently need efficiently and effectively working graph visualization services, whereas graph visualization tools may profit from research activities on graph query and graph transformation engines for the computation of graph abstractions or views. We, therefore, hope that this workshop encourages researchers to start new cooperations, such as adapting graph visualization tools to the needs of graph manipulation tools or exploiting graph manipulation and transformation techniques to compute sensible abstractions of huge graphs.

2.3 Graph Queries and Graph Algorithms

Most, if not all, graph-based tools use to a certain degree software components (libraries, subsystems, etc.) for executing graph queries and/or various kinds of standard graph algorithms. For example, graph transformation tools rely on rather sophisticated means for computing graph matches (rule occurrences) and graph-based reverse engineering tools need rather powerful query engines for determining critical substructures of software architectures. On the other hand, quite a number of database management systems have already been developed using graphs (networks of related objects) as the underlying data model and offering query languages based on graph path expressions or even graph transformations [6]. Vice versa, graph transformation languages like PROGRES [17] are not only used for specifying and visualizing graph algorithms, but incorporate many elements of database query languages such as means for the construction of indexes, the materialization and incremental update of views, etc. Therefore, we like to encourage tool developers again to start cooperating across the boundaries of research areas.

2.4 Graph Transformation

Graph transformation means the rule-based manipulation of graphs. Several graph transformation approaches have emerged which differ w.r.t. to the underlying kind of graphs as well as in the way how rules are applied to graphs, i.e. graph transformation takes place [16,8]. The kind of graphs used by these tools include labeled, directed graphs, hypergraphs, and graph structures. Their rules, the basic means to manipulate graphs, differ w.r.t. to the formal definition of their semantics, the way how occurrences (matches) are searched for, and how matching rules are applied eventually.

In tools, graph transformation is usually applied to visual languages, specification, code generation, verification, restructuring, evolution and programming of software systems. Developers of graph transformation tools may profit from other workshop participants concerning more efficient realizations of basic functionality, while developers of other graph-based tools might find the graph transformation paradigm attractive to implement certain graph manipulations. The workshop may also provide insights to apply these tools to other application domains.

2.5 A Common Exchange Format for Graphs

To support interoperability between various graph-based tools, several initiatives on the development of common exchange formats for graphs have been founded. These formats are all based on the extensible markup language XML developed to interchange documents of arbitrary types. Preceding events like three subgroup meetings of the EU Working Group APPLIGRAPH [4], a Workshop on Standard Exchange Formats [2], and a satellite workshop of the 8th intl. Symposium on Graph Drawing (GD 2000) [1] discussed various ideas which are currently converging to one format being GXL [3]. During the GraBaTs workshop a further discussion round on this topic will be organized focusing especially on graph layout and graph attributes. Another topic of interest for this discussion is an exchange format for graph transformation systems called GTXL, which is under development and which will be built on top of GXL.

3 Workshop Organizers

The Program Committee of the workshop consists of:

- Luciano Baresi (Italy)
- Giuseppe Di Battista (Italy)
- Ulrik Brandes (Germany)
- Scott Marshall (The Netherlands)
- Tom Mens (Belgium) (Co-chair)
- Andy Schürr (Germany) (Co-chair)
- Gabriele Taentzer (Germany) (Co-chair)
- Andreas Winter (Germany)
- Albert Zündorf (Germany)

More information about the workshop including its program and an electronic version of all accepted papers appearing in the electronic notes of Theoretical Computer Science (ENTCS) can be found on the workshop web page: http://tfs.cs.tu-berlin.de/grabats

References

1. *Satellite workshop of 8. Int. Symposium on Graph Drawing 2000* *http://www.cs.virginia.edu/~gd2000/gd-satellite.html*, 2000.
2. *Workshop on Standard Exchange Formats* *http://www.cs.toronto.edu/~simsuz/wosef/*, 2000.
3. *GXL* *http://www.gupro.de/GXL*, 2002.
4. *GXL/GTXL* *http://tfs.cs.tu-berlin.de/projekte/gxl-gtxl.html*, 2002.
5. R. Bardohl. A visual environment for visual languages. *Science of Computer Programming*, 44(3), 2002.
6. M. Costabile and T. Catarci, editors. *Special Issue on Visual Query Systems*, volume 6. Academic Press, 1995.
7. I.F. Cruz and P. Eades, editors. *Special Issue on Graph Visualization*, volume 6. Academic Press, 1995.
8. H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors. *Handbook on Graph Grammars and Computing by Graph Transformation: Applications, Languages, and Tools*, volume 2. World Scientific, Singapore, 1999.
9. C. Ermel, M. Rudolf, and G. Taentzer. The AGG-Approach: Language and Tool Environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, volume 2: Applications, Languages and Tools*, pages 551–603. World Scientific, 1999. available at: `http://tfs.cs.tu-berlin.de/agg`.
10. *Fujaba Project Group*, 2002. Available at `http://www.fujaba.de`.
11. K. Marriott and B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal on Visual Languages and Computing*, 8(4):375–402, 1997.
12. M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(3):157 – 180, 2002.
13. *Meta Object Facilities – version 1.4*, 2002. Available at `http://www.omg.org/technology/documents/formal/mof.htm`.
14. M. Nagl, M. Münch, and A. Schürr, editors. *Application of Graph Transformation with Industrial Relevance, Proc. AGTIVE Workshop, Castle Rolduc*, volume 1779 of *LNCS*, Berlin, 2000. Springer Verlag.
15. Blaine A. Price, Ronald M. Baecker, and Ian S. Small. A principled taxonomy of software visualization. *Journal on Visual Languages and Computing*, 4(3), 1993.
16. G. Rozenberg, editor. *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.
17. A. Schürr, A. Winter, and A. Zündorf. The PROGRES-approach: Language and environment. In H. Ehrig, G. Engels, H.-J. Kreowski, and G. Rozenberg, editors, *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 2: Applications, Languages and Tools*. World Scientific, 1999. available at: `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`.
18. *Unified Modeling Language – version 1.4*, 2002. Available at `http://www.omg.org/technology/documents/formal/uml.htm`.

