Electronic Communications of the EASST
Pre-Proceedings GT-VMT 2008

Pre-Proceedings of the Seventh International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2008)

A Satellite Event of

Guest Editors: Claudia Ermel, Reiko Heckel, Juan de Lara
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Preface

GT-VMT 2008 is the seventh workshop of a series that serves as a forum for all researchers and practitioners interested in the use of graph transformation-based notations, techniques and tools for the specification, modelling, validation, manipulation and verification of complex systems. Due to the variety of languages and methods used in different domains, the aim of the workshop is to promote engineering approaches that starting from high-level specifications and robust formalizations allow for the design and the implementation of such visual modeling techniques, hence providing effective tool support at the semantic level (e.g., for model analysis, transformation, and consistency management). The workshop gathers communities working on popular visual modelling notations like UML, Petri nets, Graph Transformation and Business Process/Workflow Models.

This year's workshop has an additional focus on visualization, simulation, and animation of models as means of providing an intuitive representation of both their static semantics and for the validation of model behavior. For this purpose, two invited talks have been scheduled, one from industry and the other from academia. The first invited speaker is Juha-Pekka Tolvanen, from MetaCase (Finland), and his presentation is entitled "Domain-Specific Modelling in Practice". The second invited speaker is Hans Vangheluwe from the Modelling, Simulation and Design Lab (McGill University of Montreal in Canada), whose presentation will be on the subject "Foundations of Modelling and Simulation of Complex Systems".

Regarding scientific contributions, we had 41 submissions, from which 24 were accepted. The topics of the papers range a wide spectrum, including model integration, verification of model transformations, object oriented notations, visual language processing and grid computing. The accepted papers balance theoretical and applied concepts, including tool issues. The workshop program has been organized in six technical sessions, in two days:

Saturday, March 29, 2008	**Sunday, March 30, 2008**
Model Transformations and Queries	Dynamic Reconfiguration
Distribution and Semantics	Verification and Programming
Analysis and Visualization	Case Studies and Tools

We would like to thank the members of the Program Committee and the secondary reviewers for their excellent work in selecting the papers of this workshop, they are listed below. We would also like to thank the organizing committee of ETAPS for their constant support.

February 2008.

Claudia Ermel, Reiko Heckel, Juan de Lara.

PC chairs of GT-VMT 2008.

Programme Chairs

Claudia Ermel (TU Berlin, Germany)
Reiko Heckel (University of Leicester, UK)
Juan de Lara (University Autónoma of Madrid, Spain)

Programme Committee

Paolo Baldan (Università degli Studi di Padova, Italy)
Paolo Bottoni (Università di Roma ”La Sapienza”, Italy)
Andrea Corradini (Università di Pisa, Italy)
Karsten Ehrig (University of Leicester, UK)
Gregor Engels (Universität Paderborn, Germany)
Claudia Ermel (TU Berlin, Germany)
Holger Giese (Universität Paderborn, Germany)
Reiko Heckel (University of Leicester, UK)
Gabor Karsai (Vanderbilt University, US)
Jochen Küster (IBM Zürich Research)
Juan de Lara (Universidad Autónoma de Madrid, Spain)
Mark Minas (Universität der Bundeswehr München, Germany)
Francesco Parisi-Presicce (Università di Roma ”La Sapienza”, Italy)
Arend Rensink (Universiteit Twente, The Netherlands)
Andy Schürr (Universität Darmstadt, Germany)
Gabriele Taentzer (Universität Marburg, Germany)
Daniel Varró (Budapest University of Technology and Economics, Hungary)
Martin Wirsing (Ludwig-Maximilians-Universität München, Germany)
Albert Zündorf (Universität Kassel, Germany)

External Reviewers

Jan-Christopher Bals, Luciano Baresi, Basil Becker, Clara Bertolissi, Enrico Biermann, Denes Bisztray, Florian Brieler, Roberto Bruni, Troels Damgaard, Ira Diethelm, Hartmut Ehrig, Alexander Foerster, Leif Geiger, Esther Guerra, Stephan Hildebrandt, Kathrin Hoffmann, Leen Lambers, Thomas Maier, Sonja Maier, Steffen Mazanek, Tony Modica, Stefan Neumann, Ulrike Prange, Carsten Reckord, Tim Schattkowsky, Andreas Seibel, Christian Soltenborn

Table of Contents

Session 1: Invited Talk

Domain-Specific Modeling in Practice 7
Juha-Pekka Tolvanen (MetaCase and University of Jyväskylä, Finland)

Session 2: Model Transformations and Queries

From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars 9
Frank Hermann, Hartmut Ehrig, Karsten Ehrig

Verifying Model Transformations by Structural Correspondence 23
Anantha Narayanan, Gabor Karsai

Extending Graph Query Languages by Reduction 37
Erhard Weinell

Improved Live Sequence Chart to Automata Translation for Verification 51
Rahul Kumar, Eric Mercer

Session 3: Distribution and Semantics

Composing Control Flow and Formula Rules for Computing on Grids 65
Paolo Bottoni, Nikolay Mirenkov, Yutaka Watanobe, Rentaro Yoshioka

A Graph-Based Semantics for UML Class and Object Diagrams 79
Anneke Kleppe, Arend Rensink

Graph Transformations for the Resource Description Framework 95
Benjamin Braatz, Christoph Brandt

Controlling resource access in Directed Bigraphs 109
Davide Grohmann, Marino Miculan

Interaction nets: programming language design and implementation 123
Abubaker Hassan, Ian Mackie, Shinya Sato

Session 4: Analysis and Visualization

Sufficient Criteria for the Applicability and Non-Applicability of Rule Sequences . . . 137
Leen Lambers, Hartmut Ehrig, Gabriele Taentzer

Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars 151
Florian Brieler, Mark Minas

A Static Layout Algorithm for DiaMeta 165
Sonja Maier, Mark Minas

Session 5: Invited Talk

Foundations of Modelling and Simulation of Complex Systems 179
Hans Vangheluwe (Modelling, Simulation and Design Lab, McGill University, Canada)

Session 6: Dynamic Reconfiguration

Dynamic Software Architectures Verification using DynAlloy 181
Antonio Bucchiarone, Juan Galeotti

Reconfiguration of Reo Connectors Triggered by Dataflow 195
Christian Koehler, David Costa, Jose Proenca, Farhad Arbab

Negative Application Conditions for Reconfigurable Place/Transition Systems 208
Alexander Rein, Ulrike Prange, Leen Lambers, Kathrin Hoffmann, Julia Padberg

Independence Analysis of Firing and Rule-based Net Transformations in Reconfigurable Object Nets 222
Enrico Biermann, Tony Modica

Session 7: Verification and Programming

The GP Programming System 235
Greg Manning, Detlef Plump

Type Checking C++ Template Instantiation by Graph Programs 249
Karl Azab, Karl-Heinz Pennemann

A Graph-Based Type Representation for Objects 263
Cong-Cong Xing

Using Graph Transformation Systems to Specify and Verify Data Abstractions 277
Luciano Baresi, Carlo Ghezzi, Andrea Mocci, Mattia Monga

Parsing of Hyperedge Replacement Grammars with Graph Parser Combinators 291
Steffen Mazanek, Mark Minas

Session 8: Case Studies and Tools

Visual Design and Reasoning with the Use of Hypergraph Transformations 305
Grażyna Ślusarczyk, Ewa Grabska, Truong Le

Graph Transformation Model of a Triangulated Network of Mobile Units 319
Stefan Gruner

Some Applications of Graph Transformations in Modeling of Mechanical Systems . . 332
Stan Zawiślak, Lukasz Szypula, Miroslaw Myśliwiec, Adam Jagosz

Domain-Specific Modeling in Practice

Juha-Pekka Tolvanen[1]

MetaCase
Ylistönmäentie 31
FI-40500 Jyväskylä
Finland
[1]jpt@metacase.com

Abstract

Languages and transformations have played a significant role in raising the level of abstraction for software development. A key to the success has been that work in the higher level of abstraction is automatically transformed to the lower level. Today, Domain-Specific Modeling (DSM) languages provide a viable solution for continuing to raise the abstraction level beyond coding, making development faster and easier. With DSM, the models are composed of elements representing concepts that are part of the domain world, not the code world. In many cases, full, final product code can be automatically generated from these high-level specifications using domain-specific code generators. This automation is possible because both the language and generators are domain-specific rather than general purpose: they are narrowed to address specific needs, often those of a single company and domain.

In this talk, we describe DSM and how it differs from other modeling and code generation approaches: a level of abstraction higher than in code, closer fit with the problem domain, and the transference of control from tool vendors to company expert developers. In particular, we describe the various roles transformations have between language specifications (metamodels), system/software specifications (models), and the final application code. Throughout the talk, we inspect DSM examples from industry to demonstrate DSM practices, and to answer questions companies adopting transformations face. This leads to suggestions for possible future research in this area.

From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars

Hartmut Ehrig[1] **, Karsten Ehrig**[2] **and Frank Hermann**[1]

[1] [ehrig, frank]@cs.tu-berlin.de
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

[2] karsten@mcs.le.ac.uk
Department of Computer Science
University of Leicester, United Kingdom

Abstract: Success and efficiency of software and system design fundamentally relies on its models. The more they are based on formal methods the more they can be automatically transformed to execution models and finally to implementation code. This paper presents model transformation and model integration as specific problem within bidirectional model transformation, which has shown to support various purposes, such as analysis, optimization, and code generation.

The main purpose of model integration is to establish correspondence between various models, especially between source and target models. From the analysis point of view, model integration supports correctness checks of syntactical dependencies between different views and models.

The overall concept is based on the algebraic approach to triple graph grammars, which are widely used for model transformation. The main result shows the close relationship between model transformation and model integration. For each model transformation sequence there is a unique model integration sequence and vice versa. This is demonstrated by a quasi-standard example for model transformation between class models and relational data base models.

Keywords: model transformation, model integration, syntactical correctness

1 Introduction

Whenever one can expect benefits out of different modeling languages for the same specific task there is a substantial motivation of combining at least two of the them. For this purpose it is useful to have model transformations between these modeling languages together with suitable analysis and verification techniques. In cases of bidirectional model transformation the support for the modeling process increases, for instance, if results of analysis can be translated backwards to mark the original source of deficiency or defect, respectively.

In [EEE⁺07] Ehrig et al. showed how to analyze bi-directional model transformations based on triple graph grammars [Sch94, KS06] with respect to information preservation, which is especially important to ensure the benefits of other languages for all interesting parts of models.

Triple graph grammars are based on triple rules, which allow to generate integrated models G consisting of a source model G_S, a target model G_T and a connection model G_C together with correspondences from G_C to G_S and G_T. Altogether G is a triple graph $G = (G_S \leftarrow G_C \rightarrow G_T)$. From each triple rule tr we are able to derive a source rule tr_S and a forward rule tr_F, such that the source rules are generating source models G_S and the forward rules allow to transform a source model G_S into its corresponding target model G_T leading to a model transformation from source to target models. On the other hand we can also derive from each triple rule tr a target rule tr_T and a backward rule tr_B, such that the target rules are generating target models G_T and backward rules transform target models to source models. The relationship between these forward and backward model transformation sequences was analyzed already in [EEE+07] based on a canonical decomposition and composition result for triple transformations.

In this paper we study the model integration problem: Given a source model G_S and a target model G_T we want to construct a corresponding integrated model $G = (G_S \leftarrow G_C \rightarrow G_T)$. For this purpose, we derive from each triple rule tr an integration rule tr_I, such that the integration rules allow to define a model integration sequence from (G_S, G_T) to G. Of course, not each pair (G_S, G_T) allows to construct such a model integration sequence. In our main result we characterize existence and construction of model integration sequences sequences from (G_S, G_T) to G by model transformation sequences from G_S to G_T. This main result is based on the canonical decomposition result mentioned above [EEE+07] and a new decomposition result of triple transformation sequences into source-target- and model integration sequences.

In Section 2 we review triple rules and triple graph grammars as introduced in [Sch94] and present as example the triple rules for model transformation and integration between class models and relational data base models. Model transformations based on our paper [EEE+07] are introduced in Section 3, where we show in addition syntactical correctness of model transformation. The main new part of this paper is model integration presented in Section 4 including the main results mentioned above and applied to our example. Related and future work are discussed in sections 5 and 6, respectively.

2 Review of Triple Rules and Triple Graph Grammars

Triple graph transformation [Sch94] has been shown to be a promising approach to consistently co-develop two related structures. Bidirectional model transformation can be defined using models consisting of a pair of graphs which are connected via an intermediate correspondence graph together with its embeddings into the source and target graph. In [KS06], Königs and Schürr formalize the basic concepts of triple graph grammars in a set-theoretical way, which was generalized and extended by Ehrig et. el. in [EEE+07] to typed, attributed graphs. In this section, we shortly review main constructions and relevant results for model integration as given in [EEE+07].

Definition 1 (Triple Graph and Triple Graph Morphism) Three graphs SG, CG, and TG, called source, connection, and target graphs, together with two graph morphisms $s_G : CG \rightarrow SG$ and $t_G : CG \rightarrow TG$ form a triple graph $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\rightarrow} TG)$. G is called *empty*, if SG, CG, and TG are empty graphs.

A triple graph morphism $m = (s,c,t) : G \to H$ between two triple graphs $G = (SG \overset{s_G}{\leftarrow} CG \overset{t_G}{\to} TG)$ and $H = (SH \overset{s_H}{\leftarrow} CH \overset{t_H}{\to} TH)$ consists of three graph morphisms $s : SG \to SH$, $c : CG \to CH$ and $t : TG \to TH$ such that $s \circ s_G = s_H \circ c$ and $t \circ t_G = t_H \circ c$. It is injective, if morphisms s, c and t are injective.

Triple graphs G are typed over a triple graph $TG = (TG_S \leftarrow TG_C \to TG_T)$ by a triple graph morphism $t_G : G \to TG$. Type graph of the example is given in Fig. 1 showing the structure of class diagrams in source component and relational databases in target component. Where classes are connected via associations the corresponding elements in databases are foreign keys. Though, the complete structure of correspondence elements between both types of models is defined via the connection component of TG. Throughout the example, originating from [EEE+07], elements are arranged left, center, and right according to the component types source, correspondence and target. Morphisms starting at a connection part are given by dashed arrow lines.

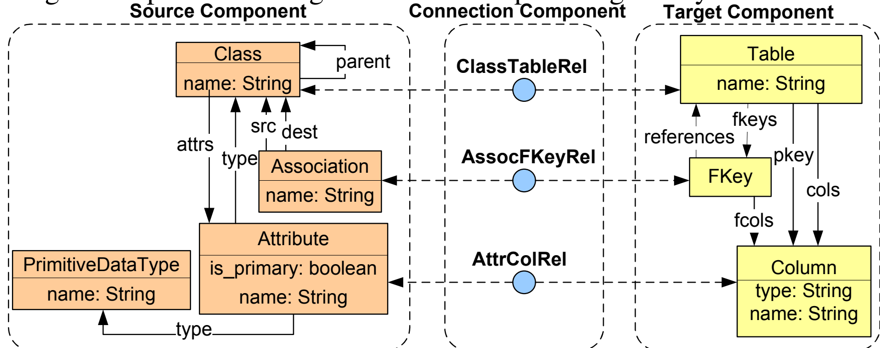


Figure 1: Triple type graph for *CD2RDBM* model transformation

A triple rule is used to build up source and target graphs as well as their connection graph, i.e. to build up triple graphs. Structure filtering which deletes parts of triple graphs, is performed by projection operations only, i.e. structure deletion is not done by rule applications. Thus, we can concentrate our investigations on non-deleting triple rules without any restriction.

Definition 2 (Triple Rule tr and Triple Transformation Step)
A *triple rule* tr consists of triple graphs L and R, called left-hand and right-hand sides, and an injective triple graph morphism $tr = (s,c,t) : L \to R$.

$$
\begin{array}{ccccccc}
L & = & (SL & \overset{s_L}{\longleftarrow} & CL & \overset{t_L}{\longrightarrow} & TL) \\
{\scriptstyle tr}\downarrow & & {\scriptstyle s}\downarrow & & {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle t} \\
R & = & (SR & \underset{s_R}{\longleftarrow} & CR & \underset{t_R}{\longrightarrow} & TR)
\end{array}
$$

Given a triple rule $tr = (s,c,t) : L \to R$, a triple graph G and a triple graph morphism $m = (sm, cm, tm) : L \to G$, called triple match m, a triple graph transformation step (TGT-step)$G \overset{tr,m}{\Longrightarrow} H$ from G to a triple graph H is given by three pushouts (SH, s', sn), (CH, c', cn) and (TH, t', tn) in category **Graph** with induced morphisms $s_H : CH \to SH$ and $t_H : CH \to TH$. Morphism $n = (sn, cn, tn)$ is called comatch.

$$
\begin{array}{ccccc}
 & SL & \longleftarrow & CL & \longrightarrow & TL \\
 & {\scriptstyle sm}\swarrow & & {\scriptstyle cm}\swarrow & & {\scriptstyle tm}\swarrow \\
G = (SG & \longleftarrow & CG & \longrightarrow & TG) & \\
 & SR & \longleftarrow & CR & \longrightarrow & TR \\
{\scriptstyle tr}\Downarrow \; {\scriptstyle s'}\downarrow & {\scriptstyle sn}\swarrow & {\scriptstyle c'}\downarrow & {\scriptstyle cn}\swarrow & {\scriptstyle t'}\downarrow & {\scriptstyle tn}\swarrow \\
H = (SH & \underset{s_H}{\longleftarrow} & CH & \underset{t_H}{\longrightarrow} & TH) &
\end{array}
$$

Moreover, we obtain a triple graph morphism $d : G \to H$ with $d = (s', c', t')$ called transformation morphism. A sequence of triple graph transformation steps is called triple (graph) transformation sequence, short: TGT-sequence. Furthermore, a triple graph grammar $TGG = (S, TR)$

consists of a triple start graph S and a set TR of triple rules. Given a triple rule tr we refer by $L(tr)$ to its left and by $R(tr)$ to its right hand side.

Remark 1 (gluing construction) *Each of the pushout objects SH, CH, TH in Def. 2 can be constructed as a gluing construction, e.g. $SH = SG +_{SL} SR$, where the S-components SG of G and SR of R are glued together via SL.*

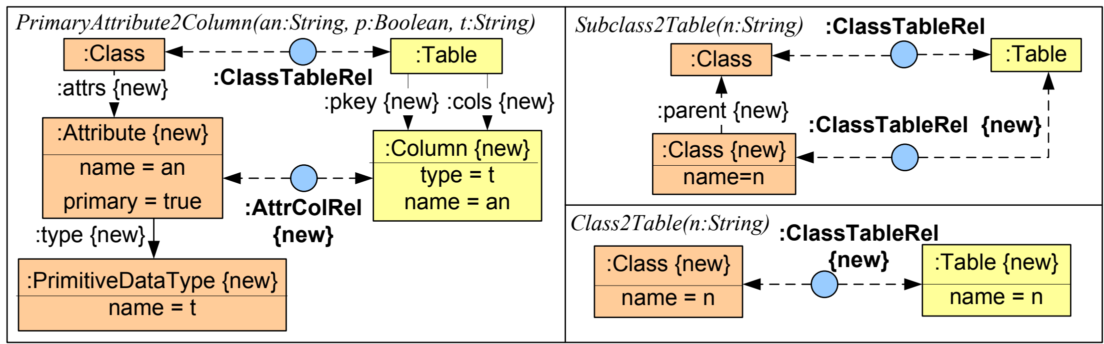


Figure 2: TGT-rules for *CD2RDBM* model transformation

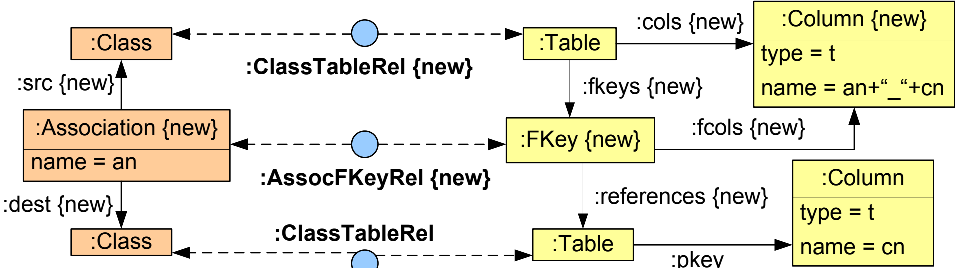


Figure 3: Rule $Association2ForeignKey(an : String)$ for *CD2RDBM* model transformation

Examples for triple rules are given in Fig. 2 and Fig. 3 in short notation. Left and right hand side of a rule are depicted in one triple graph. Elements, which are created by the rule, are labeled with "new" and all other elements are preserved, meaning they are included in the left and right hand side. Rule "*Class2Table*" synchronously creates a class in a class diagram with its corresponding table in the relational database. Accordingly the other rules create parts in all components. For rule "*PrimaryAttribute2Column*" there is an analogous rule "*Attribute2Column*" for translation of non primary attributes, which does not add the edge ":pkey" in the database component.

3 Model transformation

The triple rules TR are defining the language $VL = \{G \,|\, \emptyset \Rightarrow^* G \; via \; TR\}$ of triple graphs. As shown already in [Sch94] we can derive from each triple rule $tr = L \rightarrow R$ the following source and forward rule. Forward rules are used for model transformations from a model of a source language to models of the target language. Source rules are important for analyzing properties of forward transformations such as information preservation, presented in [EEE+07].

$$\begin{array}{ccccccc} L & = & (SL & \xleftarrow{s_L} & CL & \xrightarrow{t_L} & TL) \\ tr\downarrow & & s\downarrow & & c\downarrow & & \downarrow t \\ R & = & (SR & \xleftarrow[s_R]{} & CR & \xrightarrow[t_R]{} & TR) \end{array}$$

triple rule tr

$$\begin{array}{ccccc} (SR & \xleftarrow{s\circ s_L} & CL & \xrightarrow{t_L} & TL) \\ id\downarrow & & c\downarrow & & \downarrow t \\ (SR & \xleftarrow{s_R} & CR & \xrightarrow{t_R} & TR) \end{array}$$

forward rule tr_F

$$\begin{array}{ccccc} (SL & \longleftarrow & \emptyset & \longrightarrow & \emptyset) \\ s\downarrow & & \downarrow & & \downarrow \\ (SR & \longleftarrow & \emptyset & \longrightarrow & \emptyset) \end{array}$$

source rule tr_S

For simplicity of notation we sometimes identify source rule tr_S with $SL \xrightarrow{s} SR$ and target rule tr_T with $TL \xrightarrow{t} TR$.

Theses rules can be used to define a model transformation from source graphs to target graphs. Vice versa using backward rules - which are dual to forward rules - it is also possible to define backward transformations from target to source graphs and altogether bidirectional model transformations. In [EEE+07] we have shown that there is an equivalence between corresponding forward and backward TGT sequences. This equivalence is based on the canonical decomposition and composition result (Thm. 1) and its dual version for backward transformations.

Definition 3 (Match Consistency) Let tr_S^* and tr_F^* be sequences of source rules tri_S and forward rules tri_F, which are derived from the same triple rules tri for $i = 1,\ldots,n$. Let further $G_{00} \xRightarrow{tr_S^*} G_{n0} \xRightarrow{tr_F^*} G_{nn}$ be a TGT-sequence with (mi_S, ni_S) being match and comatch of tri_S (respectively (mi, ni) for tri_F) then match consistency of $G_{00} \xRightarrow{tr_S^*} G_{n0} \xRightarrow{tr_F^*} G_{nn}$ means that the S-component of the match mi is uniquely determined by the comatch ni_S $(i = 1,\ldots,n)$.

Theorem 1 (Canonical Decomposition and Composition Result - Forward Rule Case)

1. **Decomposition:** *For each* TGT*-sequence based on triple rules* tr^*
   (1) $G_0 \xRightarrow{tr^*} G_n$ *there is a canonical match consistent* TGT*-sequence*
   (2) $G_0 = G_{00} \xRightarrow{tr_S^*} G_{n0} \xRightarrow{tr_F^*} G_{nn} = G_n$ *based on corresponding source rules* tr_S^* *and forward rules* tr_F^*.
2. **Composition:** *For each match consistent transformation sequence (2) there is a canonical transformation sequence (1).*
3. **Bijective Correspondence:** *Composition and Decomposition are inverse to each other.*

Proof. See [EEE+07]. □

Now we want to discuss under which conditions forward transformation sequences $G_1 \xRightarrow{tr_F^*} G_n$ define a model transformation between suitable source and target languages. In fact we have different choices: On the one hand we can consider the projections $VL_S = proj_S(VL)$ and $VL_T = proj_T(VL)$ of the triple graph language $VL = \{G \,|\, \emptyset \Rightarrow^* G \; via \; TR\}$, where $proj_X$ is a projection defined by restriction to one of the triple components, i. e. $X \in \{S, C, T\}$. On the other hand we can use the source rules $TR_S = \{tr_S \,|\, tr \in TR\}$ and the target rules $TR_T = \{tr_T \,|\, tr \in TR\}$ to define the source language $VL_{S0} = \{G_S \,|\, \emptyset \Rightarrow^* G_S \; via \; TR_S\}$ and the target language $VL_{T0} = \{G_T \,|\, \emptyset \Rightarrow^* G_T \; via \; TR_T\}$. Since each sequence $\emptyset \Rightarrow^* G \; via \; TR$ can be restricted to a source sequence $\emptyset \Rightarrow^* G_S$ via TR_S and to a target sequence $\emptyset \Rightarrow^* G_T$ via TR_T we have $VL_S \subseteq VL_{S0}$ and

$VL_T \subseteq VL_{T0}$, but in general no equality. In case of typed graphs the rules in TR are typed over TG with $TG = (TG_S \leftarrow TG_C \rightarrow TG_T)$ and rules of TR_S and TR_T typed over $(TG_S \leftarrow \emptyset \rightarrow \emptyset)$ and $(\emptyset \leftarrow \emptyset \rightarrow TG_T)$, respectively. Since G_S and G_T are considered as plain graphs they are typed over TG_S and TG_T, respectively.