Workshop on Graph Transformation and Visual Modeling Techniques

Paolo Bottoni[1] and Mark Minas[2]

[1] Università di Roma "La Sapienza", Italy
[2] University of Erlangen, Germany

1 Introduction

Diagrammatic notations have accompanied the development of technical and scientific disciplines in fields as diverse as mechanical engineering, quantum physics, category theory, and software engineering. In general, diagrammatic notations allow the construction of images associated with an interpretation based on considering as significant some well-defined spatial relations among graphical tokens. These tokens either derive from conventional notations employed in a user community or are elements specially designed to convey some meaning. The notations serve the purpose of defining the (types of) entities one is interested in and the types of relations among these entities. Hence, types must be distinguishable from one another and no ambiguity may arise as to their interpretation. Moreover, the set of spatial relations to be considered must be clearly defined, and the holding of any relation among any set of elements must be decidable.

The evolution of diagrammatic notations usually follows a pattern that, from their usage as illustrations of sentences written in some formal or natural language, leads to the definition of "modeling languages". These languages are endowed with rules for the construction of "visual sentences" from some elementary graphical components, and for interpreting the meaning of these sentences with respect to the modeled domain, up to rules for mapping the behaviour of the modeled systems onto the behaviour of the visual elements in the model.

The problem researchers in the area are faced with is to precisely define the lexicons, the syntaxes and the semantics of the diagrams forming visual models. These definitions must allow the non-ambiguous interpretation of a diagram, as well as provide a basis for the construction of systems facilitating the modelers in defining their models or even in creating new languages.

Several traditions and research lines compete. Several authors have studied abstract definitions of visual languages as a basis to define formal semantics on them, or for the construction of visual tools. Abstract definitions of visual languages can be introduced in the form of graphs [6] or hypergraphs [12], possibly distinguishing between low-level and high-level interpretations occurring on distinct graphs [1,13]. Extensions of traditional Chomsky-like grammars have been employed for the definition and parsing of visual languages, working on string [4] or multiset [11] representations of the visual sentences. Algebraic specifications

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 445–449, 2002.
© Springer-Verlag Berlin Heidelberg 2002

have been adapted to express the admissible structures and transformations of visual elements and sentences [3,5,14].

Metamodel approaches are gaining interest following their success in defining the UML different visual languages [7]. A metamodel approach is implicit in most generators of diagrammatic editors, in which at least an abstract notion of graphical object has to be defined. Most such generators, however, are based on the translation of some formal syntax into a set of procedures controlling user interaction, and on constructing the semantic interpretation of the diagram in a parsing process. Examples of such tools where the formal syntax is some variant of graph rewriting are DiaGen [12] (based on hypergraph rewriting) and GenGEd [1] (in which constraints are used to define lexical elements). A constructive approach can also be derived from metamodels, defining ways to build correct visual sentences by manipulating the concrete notation, rather than its abstract structure, as done in GenIAL [2].

Concrete notations are also explored to investigate their ability to support different kinds of inferences and reasoning [8,9]. A perspective on the wealth of existing different approaches can be obtained from surveys such as the one in [10], or from the series of proceedings of conferences such as the ones on Visual Languages, now Human Centric Computing Languages and Environments, on Graph Transformations, on Diagrammatic Reasoning, or journals such as the Journal of Visual Language and Computing.

In its 3rd edition, now a satellite workshop of ICGT 2002 after being linked with ICALP, the Workshop on Graph Transformation and Visual Modeling Techniques has attracted 19 submissions covering almost the whole spectrum of approaches mentioned here. Of these, 12 papers have been selected for presentation in four paper sessions. The four sessions together with their papers are detailed in the following.