Given a forward transformation sequence $G_1 \xrightarrow{tr_F^*} G_n$ we want to ensure the source component of G_1 corresponds to the target component of G_n, i.e. the transformation sequence defines a a model transformation MT from VL_{S0} to VL_{T0}, written $MT : VL_{S0} \Rrightarrow VL_{T0}$, where all elements of the source component are translated. Thus given a class diagram as instance of the type graph in Fig. 1 all corresponding tables, columns and foreign keys of the corresponding data base model shall be created in the same way they could have been synchronously generated by the triple rules of TR. An example forward transformation is presented in [EEE+07]. Since $G_S \in VL_{S0}$ is generated by TR_S-rules we have a source transformation $\emptyset \Rightarrow^* G_S$ via TR_S. In order to be sure that $G_1 \xrightarrow{tr_F^*} G_n$ transforms all parts of G_1, which are generated by $\emptyset \Rightarrow^* G_S$, we require that $\emptyset \Rightarrow^* G_S$ is given by $\emptyset \xrightarrow{tr_S^*} G_1$ with $G_1 = (G_S \leftarrow \emptyset \rightarrow \emptyset)$, i.e. $proj_S(G_1) = G_S$ based on the same triple rule sequence tr^* as $G_1 \xrightarrow{tr_F^*} G_n$. Finally we require that the TGT-sequence $\emptyset \xrightarrow{tr_S^*} G_1 \xrightarrow{tr_F^*} G_n$ is match consistent, because this implies – by Fact 1 below – that $G_S \in VL_S$ and $G_T \in VL_T$ and that we obtain a model transformation $MT : VL_S \Rrightarrow VL_T$ (see Fact 1).

Definition 4 (Model Transformation) A model transformation sequence $(G_S, G_1 \xrightarrow{tr_F^*} G_n, G_T)$ consists of a source graph G_S, a target graph G_T, and a source consistent forward TGT-sequence $G_1 \xrightarrow{tr_F^*} G_n$ with $G_S = proj_S(G_1)$ and $G_T = proj_T(G_n)$.
Source consistency of $G_1 \xrightarrow{tr_F^*} G_n$ means that there is a source transformation sequence $\emptyset \xrightarrow{tr_S^*} G_1$, such that $\emptyset \xrightarrow{tr_S^*} G_1 \xrightarrow{tr_F^*} G_n$ is match consistent. A model transformation $MT : VL_{S0} \Rrightarrow VL_{T0}$ is defined by model transformation sequences $(G_S, G_1 \xrightarrow{tr_F^*} G_n, G_T)$ with $G_S \in VL_{S0}$ and $G_T \in VL_{T0}$.

Remark 2 *A model transformation $MT : VL_{S0} \Rrightarrow VL_{T0}$ is a relational dependency and only in special cases a function.*

This allows to show that $MT : VL_{S0} \Rrightarrow VL_{T0}$ defined above is in fact $MT : VL_S \Rrightarrow VL_T$

Fact 1 (Syntactical Correctness of Model Transformation MT) *Given $G_S \in VL_{S0}$ and $G_1 \xrightarrow{tr_F^*} G_n$ source consistent with $proj_S(G_1) = G_S$ then $G_T = proj_T(G_n) \in VL_T$ and $G_S \in VL_S$, i.e. $MT : VL_S \Rrightarrow VL_T$.*

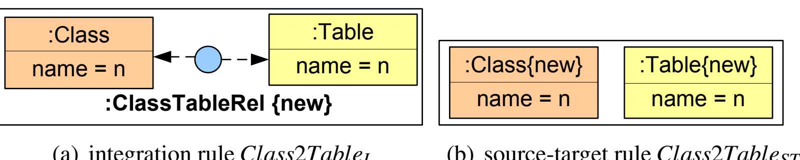
Proof. Given $G_1 \xrightarrow{tr_F^*} G_n$ source consistent, we have $\emptyset \xrightarrow{tr_S^*} G_1 \xrightarrow{tr_F^*} G_n$ match consistent and hence, by Theorem 1 above with $G_0 = \emptyset \xrightarrow{tr^*} G_n$ which implies $G_n \in VL$. Now we have $proj_S(G_n) = proj_S(G_1) = G_S \in VL_S$ and $proj_T(G_n) = G_T \in VL_T$. □

4 Model Integration

Given models $G_S \in VL_{S0}$ and $G_T \in VL_{T0}$ the aim of model integration is to construct an integrated model $G \in VL$, such that G restricted to source and target is equal to G_S and G_T, respectively, i.e. $proj_S G = G_S$ and $proj_T G = G_T$. Thus, given a class diagram and a data base model as instance of the type graph in Fig. 1 all correspondences between their elements shall be recovered or detected, respectively. Similar to model transformation we can derive rules for model integration based on triple rule tr. The derived rules are source-target rule tr_{ST} and integration rule tr_I given by

$$\begin{array}{ccc}
\begin{array}{ccccc}
(SL & \xleftarrow{s_L} & CL & \xrightarrow{t_L} & TL) \\
{\scriptstyle s}\downarrow & & {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle t} \\
(SR & \xleftarrow[s_R]{} & CR & \xrightarrow[t_R]{} & TR)
\end{array}
&
\begin{array}{ccccc}
(SL & \longleftarrow & \emptyset & \longrightarrow & TL) \\
{\scriptstyle s}\downarrow & & \downarrow & & \downarrow{\scriptstyle t} \\
(SR & \longleftarrow & \emptyset & \longrightarrow & TR)
\end{array}
&
\begin{array}{ccccc}
(SR & \xleftarrow{s \circ s_L} & CL & \xrightarrow{t \circ t_L} & TR) \\
{\scriptstyle id}\downarrow & & {\scriptstyle c}\downarrow & & \downarrow{\scriptstyle id} \\
(SR & \xleftarrow{s_R} & CR & \xrightarrow{t_R} & TR)
\end{array}
\\
\text{triple rule } tr & \text{source-target rule } tr_{ST} & \text{integration rule } tr_I
\end{array}$$

An example for both kinds of rules is given in Fig. 4 for the triple rule *Class2Table* in Fig. 2.



(a) integration rule $Class2Table_I$ (b) source-target rule $Class2Table_{ST}$

Figure 4: Derived rules for $Class2Table()$

Similar to the canonical decomposition of TGT-sequences $G_0 \xRightarrow{tr^*} G_n$ into source and forward transformation sequences we also have a canonical decomposition into source-target and integration transformation sequences of the form $\emptyset \xRightarrow{tr^*_{ST}} G_0 \xRightarrow{tr^*_I} G_n$. Such a sequence is called S-T-consistent, if the S- and T-component of the comatch of tri_{ST} is completely determined by that of the match of tri_I for $tr = (tri)_{i=1...n}$.

Theorem 2 (Canonical Decomposition and Composition Result - Integration Rule Case)

1. **Decomposition:** *For each* TGT-*sequence based on triple rules* tr^*
   (1) $G_0 \xRightarrow{tr^*} G_n$ *there is a canonical S-T-match consistent* TGT-*sequence*
   (2) $G_0 = G_{00} \xRightarrow{tr^*_{ST}} G_{n0} \xRightarrow{tr^*_I} G_{nn} = G_n$ *based on corresponding source-target rules* tr^*_{ST} *and integration rules* tr^*_I.
2. **Composition:** *For each S-T-match consistent transformation sequence (2) there is a canonical transformation sequence (1).*
3. **Bijective Correspondence:** *Composition and Decomposition are inverse to each other.*

In the following we give the proof of Theorem 2 which is based on the Local-Church-Rosser and the Concurrency Theorem for algebraic graph transformations (see [Roz97], [EEPT06]). The proof uses two lemmas, where the proof of the lemmas is given in [EEH08]. In Lemma 1 we show that a triple rule tr can be represented as concurrent production $tr_{ST} *_E tr_I$ of the corresponding source-target rule tr_{ST} and integration rule tr_I, where the overlapping E is equal

to $L(tr_I)$, the left hand side of tr_I. Moreover E-related sequences in the sense of the Concurrency Theorem correspond exactly to S-T-match-consistent sequences in Theorem 2. In Lemma 2 we show compatibility of S-T-match consistency with sequential independence in the sense of the Local-Church-Rosser-Theorem. Using Lemma 1 we can decompose a single TGT-transformation $G_0 \xRightarrow{tr} G_1$ into an S-T-match consistent sequence $G_0 \xRightarrow{tr_{ST}} G_{10} \xRightarrow{tr_I} G_1$ and vice versa. Lemma 2 allows to decompose TGT-sequences $G_0 \xRightarrow{tr^*} G_n$ into S-T-match consistent sequences $G_0 \xRightarrow{tr^*_{ST}} G_{n0} \xRightarrow{tr^*_I} G_n$ and vice versa.

All constructions are done in the category $\mathbf{TripleGraph_{TG}}$ of typed triple graphs and typed triple graph morphisms, which according to Fact 4.18 in [EEPT06] is an adhesive HLR category. This implies that the Local-Church-Rosser and Concurrency Theorem are valid for triple rules with injective morphisms (see Chapter 5 in [EEPT06]).

Lemma 1 (Concurrent Production $tr = tr_{ST} *_E tr_I$) *Let $E = L(tr_I)$ with $e1 = (id, \emptyset, id) : R(tr_{ST}) \to E$ and $e2 = id : L(tr_I) \to E$ then tr is given by the concurrent production $tr = tr_{ST} *_E tr_I$. Moreover, there is a bijective correspondence between a transformation $G_1 \xRightarrow{tr,m} G_2$ and match-consistent sequences $G_1 \xRightarrow{tr_{ST},m1,n1} H \xRightarrow{tr_I,m2,n2} G_2$, where $S-T$-match consistency means that the $S-$ and $T-$components of the comatch $n1$ and the match $m2$ are equal, i.e. $n1_S = m2_S$ and $n1_T = m2_T$. Construction of concurrent production:*

$$\begin{array}{ccccccc}
L(tr_{ST}) & \xrightarrow{tr_{ST}} & R(tr_{ST}) & & L(tr_I) & \xrightarrow{tr_I} & R(tr_I) \\
l\downarrow & (1) & & \searrow^{e1} \quad {}^{e2}\swarrow & & (2) & \downarrow r \\
L(tr) & & \xrightarrow[d1]{} & E & \xrightarrow[d2]{} & & R
\end{array}$$

$E-$concurrent rule

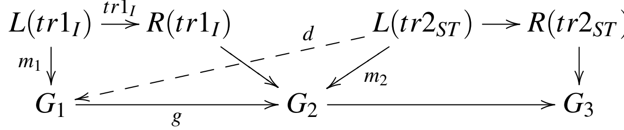
Lemma 2 (Compatibility of $S-T$-match consistency with independence)
Given the TGT-sequences on the right with independence in (4) and matches m_i, m'_i and comatches n_i, n'_i. Then we have:

$$\begin{array}{ccccccc}
 & & & G_{20} & & & \\
 & & \overset{tr2_{ST}}{\nearrow}{}_{m2',n2'} & & \overset{tr1_I}{\searrow}{}_{m1',n1'} & & \\
G_{00} & \xRightarrow[m0,n0]{tr1_{ST}} & G_{10} & & & G_{21} & \xRightarrow[m3,n3]{tr2_I} G_{22} \\
 & & \underset{m1,n1}{\overset{tr1_I}{\searrow}} & & \underset{m2,n2}{\overset{tr2_{ST}}{\nearrow}} & & \\
 & & & G_{11} & & &
\end{array}$$

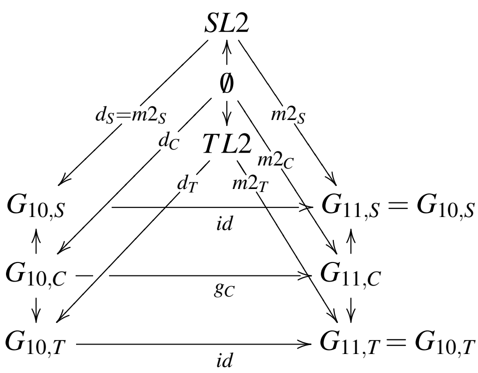
(1) $G_{00} \xRightarrow{tr1_{ST}} G_{10} \xRightarrow{tr1_I} G_{11}$ *$S-T$-match consistent* $\Leftrightarrow$
(2) $G_{00} \xRightarrow{tr1_{ST}} G_{10} \xRightarrow{tr2_{ST}} G_{20} \xRightarrow{tr1_I} G_{21}$ *$S-T$-match consistent*
and
(3) $G_{11} \xRightarrow{tr2_{ST}} G_{21} \xRightarrow{tr2_I} G_{22}$ *$S-T$-match consistent* $\Leftrightarrow$
(4) $G_{10} \xRightarrow{tr2_{ST}} G_{20} \xRightarrow{tr1_I} G_{21} \xRightarrow{tr2_I} G_{22}$ *$S-T$-match consistent*

Proof of Theorem 2.
1. Decomposition: Given (1) we obtain (for $n = 3$) by Leamma 1 a decomposition into triangles $(1), (2), (3)$, where the corresponding transformation sequences are $S-T$-match consistent.

$$\begin{array}{ccccccccccccc}
 & & & & & & G_{30} & & & & & & \\
 & & & & & \overset{tr3_{ST}}{\nearrow} & (6) & \overset{tr1_I}{\searrow} & & & & & \\
 & & & & G_{20} & & & & G_{31} & & & & \\
 & & & \overset{tr2_{ST}}{\nearrow} & (4) & \overset{tr1_I}{\searrow} & & \overset{tr3_{ST}}{\nearrow} & (5) & \overset{tr2_I}{\searrow} & & & \\
 & & G_{10} & & & & G_{21} & & & & G_{32} & & \\
 & \overset{tr1_{ST}}{\nearrow} & (1) & \overset{tr1_I}{\searrow} & & \overset{tr2_{ST}}{\nearrow} & (2) & \overset{tr2_I}{\searrow} & & \overset{tr3_{ST}}{\nearrow} & (3) & \overset{tr3_I}{\searrow} & \\
G_0 = G_{00} & & \xRightarrow[tr1]{} & & G_{11} & & \xRightarrow[tr2]{} & & G_{22} & & \xRightarrow[tr3]{} & & G_{33} = G_3
\end{array}$$

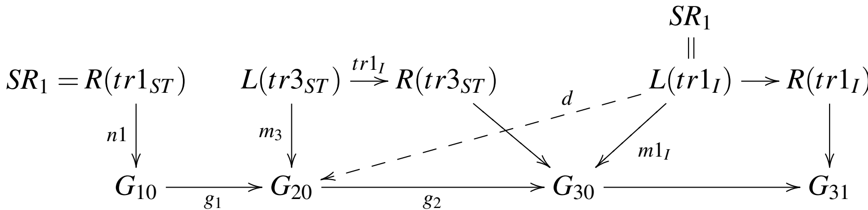


In the next step we show that $G_{10} \xrightarrow{tr1_I} G_{11} \xrightarrow{tr2_{ST}} G_{21}$ is sequentially independent leading by the Local Church Rosser Theorem to square (4) sequential independence in this case means existence of $d: L(tr2_{ST}) \to G_{10}$ with $g \circ d = m2$.



The diagram on the right shows that $d = (d_S, d_C, d_T) = (m2_S, \emptyset, m2_T)$ satisfies this property. $(1)-(4)$ leads to the following transformation sequence $G_{00} \xrightarrow{tr1_{ST}} G_{10} \xrightarrow{tr2_{ST}} G_{20} \xrightarrow{tr1_I} G_{21} \xrightarrow{tr2_I} G_{22} \xrightarrow{tr3_{ST}} G_{32} \xrightarrow{tr3_I} G_{33}$ which is again $S-T$-match consistent due to shift equivalence of corresponding matches in the Local Church Rosser Theorem (see Lemma 2). Similar to above we can show that $G_{21} \xrightarrow{tr2_I} G_{22} \xrightarrow{tr3_{ST}} G_{32}$ are sequentially independent leading to (5) and in the next step to (6) with corresponding $S-T$-match consistent sequences.

2. *Composition:* Vice versa, each $S-T$-match consistent sequence (2) leads to a canonical $S-T$-match consistent sequence of triangles $(1),(2),(3)$ and later by Lemma 1 to TGT-sequence (1). We obtain the triangles by inverse shift equivalence, where subsequence 1 as above is $S-T$-match consistent. In fact $S-T$-match consistency of (2) together with Lemma 2 implies that the corresponding sequences are sequentially independent in order to allow inverse shifts according to the Local Church Rosser Theorem. Sequential independence for (6) is shown below



By $S-T$-match consistency we have $m1_{I,S} = g2_S \circ g1_S \circ n1_S$. Define $d_S = g1_S \circ n1_S$, then $g2_S \circ d_S = g2_S \circ g1_S \circ n1_S = m1_{I,S}$ and similar for the T-component, while $d_C = m1_{I,C}$ using $g2_C = id$.
3. *Bijective Correspondence:* by that of the Local Church Rosser Theorem and Concurrency Theorem. □

Given an integration transformation sequence $G_0 \xrightarrow{tr_I^*} G_n$ with $proj_S(G_0) = G_S, proj_T(G_0) = G_T$ and $proj_C(G_0) = \emptyset$, we want to make sure that the unrelated pair $(G_S, G_T) \in VL_{S0} \times VL_{T0}$ is transformed into an integrated model $G = G_n$ with $proj_S(G) = G_S, proj_T(G) = G_T$. Of course this is not possible for all pairs $(G_S, G_T) \in VL_{S0} \times VL_{T0}$, but only for specific pairs. In any case $(G_S, G_T) \in VL_{S0} \times VL_{T0}$ implies that we have a source-target transformation sequence $\emptyset \Rightarrow^* G_0$ via $TR_{ST} = \{tr_{ST} \,|\, tr \in TR\}$. In order to be sure that $G_0 \xrightarrow{tr_I^*} G_n$ integrates all parts of G_S and G_T, which are generated by $\emptyset \Rightarrow^* G_0$, we require that $\emptyset \Rightarrow^* G_0$ is given by $\emptyset \xrightarrow{tr_{ST}^*} G_0$ based on

the same triple rule sequence tr^* as $G_0 \xrightarrow{tr_I^*} G_n$. Moreover, we require that the TGT-sequence $\emptyset \xrightarrow{tr_{ST}^*} G_0 \xrightarrow{tr_I^*} G_n$ is S-T-match consistent because this implies - using Theorem 2 - that $G_S \in VL_S, G_T \in VL_T$ and $G \in VL$ (see Theorem 2).

Definition 5 (Model Integration) A model integration sequence $((G_S,G_T),G_0 \xrightarrow{tr_I^*} G_n,G)$ consists of a source and a target model G_S and G_T, an integrated model G and a source-target consistent TGT-sequence $G_0 \xrightarrow{tr_I^*} G_n$ with $G_S = proj_S(G_0)$ and $G_T = proj_T(G_0)$.
Source-target consistency of $G_0 \xrightarrow{tr_I^*} G_n$ means that there is a source-target transformation sequence $\emptyset \xrightarrow{tr_{ST}^*} G_0$, such that $\emptyset \xrightarrow{tr_{ST}^*} G_0 \xrightarrow{tr_I^*} G_n$ is match consistent. A model integration $MI : VL_{S0} \times VL_{T0} \Rrightarrow VL$ is defined by model integration sequences $((G_S,G_T),G_0 \xrightarrow{tr_I^*} G_n,G)$ with $G_S \in VL_{S0}$, $G_T \in VL_{T0}$ and $G \in VL$.

Remark 3 *Given model integration sequence* $((G_S,G_T),G_0 \xrightarrow{tr_I^*} G_n,G)$ *the corresponding source-target* TGT*-sequence* $\emptyset \xrightarrow{tr_{ST}^*} G_0$ *is uniquely determined. The reason is that each comatch of* tri_{ST} *is completely determined by* S*- and* T*-component of the match of* tri_I*, because of embedding* $R(tri_{ST}) \rightarrowtail L(tri_I)$*. Furthermore, each match of* tri_{ST} *is given by uniqueness of pushout complements along injective morphisms with respect to non-deleting rule* tri_{ST} *and its comatch. Moreover, the source-target* TGT*-sequence implies* $G_S \in VL_{S0}$ *and* $G_T \in VL_{T0}$*.*

Fact 2 (Model Integration is syntactically correct) *Given model integration sequence* $((G_S,G_T),$ $G_0 \xrightarrow{tr_I^*} G_n,\ G)$ *then* $G_n = G \in VL$ *with* $proj_S(G) = G_S \in VL_S$ *and* $proj_T(G) = G_T \in VL_T$*.*

Proof. $G_0 \xrightarrow{tr_I^*} G_n$ source-target consistent
$\Rightarrow\ \exists\, \emptyset \xrightarrow{tr_{ST}^*} G_0$ s.t. $\emptyset \xrightarrow{tr_{ST}^*} G_0 \xrightarrow{tr_I^*} G_n$ S-T-match consistent
$\overset{Thm2}{\Rightarrow}\ \emptyset \xrightarrow{tr^*} G_n$, i.e. $G_n = G \in VL$ □

Finally we want to analyze which pairs $(G_S,G_T) \in VL_S \times VL_T$ can be integrated. Intuitively those which are related by the model transformation $MT : VL_S \Rrightarrow VL_T$ in Theorem 1. In fact, model integration sequences can be characterized by unique model transformation sequences.

Theorem 3 (Characterization of Model Integration Sequences) *Each model integration sequence* $((G_S,G_T),G_0 \xrightarrow{tr_I^*} G_n,G)$ *corresponds uniquely to a model transformation sequence* $(G_S,G_0' \xrightarrow{tr_F^*} G_n,G_T)$*, where* tr_I^* *and* tr_F^* *are based on the same rule sequence* tr^**.*

Proof. $((G_S,G_T),G_0 \xrightarrow{tr_I^*} G_n,G)$ is model integration sequence
$\overset{def}{\Leftrightarrow}$ source-target consistent $G_0 \xrightarrow{tr_I^*} G_n$ with $proj_S(G_0) = proj_S(G_n) = G_S$, $proj_C(G_0) = \emptyset$, $proj_T(G_0) = proj_T(G_n) = G_T$ and $G_n = G$
$\overset{def}{\Leftrightarrow}\ \emptyset \xrightarrow{tr_{ST}^*} G_0 \xrightarrow{tr_I^*} G_n$ S-T-match consistent with $proj_S(G_n) = G_S$ and $proj_T(G_n) = G_T$
$\overset{Thm2}{\Leftrightarrow}\ \emptyset \xrightarrow{tr^*} G_n$ with $proj_S(G_n) = G_S$ and $proj_T(G_n) = G_T$
$\overset{Thm1}{\Leftrightarrow}\ \emptyset \xrightarrow{tr_S^*} G_0' \xrightarrow{tr_F^*} G_n$ match consistent with $proj_S(G_n) = G_S$ and $proj_T(G_n) = G_T$

$\overset{def}{\Leftrightarrow}$ $G'_0 \xrightarrow{tr^*_F} G_n$ source consistent with $proj_S(G'_0) = proj_S(G_n) = G_S$ and $proj_T(G_n) = G_T$
$\overset{def}{\Leftrightarrow}$ $(G_S, G'_0 \xrightarrow{tr^*_F} G_n, G_T)$ is model transformation sequence. □

Coming back to the example of a model transformation from class diagrams to database models the relevance and value of the given theorems can be described from the more practical view. Fig. 6 shows a triple graph, which defines a class diagram in its source component, database tables in its target component and the correspondences in between. Since this model is already fully integrated, it constitutes the resulting graph G of example model integration sequence $((G_S, G_T), G_0 \xrightarrow{tr^*_I} G_n, G)$. The starting point is given by G_S as restriction of G to elements of the class diagram, indicated by pink, and G_T containing the elements of the database part, indicated by yellow colour. Now, the blue nodes for correspondence as well as the morphisms between connection component to source and target component are created during the integration process. All elements are labeled with a number to specify matches and created objects for each transformation step. The sequence of applied rules is

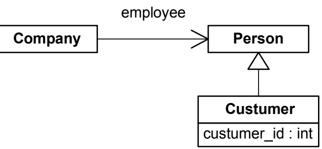


Figure 5: Source component of Fig. 6 in concrete syntax

$$G_0 \xrightarrow{Class2table} G_1 \xrightarrow{Class2table} G_2 \xrightarrow{Subclass2Table} G_3 \xrightarrow{PrimaryAttribute2Column}$$
$$G_4 \xrightarrow{Association2ForeignKey} G_5 = G_n.$$

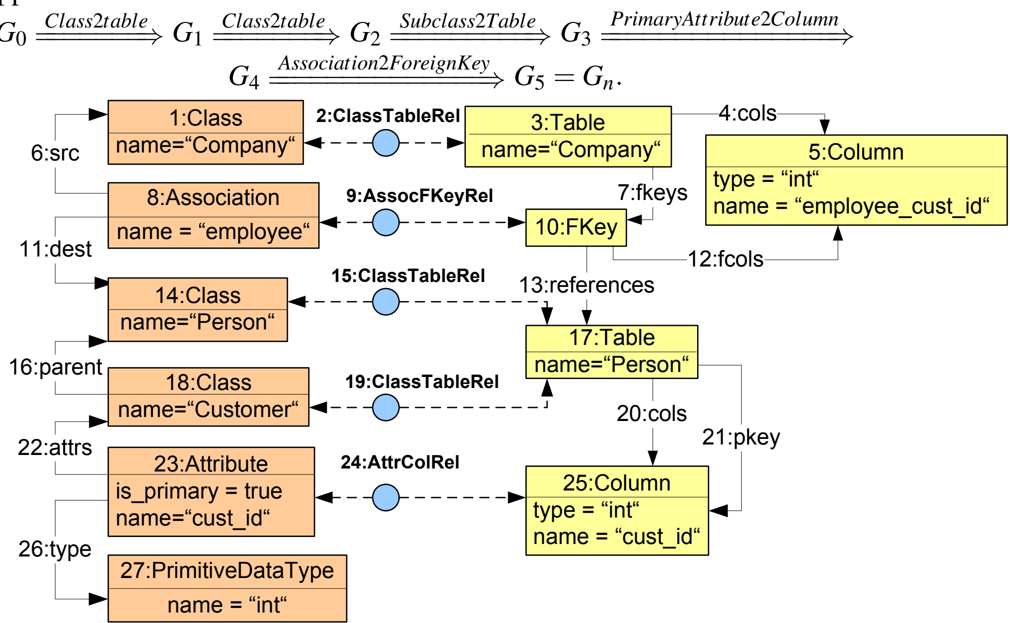


Figure 6: Example of model integration for model transformation *Class2Table*

Now, Table 1 shows all matches of this sequence for both cases of Theorem 3 being the model integration sequence $G_0 \xrightarrow{tr^*_I} G_n$ and the forward transformation sequence $G'_0 \xrightarrow{tr^*_I} G_n$, where G_0 contains the elements of G except correspondence parts and G'_0 is G leaving out all elements of target and connection component. The column "Created" in the table lists the elements which are created at each transformation step. According to the numbers for the elements, the correspondence component is completely created during the model integration sequence and the elements of each match are created by the corresponding source-target rule application in

Step and Rule	Integration Sequence Elements		Forward Sequence Elements	
	Matched	Created	Matched	Created
1	1,3	2	1	2,3
2	14,17	15	14	15,17
3	14-18	19	14-18	19
4	17-20, 22,23, 25-27	24	17-19, 22,23, 26,27	20,21, 24,25
5	1-8, 10-15, 17,21,25	9	1-3,6,8, 11,14,15, 17,21,25	4,5,7,9,10,12,13

Table 1: Steps of example integration sequence

$\emptyset \xRightarrow{tr_{ST}^*} G_0$. Therefore, $\emptyset \xRightarrow{tr_{ST}^*} G_0 \xRightarrow{tr_I^*} G_n$ is match consistent. Analogously $\emptyset \xRightarrow{tr_S^*} G_0'$ consists of the specified steps in Table 1, where comatches are given by the elements of the match in the forward transformation sequence implying $\emptyset \xRightarrow{tr_S^*} G_0' \xRightarrow{tr_F^*} G_n$ being match consistent. Both integration and forward transformation sequence can be recaptured by analyzing the other, which corresponds to Theorem 3.