The workshop program is completed by an invited talk by Martin Gogolla (University of Bremen, Germany) and a special session on *Case Studies for Visual Modelling Techniques*, held jointly with the ICGT 2002-Workshop on Software Evolution through Transformations (SET'02).

2 Session on Geometry and Visualization

The first paper of this session by *Power* and *Tourlas* [15] is basically a criticism of visual language specification techniques that ignore the geometry of the visual language. Instead, the paper advocates consideration for concrete geometric aspects when dealing with visual languages, to complement reasoning at the abstract level, as performed on some form of abstract syntax. The authors show that geometry simplifies certain arguments when formally defining visual languages.

Simplifying the process of understanding of abstract problems by visualization is also the topic of the paper by *Bauderon* and *Mosbah* [16]. The paper presents a toolkit for designing, implementing and visualizing distributed algorithms described by graph relabelling systems. It is useful for pedagogical pur-

poses to explain the execution of distributed algorithms, and also for researchers in distributed algorithms who require tools for tests and experiments.

3 Session on Euler/Venn Diagrams

Among the earliest diagrammatic notations in mathematics, Euler and Venn diagrams have recently been the foundation of new visual languages (e.g., spider diagrams [9]). The paper by *Flower*, *Howse*, and *Taylor* [17] investigates nesting in Euler diagrams, and how nesting affects the interpretation and construction of such diagrams. Dual representation of Euler diagrams by graphs play an important role in those considerations which are particularly helpful when building tools for visual languages based on Euler diagrams.

The other paper of this session by *Swoboda* and *Allwein* [18] is also on Euler and Venn diagrams, and it uses graphs as a dual representation, too. However, since this paper investigates reasoning processes with Euler/Venn diagrams, they use directed acyclic graphs (DAGs). First order logic reasoning rules are represented as DAG transformation rules.

4 Session on Frameworks and Tools

Costagliola et al. [19] propose an approach to the construction of meta-CASE workbenches based on the UML metamodelling techniques. UML class diagrams specifying the abstract syntax of visual languages, together with OCL constraints, are used to produce skeletons of extended positional grammars [4] describing the concrete syntax. These are further processed to generate a parser which is at the core of an environment in which a modeller can construct sentences of the specified visual language.

A metamodel approach is also used by *de Lara et al.* [20] where metamodels can describe not only the characteristics of a visual language, but also characteristics of the interface by which the user is going to create sentences of the specified language. Graph-rewriting rules are used to manipulate models so as to allow user customisation of the generated environment.

In Varró's paper [21] graph rewriting rules are used to specify the semantics of diagrammatic sentences representing users' models, abstractly represented as graphs. Graph transformation systems are then transformed into Kripke structures in order to perform verification of properties of the models, using model checking tools.

In the last paper of this session, *Ermel* and *Bardohl* [22] report on progresses in the GenGEd framework. Two-level graph transformations are exploited to simultaneously specify modifications at the abstract and concrete level of a visual sentence. This feature is employed to animate concrete representations according to the evolution of the underlying model.

5 Session on Components, Models, and Semantics

The session is opened by the work by *Braun* and *Marschall* [23] who propose an approach to the integration of data models possibly deriving from heterogeneous

applications. The metamodels of the applications are expressed as class diagrams, so that the transformation of a model into another one is expressed by rules on instance diagrams together with a system of equations resolved by assigning values to attributes of elements in instance diagrams representing the models of the concrete systems.

The work by *Ehrig et al.* [24] presents an abstract component framework based on High-Level Replacement Systems in which a component is specified by its import and export interfaces with the rest of the environment and its body, plus two connections: one between the import and the body, and one from the body to the export interface. Composition is achieved by connecting the export interface of the first component to the import interface of the second. The body of the resulting component extends the body of the first component according to the import connection of the second component.

In the third paper, *Hausmann et al.* [25] extend the Dynamic Meta Modeling framework for the specification of UML semantics to accommodate time, so as to provide a formal semantics to models describing multimedia applications using continuous media, such as video. Time is described by local logical clocks, and graph transformations are enriched with rules to update local clocks in a consistent way. This approach allows also the specification of situations in which several transformations could be performed concurrently.

The session is closed by *Roubtsova* and *Kuiper* [26], who introduce an approach to inheritance of component specifications, considered as *interface suites* formed by an interface role diagram and a set of sequence diagrams. Sequence diagrams are translated to suitable process algebras so that checking if one component inherits from another is reduced to check if the corresponding processes satisfy the inheritance relation.

Acknowledgements

We thank members of the programming committee and additional reviewers for their careful evaluation of the submissions. Their names can be found on the GT-VMT'02 web page www2.cs.fau.de/GTVMT02/ and in the upcoming special issue of Electronic Notes of Theoretical Computer Science where the accepted papers will appear.