5 Related Work

Various approaches for model transformation in general are discussed in [MB03] and [OMG07] using BOTL and QVT respectively. For a taxonomy of model transformation based on graph transformation we refer to [MG06]. Triple Graph Grammars have been proposed by A. Schürr in [Sch94] for the specification of graph transformations. A detailed discussion of concepts, extensions, implementations and applications scenarios is given by E. Kindler and R. Wagner in [KW07]. The main application scenarios in [KW07] are model transformation, model integration and model synchronization. These concepts, however, are discussed only on an informal level using a slightly different concept of triple graphs compared with [Sch94].

In this paper we use the original definition of triple graphs, triple rules, and triple transformations of [Sch94] based on the double pushout approach (see [Roz97], [EEPT06]). In our paper [EEE+07] we have extended the approach of [Sch94] concerning the relationship between TGT-sequences based on triple rules $G_0 \overset{tr^*}{\Rightarrow} G_n$ and match consistent TGT-sequences $G_0 \overset{tr_S^*}{\Rightarrow} G_{n0} \overset{tr_F^*}{\Rightarrow} G_m$ based on source and forward rules leading to the canonical Decomposition and Composition Result 1 (Thm 1). This allows to characterize information preserving bidirectional model transformations in [EEE+07].

In this paper the main technical result is the Canonical Decomposition and Composition Result 2 (Thm 2) using source-target rules tr_{ST} and integration rules tr_I instead of tr_S and tr_F. Both results are formally independent, but the same proof technique is used based on the Local Church–Rosser and Concurrency Theorem for graph transformations. The main result of [EEPT06] is based on these two decomposition and composition results. For a survey on tool integration with triple graph grammars we refer to [KS06].

6 Future Work and Conclusion

Model integration is an adequate technique in system design to work on specific models in different languages, in order to establish the correspondences between these models using rules which can be generated automatically. Once model transformation triple rules are defined for translations between the involved languages, integration rules can be derived automatically for maintaining consistency in the overall integrated modelling process.

Main contributions of this paper are suitable requirements for existence of model integration as well as composition and decomposition of source-target and integration transformations to and from triple transformations. Since model integration may be applied at any stage and several times during the modelling process, results of model integrations in previous stages can be used as the starting point for the next incremental step.

All concepts are explained using the well known case study for model transformation between class diagrams and relational data bases. While other model transformation approaches were applied to the same example for translation between source and target language, triple graph grammars additionally show their general power by automatic and constructive derivation of an integration formalism. Therefore, model integration in the presented way can scale up very easily, only bounded by the effort to build up general triple rules for parallel model evolution.

Usability extends when regarding partly connected models, which shall be synchronized as discussed on an informal level in [KW07]. On the basis of model integration rules model synchronization can be defined in future work as model integration using inverse source and target rules, standard source and target rules as well as integration rules in a mixed way, such that the resulting model is syntactically correct and completely integrated. Another interesting aspect for future work is the extension of triple graph rules and corresponding transformation and integration rules by negative application conditions (see [HHT96]), or by more general graph constraints (see [HP05]).

Bibliography

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In Dwyer and Lopes (eds.), *Fundamental Approaches to Software Engineering*. LNCS 4422, pp. 72–86. Springer, 2007.
http://tfs.cs.tu-berlin.de/publikationen/Papers07/EEE+07.pdf

[EEH08] H. Ehrig, K. Ehrig, F. Hermann (eds.). *From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars (Long Version)*. Februrary 2008. published as Technical Report, TU Berlin, No. 2008-3.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer Verlag, 2006.
http://www.springer.com/3-540-31187-4

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae* 26(3,4):287–313, 1996.

[HP05] A. Habel, K.-H. Pennemann. Nested Constraints and Application Conditions for High-Level Structures. In Kreowski et al. (eds.), *Formal Methods in Software and Systems Modeling*. Lecture Notes in Computer Science 3393, pp. 293–308. Springer, 2005.
http://dx.doi.org/10.1007/b106390

[KS06] A. König, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In *Heckel, R. (eds.): Elsevier Science Publ. (pub.), Proceedings of the SegraVis School on Foundations of Visual Modelling Techniques, Vol. 148, Electronic Notes in Theoretical Computer Science pp. 113-150, Amsterdam*. 2006.
http://dx.doi.org/10.1016/j.entcs.2005.12.015

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report tr-ri-07-284, Software Engineering Group, Department of Computer Science, University of Paderborn, June 2007.
http://www.uni-paderborn.de/cs/ag-schaefer/Veroeffentlichungen/Quellen/Papers/2007/tr-ri-07-284.pdf

[MB03] F. Marschall, P. Braun. Model Transformations for the MDA with BOTL. In *Proc. of the Workshop on Model Driven Architecture: Foundations and Applications (MDAFA 2003), Enschede, The Netherlands*. Pp. 25–36. 2003.
http://citeseer.ist.psu.edu/marschall03model.html

[MG06] T. Mens, P. V. Gorp. A Taxonomy of Model Transformation. In *Proc. International Workshop on Graph and Model Transformation (GraMoT'05), number 152 in Electronic Notes in Theoretical Computer Science, Tallinn, Estonia, Elsevier Science*. 2006.
http://tfs.cs.tu-berlin.de/gramot/Gramot2005/FinalVersions/PDF/MensVanGorp.pdf

[OMG07] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, Final Adopted Specification (07-07-2007). 2007.
http://www.omg.org/docs/ptc/07-07-07.pdf

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations*. World Scientific, 1997.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *G. Tinhofer, editor, WG94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science, volume 903 of Lecture Notes in Computer Science, pages 151–163, Springer Verlag, Heidelberg*. 1994.
http://dx.doi.org/10.1007/3-540-59071-4_45

Verifying Model Transformations by Structural Correspondence

Anantha Narayanan[1] **and Gabor Karsai**[1]

[1]Institute for Software Integrated Systems,
Vanderbilt University
Nashville, TN 37203 USA

Abstract: Model transformations play a significant role in model based software development, and the correctness of the transformation is crucial to the success of the development effort. We have previously shown how we can use bisimulation to verify the preservation of certain behavioral properties across a transformation. However, transformations are often used to construct structurally different models, and we might wish to ensure that there is some structural correspondence to the original model. It may be possible to verify such transformations without having to explicitly specify the dynamic semantics of the source and target languages. In this paper, we present a technique to verify such transformations, by first specifying certain structural correspondence rules between the source and target languages, and extending the transformation so that these rules can be easily evaluated on the instance models. This will allow us to conclude if the output model has the expected structure. The verification is performed at the instance level, meaning that each execution of the transformation is verified. We will also look at some examples using this technique.

Keywords: Verification, Model transformations

1 Introduction

Model transformations that translate a source model into an output model are often expressed in the form of rewriting rules, and can be classified according to a number of categories [MV05]. However, the correctness of a model transformation depends on several factors, such as whether the transformation terminates, whether the output model complies with the syntactical rules of the output language, and others. One question crucial to the correctness of a transformation is whether it achieved the intended result of mapping the semantics of the input model into that of the output model. For instance, a transformation from a Statechart model to a non-hierarchical FSM model can be said to be correct if the output model truly reproduces the behavior of the original Statechart model.

Models can also be seen as attributed and typed graph structures that conform to an abstract syntax. Model transformations take an input graph and produce a modified output graph. In a majority of these cases, the transformation matches certain graph structures in the input graph and creates certain structures in the output graph. In such cases, correctness may be defined as whether the expected graph structures were produced in the output corresponding to the relevant structures in the input graph. If we could specify the requirements of such correspondences and trace the correspondences easily over instance models, a simple model checking process at the

end of a transformation can be used to verify if those instances were correctly transformed. In this paper, we explore a technique to specify *structural correspondence* rules, which can be used to decide if the transformation resulted in an output model with the expected structure. This will be specified along with the transformation, and evaluated for each execution of the transformation, to check whether the output model of that execution satisfies the correspondence rules.

2 Background

2.1 GReAT

GReAT [AKL03] is a language framework for specifying and executing model transformations using graph transformation. It is a meta-model based transformation tool implemented within the framework of GME [LBM+01]. One of the key features of GReAT is the ability to define cross-language elements by composing the source and target meta-models, and introducing new vertex and edge types that can be temporarily used during the transformation. Such cross-meta-model associations are called *cross-links*. Note that a similar idea is present in Triple Graph Grammars [Sch95]. This ability of GReAT allows us to track relationships between elements of the source and target models during the course of the transformation, as the target model is being constructed from the source model. This feature plays a crucial role in our technique to provide assurances about the correctness of a model transformation.

2.2 Instance Based Verification of Model Transformations

Verifying the correctness of model transformations in general is as difficult as verifying compilers for high-level languages. But for practical purposes, a transformation may be said to have 'executed correctly' if a certain instance of its execution produced an output model that preserved certain properties of interest. We call that instance 'certified correct'. This idea is similar to the work of Denney and Fischer in [DF05], where a program generator is extended to produce logical annotations necessary for formal verification of certain safety properties. An automated theorem prover uses these annotations to find proofs for the safety properties for the generated code. Note that this does not prove the safety of the code generator, but only of a particular instance of generated code.

In our previous effort [NK06], we have shown that it is both practical and prudent to verify the correctness of every execution of a model transformation, as opposed to finding a correctness proof for the transformation specification. This makes the verification tractable, and can also find errors introduced during the implementation of a transformation that may have been specified correctly.

This technique was applied to the specific case of preservation of reachability related properties across a model transformation. Reachability is a fairly well-understood property, and can be verified easily for a labeled transition system (LTS), for instance by model checking [Hol97]. If two labeled transition systems are *bisimilar*, then they will have the same reachability behavior. In our approach, we treated the source and target models as labeled transition systems, and verified the transformation by checking if the source and target instances were bisimilar.

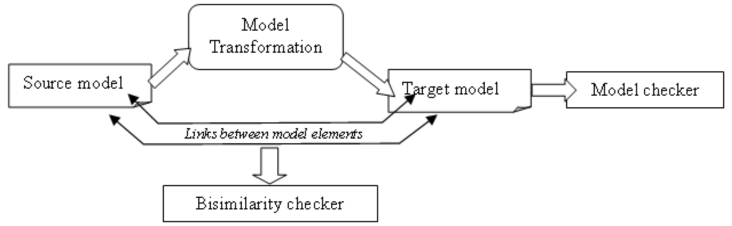


Figure 1: Architecture for Verifying Reachability Preservation in a Transformation

Given an LTS (S, Λ, →), a relation $R \subseteq S \times S$ is a *bisimulation* [San04] if:

$$(p,q) \in R \text{ and } p \overset{\alpha}{\to} p' \text{ implies that there exists a } q' \in \text{S}, \\ \text{such that } q \overset{\alpha}{\to} q' \text{ and } (p',q') \in R,$$

and conversely,

$$q \overset{\alpha}{\to} q' \text{ implies that there exists a } p' \in \text{S}, \\ \text{such that } p \overset{\alpha}{\to} p' \text{ and } (p',q') \in R.$$

We used cross-links to relate source and target elements during the construction of the output model. These relations were then passed to a bisimilarity checker, which determined whether the source and target instances were bisimilar. If the instances were determined to be bisimilar, we could conclude that the execution of the transformation was correct. Figure 1 shows an overview of the architecture for this approach.

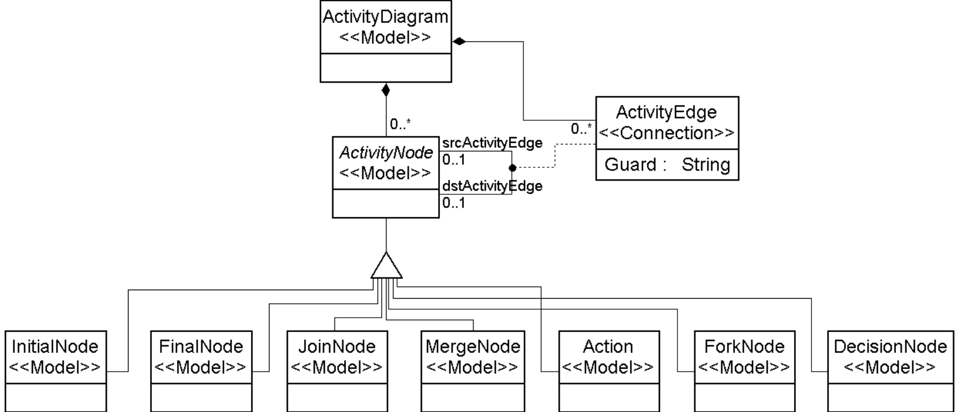


Figure 2: Meta-model for UML Activity Diagrams

2.3 UML to CSP Transformation

The UML to CSP transformation was presented as a case study at AGTIVE ’07 [BEH07], to compare the various graph transformation tools available today. We provide an overview of this

case study here from a GReAT point of view, and we will use this as an example to explain our technique for verifying model transformations.

The objective of this transformation is to take a UML Activity Diagram [OMG06] and generate a Communicating Sequential Process [Hoa78] model with the equivalent behavior. The Activity Diagram consists of various types of *Activity Nodes*, which can be connected by directed *Activity Edges*. Figure 2 shows the GME meta-model for UML Activity Diagrams. A CSP *Process* is defined by a *Process Assignment*, which assigns a *Process Expression* to a *Process Id*. *Process Expressions* can be a simple *Process*, a *Prefix* operator or a *BinaryOperator*. Figure 3 shows the GME meta-model for CSP, highlighting the relevant parts for our example.

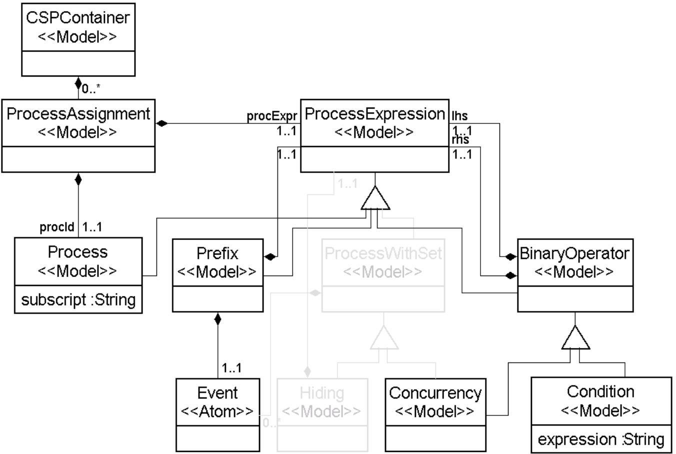


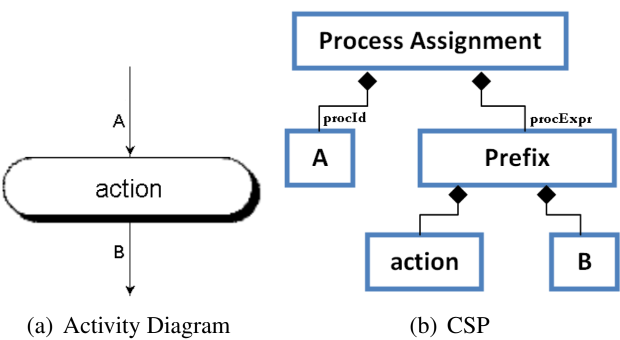
Figure 3: Meta-model for CSP

The UML to CSP mapping assigns a *Process* for each *Activity Edge*. For each type of *Activity Node*, a *Process Assignment* is created, which assigns the Process corresponding to the incoming Activity Edge to a *Process Expression* depending on the type of the Activity Node. Figure 4 shows one such mapping, for the type *Action Node*. This assigns the incoming Process to a *Prefix* expression. The resulting CSP expression can be written as $A = action \longrightarrow B$, which is shown in Figure 4 as a model instance compliant with the CSP meta-model.

3 Structural Correspondence

As in the UML to CSP case, model transformations can be used to generate a target model of a certain structure (CSP) from a source model of a different structure (UML). Specific structural configurations in the source model (such as an *Action Node* in the UML model) produce specific structural configurations in the target model (such as a *Prefix* in the CSP model). The rules to accomplish the structural transformations may be simple or complicated. However, it is fairly straightforward to compare and verify that the correct structural transformation was made, if we already know which parts of the source structure map to which parts of the target structure.

In our technique to verify a transformation by structural correspondence, we will first define



(a) Activity Diagram (b) CSP

Figure 4: CSP Process Assignment for Action Node

a set of structural correspondence rules specific to a certain transformation. We will then use cross-links to trace source elements with the corresponding target elements, and finally use these cross-links to check whether the structural correspondence rules hold.

In essence, we expect that the correspondence conditions are independently specified for a model transformation, and an independent tool checks if these conditions are satisfied by the instance models, after the model transformation has been executed. In other words, the correspondence conditions depend purely on the source and target model structures and not on the rewriting rules necessary to effect the transformation. Since the correspondence conditions are specified in terms of simple queries on the model around previously chosen context nodes, we expect that they will be easier to specify, and thus more reliable than the transformation itself. We also assume that the model transformation builds up a structure for bookkeeping the mapping between the source and target models.

3.1 Structural Correspondence Rules for UML to CSP Transformation

A structural correspondence rule is similar to a precondition-postcondition style axiom. We will construct them in such a way that they can be evaluated easily on model instances. We will use the UML to CSP example to illustrate structural correspondence. Consider the case for the *Action Node*, as shown in Figure 4. The *Action Node* has one incoming edge and one outgoing edge. It is transformed into a *Process Assignment* in the CSP. The CSP structure for this instance consists of a *Process Id* and a *Prefix*, with an *Event* and a target *Process*. This is the structural transformation for each occurrence of an Action Node in the Activity Diagram.

We can say that for each Action Node in the Activity Diagram, there is a *corresponding* Process Assignment in the CSP, with a Prefix Expression. When our transformation creates this corresponding Process Assignment, we can use a cross-link to track this correspondence. The structural correspondence is still not complete, as we have to ensure that the Process Id and the Prefix Expression are created correctly. We use a kind of *path expression* to specify the correctness of corresponding parts of the two structures, and the correspondence is expressed in the form $SourceElement = OutputElement$. Let us denote the Action Node by *AN*, and the Process Assignment by *PA*. Then the necessary correspondence rules can be written using path expressions as shown in Table 1.

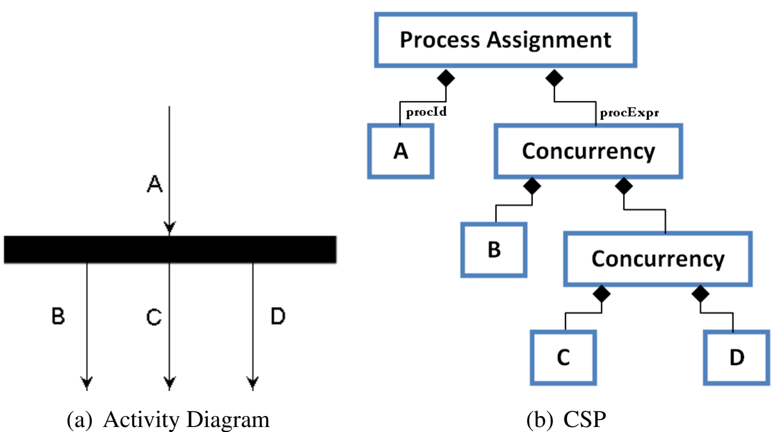
Rule	Path expression
The *Action Node* corresponds to a Process Assignment with a Prefix	$PA.procExpr.type = Prefix$
The incoming edge in the UML corresponds to the Process Id	$AN.inEdge.name = PA.procId.name$
The outgoing edge corresponds to the target Process	$AN.outEdge.name = PA.procExpr.Process.name$
The *action* of the Action Node corresponds to the Event	$AN.action = PA.procExpr.event$

Table 1: Structural Correspondence Rules for Action Node

These rules together specify the complete structural correspondence for a section of the Activity Diagram and a section of its equivalent CSP model. The different types of Activity Nodes result in different structures in the CSP model, some of which are more complex than the fairly straightforward case for the Action Node. Next, we look at the structural mapping for some of the other nodes.

A *Fork Node* in the Activity Diagram is transformed into a Process Assignment with a *Concurrency* Expression. Figure 5 shows a Fork Node with an incoming edge A and three outgoing edges B, C and D. This is represented by the CSP expression $A = B \parallel (C \parallel D)$, where $\parallel$ represents concurrency (the actual ordering of B, C and D is immaterial). The structural representation of this expression as an instance of the CSP meta-model is shown in Figure 5. The Fork Node is transformed into a CSP Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Fork Node, and a Process Expression of type *Concurrency*. The Concurrency Expression consists of Processes and other Concurrency nodes, depending on the number of outgoing Activity Edges.

If we denote the Fork Node by FN and the Process Assignment by PA, the structural corre-

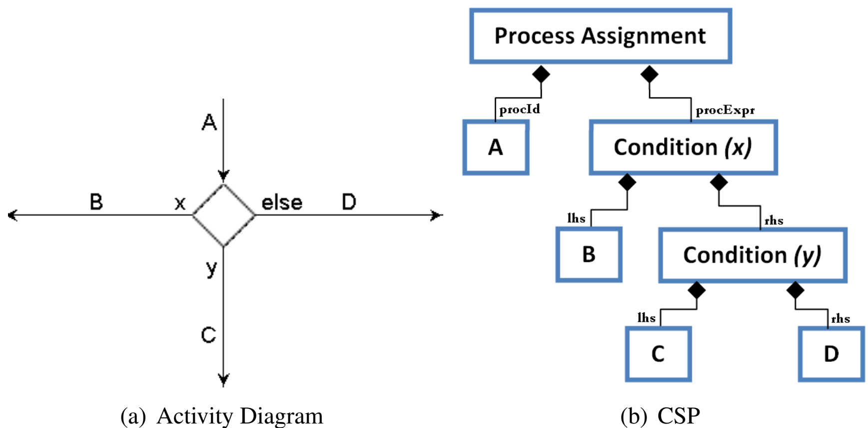


(a) Activity Diagram (b) CSP

Figure 5: CSP Process Assignment for Fork Node

Rule	Path expression
The *Fork Node* corresponds to a Process Assignment with a Concurrency	$PA.procExpr.type = Concurrency$
The incoming edge in the UML corresponds to the Process Id	$FN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Process in the Process Expression	$\forall o \in FN.outEdge$ $\exists p \in PA.procExpr..Process : o.name = p.name$

Table 2: Structural Correspondence Rules for Fork Node



(a) Activity Diagram (b) CSP

Figure 6: CSP Process Assignment for Decision Node

spondence rules can be described using path expressions as shown in Table 2. We will use the double-dot '..' to denote the *descendant* operator (similar to '//' in XPath queries), to specify ancestor-descendant relationships.

By evaluating these rules on the Activity Diagram and the CSP models, we can determine whether the structural correspondence was satisfied for Fork Nodes. Another type of node in the Activity Diagram is the *Decision Node*. The transformation mapping for the Decision Node is a slight variation of the Fork Node. Figure 6 shows a Decision Node with an incoming edge A, and three outgoing edges B, C and D, with guards x, y and *else* respectively (in this case study, we will assume that the Decision Node always has exactly one 'else' edge). This is represented by the CSP expression $A = B \not\prec x \not\succ (C \not\prec y \not\succ D)$, where the operator $C \not\prec y \not\succ D$ is the *condition* operator with the meaning that if y is *true*, then the process behaves like C, else like D.

The Decision Node is transformed to the CSP model shown in Figure 6 as a model instance of the CSP meta-model. The Decision Node is transformed into a Process Assignment that consists of a Process Id corresponding to the incoming Activity Edge of the Decision Node, and a Process Expression of type *Condition*. The Condition's *expression* attribute is set to the *guard* of the corresponding Activity Edge, and a Process corresponding to the Activity Edge is created as it's LHS. For the final 'else' edge, a Process is created in the last Condition as it's RHS. The

Rule	Path expression
The *Decision Node* corresponds to a Process Assignment with a Condition	$PA.procExpr.type = Condition$
The incoming edge in the UML corresponds to the Process Id	$DN.inEdge.name = PA.procId.name$
For each outgoing edge, there is a corresponding Condition in the Process Expression and a corresponding Process in the Condition's LHS	$\forall o \in DN.outEdge \wedge o.guard \neq else$ $\exists c \in PA.procExpr..Condition :$ $c.expression = o.guard \wedge c.lhs.name = o.name$
For the outgoing 'else' edge, there is a Condition in the Process Expression with a corresponding Process as it's RHS	$\forall o \in DN.outEdge \wedge o.guard = else$ $\exists c \in PA.procExpr..Condition :$ $c.rhs.name = o.name$

Table 3: Structural Correspondence Rules for Decision Node

structural correspondence rules for this mapping are shown in Table 3.

3.2 Specifying Structural Correspondence Rules in GReAT

Specifying the structural correspondence for a transformation consists of two parts:

1. Identifying the significant source and target elements of the transformation
2. Specifying the structural correspondence rules for each element using path expressions

The first step is accomplished in GReAT by using cross-links between the source and target elements. A composite meta-model is added to the transformation, by associating selected source and target elements using a temporary 'Structural Correspondence' class. There will be one such class for each pair of elements. This class will have a string attribute, which is set to the path expressions necessary for structural correspondence for that pair. Figure 7 shows a composite meta-model specifying the structural correspondence for Fork Nodes. The *Fork Node* class comes from the Activity Diagram meta-model, and the *Process Assignment* class comes from the CSP meta-model.

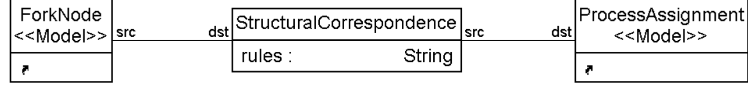


Figure 7: Composite Meta-model to Specify Structural Correspondence

Once the structural correspondence has been specified for all the relevant items, the transformation is enhanced to create the cross-link when creating the respective target elements. Figure 8 shows the GReAT rule in which the Process Assignment for a Fork Node is created, enhanced to

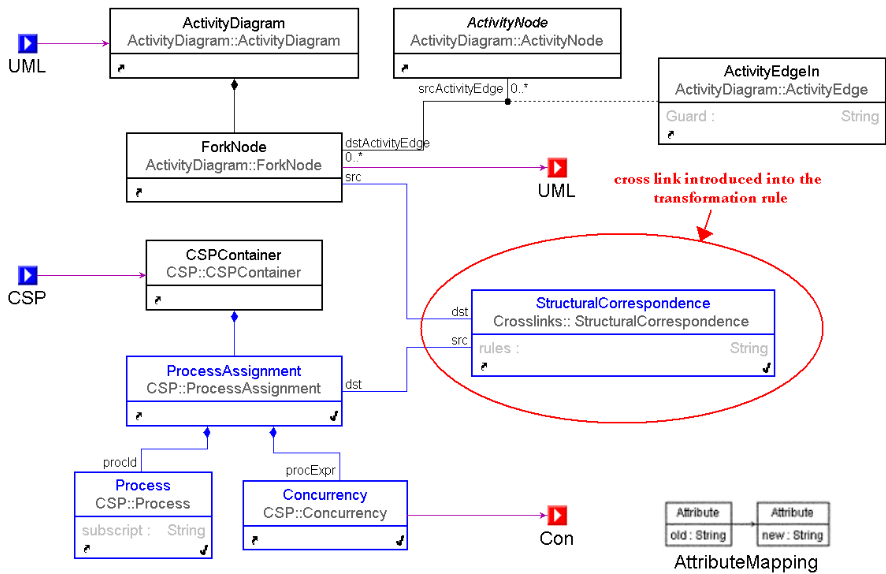


Figure 8: GReAT Rule with Cross-link for Structural Correspondence

create the cross-link for structural correspondence. Note that in incoming Activity Edge is represented as an *Association Class* named `ActivityEdgeIn`. The transformation for the Fork Node is actually accomplished in a sequence of several rules executed recursively, as shown in Figure 9. First all the Fork Nodes are collected, and a sequence of rules are executed for each node. These rules iterate through the out-edges of each Fork node, creating the Concurrency tree. Though several rules are involved in the transformation for Fork nodes, the cross-link needs to be added to one rule only.

It must be noted that it is necessary to specify the structural correspondence rules only once in the composite meta-model. The cross-link must however be added to the transformation rules, and in most cases will be required only once for each pair of source and target element.

3.3 Evaluating the Structural Correspondence Rules

Once the structural correspondence rules have been specified, and the cross-links added to the transformation, the correspondence rules are evaluated on the instance models after each execution of the transformation. These rules can be evaluated by performing a simple depth first search on the instance models, and checking if the correspondence rules are satisfied at each relevant stage.

This consists of two phases. The first phase is to generate the code that will traverse the instance models and evaluate the correspondence rules. Since the meta-models of both the source and target languages are available with the transformations, and the path expressions are written in a standard form that can be parsed automatically, the model traverser code can be automatically generated from the structural correspondence specification. This needs to be done only once each

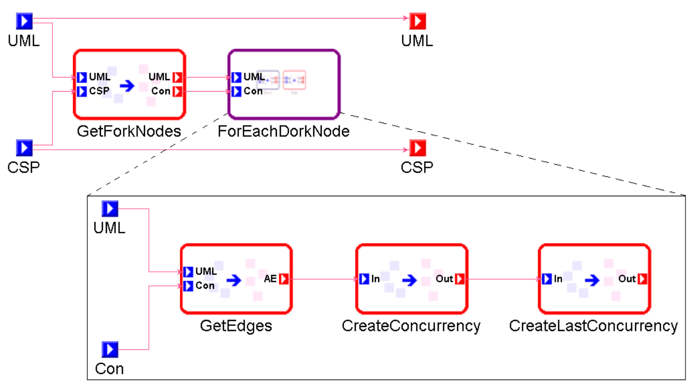


Figure 9: Sequence of GReAT Rules for Fork Node Transformation

time the structural correspondence specification changes. The second phase is to call the model traverser code at the end of each execution of the transformation, supplying to it the source and target model instances along with the cross-links.

In the case of the UML to CSP transformation, we traverse the input Activity Diagram model and evaluate the correspondence rules at each activity node. For each Activity Node, the cross-link is traversed to find the corresponding Process Assignment. If a correspondence rule has been defined for an Activity Node, and no corresponding Process Assignment is found, then this signals an error in the transformation. After locating the corresponding Process Assignment, the path expressions are evaluated. If any of the rules are not satisfied, the error is reported. If all the rules are satisfied for all the nodes, then we can conclude that the transformation has executed correctly.

The instance model is traversed in a depth-first manner. The corresponding elements are located using the cross-links, which will take constant time. The path expressions are evaluated on the instances, which will take polynomial time in most cases. Thus, the overall verification process does not incur a significant performance overhead in most cases.

3.4 Remarks

The structural correspondence based verification described here can provide important assurances about the correctness of transformations, while being practically applicable in most common transformations.

The use of path expressions to specify correspondence rules makes it easy to specify correctness. The path expressions use a simple query language that can be easily evaluated on the instance models. Our future research concentrates on the requirements of such a query language. Most complex transformations may involve multiple rules executing recursively to transform a particular part of a model. However, it may be possible to specify the correspondence for that part of the model using a set of simple path expressions. Such a specification would be simpler and easier to understand than the complex transformation rules.

The structural correspondence is also specified orthogonal to the transformation specification.

Thus, these rules may be written by someone other than the original transformation writer, or even supplied along with the requirements document.

4 Related Work

[KÖ4], [KHE03] present ideas on validating model transformations. In [KHE03], the authors present a concept of rule-based model transformations with control conditions, and provide a set of criteria to ensure termination and confluence. In [KÖ4], Küster focuses on the syntactic correctness of rule-based model transformations. This validates whether the source and target parts of the transformation rule are syntactically correct with respect to the abstract syntax of the source and target languages. These approaches are concerned with the functional behavior and syntactic correctness of the model transformation. [de] also discusses validation of transformations on these lines, but also introduces ideas of syntactic consistency and behavior preservation. Our technique addresses semantic correctness of model transformations, addressing errors introduced due to loss or mis-representation of information during a transformation.

In [BH07], Bisztray and Heckel present a rule-level verification approach to verify the semantic properties of business process transformations. CSP is used to capture the behavior of the processes before and after the transformation. The goal is to ensure that every application of a transformation rule has a known semantic effect. We use path expressions to capture the relation between structures before and after a transformation. These path expressions are generic (they do not make any assumptions about the underlying semantics of the models involved), and can be applied to a wide variety of transformations.

Ehrig et. al. [EEE+07] study bidirectional transformations as a technique for preserving information across model transformations. They use triple graph grammars to define bi-directional model transformations, which can be inverted without specifying a new transformation. Our approach offers a more relaxed framework, which will allow some loss of information (such as by abstraction), and concentrates on the crucial properties of interest. We also feel that our approach is better suited for transformations involving multiple models and attribute manipulations.

In other related work, [VP03] presents a model level technique to verify transformations by model checking a selected semantic property on the source model, and transforming the property and validating it in the target domain. The validation requires human expertise. After transforming the property, the target model is model checked. In our approach, since the properties are specified using cross links that span over both the source and target languages, we do not need to transform them. [LLMC06] discuss an approach to validate model transformations by applying OCL constraints to preserve and guarantee certain model properties. [GGL+06] is a language level verification approach which addresses the problem of verifying semantic equivalence between a model and a resulting programming language code.

4.1 MOF QVT Relations Language

The MOF 2.0 Query / View / Transformation specification [OMG05] addresses technology pertaining to manipulation of MOF models. A *relations language* is prescribed for specifying relations that must hold between MOF models, which can be used to effect model transformations.

Relations may be specified over two or more domains, with a pair of *when* and *where* predicates. The *when* predicate specifies the conditions under which a relation must hold, and the *where* predicate specifies the condition that all the participating model elements must satisfy. Additionally, relations can be marked as *checkonly* or *enforced*. If it is marked *checkonly*, the relation is only checked to see if there exists a valid match that satisfies the relationship. If it is marked *enforced*, the target model is modified to satisfy the relationship whenever the check fails.

Our approach can be likened to the *checkonly* mode of operation described above. However, in our case, the corresponding instances in the models are already matched using cross links, and the correspondence conditions are evaluated using their context. The cross links help us to avoid searching the instances for valid matches. Specifying the correspondence conditions using context nodes simplifies the model checking necessary to evaluate the conditions, thus simplifying the verification process. Since we verify the correspondence conditions for each instance generated by the transformation, these features play an important role.

4.2 Triple Graph Grammars

Triple Graph Grammars [Sch95] are used to describe model transformations as the evolution of graphs by applying graph rules. The evolving graph complies with a graph schema that consists of three parts. One graph schema represents the source meta model, and one represents the target meta-model. The third schema is used to track correspondences between the source and target meta models. Transformations are specified declaratively using triple graph grammar rules, from which operational rules are derived to effect model transformations.

The schema to track correspondences between the source and target graphs provides a framework to implement a feature similar to cross links in GReAT. If the correspondence rules can be encoded into this schema, and the correspondence links persisted in the instance models, our verification approach can be implemented in this scenario.

5 Conclusions and Future Work

In this paper, we have shown how we can provide an assurance about the correctness of a transformation by using structural correspondence. The main errors that are addressed by this type of verification is the loss or misrepresentation of information during a model transformation.

We continue to hold to the idea that it is often more practical and useful to verify transformations on an instance basis. The verification framework must be added to the transformation only once, and is invoked for each execution of the transformation. The verification process does not add a significant overhead to the transformation, but provides valuable results about the correctness of each execution.

The path expressions must use a simple and powerful query language to formulate queries on the instance models. While existing languages such as OCL may be suitable for simple queries, we may need additional features, such as querying children to an arbitrary depth. Our future research concentrates on the requirements of such a language.

While the path expressions can be parsed automatically and evaluated on the instances, the cross-link for the relevant elements must be manually inserted into the appropriate transformation

rules. However, in most cases, it may be possible to infer where the cross-links must be placed. If the cross-links could be inserted into the rules automatically, the transformation can remain a black box. The main concern with this is that the cross-links are crucial to evaluating the correspondence rules correctly and also to keep the complexity down.

We have seen simple string comparisons added to the path expressions in this paper. Some transformations may require more complex attribute comparisons, or structure to attribute comparisons such as counting. We wish to explore such situations in further detail in future cases, to come up with a comprehensive language for specifying the path expressions.

Bibliography

[AKL03] A. Agrawal, G. Karsai, A. Ledeczi. An end-to-end domain-driven software development framework. In *OOPSLA '03: 18th annual ACM SIGPLAN conference on OOP, systems, languages, and applications*. Pp. 8–15. ACM Press, New York, NY, USA, 2003.
doi:http://doi.acm.org/10.1145/949344.949347

[BEH07] D. Bisztray, K. Ehrig, R. Heckel. Case Study: UML to CSP Transformation. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. 2007.

[BH07] D. Bisztray, R. Heckel. Rule-Level Verification of Business Process Transformations using CSP. In *Proceedings of the 6th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'07)*. 2007.

[de] de Lara, J. and Taentzer, G. Automated Model Transformation and its Validation with AToM3 and AGG. In *Lecture Notes in Artificial Intelligence, 2980*. Pp. 182–198. Springer.

[DF05] E. Denney, B. Fischer. Certifiable Program Generation. In Glück and Lowry (eds.), *GPCE*. Lecture Notes in Computer Science 3676, pp. 17–28. Springer, 2005.

[EEE+07] H. Ehrig, K. Ehrig, C. Ermel, F. Hermann, G. Taentzer. Information Preserving Bidirectional Model Transformations. In *Fundamental Approaches to Software Engineering*. Pp. 72–86. 2007.
doi:http://dx.doi.org/10.1007/978-3-540-71289-3_7

[GGL+06] H. Giese, S. Glesner, J. Leitner, W. Schfer, R. Wagner. Towards Verified Model Transformations. October 2006.

[Hoa78] C. A. R. Hoare. Communicating Sequential Processes. *Commun. ACM* 21(8):666–677, 1978.

[Hol97] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.* 23(5):279–295, 1997.
doi:http://dx.doi.org/10.1109/32.588521

[KÖ4] J. M. Küster. Systematic Validation of Model Transformations. In *Proceedings 3rd UML Workshop in Software Model Engineering (WiSME 2004)*. October 2004.

[KHE03] J. M. Küster, R. Heckel, G. Engels. Defining and validating transformations of UML models. In *HCC '03: Proceedings of the 2003 IEEE Symposium on Human Centric Computing Languages and Environments*. Pp. 145–152. IEEE Computer Society, Washington, DC, USA, 2003.

[LBM+01] A. Ledeczi, A. Bakay, M. Maroti, P. Volgyesi, G. Nordstrom, J. Sprinkle, G. Karsai. Composing Domain-Specific Design Environments. *Computer* 34(11):44–51, 2001. doi:http://dx.doi.org/10.1109/2.963443

[LLMC06] L. Lengyel, T. Levendovszky, G. Mezei, H. Charaf. Model-Based Development with Strictly Controlled Model Transformation. In *The 2nd International Workshop on Model-Driven Enterprise Information Systems, MDEIS 2006*. Pp. 39–48. Paphos, Cyprus, May 2006.

[MV05] T. Mens, P. Van Gorp. A taxonomy of model transformation. In *Proc. Int'l Workshop on Graph and Model Transformation*. 2005.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In Bruni and Varro (eds.), *Graph Transformation and Visual Modeling Techniques GT-VMT 2006*. Electronic Notes in Theoretical Computer Science, pp. 185–194. 2006.

[OMG05] OMG. Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification. 2005. http://www.omg.org/cgi-bin/doc?ptc/2005-11-01.

[OMG06] OMG. Unifed Modeling Language, version 2.1.1. 2006. "http://www.omg.org/technology/documents/formal/uml.htm"

[San04] D. Sangiorgi. Bisimulation: From The Origins to Today. In *LICS '04: Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*. Pp. 298–302. IEEE Computer Society, Washington, DC, USA, 2004. doi:http://dx.doi.org/10.1109/LICS.2004.13

[Sch95] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG '94: Proceedings of the 20th International Workshop on Graph-Theoretic Concepts in Computer Science*. Pp. 151–163. Springer-Verlag, London, UK, 1995.

[VP03] D. Varró, A. Pataricza. Automated Formal Verification of Model Transformations. In Jürjens et al. (eds.), *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*. Technical Report TUM-I0323, pp. 63–78. Technische Universität München, September 2003.

Extending Graph Query Languages by Reduction

Erhard Weinell

RWTH Aachen University of Technology, Department of Computer Science 3,
Ahornstrasse 55, D-52074 Aachen, Germany,
Weinell@cs.rwth-aachen.de
http://se.rwth-aachen.de/weinell

Abstract: Graph grammars have a long history in visual programming dating back to the seventies. Due to their declarative nature, even complex systems can be captured by clear and concise specifications. Recently, with the OMG's standard for model transformations QVT, graph grammar concepts have also found their way into an industrial scale. Although numerous languages and tools for graph transformations exist, they have no technical basis such as an execution framework in common. Instead, graph transformation machineries are usually implemented anew for each of these tools.

The DRAGOS graph database is especially well-suited for building graph transformation systems, as it is able to store complex graph structures directly. Besides its storage functionality, the database also provides a Query & Transformation Mechanism which is able to handle complex queries upon the stored graphs, and to modify them accordingly. Being designed as a basis for graph and model transformation tools, this mechanism is required to allow a flexible adaptation and extension according to the respective applications' needs. The present paper discusses how this requirement is covered by the proposed Query & Transformation Mechanism.

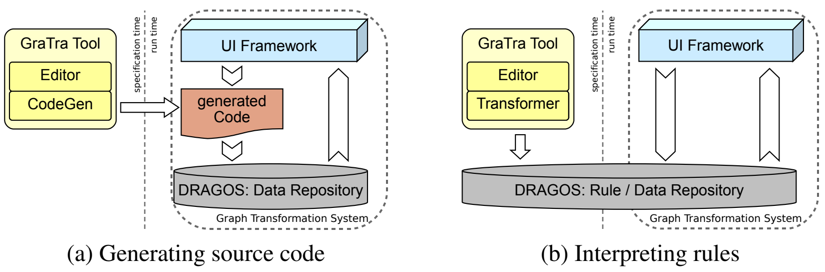
Keywords: graph database, extensibility, constraint satisfaction

1 Introduction

Model transformations are an enabling technique for model-driven software engineering, as they allow the formal definition of automated model translations. For example, model transformations can be used to enrich generic models by platform-specific informations, or to define refactoring rules at model level. The sheer amount of recent publications on the use of *graph transformations* for these purposes indicates their well-suitedness for this task. Presumably, this is caused by the fact that they rely on a mature and formally defined background with proper tool support. Nevertheless, the all-encompassing graph (or model) transformation language does not seem to exist, as new ones are proposed regularly. All of them share a common requirement: A proper data repository to store graph structures persistently, and an according execution framework to carry out the specified transformation rules.

Applications specified using graph transformation languages are called *graph transformation systems* (GTS). These system often utilize memory-based solutions as graph storage, which provide a direct access to the stored data. However, large-scaled applications usually require additional functionality, such as persistent transactional storage (instead of dedicated save actions).

Furthermore, support for concurrency isolation, and the ability to store large graph structures which are too unhandy for continuous transfer between file and memory, come into play. DRAGOS, a graph-oriented database management system (graph-database for short), is especially designed for this purpose. In contrast to traditional databases, DRAGOS provides a data model based on graphs. Therefore, graph structures can be stored directly, without any need for technical helper elements, such as tables for n-to-m relations.



(a) Generating source code (b) Interpreting rules

Figure 1: Applying DRAGOS in graph transformation systems

For implementing graph transformation systems, basically two alternatives exist (as shown in Figure 1). *First*, Figure 1a depicts an approach based on *generating* source code from the graph transformation rules. This code invokes operations on the graph database to retrieve and manipulate individual entities. *Second*, transformation rules can be executed directly by the database, as indicated by Figure 1b. As the figure suggests, the corresponding UI framework does not need to incorporate any generated code, but only relies on the functionality provided by the database. However, the database has to be able of interpreting the respective graph transformation language. This is currently not supported by the DRAGOS graph database, for which reason we develop an according mechanism. As discussed in [Wei07], this solution provides an easier integration with graph transformation tools and a larger optimization potential. To subsume the second argumentation, the code generation approach is less suitable for GTS based on databases. Due to the fact that operations utilized by the generated code are usually situated on a very low level of abstraction, they cannot consider the database-internal specifics. As result, a lot of simple operations are invoked, whereas few complex operations would cause less overhead.

In order to support not only a single, but arbitrary graph transformation approaches and their respective languages, DRAGOS only offers a *basic*, yet *extensible* core language. High-level languages are used to provide a user-friendly *concrete syntax*, which is not provided by the core language. The integration of these languages is basically performed as follows: First, rule definitions of the high-level language are *imported* into the database, e.g. by parsing textual specifications. Graph transformations then *convert* the imported rules to the DRAGOS Query & Transformation Language. The resulting graphs are *evaluated* by the underlying rule processor at runtime. Thus, application integration is achieved through graph transformations, instead of providing a complex code generation module as required in Figure 1a.

The conversion of graph transformation rules is usually complicated by complex mappings of high-level language constructs to the low-level ones provided by DRAGOS. We therefore allow to *extend* the core language by additional constructs to yield a more concise conversion. Like-

wise, the extended language constructs can be re-used for integrating other graph transformation languages, which is not possible for conversion rules. In this paper, we present the DRAGOS Query & Transformation Language and, as novel contribution, show how the core language can be extended. This is achieved by *reducing* new language constructs to existing ones.

The rest of this paper is structured as follows: We first introduce the basic functionality of DRAGOS in Section 2, and afterwards the Query & Transformation Language in Section 3. The following Section 4 presents how the language can be extended by additional language constructs. The paper finally discusses relations to other projects in Section 5 and gives an outlook on future work in Section 6.

2 Graph database DRAGOS

The DRAGOS database[1] allows to store and retrieve graph structures. Its data model is based on graphs, which are able to capture even complex data structures without need to introduce technical helper elements. For example, the relational data model often requires additional elements, such as extra tables to store many-to-many relations.

Architecture. Figure 2 shows a coarse-grained overview of the DRAGOS architecture. In the middle, the DRAGOS Kernel encapsulates the core graph model and a set of basic services. The services' responsibilities include opening and closing of databases as well as transaction and event management.

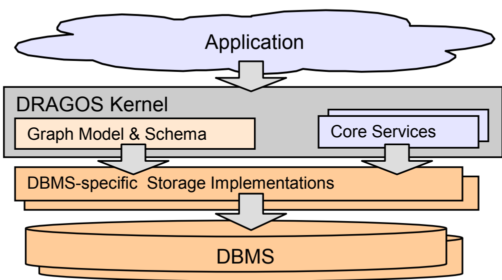
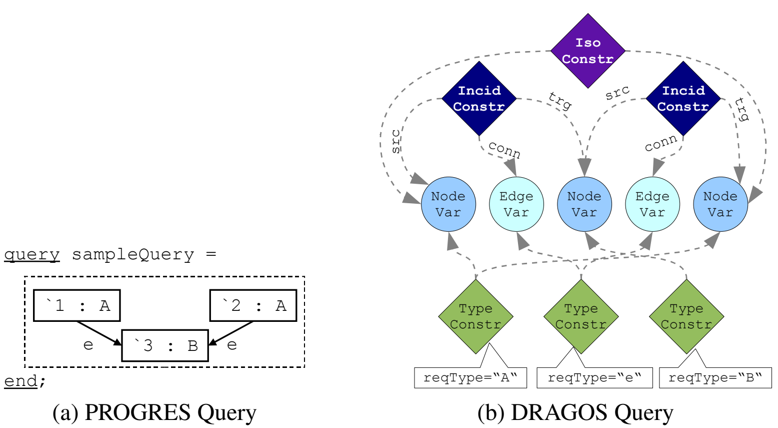


Figure 2: DRAGOS architecture

DRAGOS does not implement an own graph storage module. Instead, several implementations of the core graph model exist, which utilize existing database management systems as storage facility. Implementations are available for various databases accessible through JDBC and for the Java Data Objects framework. For testing purposes, an in-memory storage is provided. Database-specific implementations initialize connections to the database and perform queries and updates according to the operations invoked on the core model.