References

1. R. Bardohl, T. Schultzke, G. Taentzer. Visual Language Parsing in GenGEd. *Proc. 2nd Int. Workshop on Graph Transformation and Visual Modeling Techniques GT-VMT'01.* In *Electronic Notes in Theoretical Computer Science* 50:3, 2001.
2. P. Bottoni, M.F. Costabile, P. Mussio. Specification and Dialogue Control of Visual Interaction through Visual Rewriting Systems. *ACM TOPLAS*, 21(6):1077-1136, 1999.
3. A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation - Part I: Basic Concepts and Double Pushout Approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1997, pp. 163–245.

4. G. Costagliola, A. De Lucia, S. Orefice, G. Tortora. A Framework of Syntactic Models for the Implementation of Visual Languages. In *Proc. 1997 IEEE Symposium on Visual Languages*, pp. 8–67, 1997.
5. H. Ehrig, R. Heckel, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. Algebraic Approaches to Graph Transformation - Part II: Single Pushout Approach and Comparison with Double Pushout Approach. In *Handbook of Graph Grammars and Computing by Graph Transformation*, World Scientific, 1997, pp. 247–312.
6. M. Erwig. Abstract Syntax and Semantics of Visual Languages. *Journal of Visual Languages and Computing*. 9(5):461–483, 1998.
7. R.I. Ferguson, A. Hunter, C. Hardy. MetaBuilder: The Diagrammer's Diagrammer. In M. Anderson, P. Cheng, V. Haarslev eds. *Theory and Application of Diagrams*. Springer, 2000, pp. 407–421.
8. C. Gurr. On the Isomorphism, or Lack of It, of Representations. In *Visual Language Theory*. K. Marriott, B. Meyer eds., pp. 293–305, Springer, 1998.
9. J. Howse, F. Molina, S. Kent, J. Taylor. Reasoning with Spider Diagrams. In *Proc. IEEE Symp. on Visual Languages '99*, pp. 138–145, IEEE CS Press, 1999.
10. K. Marriott, B. Meyer, K. Wittenburg. A survey of visual language specification and recognition. In *Visual Language Theory*. K. Marriott, B. Meyer eds., pp. 5–85, Springer, 1998.
11. K. Marriott, B. Meyer. On the classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):374–402, 1997.
12. M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming*, 44(2):157–180, 2002.
13. J. Rekers, A. Schürr. Defining and Parsing Visual Languages with Layered Graph Grammars. *Journal of Visual Languages and Computing*, 8(1):27–55, 1998.
14. D. Wang, H. Zeevat. A Syntax Directed Approach to Picture Semantics. In *Visual Language Theory*. K. Marriott, B. Meyer eds., pp. 307–323, Springer, 1998.
15. J. Power, K. Tourlas. *On the Geometric Modelling of Visual Languages*, GT-VMT'02.
16. M. Bauderon, M. Mosbah. *A Unified Framework for Designing, Implementing and Visualizing Distributed Algorithms*, GT-VMT'02.
17. J. Flower, J. Howse, J. Taylor. *Nesting in Euler diagrams*, GT-VMT'02.
18. N. Swoboda, G. Allwein. *Using DAG Transformations to Verify Euler/Venn Homogeneous and Euler/Venn FOL Heterogeneous Rules of Inference*, GT-VMT'02.
19. G. Costagliola, V. Deufemia, F. Ferruci, C. Gravino. *Exploiting Visual Languages Generation and UML Meta Modelling to Construct Meta-CASE Workbenches*, GT-VMT'02.
20. J. de Lara, H. Vangheluwe, M. Alfonseca. *Using Meta-Modelling and Graph-Grammars to Create Modelling Environments*, GT-VMT'02.
21. D. Varró. *Towards symbolic analysis of visual modeling languages*, GT-VMT'02.
22. C. Ermel, R. Bardohl. *Multipe Views of Visual Behavior Models in GenGED*, GT-VMT'02.
23. P. Braun, F. Marschall. *Transforming Object Oriented Models with BOTL*, GT-VMT'02.
24. H. Ehrig, F. Orejas, B. Braatz, M. Klein, M. Piirainen. *A Component Framework Based on High-Level Replacement Systems*, GT-VMT'02.
25. J.H. Hausmann, R. Heckel, S. Sauer. *Dynamic Meta Modeling with Time: Specifying the Semantics of Multimedia Sequence Diagrams*, GT-VMT'02.
26. E.E. Roubtsova, R. Kuiper. *Process semantics for UML component specifications to assess inheritance*, GT-VMT'02.