[1] Database Repository for Applications using Graph-Oriented Storage, previously called *Gras/GXL*.



(a) PROGRES Query (b) DRAGOS Query

Figure 3: Sample query searching three connected nodes

Graph model. DRAGOS offers a rich graph model originally inspired by the Graph eXchange Language (GXL) [HWS00]. Among other things, DRAGOS supports hierarchical graphs including graph-crossing connections. Nodes, graphs, edges and relations are treated as first-class citizens, and thus can be identified and attributed. This enables flexible connections between entities, e.g. edges connecting edges and the attribution of all entities. All entities need to be typed by some graph entity class. Type structuring is supported, including multiple inheritance.

3 Queries & Transformations for DRAGOS

In this section, we present the Query & Transformation Language by means of an example, relating it to the well-known graph transformation language PROGRES [SWZ99]. The language's abstract syntax is presented and its semantics are sketched. Unfortunately, no comprehensive definition of the DRAGOS Query & Transformation Language can be given here due to the lack of space. Also, only the *query* aspect of the language is handled in this paper. For the transformation of graphs, the reader is referred to [Wei07].

3.1 Introductory example

Figure 3a shows a simple visual query modeled using the PROGRES graph transformation language. This query checks whether three nodes connected by edges of proper type and direction exist in the host graph. Another (intuitive, but rather implicit) condition is that indeed two different nodes ‘1 resp. ‘2 exist.

As the DRAGOS graph model is a lot more complex than the PROGRES model, queries according to the PROGRES syntax would be hard to represent. Therefore, the Query & Transformation Language separates between graph entities to be *searched* from the conditions that need to be *fulfilled* by these entities. The DRAGOS query shown in Figure 3b contains a set of *variables* (middle row, depicted as circles). In order to confirm the query, each of these variables has to be bound to a graph entity from the host graph, otherwise the query fails.

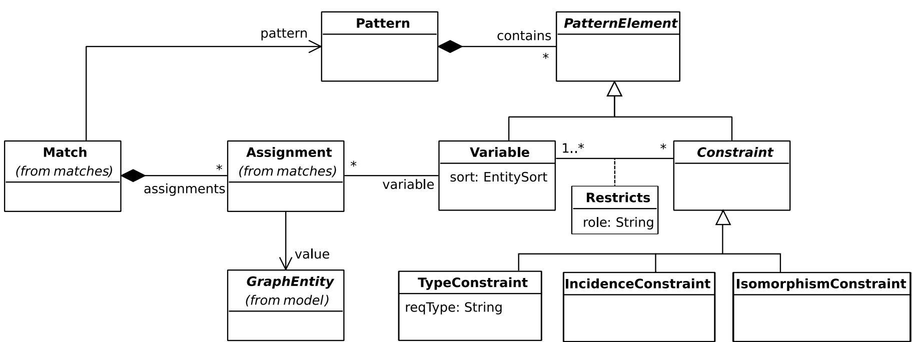


Figure 4: Meta-Model of the Query & Transformation Language (simplified)

Constraints (depicted as diamonds in Figure 3b) are used to restrict the queries' results in several ways: IncidenceConstraints demand connectivity of entities, using role names to distinguish between variables for the source, the target and the connector. This distinction is necessary as DRAGOS allows edges to be connected to other edges, and so querying these structures needs to be supported. TypeConstraints restricts legal values to a certain type, where the desired type is indicated by the reqType attribute. The IsomorphismConstraint is used to ensure that attached variables are bound to *pairwise different* entities. Its name stems from the theoretical concept of searching an isomorphic mapping of queried entities to host graph entities, although it could be called NonIdentityConstraint as well. It is only added between variables of the same type, as inheritance is not considered in the current example.

3.2 Syntax & Semantics

The language's abstract syntax is depicted in Figure 4 by means of its meta-model. According to this model, each Pattern consists of a set of PatternElements, which are sub-divided into Variables and Constraints. Constraints are connected to at least one Variable via Restricts edges, which can be distinguished using the role attribute. To support manipulation of graphs, the complete meta-model additionally provides Operators, which are not discussed in this paper.

Figure 4 only defines the basic structure of patterns, but does neither define static semantics (e.g. well-formedness of patterns) nor dynamic semantics (the actual meaning of the pattern). Here, these two kinds of semantics are introduced for a small subset of the Query & Transformation Language. We utilize the OMG's Object Constraint Language (OCL), as it allows to combine first-order predicate formulae with object-oriented concepts. Nevertheless, it should be noted that the OCL has not been comprehensively defined in a formal way, so that no unique interpretation of the presented formulae can be given. However, several research activities [BW02] strive to define the OCL's semantics, which would lead to an unambiguous understanding.

Besides the language's meta-model depicted above, several well-formedness conditions for patterns exist, which cannot be expressed using class-diagrams in a convenient way. For example, the following OCL invariant defines conditions on the IncidenceConstraint:

```
context IncidenceConstraint
  def: 𝒮: Collection(Variable) = self.restricts→select(r | r.role = "src")
  def: 𝒯: Collection(Variable) = self.restricts→select(r | r.role = "trg")
  def: 𝒞: Collection(Variable) = self.restricts→select(r | r.role = "conn")

  inv: wellformedness =
    self.𝒞→size() = 1 and self.𝒞.sort = VariableSort.EDGE and
    self.𝒮→size() ≤ 1 and
    self.𝒯→size() ≤ 1 and
    1 ≤ self.𝒮→size() + self.𝒯→size()
```

This invariant requires that the constraint is connected to exactly one Variable via a Restricts edge with role conn (connector). This variable has to specify the meta-class EDGE, i.e. it must query edges from the database. In addition, either a unique source variable (role src), or an unique target variable (role trg), or both, have to be given.

An assignment of graph entities to a Pattern's Variables not violating any Constraints is called a Match. Matches are instantiated by the language implementation according to the given Pattern and the contents of the graph database. As specified by the class diagram, each Match holds a (possibly empty) set of Assignments, each of which points to a Variable and its corresponding value. In addition, Matches have to comply to the following invariants.

```
context Assignment
  inv: validity =
    (self.variable.sort = VariableSort.NODE implies self.value.oclIsTypeOf(Node)) and
    (self.variable.sort = VariableSort.EDGE implies self.value.oclIsTypeOf(Edge)) and
        [...]
```

The *validity* invariant requires that each Assignment relates Variables to *proper* entities in the database. Therefore, i.e. a Variable of sort EDGE may only be related to an Edge in the database.

```
context Match
  inv: completeness =
    let 𝒱: Collection(Variable) =
      self.pattern.contains→select(oclIsKindOf(Constraint))→collect(c | c.variable)
    in self.assignments→collect(a | a.variable)→includesAll(𝒱)

  inv: uniqueness =
    self.assignments→forAll(a1 | self.assignments→forAll(a2 |
      a1.variable = a2.variable implies a1 = a2))

  inv: correctness =
    self.pattern.contains→select(oclIsKindOf(Constraint))→forAll(c | c.fulFilled(self))
```

Besides the Assignments' validity, a Match has to be complete, unique, and correct.

- For *completeness*, an Assignment has to exist for all Variables which are referred to by any of the Pattern's Constraints. Hence, all restricted Variables must have a value assigned.
- The *uniqueness* invariant demands that each Match holds at most one Assignment for each Variable. This restriction eases the definition of Constraints.
- *correctness* means that a Match fulfills every Constraint of its Pattern. Fulfilledness is defined depending on the respective Constraint's type (see below).

```
context TypeConstraint
  def: fulFilled(m: Match): Boolean =
    self.variable→
      forAll(v | m.assignments→select(a | v = a.variable).value.type = self.reqType)

context IncidenceConstraint
  def: fulFilled(m: Match): Boolean =
    let c = m.assignments→select(a | 𝒞 = a.variable).value
    in (𝒮 →isEmpty() or m.assignments→select(a | 𝒮 = a.variable).value = c.source)
      and (𝒯 →isEmpty() or m.assignments→select(a | 𝒯 = a.variable).value = c.target)
```

A TypeConstraint is fulfilled iff the values of all attached variables are of the type demanded by its reqType attribute. This definition does not consider any type hierarchy. The IncidenceConstraint demands that the edge assigned to the connector variable (the singleton collection $\mathscr{C}$) is the source resp. the target of the corresponding variables. This restriction only applies if an according variable is connected to the constraint.

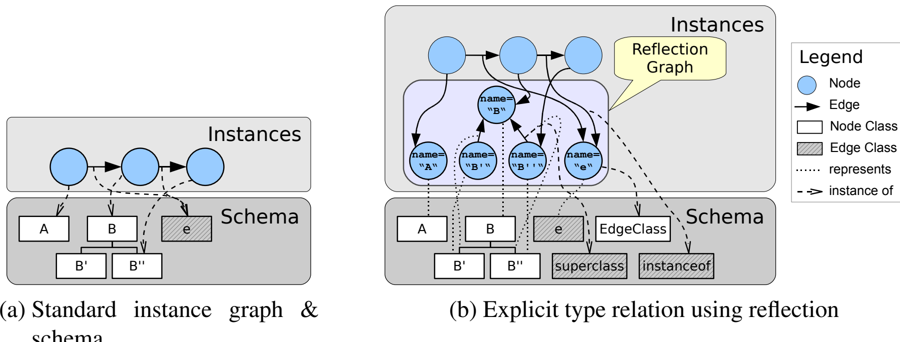
The presented invariants (partially) define the validity of Matches, but do not state how such an assignment can be computed. Language implementations therefore need to provide an operational implementation of these invariants.

4 Extending the Query & Transformation language

The core language defined in the previous section allows to model queries using a basic set of language constructs. This section introduces a technique to add additional constructs to the language, e.g. to represent special semantics of a high-level language. As example, the TypeConstraint mentioned above is extended to support *type inheritance*. This is achieved by adding an additional constraint to the language's meta-model, and by reducing its intended semantics to those of existing constraints.

4.1 Type-level reasoning

The reduction of constraints usually requires to reason on the entities' types and their relations. For this purpose, we added a mechanism which *reflects* the database graph schema into the runtime graph, as shown in Figure 5. On the left side (Figure 5a), the standard situation using separate instance and schema models is shown. Dashed arrows indicate an entities' type. However, the Query & Transformation Mechanism is not able to traverse this relation or examine the entities' types. Therefore, Figure 5b reflects the graph schema into the runtime data as special Reflection Graph. Node classes and edge classes are represented by nodes in this graph, with attributes storing the types' names. Edges model the inheritance relations. Additional edges connect entities of the regular instance graph to nodes representing their types in the Reflection Graph. The Query & Transformation Mechanism can therefore traverse and analyze this graph in the same way as regular instance graphs are handled. For the sake of clarity, some represents and instance of lines are omitted in the figure.



(a) Standard instance graph & schema

(b) Explicit type relation using reflection

Figure 5: Reflection graph to query types

4.2 Basic constraint reduction

To revive the initial example, Figure 6 (left side) shows an additional SubTypeConstraint used to check an entities type compatibility. Just like the regular type constraint, it receives the type's name by the reqType attribute. In order to evaluate this constraint based on the core language, it is *reduced* to the query on the right. Node variable ① corresponds to the original variable. An IncidenceConstraint is used to traverse the instanceof relation, as checked by the TypeConstraint of edge variable ②. The value of ③ is a node in the reflection graph representing the entities class.

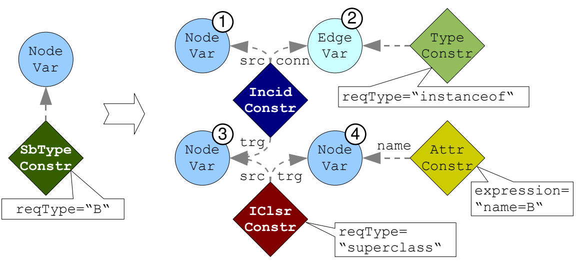


Figure 6: Definition of the SubTypeConstraint

From variable ③, a so-called IncidenceClosureConstraint traverses an arbitrary (including zero) number of edges, just like the Kleene star operator does for regular expressions. In contrast to the IncidenceConstraint, this constraint is not connected to any edge variable, as an unknown number is traversed during pattern matching. To restrict the traversed edges to a certain type, the IncidenceClosureConstraint expects an edge class passed as value of the reqType attribute. In this case, the type superclass is given, whose instances model inheritance relations in the reflection graph. According to this relation, entities assigned to the target variable ④ are again nodes of the reflection graph representing classes. Another AttributeConstraint checks the respective class name, only retaining the class named B as valid assignment.

As result of this transition, the SubTypeConstraint is fulfilled iff the pattern on the right of Figure 6 is fulfilled. For variable ①, the value's type ③ is retrieved, and all reachable supertypes are

checked whether they carry the requested name B. Variables ③ and ④ may get the same entity assigned, so the case that the value of ① is an instance of class B is covered, too. Furthermore, the reachability check also supports multiple inheritance offered by DRAGOS.

The replacement shown in Figure 6 can be expressed easily by a graph transformation rule. This replacement rule is run in a pre-processing phase before invoking the resulting query.

4.3 Nested pattern matching

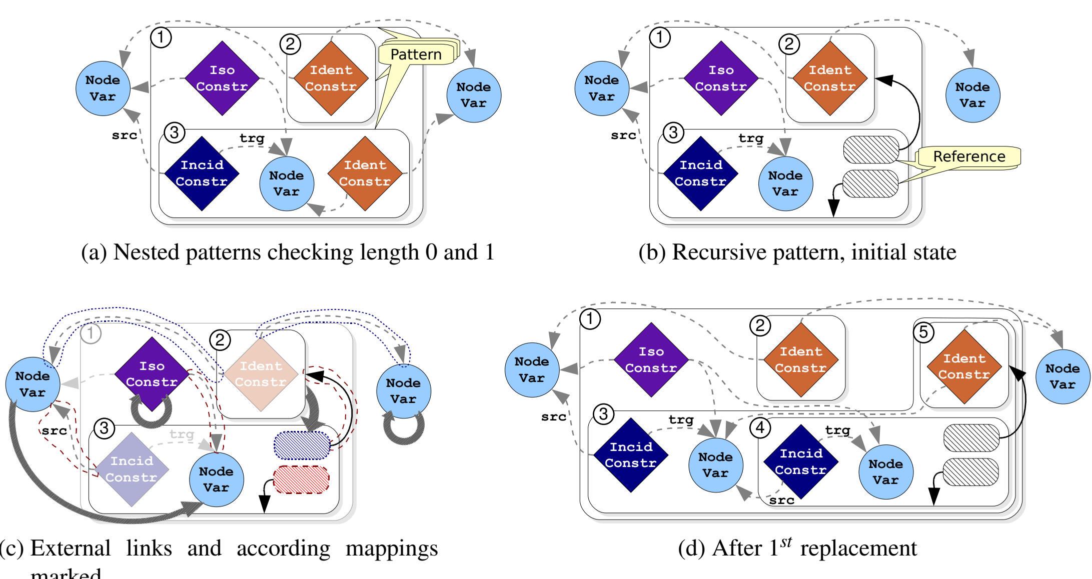
The previous subsection demonstrated a simple conversion rule to replace extended language constructs by basic ones. The DRAGOS Query & Transformation Language additionally allows to replace parts of a rule *recursively*, which is necessary to define the IncidenceClosureConstraint used above. Our approach for recursive replacements is based on the idea of *nested queries*, which is presented in the following.

On the syntactic level, the Query & Transformation Language meta-model is extended by adding Pattern to the subclasses of PatternElement (c.f. Figure 4), so that its instances may contain other patterns. Furthermore, class Match gains a reflexive association to model nested matches. For all matches, this relation has to be coherent with their respective patterns' nesting. This condition implies that a child pattern is evaluated only if a match of its parent pattern exists.

Semantically, nested patterns are matched independently from each other if constraints only refer to variables of the same pattern. The resulting set of matches (if "joining" assignments of parent and child matches) is the cross-product of matches of non-nested patterns. However, there are two possible interactions between parent and child patterns: *Firstly*, constraints can restrict variables of child patterns. As the child pattern's variable is not bound when checking fulfilledness of the parent pattern, such constraints cannot be verified. Fulfilledness of the constraint's pattern therefore only demands that no constraint is violated, thus allowing unevaluable constraints to persist. In addition, a pattern is matched only if no constraint of any ancestor pattern is violated by its variable assignments. *Secondly*, variables can be restricted by constraints of child patterns. Here, the common conditions for non-nested queries suffice, demanding fulfilledness (more generally non-violatedness) of a pattern's constraints. *However*, references between entities of *sibling* patterns are *forbidden* to keep matches independent of each other.

A final aspect on nested queries that needs to be addressed here is the *processing* of the resulting match structure. As result, we determine the validity of a match with regards to its child matches. From the application's point of view, an invalid match is treated as non-existent. Match validity can be specified w.r.t. two criteria: The *pattern condition* ensures that a match contains appropriate child matches for a *specific* child pattern. One usage of this condition is to reason on the number of these matches, e.g. *at most zero matches* to model negative application conditions. In the following, nested patterns are treated according to the intuitive *at least one* cardinality. Another approach is the *group condition*, which specifies the treatment of distinct child patterns (if any, otherwise the condition is true). Here, e.g. a boolean operator such as $\vee$ or $\wedge$ can be applied on the pattern conditions' results. In the following, we assume an $\vee$ condition, so that *at least one match* for *at least one child pattern* has to be found.

Figure 7a shows a nested pattern searching for paths of length 0 or 1. The outer variables are assumed to be bound before, in surrounding parent pattern. Pattern ① contains a single IsomorphismConstraint. As only the outer variables are bound when searching for matches of



(a) Nested patterns checking length 0 and 1

(b) Recursive pattern, initial state

(c) External links and according mappings marked

(d) After 1st replacement

Figure 7: Patterns for incidence closure

①, this constraint is always fulfilled. Therefore, a single match without any assignments exists for this pattern. The inner pattern ② checks whether the outer variables have the same value assigned, which represents a path of length 0. In contrast, pattern ③ traverses an edge (according variable and type check are omitted here), and checks whether the reached node equal the outer-right variable's value. The IsomorphismConstraint of ① requires that the target node is not identical to the outer-left variable's value, to exclude reflexive edges. Processing rules discussed above state that at least *one* of these nested patterns need to be matched to obtain a valid match for ①.

4.4 Recursive constraint reduction

Although nested queries allow to express *alternative* patterns, they can only be used to check a limited number of variables. Usually, this number cannot be given in advance, e.g. the IncidenceClosureConstraint requires to check for paths of an *arbitrary* length. The only, albeit impossible, solution would be the specification of an infinite number of patterns. Therefore, we apply a mechanism for recursive expansion of queries at *runtime*.

The language's meta-model is extended by a PatternReference class, which references a pattern defined by the developer. References are replaced by the corresponding pattern when its container pattern is matched successfully. Recursion is achieved by copying a pattern into itself, also copying the reference being expanded. If multiple references exist within the same pattern, their order of replacement is undefined. However, consistency conditions introduced below ensure that the result is indeed independent of this order. Furthermore, reference expansion should be guaranteed to terminate in recursive situations. Although this property is not ensured directly, expansion can only occur whenever a pattern is matched. Therefore, termination of reference expansion is given if only a finite number of patterns can be matched. Obviously, this can be

achieved by an IsomorphismConstraint limiting at least one variable per pattern to an entity not assigned to other variables. Therefore, finiteness of the host graph implies finiteness of matched patterns and expansion steps.

The actual application of pattern references is introduced by referring to the IncidenceClosureConstraint. Figure 7b shows a variant of the nested pattern introduced above. In contrast to Figure 7a, pattern ③ does not contain an own IdentityConstraint to check the connectedness of the path ends. Instead, two PatternReferences are given: The upper one refers to pattern ②, which means that this pattern is copied *into* pattern ③ if the latter can be matched. Furthermore, the lower reference copies pattern ③ into itself.

Reference expansion is conducted as follows: Each PatternReference is replaced by a new pattern created inside the reference's container, and filled with copies of the referenced pattern's entities. This covers the entities' types, attribute values, and connectedness to other copied entities. However, the question remains how the copied pattern's *context* should be handled. This context is defined by the edges connecting its contained entities to entities not contained in the pattern being copied, the so-called *external links*. Figure 7c highlights the external links of Figure 7b for both copied patterns. Here, this concerns Restricts edges (four times), but also the pattern referred to by the upper PatternReference.

For each pattern reference, the developer has to specify a *mapping* of entities connected to external links, relating them to entities that should be connected to the referred pattern. Identical mapping of an element to itself is a valid choice. Mappings are copied along with other pattern entities during reference expansion, which is required for recursive expansions.

In order to achieve the desired replacement in case of the IncidenceClosureConstraint, the following mappings are required (c.f. broad arrows in Figure 7c):

- The *upper reference* copies pattern ② into pattern ③ to check value-identity of the outer-right variable and the variable of ③. Therefore, the outer-left variable referenced by ② is mapped to the variable of ③, whereas the outer-right variable is mapped identically.

- Expansion of the *lower reference* should yield a query for path of length 2. Therefore, the same mapping of the outer-left variable to the variable of ③ applies here, such that the IncidenceConstraint of the *copy* of pattern ③ refers to the original's variable as source.

- The traversed node should not have been visited before, so all node variables are connected to the IsomorphismConstraint of ①, which is mapped identically for this purpose. This constraint ensures termination of the replacement, as discussed above.

- A last external link of the lower reference is the pattern referred to by the *upper* reference. Here, pattern ② is mapped to its reference, such that the copied reference will refer to the *expanded* upper reference of ③. In this case, identical mapping would lead to broken copied mappings in later expansion steps, if the lower reference is expanded first.

Using these mappings, expanding both references yields the pattern structure shown in Figure 7d. Expanding the upper reference results in ⑤, whereas the lower one is expanded to ④. The resulting query checks for paths of length 0 by matching ① and ②, and 1 by matching ①,③, and ⑤, respectively. Paths of length 2 can be found after the next step, using ①,③,④, and the expanded reference to ⑤.

This section showed how complex or application-specific language constructs (represented by constraints) can be reduced to basic ones. With the presented nested query mechanism, recursive expression can be captured as well. Although its evaluation might be inefficient, it serves as the guideline for implementing the DRAGOS Query & Transformation Language. This is required by the fact that the actual storage backend of DRAGOS is exchangable, and so is the implementation of its language. As discussed in [Wei07], such implementations may either rely on the DRAGOS core graph model, or convert rules into a backend-specific format. e.g. SQL statements. To provide an efficient implementation, language extensions might also be converted into such a backend-specific language. The modeled reduction rules in this case serve as the formal definition and as reference used in test-based validation of the specific implementations.

5 Related Work

In contrast to previous publications on DRAGOS [Böh04] and the according Query & Transformation Language [Wei07], this paper focusses on the language's extensibility. In this section, we give a brief comparison to other research in the area of graph transformations.

Graph transformations based on constraint satisfaction. The DRAGOS Query & Transformation Language is based on the theory of *constraint satisfaction problems* (CSPs) known from artificial intelligence. CSPs are well-suited to model graph pattern matching by solving the *subgraph-isomorphism problem* [LV02]. Algorithms have been proposed for this purpose which circumvent efficiency concerns arising from the problem's complexity in most situations [FSV01]. In our work, we implement the proposed language based on existing databases, and therefore extensive development of a basic constraint solver is not of crucial importance. Instead, we focus on implementations based on sophisticated storage backends like databases.

Graph transformations on databases. As briefly mentioned in Section 4, we not only provide an operational implementation of the presented language. In addition, queries and transformation rules can be converted into a language offered by the respective storage backend, e.g. SQL. Building GTS on this language has been presented by [VFV06], which is based on the construction of database views and update operations from graph transformation rules.

Ongoing work in our project generalizes this idea by deriving SQL statements from the more expressive DRAGOS Query & Transformation Language. Furthermore, its language structure allows an easier processing, as it already separates between variables and constraints. Therefore, we are able to apply an extensible rule language on various storage backends, not limited to the SQL or one of its DBMS-specific variants. Implementations do not need to cover the entire DRAGOS Query & Transformation Language, as extended language constructs may be reduced to basic ones. Moreover, evaluation may fall back on a generic implementation only based on the DRAGOS graph model, which is independent of the actual storage backend.

Complex pattern matching. A large amount of recent publications deals with the representation and semantical definition of complex graph patterns. Besides recent work in our own

department, [Ba07] proposes set-valued graph patterns by grouping, and [LLP07] discusses repeated pattern structures beyond binary path expressions. [DHJ+07] utilizes graph grammars to build transformation rules, similar to two-level grammars.

The stepwise extension of pattern references can be seen as a simple graph grammar, and indeed provides similar functionality. Therefore, arbitrary repeated structures can be expressed by building nested queries, as shown in Figure 7. This is not limited to binary path expressions, but can be applied for n-ary structures as well. The definition of proper sub-pattern interfaces, expressing how sub-patterns are to be glued together, is given by a consistent element mapping.

Graph transformation languages for visual programming. Graph transformation languages like PROGRES provide similar functionality to the DRAGOS Query & Transformation Language. In fact, PROGRES can already generate code to store the runtime data persistently using DRAGOS. However, this approach leads to inefficient applications because the generated code performs many simple operations on the DRAGOS graph model. In our approach, DRAGOS interprets transformation rules itself, and hence may utilize storage backends more appropriately.

In contrast to common graph transformation languages, the low-level DRAGOS Query & Transformation Language is not feasible for direct use by a specificator. Therefore, it should not be considered as competitor to existing languages, but as a common core for existing and new languages to build on.

6 Conclusion

In this paper, we introduced the Query & Transformation Language currently being developed for the DRAGOS graph database. This language especially focuses on extensibility, which is the core aspect of this publication. Developers may choose to add new constructs to the language in case existing ones do not suffice the application's needs or do not match its semantics. These are implemented by reduction to existing ones, also allowing recursive substitutions. In addition, language constructs may be converted into a storage-specific query such as SQL statements.

The presented work is fully implemented based on the DRAGOS graph model interface, designated as *generic* implementation in [Wei07]. Currently, we are working on an SQL-based solution. Interesting problems remain in the recursive evaluation of queries, which cannot be expressed directly in many database systems[2]. Upon completion, we will conduct performance evaluations comparing the Query & Transformation Language to DRAGOS applied in the code generation approach. Furthermore, comparisons to other graph transformation solutions based on databases are of interest.

Currently, we are embedding support for control flow into the language definition and its generic implementation. Core features of this mechanism include hierarchical rule composition, optional dataflow and rule invocation. Rule application strategies will allow non-deterministic and random (with or without backtracking) processing of multiple matches. Using this mechanism, rules can be combined to complex graph transformation systems.

As next step, we will investigate which additional language constructs are required to support different approaches to graph transformations, such as the algebraic approach or hyper-edge

[2] Altough recursive SELECT statements are defined by SQL3, support is optional and obviously not very popular.

replacement grammars [Hab93]. This way, DRAGOS can serve as a platform to develop new constructs for graph transformation languages by offering a high-level extension mechanism.

Bibliography

[Ba07] D. Balasubramanian, et al. Applying a Grouping Operator in Model Transformations. Pp. 406–421 in [SNZ07].

[Böh04] B. Böhlen. Specific Graph Models and Their Mappings to a Common Model. In Pfaltz et al. (eds.). LNCS 3062, pp. 45–60. Springer, 2004.

[BW02] A. D. Brucker, B. Wolff. A Proposal for a Formal OCL Semantics in Isabelle/HOL. In Muñoz et al. (eds.), *Theorem Proving in Higher Order Logics*. Lect. Notes in Comp. Sci. 2410, pp. 99–114. Springer, Hampton, VA, USA, 2002.

[DHJ+07] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, N. V. Eetvelde. Shaped Generic Graph Transformation. Pp. 197–212 in [SNZ07].

[FSV01] P. Foggia, C. Sansone, M. Vento. A performance comparison of five algorithms for graph isomorphism. In *Proc. of the 3rd IAPR TC-15 Workshop on Graph-based Representations in Pattern Recognition*. Pp. 188–199. 2001.

[Hab93] A. Habel. *Hyperedge Replacement: Grammars and Languages*. Lect. Notes in Comp. Sci. 643. Springer, 1993.

[HWS00] R. Holt, A. Winter, A. Schürr. GXL: Towards a Standard Exchange Format. In *Proc. of the* 7th *Working Conference on Reverse Engineering (WCRE '00)*. Pp. 162–171. IEEE Computer Society Press, 2000.

[LLP07] J. Lindqvist, T. Lundkvist, I. Porres. A Query Language With the Star Operator. In Ehrig and Giese (eds.), *Graph Transformation and Visual Modeling Techniques*. ECEASST 6, pp. 69–80. 2007.

[LV02] J. Larrosa, G. Valiente. Constraint Satisfaction Algorithms for Graph Pattern Matching. *Mathematical Structures in Computer Science* 12(4):403–422, 2002.

[SNZ07] A. Schürr, M. Nagl, A. Zündorf (eds.). *Applications of Graph Transformation with Industrial Relevance (AGTIVE'07), preliminary proceedings*. 2007.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.). Pp. 487–550. Volume 2. World Scientific, 1999.

[VFV06] G. Varró, K. Friedl, D. Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal on Software and Systems Modeling* 5(3):313–341, 2006.

[Wei07] E. Weinell. Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. Pp. 390–405 in [SNZ07].

Improving Live Sequence Chart to Automata Transformation for Verification

Rahul Kumar[1] **and Eric G Mercer**[2]

[1] rahul@cs.byu.edu, [2] egm@cs.byu.edu
Computer Science Department
Brigham Young University
Provo, UT, USA

Abstract:

This paper presents a Live Sequence Chart (LSC) to automata transformation algorithm that enables the verification of communication protocol implementations. Using this LSC to automata transformation a communication protocol implementation can be verified using a single verification run as opposed to previous techniques that rely on a three stage verification approach. The novelty and simplicity of the transformation algorithm lies in its placement of accept states in the automata generated from the LSC. We present in detail an example of the transformation as well as the transformation algorithm. Further, we present a detailed analysis and an empirical study comparing the verification strategy to earlier work to show the benefits of the improved transformation algorithm.

Keywords: live sequence charts, transformation, automata, verification, non-determinism

1 Introduction

Current trends in system development are shifting towards creating and developing larger systems using several smaller communicating sub-systems. With the increasing popularity of such modular designs comes the burden of creating, implementing, and testing the implemented communication protocols. Specification of communication protocols has been explored significantly in the past. English, which has been traditionally used as the most common language for specifying protocols, lacks the formal rigor and preciseness needed for clarity. Viable alternatives are formal specification languages such as UML, Message Sequence Charts (MSCs) and Live Sequence Charts (LSCs) [IT93, DH99, BDK+04]. The evolution of these graphical languages has led to their application to modeling and specifying communication behaviors in a variety of different domains [BHK03, KHG05, DK01]. Other research has also investigated the automatic synthesis of systems from LSCs as well as the verification and validation of requirements on the LSCs themselves [HK01, AY99, SD05]. Efficient methodologies for using these graphical languages in a formal verification environment provide the support in the development process to completely certify, test and develop a system. Since LSCs are a more expressive and semantically rich visual specification language compared to MSCs, Timing Diagrams and Sequence Diagrams in UML, we focus on techniques related to LSCs. Due to the encompassing nature of LSCs, the techniques and algorithms presented in this paper are also applicable to the afore

mentioned specification languages.

Previous work in [KTWW06, Klo03] presents a strategy to verify systems against LSC specifications by transforming the LSC to a *positive* automaton. We use the term positive automaton to denote automaton that witness chart completions. With the positive automaton, a system is verified against the LSC in three stages: reachability analysis for detecting safety violations, ACTL verification for detecting liveness errors, and finally, if the first two steps fail to provide a significant result, full LTL verification is required to completely verify the system. The authors argue that the verification algorithms are applied in increasing order of cost and for certain sub-classes of LSCs not all algorithms need to be applied, which can eventually save on the total verification cost. Although the approach presented in [KTWW06] is sound, it has several drawbacks. For any arbitrary LSC, the approach at a minimum has to apply reachability analysis as well as ACTL model checking for verifying the safety and liveness properties of the system against the LSC. In the worst case, LTL verification is required to completely verify the system, which was shown to be impractical for LSC verification [KM07]. Another drawback of the verification approach is the specialized algorithms and tools that have to be created to perform the verification, which limit the general applicability and acceptance of the approach. The approach presented in this paper only requires one verification algorithm of the same cost as reachability analysis to completely verify a system against any arbitrary LSC.

We present a direct and obvious transformation of the LSC to a *negative* automaton by changing the placement of accept states. We use the term negative automaton to denote automaton that witness chart violations as opposed to chart completions. Using this improved LSC to automaton transformation a system can be formally verified against the LSC specification by performing only language containment on the parallel composition of the system automaton and the negative automaton of the LSC created using the transformation algorithm presented in this paper. Additionally, this approach does not require the use of customized algorithms and tools to verify a system against a specification. Using our new LSC to automaton transformation, we verify systems against larger more concurrent LSCs that were previously not verifiable with direct LSC to LTL or LSC to positive automaton transformations.

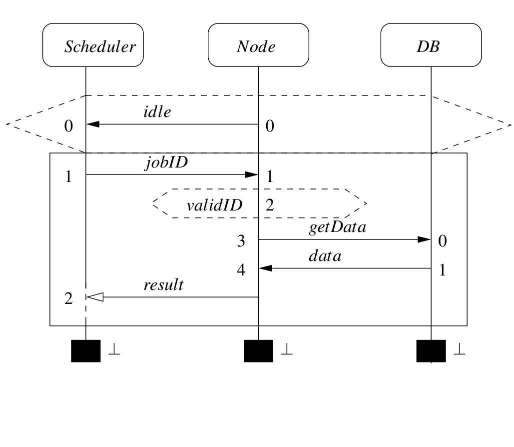
The structure of the paper is as follows. Section 2 presents a brief introduction to LSCs and an overview of the basic LSC to automaton transformation algorithm as described in [Klo03]. Section 3 discusses in detail an example of using our approach for verifying a system against an LSC. This example will be used for the remainder of the paper as well. Section 4 discusses the details of the transformation algorithm and presents the theoretical results to prove the correctness of the transformation algorithm. Section 5 presents an analysis of the improved transformation compared to the old transformation presented in [Klo03]. Section 6 presents a subset of the results using the improved verification approach in both symbolic and explicit state model checkers. Finally, 7 discusses the conclusions and future work. Proofs, details and additional results can be found in the long version of the paper at `http://vv.cs.byu.edu/~rahul/lsc2automata.pdf`.

2 LSC Overview

We briefly introduce some constructs of the LSC grammar[1]. Fig. 1(a) shows an example LSC where an idle node in a compute cluster requests and processes a job from the scheduler's queue with a possible implementation of the *Node* and *DB* process in Fig. 1(b). There are three *processes* in the example LSC: *Scheduler*, *Node* and *DB*. Each process is drawn with a rectangular instance head and a vertical life-line originating from each instance head. The life-line represents the time dimension in the LSC with time progressing in the downward direction. Communication between processes occurs via *messages* with the arrows representing the direction of communication. The *idle* message is an example of a *synchronous* message (filled arrowhead) where both the sender and receiver have to be ready for the message to be observed. The actual message communication occurs instantaneously for the sender and receiver. The *result* message is an example of an *asynchronous* message (unfilled arrowhead) where the sender does not have to block for the receiver to be ready to receive the message. The send event is written as *result*! and the receive event is written as *result*?. The example LSC also contains a *cold non-bonded condition* (second dashed hexagon) which enforces the *validID* predicate after a *jobID* has been received from the *Scheduler*. If the condition is violated, the *Node* process exits the chart. On each life-line any point where a condition or an event occurs is referred to as a *location*. Locations are unique to each life-line and in our research are represented by numbers next to the instance life-line. By default all locations are *hot* or *mandatory* locations unless specified otherwise using a dashed line for the life-line. The location for receiving the *result* message in the *Scheduler* life-line is the only cold location in the example chart. The behavior specified on a cold location is not mandatory, which implies that the *result* message may or may not be received by the *Scheduler*. Finally, behaviors described by the LSC are partitioned into the *pre-chart* (dashed hexagon before solid rectangle) and the *main chart* (rectangle after pre-chart). The pre-chart specifies the activation condition of the LSC and the main chart describes the behavior which must follow the pre-chart. In the example LSC, the main chart is a *universal* main chart (solid line), which represents behaviors that have to be observed every time the pre-chart is satisfied.

In addition to the constructs shown in the example LSC, several other constructs are also available. The main chart can be specified as an *existential* chart (drawn with a dashed rectangle) that specifies behavior the system must satisfy at least once when the pre-chart is satisfied (as opposed to every time the pre-chart is satisfied). Conditions if attached to another event are *bonded* otherwise *non-bonded*. By attaching conditions to other events, the condition is evaluated at the exact moment the bonded event occurs, as opposed to non-bonded conditions where the condition is continuously evaluated until satisfied. LSCs also allow the specification of *invariants* which are conditions spanning over multiple events in the LSC. *Co-regions* specified with a dashed line parallel to a life-line allow events to occur in any order. For example, if the messages *getData* and *data* are specified in a co-region, either message *data* or *getData* may occur first. It is only necessary for all events in a co-region to occur. Finally, conditions, messages, and locations may be specified as *hot* or *cold*. If drawn with a solid line, the construct is hot and specifies mandatory behavior, and if drawn with a dashed line, the construct specifies cold or provisional behavior.

[1] See [DH99, BDK+04] for details.



(a)

```
Process Node:
1:  if(idle) then
2:    Send(``idle'', Scheduler)
3:    Receive(``jobID'', Scheduler)
4:    if(not ``validID'')
5:      break
6:    Send(``getData'', DB)
7:    Receive(``data'', DB)
8:    Send(``result'', Scheduler)
9:  endif
10: End Process Node
11: Process DB:
12:  while(true){
13:    Receive(msg, Node)
14:    if(msg is ``data'')
15:      Receive(``data'', Node)
16:    endif
17:      RemoveData(data)
18:    endif
19:  end while
20: End Process DB
```

(b)

Figure 1: An example specification describing the interaction between a cluster node (*Node*), a database (*DB*) and a job scheduler (*Scheduler*), and a possible implementation of the *Node* and *DB* processes (a) The example LSC containing a subset of the complete LSC grammar (b) A system implementing the *Node* and *DB* processes described in the LSC.

Our method supports all the mentioned constructs of LSCs with the following commonly accepted restrictions. First, we adopt the *strict* interpretation of LSCs (i.e., no duplicate message instances are allowed within a chart). Second, the LSC and all charts within the LSC are to be acyclic. Third, we also do not consider overlapping LSCs or iterative LSCs (Kleene stars) where multiple instances of the chart may be executed simultaneously. Since most scenario based specifications in general do not deal with the constructs omitted from this research, the restrictions do not affect the general applicability of our results.

2.1 Transforming Live Sequence Charts to Automata

Past research in the area of transforming LSCs to automaton has primarily revolved around the generation of positive automaton that detect chart completions [Klo03, HK01, BH02, KW01]. Work in [Klo03] gives a detailed presentation of the algorithm to transform an LSC to positive automaton. We present an overview of this algorithm followed by a discussion of some key aspects of the algorithm.

The LSC to automaton unwinding algorithm explores all possible inter-leavings of the events defined in the LSC starting from the top and ending at the bottom of each life-line in the chart. The possible event inter-leavings are explored by considering the partial order induced by the semantics of the LSC. The partial order of the chart dictates that the locations in each instance are totally ordered unless part of a co-region; thus, implying that each instance has to progress linearly from top to bottom. For example, in the chart shown in Fig. 1(a), instance *Node* cannot

move from location 1 to location 4. From location 1, *Node* has to move to the next logical location: location 2. To maintain the current state of the LSC, we define a *cut* as a set of locations in the chart with exactly one location for each instance. The cut is used to record the current state of the chart and create successor cuts. The reachable set of cuts from the initial cut is the automaton for the chart. Each state of the automaton corresponds to a reachable cut of the chart. Successor cuts are generated using the set of enabled transitions for a given cut. The initial cut for all charts is created by placing each instance at its first location, $(0,0,0)$, where the first, second and third locations correspond to the locations for the *Scheduler*, *Node* and *DB* instances.

The enabled set of transitions for a cut is created using the chart semantics. For example, a synchronous message is enabled if both the sender and receiver of the message are at their respective send and receive locations. In our chart, the message *idle* is observed if the *Scheduler* and *Node* instances are each at locations 0. At the initial cut, $(0,0,0)$, the *idle* message is enabled. On the other hand, since the *Node* is not at location 3, the *getData* message is not enabled in the initial cut, even though the *DB* is at location 0. When the *idle* message is explored from the enabled set, a successor cut is generated where the locations for the involved instances have been updated. In this case, the locations for the *Node* and *Scheduler* instances are updated to their next logical location giving us the successor cut $(1,1,0)$. At the cut $(1,1,0)$, the *jobID* message is enabled, which leads to the cut $(2,2,0)$. Asynchronous sends are enabled by default when the corresponding instance is at the send location and asynchronous receives are enabled only if the corresponding send event has occurred and the receiving instance is at the receive location. Conditions act as a synchronization point where each participating instance should be at its respective condition location for the condition to be evaluated. A full description of these semantics can be found in [Klo03]. Multiple enabled transitions lead to multiple successor cuts from the given cut representing the concurrency in the chart.

Using the chart semantics, successor cuts are generated from the initial cut and each unique cut is processed until the final cut is reached where each instance is at the bottom of its life-line. Each unique cut of the chart corresponds to a state in the final automaton. The initial cut $(0,0,0)$ corresponds to state q_0 in Fig. 2(a). The successor cut $(1,1,0)$ corresponds to the state q_1 where the *idle* message has already been observed and the next message to be observed is *jobID*. Cut $(2,2,0)$ corresponds to state q_2 and the final cut corresponds to state q_7 where no further events are to be observed. Notice that transitions taken to generate successor cuts correspond to the transition labels in the automaton.

Finally, to create the positive automaton from the LSC, states corresponding to legal exits of the chart are marked as accept states. For example, state q_7 in Fig. 2(a) is marked as an accept state because it corresponds to the final cut of the LSC which represents a legal completion of the chart. Additionally, state q_6 is also marked as an accept state since it corresponds to the cut where the cold message *result* does not have to be received.

From the automaton in Fig. 2(a) we also notice that state q_2, where cold condition *validID* occurs is non-deterministic. This non-determinism is a result of the adopted semantics of cold conditions in [Klo03]. If *validID* is not satisfied, the automaton can either stay in state q_2 and wait for the condition to be satisfied or move to the exit state q_{exit} to signify that the cold condition was not satisfied and the chart has exited successfully. This non-determinism resulting from non-bonded conditions forces the approach of [Klo03, KTWW06] to translate the LSC automaton to an LTL property and re-perform the verification using the LTL property, which has been shown to

be ineffective for even moderate size charts due to the size of the resulting LTL formula [KM07].

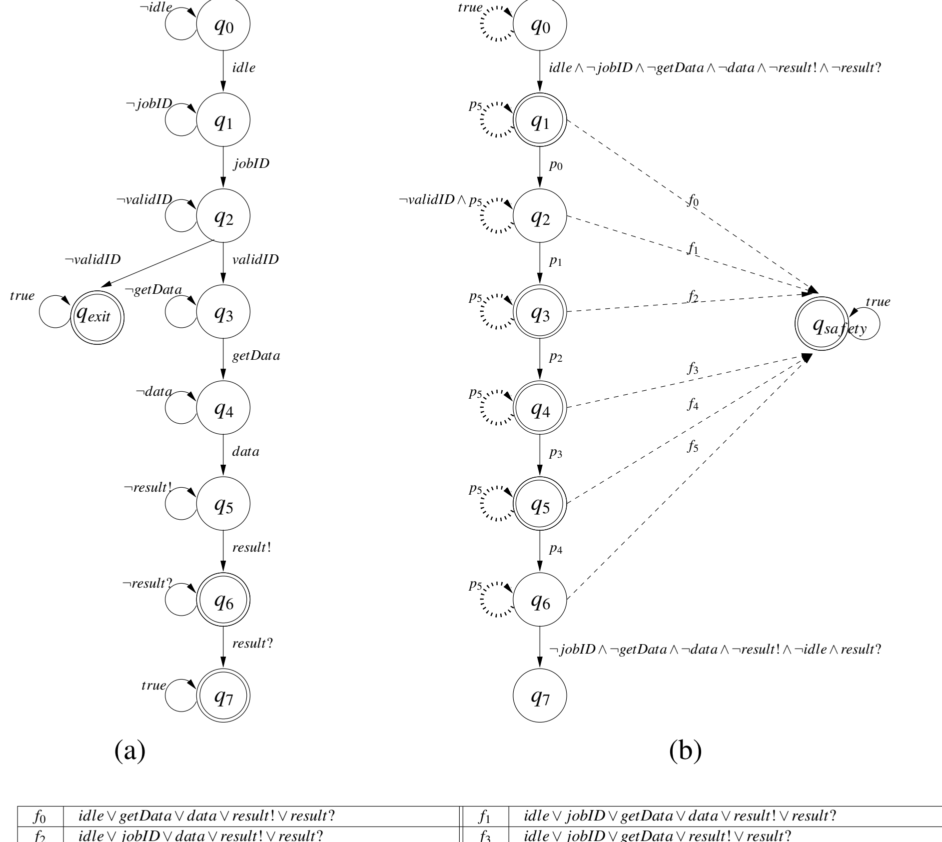
3 Transformation and Verification Example

We use the automaton produced by the unwinding algorithm discussed earlier as our initial automaton. The initial automaton from the unwinding algorithm is shown in Fig. 2(a). We transform this positive automaton to a negative automaton that can be used in our single pass verification approach. Fig. 2(b) and (c) show the transformed negative automaton.

Our approach transforms the LSC chart to a negative automaton capable of detecting chart violations (as opposed to chart completions) that is naturally suited for verifying systems using language containment. The first step in the transformation process is to remove all the accept labels from the automaton. Next the exit state q_{exit} and any transitions leading to the exit state are removed from the initial automaton. In our example of Fig. 2(a) we remove the transition from state q_2 to state q_{exit}, which also removes the non-determinism from the automaton arising from the non-bonded condition. The algorithm then introduces safety transitions (dashed edges in Fig. 2(b)) from all states that contain a transition belonging to the main chart to the safety state q_{safety}. The safety state is an accept state introduced in the automaton to capture all safety violations in the system. It has a single outgoing transition to itself predicated on $true$. The safety transitions enable the detection of safety violations which consist of duplicate messages (messages that have been observed before) and out of order messages in states that correspond to main chart states. For example, in state q_1 of Fig. 2(b), the only legal transition is if the *jobID* message is observed. Since *jobID* is a main chart transition, state q_1 corresponds to a main chart state and a safety transition is introduced. The safety transition $idle \vee getData \vee data \vee result! \vee result?$ from state q_1 to q_{safety} is taken if any message except *jobID* is observed.

After the introduction of safety transitions, the algorithm updates the self-loops on each state (dotted edges in Fig. 2(b)). The self-loops enable the automaton to remain in a given state until an event forcing progress is observed. For example, in the automaton shown in Fig. 2(b), state q_4 has a self-loop, $\neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$, that is taken until the $data$ message is observed, which moves the automaton to the next state q_5. The only exception is the self-loop for the first state and the final state. The first state q_0 contains a self-loop with the $true$ annotation to capture all possible future instances (and possible errors) of the chart in a reactive system. The final state does not have any self-loops. This is because the final state represents the successful completion of the chart and no further errors are possible unless a new chart instance is observed, which is detected in the first state.

Finally, the algorithm marks illegal end points of the main chart as accept states to facilitate detection of chart violations. For example, state q_1 in Fig. 2(b) is at the beginning of the main chart where the message *jobID* is yet to be received. If the *jobID* message is never observed, the automaton remains in state q_1 indefinitely, which should be reported as an error. To report this error, state q_1 is marked as an accept state. States containing no transitions corresponding to hot constructs in the main chart are not marked as accept states. For example, in Fig. 2(b), state q_2 is not marked as an accept state because the *validID* condition is a cold condition, and its absence does not result in an error. State q_0 is not marked as an accept state either because it does not contain any outgoing transitions corresponding to a hot construct in the main chart. If



f_0	$idle \vee getData \vee data \vee result! \vee result?$		f_1	$idle \vee jobID \vee getData \vee data \vee result! \vee result?$
f_2	$idle \vee jobID \vee data \vee result! \vee result?$		f_3	$idle \vee jobID \vee getData \vee result! \vee result?$
f_4	$idle \vee jobID \vee getData \vee data \vee result?$		f_5	$idle \vee jobID \vee getData \vee data \vee result!$
p_0	$jobID \wedge \neg idle \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$		p_1	$validID \wedge \neg jobID \wedge \neg idle \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$
p_2	$getData \wedge \neg idle \wedge \neg jobID \wedge \neg data \wedge \neg result! \wedge \neg result?$		p_3	$data \wedge \neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg result! \wedge \neg result?$
p_4	$result! \wedge \neg idle \wedge \neg jobID \wedge \neg data \wedge \neg getData \wedge \neg result?$		p_5	$\neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$

(c)

Figure 2: The initial and transformed automaton for the example LSC shown in Fig. 1(a). (a) the initial automaton (b) the transformed automaton and (c) list of transition labels.

the *idle* message is never observed, the pre-chart is not satisfied, which is not a violation of the specification. State q_6 is not marked as an accept state since the location of the *result*? event is cold implying that the *result*? event does not have to be observed. Finally, state q_7 in Fig. 2(b) is also not marked as an accept state since it is the final state where the behavior as described in the universal chart has been satisfied without errors.

Verification of the system is performed by first creating the system automaton in the usual manner. We verify the parallel composition of the system automaton and the negative automaton of the LSC by searching the behavior space of the intersection for accepting cycles. Any cycles detected correspond to errors in the system. Fig. 1(b) shows a possible implementation of the *Node* and *DB* processes in a cluster. The *Scheduler* process has not been shown in the implementation but is assumed to be correctly implemented. When idle, the *Node* process requests a job from the scheduler (line 2). The *Node* process then waits to receive the *jobID* and validates the

jobID using the predicate *validID* (lines 3 - 5). Next, the *Node* process requests data from the *DB* (line 6), processes the data and sends the result to the *Scheduler* (lines 7 - 8). The *DB* process receives and processes messages as they arrive (lines 12 - 19). In this particular implementation, the *DB* process is erroneous because it never receives/processes the *getData* message from the *Node*. Since the *getData* message is a synchronous message and the *DB* process is never ready to receive the *getData* message, the *Node* and *DB* processes never progress even though they should. Verification of the parallel composition of the system automaton (not shown) with the property automaton in Fig. 2(b) produces the word $(idle, jobID, validID, (\neg getData)^{\omega})$, with the corresponding trace: $(q_0, q_1, q_2, (q_3)^{\omega})$, where ω indicates infinite repetition. Since q_3 is marked as an accept state, the trace is reported as an accepting cycle and the violation has been discovered. Using the positive automaton in the verification approach of [KTWW06] requires two verification runs of comparable complexity to detect the same violation.