Workshop on Software Evolution through Transformations: Towards Uniform Support throughout the Software Life-Cycle

Reiko Heckel[1], Tom Mens[2,⋆], and Michel Wermelinger[3]

[1] Universität Paderborn, Germany
reiko@upb.de
[2] Vrije Universiteit Brussel, Belgium
tommens@vub.ac.be
[3] Universidade Nova de Lisboa and ATX Software SA, Lisbon, Portugal
mw@di.fct.unl.pt

Abstract. Evolution (incremental change) of software is pre-dominant over development from scratch. Transformations provide a general and uniform view of incremental software development. Based on this unifying view, this workshop provides a forum to discuss both the phenomenon of evolution as such and its support by techniques and tools. Thereby, we cover: the transformation of different artifacts, like models, code, or data, different transformation techniques, like graph transformation, term-rewriting, logic programming, etc., different motivations for transformations, like forward, reverse, and re-engineering, as well as different semantic interpretations and combinations of transformations.
Moreover, the workshop features a session on case studies for visual modelling techniques, held jointly with GT-VMT, which is open to all participants of the conference and includes problems related to evolution as one specific aspect.

1 Motivation

Businesses, organisations and society at large are increasingly reliant on software at all levels. An intrinsic characteristic of software addressing a real-world application is the need to evolve [9]. Such evolution is inevitable if the software is to remain satisfactory to its stakeholders.

Changes to software artifacts and related entities tend to be progressive and incremental, driven, for example, by feedback from users and other stakeholders. Changes may be needed for a variety of reasons, such as bug reports, requests for new features or, more generally, changes of functional requirements, or by the need to adapt to new technology, e.g., to interface to other systems. In general,

⋆ Tom Mens is a postdoctoral fellow of the Fund for Scientific Research - Flanders (Belgium)

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 450–454, 2002.
© Springer-Verlag Berlin Heidelberg 2002

evolutionary characteristics are inescapable when the problem to be solved or the application to be addressed belongs to the real world.

Transformations of artifacts like models, schemata, data, program code, or software architectures provide a uniform and systematic way to express and reason about the evolution of software systems. Literally, all activities that lead to the creation or modification of documents have a transformational aspect, i.e., they change a given structure into a new one according to pre-defined rules. In most cases, these rules manifest themselves as user-invoked operations in CASE tools or program editors. More abstract examples include rules for model refinement, for translating models to code, for recovering designs from legacy systems, or for refactoring.

2 Workshop Objectives

This workshop aims at providing a forum for the discussion of transformational techniques in software evolution with particular focus on approaches that are generally applicable throughout the software development life-cycle. Thereby, we distinguish two co-existing, complementary and mutually reinforcing views of the evolution theme. The more widespread view focuses on *the how of software evolution*, emphasising the methods and means by which software is evolved [1,11]. The other focuses on *the what and why of the evolution phenomenon*, its nature and underlying drivers [7,9]. Being mutually supportive, both views are required. A better understanding of the phenomenon will necessarily lead to more appropriate ways of achieving evolution. Both views may be supported through the general concept of *transformation*, i.e., the manual, interactive, or automatic manipulation of artifacts according to pre-defined rules, either as a conceptual abstraction of human software engineering activities, or as the implementation of mappings on and between between modelling and programming languages.

3 Workshop Topics

The workshop is scheduled for two half days and includes two sessions with invited talks by Stuart Kent [8] and Michael Löwe [10] as well as six presentations of position papers in two regular sessions on different *transformation techniques* [14,15,16] and on the *compatibility of transformations* [3,5,13]. A selection of the position papers will appear in a volume of the Electronic Notes in Theoretical Computer Science dedicated to the workshop.

The variety of problems addressed by the invited presentations and position papers underlines the generality of transformation as a concept to support and explain the evolution of software artifacts. Next, we summarise the issues raised by considering the following five questions.

3.1 What Is Transformed?

The artifacts subject to transformation include models, code, and data represented in different ways: concretely as XML documents [10,14], Smalltalk ob-

jects [15], or terms of a functional [14] or logic [16] programming language, or abstractly as graphs [10,3,5,13] or instances of a meta model [8].

Concrete representations are the basis for *implementing* transformations and thus providing tool support for (the *how*) of software evolution. Abstract presentations are inevitable for reasoning in a uniform way about transformations which work at different levels and on different concrete representations, thus contributing to a general theory of the *what and why* of evolution.

3.2 How Are Transformations Specified and Applied?

In correlation with different representations of artifacts, different transformation techniques are applied and discussed: XSLT to transform XML documents [10,14,15], programmed transformations to manipulate Smalltalk objects [15], term rewriting [14], logic programming [16], and graph transformation [10,3,5,13].

Rule-based transformations are often non-deterministic. They may be applied in an automatic way, resulting in a translation or reduction process, or interactively like editor operations. Automatic transformations are more useful to support specific development activities, interactive ones are more generally applicable as they provide a universal view of development activities as manipulation of artifacts.