4 Transformation and Verification Details

The transformation presented in this work is based on language containment and automata theory. We use *Symbolic automata*, an extension of Büchi automata, that allows observing any of a possible set of inputs on an edge. Formally Symbolic automata are given by $A = \langle \Sigma, Q, \Delta, q^0, F \rangle$ where, Σ is the finite *alphabet* of input symbols (variables), Q is the finite set of states, $q^0 \in Q$ is the initial state, $F \subseteq Q$ is the set of final/accepting states, and $\Delta \subseteq Q \times \phi \times Q$ is the transition relation. A transition $(q, \phi, q') \in \Delta$ represents the change from state q to state q' when the formula ϕ is satisfied.

We partition the set of Boolean variables Σ into three distinct sets Σ_{msgs}, $\Sigma_{\text{invariants}}$, and $\Sigma_{\text{conditions}}$, that contain the Boolean variables that are used for messages, invariants and conditions in the chart respectively. For the chart shown in Fig. 1(a), $\Sigma_{\text{msgs}} = \{idle, jobID, getData, data, result?, result!\}$ and $\Sigma_{\text{conditions}} = \{validID\}$. The set $\Sigma_{\text{main}} = \{jobID, validID, getData, data, result?, result!\}$ is the set of Boolean variables that are used in the main chart only. We also have a set $\Delta_{\text{hot}} \subseteq \Delta$ which only contains transitions that correspond to hot constructs in the chart (hot messages, hot conditions etc.).

For a set of Boolean functions $\Gamma = \{\phi_0, \phi_1, ..., \phi_n\}$ we define the function $disjunct(\Gamma)$ which returns the disjunct of the individual formulas in Γ and the function $conjunct(\Gamma)$ which returns the conjunction of the individual formulas in Γ. The function $f(\Sigma, \phi) = \{\sigma | \sigma \in \Sigma \textit{ and } \sigma \textit{ or } \neg\sigma \textit{ appears in } \phi\}$ returns the set of Boolean variables from Σ that appear in ϕ in either a positive or negative form. For example, if $\phi = idle \wedge validID$, $f(\Sigma_{\text{msgs}}, \phi) = \{idle\}$ and $f(\Sigma_{\text{condition}}, \phi) = \{validID\}$.

We take as input the automaton structure for a chart in the form of a symbolic automata structure, A, with an empty final state set. Intuitively, to capture the bad behaviors of a chart, we transform the basic automaton structure to the negative automaton that is capable of detecting safety and liveness errors by yielding accepting cycles in the verification. We do so by adding accept states to the automaton and adding/updating all transitions.

Fig. 3 shows an intuitive description of the outgoing transitions of a state in the transformed automaton. The sets ψ_c, ψ_m and ψ_i (initialized by the algorithm in Fig. 4) are sets of condition, message, and invariant letters used in the outgoing transitions of a given state. There are three

$true$

$\phi_{\text{self}} = \neg disjunct(\Sigma_{\text{msgs}}) \wedge$
$\neg disjunct(\psi_c) \wedge$
$conjunct(\psi_i)$

$\phi_{\text{safety}} = disjunct(\Sigma_{\text{msgs}} \setminus \psi_m) \vee$
$\neg conjunct(\psi_i)$

$\phi_{\text{child}} = \phi \wedge \neg disjunct(\Sigma_{\text{msgs}} \setminus f(\Sigma_{\text{msgs}}, \phi))$

Figure 3: A generic state in the transformed automaton with complete annotations for all types of outgoing transitions 1. ϕ_{self}: self-loop for non-progress, 2. ϕ_{safety}: transition to state q_{safety} for detecting safety errors, and 3. ϕ_{child} transitions to the successor states.

types of transitions that are introduced/updated for each state in the automaton. The ϕ_{safety} transition (dashed edge) leads to the safety state and is responsible for detecting any safety errors. The self-loop (dotted edge), ϕ_{self}, enables the automaton to remain in the current state until an event or condition progresses the automaton to a successor state. The ϕ_{child} transitions (solid edges) lead to the successor states. The dash-dot edge is only added to the first state of the automaton to enable verification of multiple chart instances in a reactive system.

States are marked as accept states in the automaton based on two criteria. First, the safety state is marked as an accept state for detecting safety violations such as duplicate message instances and out of order messages. Second, any state that is not a legal exit point of the chart is marked as an accept state. We now discuss in detail the creation of the transitions and the marking of accept states.

Fig. 4 shows the algorithm for transforming the input automaton. We only present an overview of the algorithm in this version of the paper and refer the reader to the long version for more details. The algorithm has a general Depth First Search (DFS) structure with line 4 enumerating the successors and line 11 making a recursive call for each successor. The algorithm is always invoked for the one initial state of the input automaton to be transformed. Lines 1 - 2 remove any transitions to the exit state q_{exit}. In the automaton shown in Fig. 2(a), the transition from state q_2 to the exit state q_{exit} is removed. Lines 5 - 7 of the algorithm build the sets of variables that are used for messages, invariants, and conditions in the transitions from the current state to the successor states.

Lines 8 - 10 update the transitions to the successor states by first removing the transition and adding a new transition with the updated label. The updated child transition ensures that only the enabled messages, invariants and conditions at a given state can enforce progress in the automaton. For example, the algorithm transforms the transition from state q_1 to state q_2 in Fig. 2(a) from $\phi = jobID$ to $\phi_{\text{child}} = jobID \wedge \neg idle \wedge \neg getData \wedge \neg data \wedge \neg result! \wedge \neg result?$.

Lines 12 - 15 update the self-loop for the current state to ensure that the automaton remains in the current state if no relevant messages are observed. For example, in state q_1 of Fig. 2(b), the self-loop $\neg idle \wedge \neg jobID \wedge \neg getData \wedge \neg data \wedge \neg result? \wedge \neg result!$ is enabled if no message is observed. As mentioned earlier, the first state of the automaton has a self-loop with the *true* label

Algorithm: $TRANSFORM(q)$

1: **for** $\forall\delta : (q,\phi,q_{\text{exit}}) \in \Delta$ **do**
2: $\quad \Delta \leftarrow \Delta \setminus \{\delta\}$
3: $\psi_m \leftarrow \emptyset$, $\psi_i \leftarrow \emptyset$, $\psi_c \leftarrow \emptyset$
4: **for** $\forall\phi,q' : (q,\phi,q') \in \Delta$ **do**
5: $\quad \psi_m \leftarrow \psi_m \cup f(\Sigma_{\text{msgs}},\phi)$
6: $\quad \psi_i \leftarrow \psi_i \cup f(\Sigma_{\text{invariant}},\phi)$
7: $\quad \psi_c \leftarrow \psi_c \cup f(\Sigma_{\text{conditions}},\phi)$
8: $\quad \Delta \leftarrow \Delta \setminus \{(q,\phi,q')\}$
9: $\quad \phi_{\text{child}} \leftarrow \phi \wedge \neg disjunct(\Sigma_{\text{msgs}} \setminus f(\Sigma_{\text{msgs}},\phi))$
10: $\quad \Delta \leftarrow \Delta \cup \{(q,\phi_{\text{child}},q')\}$
11: $\quad TRANSFORM(q')$
12: **for** $\phi : (q,\phi,q) \in \Delta$ **do**
13: $\quad \Delta \leftarrow \Delta \setminus \{(q,\phi,q)\}$
14: $\phi_{\text{self}} \leftarrow \neg disjunct(\Sigma_{\text{msgs}}) \wedge \neg disjunct(\psi_c) \wedge conjunct(\psi_i)$
15: $\Delta \leftarrow \Delta \cup \{(q,\phi_{\text{self}},q)\}$
16: **if** $\exists\phi,q' : (q,\phi,q') \in \Delta$ and $f(\Sigma_{\text{main}},\phi) \neq \emptyset$ **then**
17: $\quad \phi_{\text{safety}} \leftarrow disjunct(\Sigma_{\text{msgs}} \setminus \psi_m) \vee \neg conjunct(\psi_i)$
18: $\quad \Delta \leftarrow \Delta \cup \{(q,\phi_{\text{safety}},q_{\text{safety}})\}$
19: $\quad$ **if** $(q,\phi,q') \in \Delta_{\text{hot}}$ **then**
20: $\quad\quad F \leftarrow F \cup q$
21: return(A)

Figure 4: Algorithm for building a negated automaton from an input LSC automaton.

and the final state of the automaton has no self-loops. These special cases are not shown in the transformation algorithm in Fig. 4.

If the current state q contains a main chart transition (labels of transitions to successor states are members of the main chart alphabet Σ_{main}), then lines 16 - 18 of the algorithm add a safety transition to the safety state q_{safety}. The safety transition enables the automaton to detect message order violations or duplicate messages. For the automaton shown in Fig. 2(a), state q_1 contains a single transition for the $jobID$ message. Since $jobID$ is a member of the main chart alphabet ($jobID \in \Sigma_{main}$) a safety transition needs to be added. The safety transition for state q_1, $idle \vee getData \vee data \vee result? \vee result!$, detects the presence of any message except the one allowed message $jobID$. Because states with no main chart transitions can not violate the chart, no safety transitions are added to them.

Lines 19 - 20 of the algorithm label the current state as an accept state if it belongs to the main chart and contains a hot outgoing transition. The check for main chart transitions is performed on line 16. To check for hot outgoing transitions, each outgoing transition is checked for membership in the Δ_{hot} set (line 19). If all outgoing transitions from a state are cold, the state is not marked as an accept state. In our example, for state q_2, the only outgoing transition corresponds to a cold condition and is not part of the Δ_{hot} set; thus, state q_2 is not marked as an accept state. On the other hand state q_1 is marked as an accept state because it has one successor transition that corresponds to the hot message $jobID$.

We now state the theoretical results of the presented transformation. We first show that for any main chart state in the automaton at least one transition is enabled for any arbitrary input (i.e. the transition relation for main chart states is total). Having enabled transitions guarantees that

the automaton does not ignore any inputs which could cause violations or progress in the chart. To conserve space, all proofs have been omitted from this version of the paper but are available in the long version of the paper.

Lemma 1 *For all states containing outgoing main chart transitions, the transition relation is total. Formally, given a state q with a main chart transition* $\left(\bigvee_{\forall \phi_i, q_i:(q,\phi_i,q_i)\in\Delta} \phi_i\right) = true$.

Lemma 1 is only applicable to states containing main chart transitions. Regarding states that do not contain main chart transitions, the safety transition ϕ_{safety} is not added, resulting in an incomplete transition relation. Since these states are responsible for detecting the completion of the pre-chart and not for detecting violations or errors, the incompleteness of the transition relation does not affect the correctness of observing the pre-chart. Our next result states that for all states except the first state of the automaton, the transition relation is deterministic. The transformed automaton is non-deterministic only in the first state (self-loop annotated with $true$) to accommodate for the global verification of every possible instance of the chart in the system. Non-deterministic automata as used in [KTWW06] result in error traces that have to be validated using full LTL verification, which has been shown to be impractical for LSCs [KM07]. Using deterministic automata guarantees that any reported errors are in fact valid errors in the system.

Lemma 2 *For states q in the transformed automaton (except the initial state), the transition relation is deterministic. Formally,* $\forall q \in Q, \forall \phi_i, \phi_j : (q, \phi_i, q_i) \in \Delta \wedge (q, \phi_j, q_j) \in \Delta, (\phi_i \wedge \phi_j) = false$.

The above result guarantees that for any given input to the transformed automaton (except the first and last state) exactly one transition is ever enabled. We now state our primary result for the transformed automaton. Intuitively, we show by application of Lemma 1 and Lemma 2 that the transformed automaton accepts only those words that are not accepted by the LSC and is capable of detecting all behaviors in a system that violate the LSC. We assume that the automaton created detects all pre-chart instances correctly.

Theorem 1 *The automaton, A, generated by the transformation algorithm in Fig. 4 for a given LSC, SPEC, defined over an alphabet $\Sigma_{SPEC} \subseteq \Sigma$, reads exactly the complement of the language of the SPEC. Formally,* $\forall \theta = \theta_0 \theta_1 \theta_2 \ldots$

$$[\theta \in L(SPEC) \implies \theta \notin L(A)] \wedge [\theta \notin L(SPEC) \implies \theta \in L(A)].$$

where $L(A)$ and $L(SPEC)$ are the languages of the transformed automaton and the SPEC.

4.1 Verification Approach

For explicit state model checking, verification of a system against the specification is performed in the usual manner. The composition of the system and transformed LSC automata is computed on-the-fly and checked for accepting cycles using the Double Depth First Search (DDFS) algorithm. If the DDFS algorithm does not discover any accepting cycles, the system implements the safety and liveness behaviors as described in the chart. For symbolic model checking, we first

label accept states as fair states in the composition of the system and transformed LSC automata. This automaton is then verified against the ACTL property **EG**($true$), which searches for fair Strongly Connected Components (SCCs) reachable from the initial state. Any reported SCCs are violations of the specification.

5 Analysis

The verification approach presented in [KTWW06] utilizes at least two and in the worst case three algorithms to completely verify a system against an LSC. If reachability analysis followed by ACTL verification fails to produce a significant result (proof of correctness or a violation) the system is verified against an LTL formula generated from the LSC specification [TW06]. Compared to the verification approach of [KTWW06], the new verification approach presented in this paper only performs one verification run of comparable complexity as the reachability analysis and ACTL verification in the approach of [KTWW06]. In the average case the total verification cost is reduced by a factor of two and in the best case (worst case in old approach) by a factor of three or more.

One side effect of using the negative automaton is the inability to verify multiple instances of a chart with cold construct violations. For example, if in our example system the $Node$ receives $jobID$ but is unable to validate $jobID$, the cold condition $validID$ is never observed and the chart automaton will remain in state q_2. This is not an error since state q_2 is a non-accepting state waiting to observe the cold condition $validID$. If $Node$ restarts the job acquisition by sending the $idle$ message to the $Scheduler$, the safety transition from state q_2 to q_{safety} is taken. Consequently, a false error will be reported (duplicate message). Generally speaking, if in one instance of the chart a cold construct is never observed, no future instances of the chart can be observed in a given trace. This drawback can be limiting for highly reactive and iterative systems with multiple instances in a single trace. A solution is being investigated as future work.

6 Results

We briefly discuss our experiments and results in this section. For a detailed presentation we refer the reader to the long version of the paper. We create models with multiple communicating processes and test them against highly concurrent worst case specifications as described in [KTWW06]. All specifications are named A$c \times m$ where c and m are the number of co-regions and messages in each co-region respectively.

We first test the scalability of our approach in the symbolic model checking domain and compare it to the results presented in [KTWW06]. Table 1 shows a subset of the results for verifying the *abp* model using the NuSMV model checker. In general, our verification approach performs twice as fast as the approach presented in [KTWW06] and we scale to specification sizes that were unobtainable using the verification approach in [KTWW06]. We also test the scalability of our approach in explicit state model checking using the SPIN model checker. Table 2 shows a subset of the results for verifying the *plain* and *soko* models. Our approach performs better and scales to larger specifications when compared to the approach of [KM07].

Table 1: Results for the traditional and improved verification approaches using NuSMV.

Specification	Traditional Verification						Improved Verification	
	Reachability		ACTL		Total			
	States	Time	States	Time	States	Time	States	Time
A3x5	1.01616e+06	34	1.47142e+07	35	15730360	69	1.41696e+07	34
A3x6	1.01616e+06	237	1.01616e+06	239	2032320	477	471552	251
A3x7	879408	1568	879408	1562	1758816	3130	521504	1550

Table 2: Results for the improved verification approach using SPIN.

Specification	Model	Without Errors			With Errors		
		States	Memory	Time	States	Memory	Time
A7x6	soko	97500	17.216	125	89323	16.397	125
	plain	406	7.385	123	406	7.385	124
A8x6	soko	97500	18.491	214	89323	17.672	210
	plain	406	8.661	216	406	8.661	215
A9x6	soko	97500	20.104	325	89323	19.285	344
	plain	406	10.274	335	406	10.274	334

7 Conclusions and Future Work

The presented LSC to automaton transformation algorithm allows us to verify a system against an LSC using only language containment with readily available tools. Compared to past approaches, this approach only requires one verification run of comparable complexity as opposed to three verification runs for any arbitrary LSC. Further, we prove that the generated automaton can detect all safety and liveness violations in a system and empirically show the effectiveness of the approach. For future work we are investigating the use of LSCs for automated environment generation to test individual interfaces in a system. We are also investigating the possibility of extending the transformation algorithm to constructs such as overlapping chart instances, Kleene star and multiple instance detection with the presence of cold constructs (as discussed earlier).

Bibliography

[AY99] R. Alur, M. Yannakakis. Model Checking of Message Sequence Charts. In *CONCUR99: Proc. of the 10th Int. Con. on Concurrency Theory*. Pp. 114–129. Springer-Verlag, London, UK, 1999.

[BDK+04] M. Brill, W. Damm, J. Klose, B. Westphal, H. Wittke. Live sequence charts: An introduction to lines, arrows, and strange boxes in the context of formal verification. In *SoftSpez Final Report*. Lecture Notes in Computer Science 3147, pp. 374–399. Springer, 2004.

[BH02] Y. Bontemps, P. Heymans. Turning high-level live sequence charts into automata. In *Proc. of "Scenarios and State-Machines: Models, Algorithms, and Tools" (SCESM02) Workshop of the 24th Int. Conf. on Software Engineering (ICSE 2002)*. Orlando, FL, May 2002.

[BHK03] Y. Bontemps, P. Heymans, H. Kugler. Applying LSCs to the specification of an air traffic control system. *Proc. of the 2nd Int. Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM03), at the 25th Int. Conf. on Software Engineering (ICSE03), Portland, OR, USA*, 2003.

[DH99] W. Damm, D. Harel. LSCs: Breathing life into message sequence charts. In *Proc. of the 3rd Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS99)*. P. 451. Kluwer, B.V., Deventer, The Netherlands, 1999.

[DK01] W. Damm, J. Klose. Verification of a radio-based signaling system using the STATEMATE verification environment. *Formal Methods in System Design* 19(2):121–141, 2001.

[HK01] D. Harel, H. Kugler. Synthesizing state-sased object systems from LSC specifications. In *CIAA00: Revised Papers from the 5th Int. Conf. on Implementation and Application of Automata*. Pp. 1–33. Springer-Verlag, London, UK, 2001.

[IT93] R. ITU-T. 120: Message sequence chart (MSC). *Telecommunication Standardization Sector of ITU, Geneva*, 1993.

[KHG05] C. Knieke, M. Huhn, U. Goltz. Modeling and simulation of an automotive system using LSCs. *Proc. of the 4th Int. Workshop on Critical Systems Development Using Modelling Languages (CSDUML05)*, 2005.

[Klo03] J. Klose. *Live sequence charts: A graphical formalism for the specification of communication behavior*. PhD thesis, Fachbereich Informatik, Carl Von Ossietzky University, 2003.

[KM07] R. Kumar, E. Mercer. Improving translation of live sequence charts to temporal logic. In *Proc. of the 7th Int. Conf. on Automated Verification of Critical Systems (AVoCS07)*. Pp. 183 – 197. 2007.

[KTWW06] J. Klose, T. Toben, B. Westphal, H. Wittke. Check it out: On the efficient formal verification of live sequence charts. *Proc. of the 18th Int. Conf. on Computer Aided Verification (CAV06), LNCS* 4144:219–233, 2006.

[KW01] J. Klose, H. Wittke. An automata based interpretation of live sequence charts. In *TACAS 2001: Proc. of the 7th Int. Conf. on Tools and Algorithms for the Construction and Analysis of Systems*. Pp. 512–527. Springer-Verlag, London, UK, 2001.

[SD05] J. Sun, J. S. Dong. Model checking live sequence charts. In *ICECCS05: Proc. of the 10th IEEE Int. Conf. on Engineering of Complex Computer Systems (ICECCS05)*. Pp. 529–538. IEEE Computer Society, Washington, DC, USA, 2005.

[TW06] T. Toben, B. Westphal. On the expressive power of LSCs. In *SOFSEM 2006: 32nd Conf. on Current Trends in Theory and Practice of Computer Science, Měřín, Czech Republic, January 2006*. Volume 2, pp. 33–43. Institute of Computer Science AS CR, Prague, 2006.

Composing control flow and formula rules for computing on grids[2]

P. Bottoni[1] **and N. Mirenkov**[2] **and Y. Watanobe**[2] **and R. Yoshioka**[2]

[1] bottoni@di.uniroma1.it Dep. of Computer Science, "Sapienza" Univ. of Rome, Italy

[2](nikmir,yutaka,rentaro)@u-aizu.ac.jp Dep. of Computer Software, Univ. of Aizu, Japan

Abstract: We define computation on grids as the composition, through pushout constructions, of control flows, carried across adjacency relations between grid cells, with formulas updating the value of some attribute. The approach is illustrated in the context of the Cyberfilm visual language.

Keywords: Grids, Control flow rules, DPO

1 Introduction

Graphs have been long proposed as a universal formalism for describing the structure of system configurations and to support computational specifications of the transformations they may undergo. Moving from this common ground, the areas of graph transformations and graph algorithms have taken two divergent, possibly complementary paths.

On the one hand, graph transformations propose a declarative approach to computation based on the iteration of local modifications to the graph structure, so as to define a language of admissible graph configurations, each depicting a possible state of the system being modelled. On the other hand, algorithms on graphs exploit procedural definitions of visits to the graph structure, usually to extract some global property of it. In many cases, graph transformations – typically performed by enriching the graph with additional features such as types [CMR96], attributes [MW93, HKT02, dBE+07] or control structures on rule application [KK99, SWZ99, BKPT00, AKN+06] – are capable of replicating many relevant features of the algorithmic approach.

However, the general approach to graph transformations – based on the search for a subgraph isomorphism between the antecedent of a rule and the host graph under scrutiny – is not optimal for spatially organized structures, such as grids (in any number of dimensions), trees, or pyramids [YM02, WMYM08], as the inherent non-determinism of the matching process fails to take advantage of the existence of privileged relations among elements, and of orders for their visit.

We propose to reconcile the use of graph transformation as a general computational framework with the existence of some spatial structure on the host graph. To this end, we combine a suite of meta-models for diagrammatic languages – defining the possible spatial relations among identifiable elements [BG04] and their transformation semantics [BLG07] – with a form of algebraic composition of rules in the framework of the Double Pushout Approach to graph rewriting.

The proposal is applied to the *Cyberfilm* visual environment, which provides the user with iconic representations of computational flows on spatial structures, arranged as sequences of frames highlighting the set of nodes which at each step contribute to the production of a new

[2] Work done while P. Bottoni was at Aizu University as adjunct professor, on temporary leave from Sapienza.

result [WMYM08]. In a separate view, the formulas defining the computations can be defined, thus allowing their reuse according to different control flows. In particular, we focus on bidimensional grids on which several control flows can be defined, and use a categorical construction to provide a formal treatment of the composition of control flow and computational formulas.

In the rest of the paper, after related work in Section 2, we provide background on graph transformations and the adopted metamodels in Section 3. Section 4 introduces the categories on grids needed to define control flow rules in Section 5. Finally, Section 6 shows how to compose formulas and control flows, before drawing conclusions in Section 7.

2 Related Work

Spatial structures, such as those defined by grids or trees, have been the subject of many studies from the algorithmic point of view, in particular as regards the identification of paths with particular properties over them [IPS82]. From the algebraic point of view, trees have been studied as representations of computational structures, such as terms [HP95] or abstract syntaxes [Mos94], whereas images, rather than grids, have been studied in relation to the sets of languages definable on them [GR97]. The translation morphism discussed in this paper may be seen as an analogous of the ”positional overlapping” operation for images [BL07].

The technique for composing control flows and formulas, although equally based on pushouts, differs from that for (local) application of rules to rules in [Par94], based on finding a match from a rule component to a component of another rule, as well as from that for *action pattern*, in [BLG07], where a pattern is matched to the right-hand side of a rule to produce a rule whose effects conform to the pattern. A construction analogous to the one here, exploiting common subrules, is in [TB94], but we use it to statically construct rules, and not to identify possible agreements on a host graph.

Finally, we point out a similarity with notions of modularity and Viewpoints [GEMT00], proposed as a way to modeling a system through the integration of partial models. However, we combine different aspects of the behaviour of the system into an integrated specification, rather than considering behavioral and structural aspects together. The approach to coordination proposed in [AFGK02] is also based on pushouts (actually colimits), to allow separation of concerns when defining different aspects of a program behaviour.

3 Background: Metamodels and Graph Transformations

According to the metamodel for diagrammatic languages presented in [BG04], a *diagram* is composed of *identifiable elements* among which significant *spatial relations* exist. A whole diagram is itself an identifiable element, with global properties. An identifiable element is a recognizable unit in the language, associated with a graphical representation defined by a *complex graphic element*, composed in turn of one or more *graphic elements*, each possessing some *attach zone*, representing the geometrical support for spatial relations. The existence of a relation is assessed via a predicate `isAttached()` implemented by each zone. Symmetries may exist between spatial relations. Two relations σ and ρ are tied by a symmetry if there is a size-preserving diagram transformation changing all instances of ρ into instances of σ.

Specializations of these abstract types define language families. For example, `Adjacency` indicates a class of relations between `Cells` of regular shape tessellating the plan, and whose *borders* overlap for a finite segment. According to the type of tessellation, cells may entertain various adjacency relations, typically possessing symmetric companions, such as in `Left` and `Right` adjacency for regular arrangements of rectangles.

Based on this metamodel, we can represent diagrams as attributed typed graphs, where nodes are elements of classes in the metamodel and edges are instances of the associations.