3.3 Why Are Artifacts Transformed?

Depending on *what* is transformed, the reason (i.e., *why*) for the transformation may be:

forward engineering: to instantiate a framework [16], transform a model into an implementation or refine it by a more detailed model [8], or to manipulate models or code, e.g., to support new features or correct errors [5,13],

reverse engineering: to recover the design of an existing system (in the sense of reverse engineering), e.g., by extracting a conceptual schema from a data base [14],

re-engineering: to improve models or code through refactoring without changing their semantics [3], and to transform the corresponding data to migrate it to the new version [10]

The reason for the transformation is closely related to the next question.

3.4 What Is the Semantics of Transformations?

The semantics of transformations can be expressed as the relation between the semantics of the given and the transformed artifacts. It is therefore dependent on the semantics we associate with the entities we transform.

Refactorings, for example, are usually defined as behaviour-preserving transformation with respect to a notion of behaviour which is informally defined and which accounts only for certain aspects of the execution of a program, like access

to variables, method calls, etc. Schema transformations may be classified according to the semantic relation between the given and the transformed schema into capacity preserving, extending, or reducing transformations, etc.

In general, the semantics of transformations, even if expressed informally, reveals the purpose of (otherwise purely structural) manipulations of models, program code, or data. Often, however, semantics is left implicit or, reversing the perspective, considered as a result of the transformation rather than a primary concept (for example, if transformations are used to describe refinement or equivalence of models).

3.5 How Are Transformations Combined or Coordinated?

Software development consists of transformations with different purpose and at different levels (see above). For example, re-engineering can be described as a combination of reverse engineering, design-level evolution, and forward engineering [14].

Co-evolution requires synchronized transformations of different views, or of models and code [3,13]. Refactorings may be applied at the level of code or of models, and in this case they have to be coordinated to preserve consistency.

Run-time evolution requires coordination of computations (i.e., transformation of data and control state) and evolution (transformation of the program itself) [5].

4 Joint Session on Case Studies for Visual Modeling Techniques

In cooperation with the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2002), this session is organized as a forum to propose problems from different domains as case studies for the evaluation of visual modeling techniques. Once a set of such problems has been defined, a call shall be issued to invite interested researchers to approach these problems with their favorite techniques and tools.

On the basis of the submitted case studies, visual modeling techniques and tools shall be compared, pointing out their complementarity. This shall serve as a basis for combining and integrating different techniques and tools and providing guidance for their use in different domains.

These activities shall be part of the work proposed for the SegraVis research training network [2] with the objective *to employ and improve visual modeling techniques in specific application domains*, including (but not limited to)

- modeling support for software evolution and refactoring
- modeling of component-based software architectures
- specification of applications with mobile soft- and hardware

Beside a general discussion of these ojectives, the session will consist in presentations of two submitted case studies [4,12] and position statements by the SegraVis objective coordinators.

Acknowledgement

This workshop is organised as an official activitiy of the scientific research network *Foundations of Software Evolution*, funded by the Fund for Scientific Research - Flanders (Belgium). The workshop is also supported by the European Research Training Network SegraVis *Syntactic and Semantic Integration of Visual Modelling Techniques*.

References

1. *International Workshops on Principles of Software Evolution*, 1998-2002.
2. SegraVis — Syntactic and Semantic Integration of Visual Modelling Techniques, 2002-2006. EU Research Training Network (RTN), `www.upb.de/cs/ag-engels/ag_engl/Segravis`.
3. P. Bottoni, F. Parisi-Presicce, and G. Taentzer. Coordinated distributed diagram transformation for software evolution. In Heckel et al. [6]. Position paper.
4. S. Demeyer, D. Janssens, and T. Mens. Case study: Simulation of a LAN, 2002. Joint case study session of GT-VMT / SET 2002, `www.upb.de/cs/ag-engels/Conferences/ICGT02/CS`.
5. L. Groenewegen, J.H. Hausmann, and R. Heckel. Evolution on-the-fly as coordinated transformation of system description and state. In Heckel et al. [6]. Position paper.
6. R. Heckel, T. Mens, and M. Wermelinger, editors. *ICGT 2002 Workshop on Software Evolution Through Transformations: Towards Uniform Support Throughout the Software Life-Cycle*, Barcelona, Spain, October 2002. Selected papers to appear in volume 72.4 of Electronic Notes in TCS.
7. C. Kemerer and S. Slaughter. An empirical approach to studying software evolution. *IEEE Trans. Software Engineering*, 1999.
8. S. Kent. Invited talk. In Heckel et al. [6]. Selected papers to appear in volume 72.4 of Electronic Notes in TCS.
9. M.M. Lehman and J.F. Ramil. Software evolution. invited keynote paper. In *International Workshop on Principles of Software Evolution, Vienna, Austria*, September 2001. A revised and extended version on an article to appear in Marciniak J. (ed.), Encyclopedia of Software Engineering, 2nd. Ed., Wiley, 2002.
10. M. Löwe. Invited talk. In Heckel et al. [6]. Selected papers to appear in volume 72.4 of Electronic Notes in TCS.
11. T. Mens and M. Wermelinger, editors. *Workshop on Formal Foundations of Software Evolution*, Lisbon, Portugal, 2001.
12. M. Özhan, M. Piirainen, and M. Klein. Case study: Agent-based material flow, 2002. Joint case study session of GT-VMT / SET 2002, `www.upb.de/cs/ag-engels/Conferences/ICGT02/CS`.
13. J. Padberg. Basic ideas for transformations of specification architectures. In Heckel et al. [6]. Position paper.
14. J. Pérez, V. Anaya, J.M. Cubel, and J.Á. Ramos. Data reverse engineering of legacy databases to object oriented conceptual schemas. In Heckel et al. [6]. Position paper.
15. N. Revault. Model transformation based on production rules. In Heckel et al. [6]. Position paper.
16. T. Tourwé and T. Mens. High-level transformations to support framework-based software development. In Heckel et al. [6]. Position paper.

Workshop on Logic, Graph Transformations and Discrete Structures

Bruno Courcelle and Pascal Weil

LaBRI, Université Bordeaux-1 and CNRS
351 cours de la Libération
F-33405 Talence Cedex, France
{bruno.courcelle,pascal.weil}@labri.fr

Presentation of the Workshop

Logic plays an essential role in Fundamental Computer Science, especially in Complexity Theory, in Semantics, and in the Theory of Formal Languages. This includes the description of sets of finite and infinite words, trees, graphs and related discrete structures by grammars, equation systems, rewriting systems, automata, and logical formulas. Transformations of such structures are also essential. They too can be described by automata, by rewriting systems or by logical formulas.

Monadic Second-Order logic (MS-logic for short), the extension of First-Order logic with variables denoting subsets of the considered domain is especially useful in this respect. It is an essential tool for dealing with context-free graph grammars, somewhat similar to finite-state automata with respect to context-free (word) grammars. The equivalence of MS-formulas and finite-state automata on words and trees is a deep result having many applications. For graphs, we do not have equivalence but an implication: MS-definablity implies recognizability (by finite congruences, not by automata). The intersection of a context-free set of graphs and an MS-definable one is context-free. Furthermore, context-free sets of graphs are generated from binary trees by MS-Transductions, like context-free languages are generated from the Dyck language (which encodes finite trees by means of parentheses) by rational transductions. MS-logic is decidable on each context-free set of graphs. Finally, MS-definable graph properties (in particular NP-complete properties like 3-colorability or Hamiltonicity) are decidable in linear time on graphs of tree-width at most k (the constant depends on k).

Thus, there is a vast body of results showing intimate relations between context-free graph grammars and Monadic Second-Order logic.

The purpose of the workshop is to survey recent extensions of these results, in various directions: partially ordered sets and applications, efficient computations of some combinatorial graph invariants, complexity of graph properties expressible in MS-logic or some sublanguages of it, and hierarchical descriptions of graphs.

A. Corradini et al. (Eds.): ICGT 2002, LNCS 2505, pp. 455–457, 2002.
© Springer-Verlag Berlin Heidelberg 2002

Partial Orders

One extension consists in considering partial orders instead of graphs. Grammars, automata, logical formalisms have been developped independently of Graph Grammar Theory for dealing with partial orders, with intended applications to traces and to Petri net languages. The survey by B. Courcelle and P. Weil connects graph grammars to grammars generating posets via the notion of Hasse diagram. D. Kuske considers in detail first-order definable poset languages, and proposes characterizations of these languages in algebraic terms – with special attention to languages of series-parallel posets and of Message Sequence Charts (MSC). MSC were introduced by system designers to model the communication dimension of distributed protocols. They can be viewed as particular posets, which are disjoint unions of totally ordered subsets. Efficient algorithms for the synthesis or the verification of specifications on MSC languages (say, expressed in MS logic) are of practical interest: this is the topic of A. Muscholl's lecture.

Combinatorics

Going from graphs to posets can be seen as a move to other combinatorial structures than graphs. Motivated by the study of knots, that can be seen as combinatorial descriptions of embeddings of graphs in the three-dimensional space, J. Makowsky considers cases where the evaluation of certain polynomials associated with graphs and in particular with graphs representing knots, can be done efficiently. Again MS logic is useful, as it can be used to express counting functions and not only graph properties.

Complexity

Complexity of model-checking is an essential issue, and the linearity of evaluation of MS formulas on structures of bounded tree-width is largely responsible for the popularity of this logical language. M. Frick will review the classical upperbounds on evaluation of first-order and monadic second-order formulas and will present improved upperbounds, in the framework of parametrized complexity (a theory developped by R. Downey and M.Fellows, based on the paradigm of NP-complete problems which are linearly solvable on graphs of tree-width at most k, where k is the parameter). He will also present new results on non-elementary lower bounds for MS logic under the standard assumption that P is not equal to NP.

D. Seese will consider the analysis of dynamic systems in terms of computation graphs and their tree-width, and explain why certain problems are undecidable or intractable. The presence of large grids in these graphs appears a major obstacle to efficient computability.

Hierarchical Descriptions of Graphs

Tree-width and clique-width are parameters yielding Parametric Tractability as they characterize the existence of certain hierarchical descriptions of graphs.

These hierarchical descriptions can be formulated in algebraic terms: graphs of tree-width (or clique-width) at most k, can be considered as the values of terms (hence can be described in terms of trees) written with symbols denoting certain operations on graphs. The linear evaluation of MS formulas is actually based on the existence of a finite-state tree-automaton recognizing the terms describing the graphs of tree-width (or clique-width) at most k satisfying this property. A. Glikson will present a unified presentation, in model-theoretical terms, of the known graph operations that fit this objective. He will discuss the minimal conditions that alternative graph operations should satisfy to yield graph grammars with good logical properties.

The relations between graph decompositions and the decidability of MS logic have been put in light by D. Seese who formulated the conjecture that every set of graphs in which the MS properties are decidable, is the image of a set of trees by an MS transduction, or equivalently, has bounded clique-width. This conjecture is still open and B. Courcelle will present different special cases for which it has been established.

Problem Sessions

Problem sessions will be open to the audience. Anybody wishing to present a problem can send a note to P. Weil or B. Courcelle (`bruno.courcelle@labri.fr` and `pascal.weil@labri.fr`) but problems coined on the spot can be presented as well.

Preliminary Program

Bruno Courcelle (LaBRI, Bordeaux) : On a conjecture by D. Seese

Markus Frick (LFCS, Edinburgh) : The complexity of first order and monadic second order logic

Alex Glikson (Technion, Haïfa) : Finite Model Theory characterization of some graph grammars

Dietrich Kuske (University of Leicester) : Algebraic characterizations of first-order logic on posets

Janos Makowsky (Technion, Haïfa) : Graph polynomials and generating functions definable in Monadic Second Order Logic

Anca Muscholl (LIAFA, Paris) : Specification and verification using Message Sequence Charts

Detlef Seese (AIFB, Karlsruhe) : Complexity and dynamic systems: tree-width, regularity and information flow

Pascal Weil (LaBRI, Bordeaux) : Partial orders, graph grammars and logic

Author Index

Abreu, Marcelo M. de, 254

Baldan, Paolo, 14
Bardohl, Roswitha, 30
Baresi, Luciano, 402
Bauderon, Michel, 45
Beck, J. Christopher, 60
Borkowski, Adam, 363
Bottoni, Paolo, 445

Carrère, Frédérique, 45
Courcelle, Bruno, 455

Demeyer, Serge, 286

Enders, Bettina E., 334
Ermel, Claudia, 30

Fernández, Maribel, 75

Gerber, Anna, 90
Ghezzi, Carlo, 1
Godard, Emmanuel, 106
Gyapay, Szilvia, 120

Habel, Annegret, 135
Hage, Jurriaan, 148
Harel, David, 6
Harju, Tero, 148, 430
Heckel, Reiko, 120, 161, 402, 450
Hoffmann, Berthold, 177

Janssens, Dirk, 286

Klempien-Hinrichs, Renate, 192
Knirsch, Peter, 207
Koch, Manuel, 223
König, Barbara, 14
Kuske, Sabine, 207
Küster, Jochen Malte, 161

Lara, Juan de, 239
Lawley, Michael, 90
Lima Reis, Carla A., 254

Mackie, Ian, 75
Marburger, André, 270
Mens, Tom, 286, 440, 450
Métivier, Yves, 106
Milner, Robin, 8
Minas, Mark, 302, 445
Montanari, Ugo, 317
Mosbah, Mohamed, 106

Nunes, Daltro J., 254

Padberg, Julia, 30, 334
Parisi-Presicce, Francesco, 223
Petre, Ion, 430
Pfaltz, John L., 351
Plump, Detlef, 135, 435
Prosser, Patrick, 60

Quites Reis, Rodrigo, 254

Raymond, Kerry, 90
Ribeiro, Leila, 317
Rozenberg, Grzegorz, 430

Schlebbe, Heribert, 254
Schürr, Andy, 363, 440
Selensky, Evgeny, 60
Sellami, Afif, 106
Steel, Jim, 90
Strüber, Frank, 302
Szuba, Janusz, 363

Taentzer, Gabriele, 161, 440

Vangheluwe, Hans, 239
Varró, Dániel, 120, 378

Weil, Pascal, 455
Welzl, Emo, 148
Wermelinger, Michel, 450
Westfechtel, Bernhard, 270
Woo, Sung-Ho, 393
Wood, Andrew, 90

Yang, Sung-Bong, 393