Formally, a *type graph* is a construct $TG = (N_T, E_T, s^T, t^T)$ with N_T and E_T sets of node and edge types. $s^T : E_T \to N_T$ and $t^T : E_T \to N_T$ define the *source* and *target* node types for each edge type. A typed graph on TG is a graph $G = (N, E, s, t)$ with a graph morphism $type\colon G \to TG$ composed of $type_N\colon N \to N_T$ and $type_E\colon E \to E_T$, s.t. $type_N(s(e)) = s^T(type_E(e))$ and $type_N(t(e)) = t^T(type_E(e))$. Type graphs with node inheritance exploit a pair $TGI = (TG, I)$, where $I = (N_I, E_I, s^I, t^I)$ is a node inheritance graph, with $N_I = N_T$, i.e. I has the same nodes as TG, but its edges are the inheritance relations. The inheritance *clan* of a node n is the set of all its children nodes (including n itself): $clan(n) = \{n' \in N_I | \exists \text{ path } n' \to^* n \text{ in } I\} \subseteq N_I$.

Typed attributed graphs are typed graphs with additional *data* nodes and *attribute edges* (nodes of G are now called *object nodes*). A type graph TG has a set Δ of *data type nodes* and a set A of *attribute type edges*, denoting the domains of the attribute nodes and the set of attributes associated with nodes, together with functions $\sigma_A : N_T \to \mathscr{P}(A)$, defining the attributes for a given type, and $\tau_A\colon A \to \Delta$, defining the admissible domain for each attribute. These elements define a *type graph with attributes* TG_A. A typed attributed graph on TG_A is a construct $(TG, G, N_\Delta, E_A, s_A, t_A)$, where TG and G are as before, N_Δ is the set of data nodes, coinciding with the disjoint union of the domains of attributes, and E_A is the set of *attribute edges*, from object nodes to data nodes. Edges are typed on A and associate object nodes with the values of its attributes. $s_A\colon E_A \to N$ and $t_A\colon E_A \to N_\Delta$ define the valuation of attributes for a given node, coherently with σ_A and τ_A. We represent data nodes as typed items and distinguish them from object nodes through a dashed contour, following the convention in Example 8.5 of [EEPT06].

Attributed typed graphs form the adhesive HLR category [LS04] $\mathbf{AGraph_{ATG}}$, so that transformations can be expressed through Double Pushout (DPO) derivations [EEPT06], in which rules are spans $L \leftarrow K \to R$ and K defines the part which is left unchanged by the rule application. We also exploit application conditions, as shown in Section 4.

4 Categories on Grids

Rectangular grids are regular arrangements of cells according to symmetrical pairs of *vertical* and *horizontal* adjacency relations, with *nrow* rows and *ncol* columns, conforming to the family of diagrammatic languages depicted in Figure 1. Hence, nodes are instances of `Cell`, with a position given by their *row* and *col* attributes and boolean values to distinguish border cells. Adjacency relations can be of four types, with the obvious constraints on their pairing. Moreover, there exists a set of additional constraints stating that a grid has to form a rectangle (i.e. its top and bottom borders must have the same number of elements, as must its left and right borders), and all border elements are adjacent to three other cells except the four corner elements, adjacent to two. Mirror and rotation symmetries exists between pairs of adjacency relations.

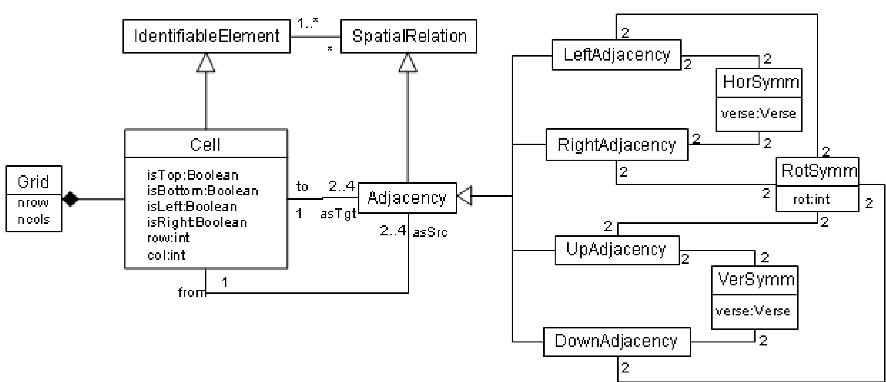


Figure 1: The family of bidimensional grids.

Grids thus give rise to a subcategory $\mathbf{AGrid_T}$ of $\mathbf{AGraph_{ATG}}$, whose morphisms are composed of a *structural* part, involving nodes of type `Cell` and `Adjacency`, and a *data* part involving attributes. The structural part uses *translations* as morphisms. A translation exists from a grid G_1 to a grid G_2 if G_2 is such that an isomorphism exists from G_1 to a subset of its cells, preserving its connectivity and the relative directions. A translation is uniquely determined by the position of the upper left corner (or any other cell) of the original grid in the context of the target grid. Figure 2 shows the composition of two translations, where the highlighted rectangles show the new positions of the original grid. We assume translations occur only rightwards and downwards. The pairs (r,c) labeling the morphisms indicate the offsets at which the nodes of the original grid are found in the new grid. The size of G_2 is at least equal to that of G_1, but can also be larger. The identity morphism is the translation $(0,0)$ from a grid into itself, and morphism composition is the vectorial sum of the translations. We now study the subcategory $\mathbf{TGrid_T}$, obtained by taking the structural part of $\mathbf{AGrid_T}$, i.e. maintaining the type information, but forgetting attributes.

In $\mathbf{TGrid_T}$, a pushout $G_1 \xrightarrow{p_1} P \xleftarrow{p_2} G_2$ for a span $G_1 \xleftarrow{t_1} G \xrightarrow{t_2} G_2$ between two grids can be constructed in the same way as the pushout in **Graph** if and only if one of the following is true:

(1) either t_1 or t_2 is an identity;
(2a) t_1 has a label of the form $(r_1,0)$ and $G_1.ncol == G.ncol$ AND
(2b) t_2 has a label of the form $(0,c_1)$ and $G_2.nrow == G.nrow$;
(3a) t_1 has a label of the form $(0,c_2)$ and $G_1.nrow == G.nrow$ AND
(3b) t_2 has a label of the form $(r_2,0)$ and $G_2.ncol == G.ncol$.

Then, P has size $(max(G1.nrow, G2.nrow), max(G1.ncol, G2.ncol))$; morphisms $p_1\colon G_1 \to P$ and $p_2\colon G_2 \to P$ are labeled by $(max(r_1,r_2)-r_1,\ max(c_1,c_2)-c_1)$, and $(max(r_1,r_2)-r_2,\ max(c_1,c_2)-c_2)$, respectively, so that parallel arrows have the same label (see Figure 3). The pushout complement $G \xrightarrow{x_1} C \xrightarrow{x_2} G'$, for the composition $G \xrightarrow{t_1} G_1 \xrightarrow{t_2} G'$, where G is an object of size (r,c), G_1 of size (r_1,c_1), and G' of size (r',c'), uniquely exists only if t_1 and t_2 satisfy the constraints above, and has size $((r'-r_1)+r, (c'-c_1)+c)$, with x_1 and x_2 labeled as t_1 and t_2.

We can now define *structural* DPO rules in $\mathbf{TGrid_T}$ in accordance with the construction above. In particular, in order to satisfy the dangling condition, rules are non-deleting (i.e. $L \leftarrow K$ is an

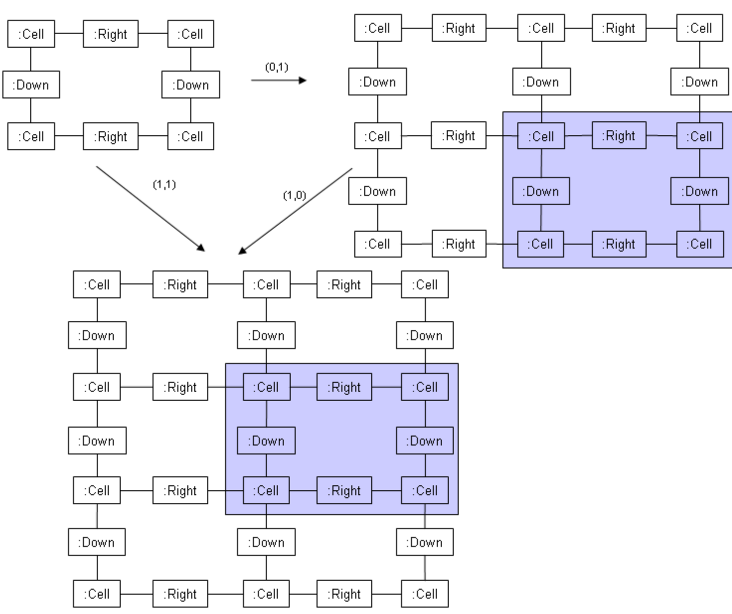


Figure 2: Translation morphism and composition.

identity). The gluing condition, if $K \to R$ is not an identity, requires border cells of L to be matched to border cells of the host grid G. The pushout complement object D is now always equal to G. Hence, grids can be generated so that the constraints on their rectangular form are maintained by the pushout construction, without having to adopt regulatory mechanisms for rewriting, such as those needed for the so-called Indian grammars: they first use a set of *horizontal* rules to create the upper row, and then apply *vertical* rules in parallel to populate the columns [SK74]. The pushout construction can now be lifted to considering also attributes. In particular, as typical of attributed graph rewriting, data morphisms are identities (no domain element can be created or deleted). Hence, the effect of the rule can only be the addition of structural nodes and edges and the deletion and creation of attribute edges. Note that, *instantiation* morphisms are used to match nodes containing variables to data nodes. These preserve the domains and equality on variable names, while need not be injective. Relations between variables are expressed via application conditions. All grids in $\mathbf{AGrid_T}$ can now be generated by the iterated use of the two rules in Figure 4 and Figure 5, in which the identifiers of the `Adjacency` nodes indicate their directions and an application condition defines the coordinates of the new cell.

In both cases, we show the classical representation of DPO rules at the top of the figure, and use, at its bottom, a compact notation, already exploited in [dGB07]: the difference between K, L, and R is shown by highlighting the deleted and produced parts with different colours and marking them with tags {`del`} and {`new`}. The elements outside the tagged areas are those belonging to the K component. Differently from [EEPT06], we explicitly show K as presenting isolated data nodes, which are connected to different object nodes in L and R.

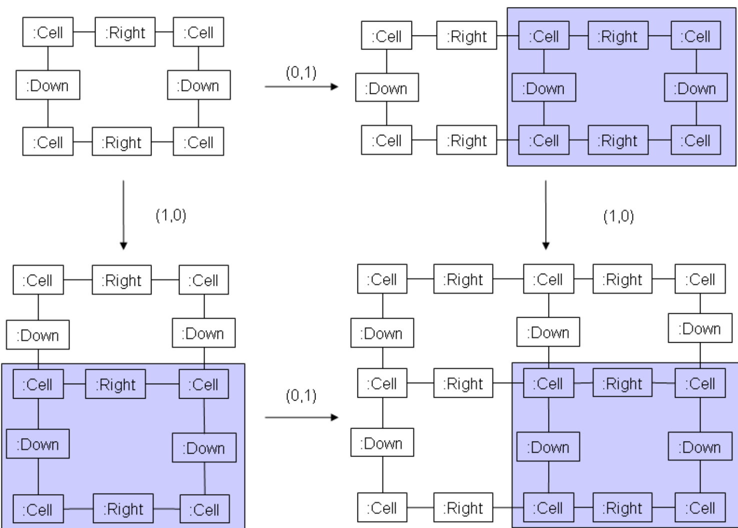


Figure 3: The pushout construction for grids

5 Control Flow Rules

Figure 6 presents the triple metamodel describing the correspondences induced by assigning the semantic meaning of a *carrier* to the adjacency relation in the visual representation. Along a carrier either data or modifications in the *activation state* travel from and to *active elements*. The left upper part of Figure 6 constitutes the static semantics for the control flow variety on spatial structures, while the right upper part models the data variety, here simplified by considering a simple integer-valued attribute, called `level`. By applying the construction in [dGB07] one can incrementally define the flow structure through triple graph rules which introduce carriers in correspondence with the installation of adjacency relations in specified directions. Hence, one can model the *permeability* of the cell wall to control or data flow. For example, flows can travel rightwards and leftwards, but not downwards and upwards.

A control flow (*cf*) rule is a DPO rule in $\mathbf{AGraph_{ATG}}$ with graphs conforming to the left upper part of Figure 6, so that A contains the attribute $state$ with values in some finite domain $ActivationState \in \Delta$.

In general, as shown in the rule (in compact form) on the left of Figure 7[3], the activation state may vary during transportation, e.g. a flow can decrease its intensity. The element reached by the flow could have possessed some other activation value and the one from which the flow originated may gain a new one, as defined by application conditions. The basic rule on the right of Figure 7 deals with the case of cells entering an *active* state as the control flow reaches them traveling the grid rightwards from the origin to the destination, while the origin enters a *quiescent* state. The carrier identifier indicates the value of its `Direction` attribute. Similar rules are defined for other directions, exploiting rotational and mirror symmetries. *cf*-rules are composed

[3] Abbreviations are used for names of values and types.

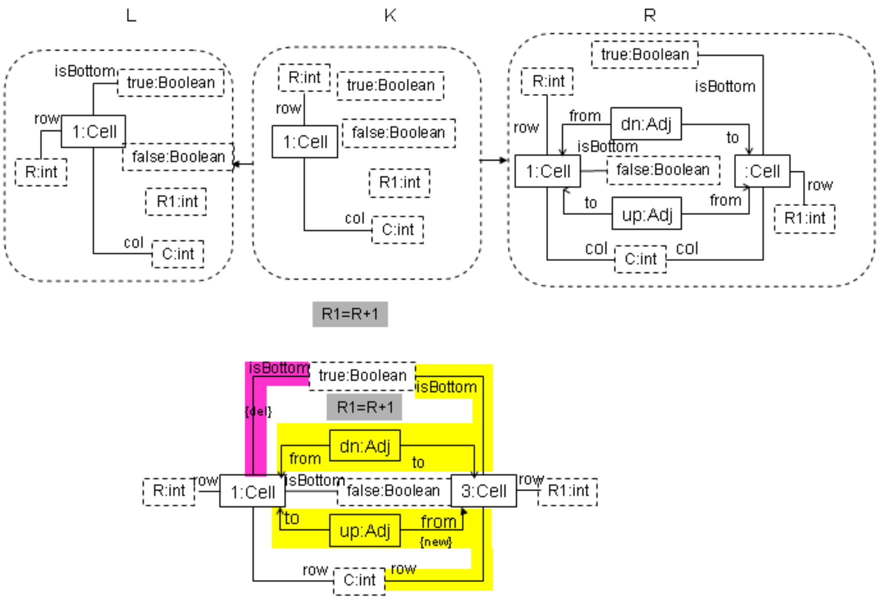


Figure 4: The rule for letting a grid grow horizontally.

to form more complex ones using a componentwise pushout construction in $\mathbf{AGraph_{ATG}}$, as shown in Figure 8, where $L \leftarrow K \rightarrow R$ is the maximal intersection of $L_1 \leftarrow K_1 \rightarrow R_1$ and $L_2 \leftarrow K_2 \rightarrow R_2$, all the squares commute and those with curved arrows are pushouts. As an example, directional rules compose through a rule on a single cell passing from the *active* to the *quiescent* state. Figure 9 illustrates the case of the two horizontal movements, while Figure 10 that of one directional and one vertical movement.

6 Composing Control Flow and Computation Formulas

We now introduce *data*-rules to specify the transformation of some attribute according to some formula. These are defined on the type graph in the right upper part of Figure 6. Data and *cf*-rules are composed, again with a pushout construction, to produce rules which both apply the formula and propagate the flow, when an active element is reached by the control flow.

The rule in Figure 11 doubles the value of `level`. X and Y are variables to indicate generic instances of an integer. Rule composition requires the use of four different types of graphs. The intersection is defined in a type graph where the type `ActiveElement` abstracts on both `ControlActiveElement` and `DataActiveElement` and has no attribute, while the pushout object complies with a type graph formed by taking the quotient of the disjoint union of the two type systems from Figure 6 and identifying the activity types in a `FullActiveElement` type (abbreviated in *FActv*). Node morphisms go from less to more specific types.

In Figure 12, the bidirectional rule of Figure 10 is composed with the formula of Figure 11, so that the latter is now evaluated only when the activation front leaves an element in both directions.

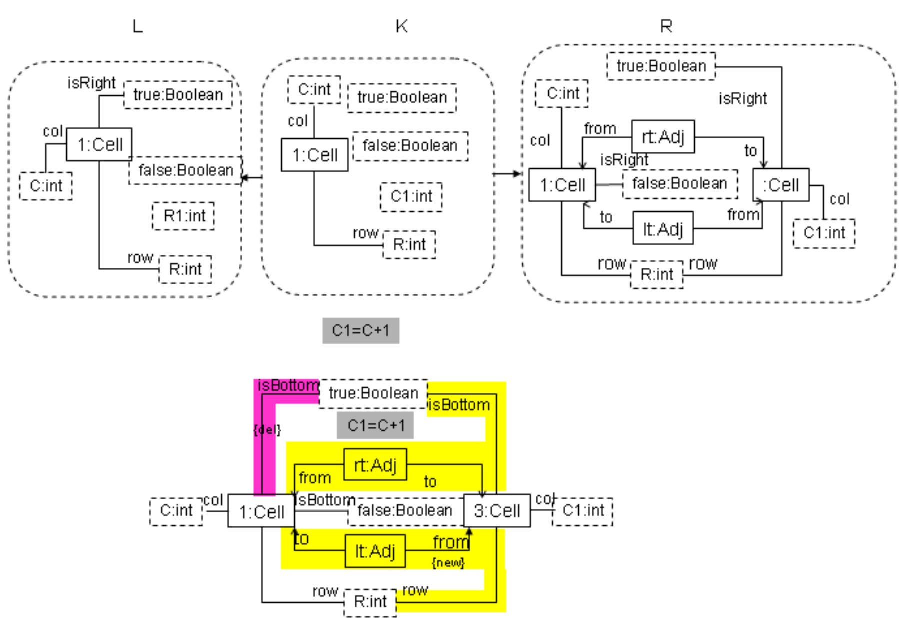


Figure 5: The rule for letting a grid grow vertically.

Using a different morphism from the intersection to the *cf*-rule, the formula would be evaluated when the control flow reaches an element from a specific direction. The resulting rule does not specify the `level` values for the other elements. Rules can be applied sequentially or in parallel, if they do not conflict on their result. As an example, the rule of Figure 12 agrees with itself on any node whose upper and left neighbours are both mapped, by two distinct matches, to the cell identified by 1. Rules can be enriched with parameters and applied via rule expressions to realize complex computations [BKPT00].

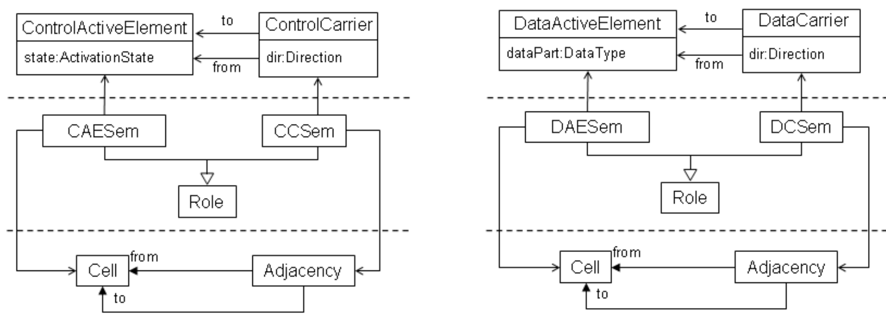


Figure 6: The triple metamodel for control and data flows on grids.

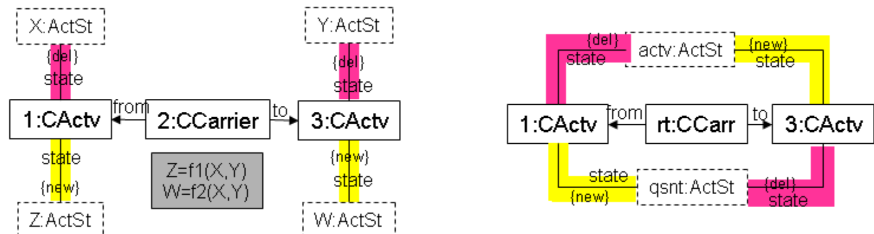


Figure 7: A generic rule for transmission of control flows and a basic rule.

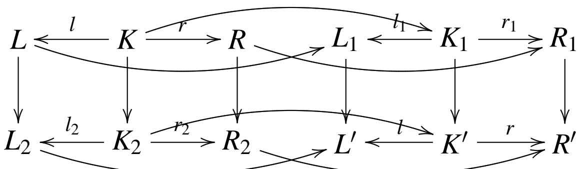


Figure 8: The construction for rule composition.

6.1 Types of activation in *Cyberfilm*

The *Cyberfilm* language [YM02, WMYM08] provides a collection of predefined control flows, associated with program templates defining the loops realizing them, and with sequences of iconic schemes for an intuitive visualization of the main steps in the execution flow. *Cyberfilm* allows the separated definition of computational formulae and control flow specifications. Hence, the constructions above can be exploited to provide a compositional mechanism for it.

In particular, control flow in *Cyberfilm* is defined by the presence of two types of activation state, called *full* and *contour* flashing, according to the visualizations suggested for them[4].

An element in the full flashing state computes the associated formula, while in the contour flashing state it can contribute to some computational activity, i.e. its content is read by a cell, but it cannot change. At any time, a cell is in only one possible state. The control flows of the *full* and *contour* flashing can be independent or coordinated. Independent flows can be specified as described in Section 5, whereas coordinated flows require the identification of the conditions under which a cell is able to receive the contributions of other cells.

Figure 13 shows the composition of a coordinated *cf*-rule, for rightward transmission of both *full* and *contour* flows, with a formula rule where an element reads the value of its down neighbour (without changing it), to compute its new level. The upward adjacency relation is mapped, in the formula rule, to a *data carrier* (DC) element, as control and data may flow in different directions, i.e. cells can have different *permeability* to data and control flows.

In this case, the *cf*-rule is bound to consider both the *full* and *contour* flows simultaneously. The same effect could be achieved by considering the two flows independently. Figure 14 shows the first step of the relative construction, in which the rightwards movement for the full flash-

[4] Actually, there are several types of flashing for different classes of activity. For example, *half-flashing* indicates decision nodes. These other types are not considered here.

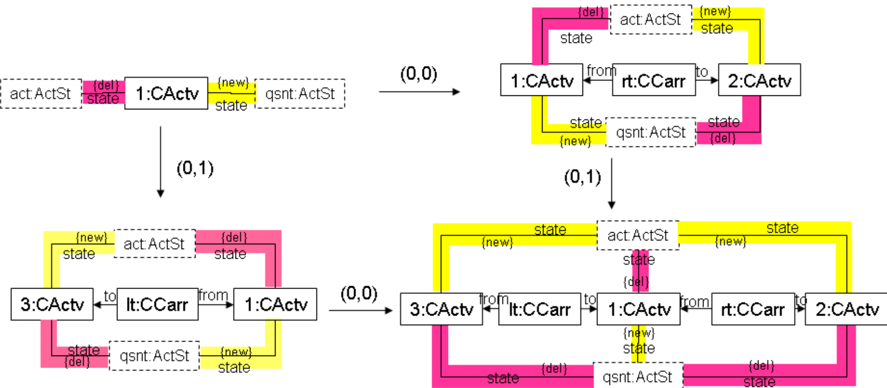


Figure 9: The construction of the rule propagating control flow horizontally in both verses.

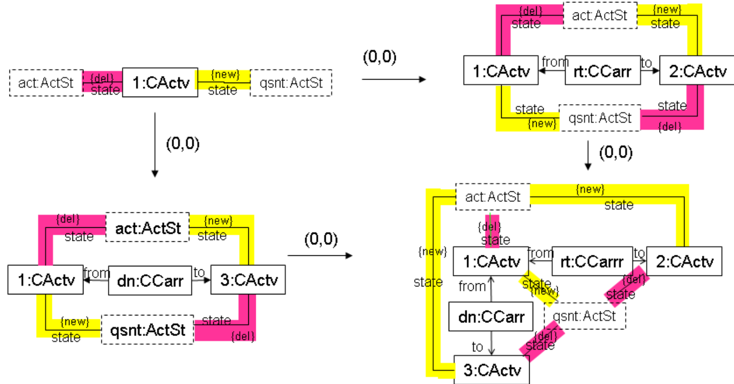


Figure 10: The construction of a bidirectional rule.

ing is combined with the same formula to produce a rule which does not affect the state of the `ControlActiveElement` identified by 2. In Figure 15, the obtained rule is composed with that for rightward movement of contour flashing. While the final effect is the same, the intermediate step could be combined with other movement rules. Several such rules might be defined, for example to propagate the flow across several cells, so that only some elements are activated.

7 Conclusions

We have proposed an approach, based on componentwise pushout of DPO rules in the category of attributed typed graphs, to the specification of computations on grids. The approach allows the independent definition of two types of rules, one to specify control flow and the other to specify the actual computations. The construction is symmetrical in control and formula rules, so that it can be flexibly applied starting from either specification. Symmetries between adjacency

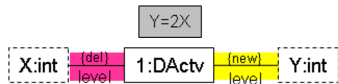


Figure 11: A rule expressing a computational formula.

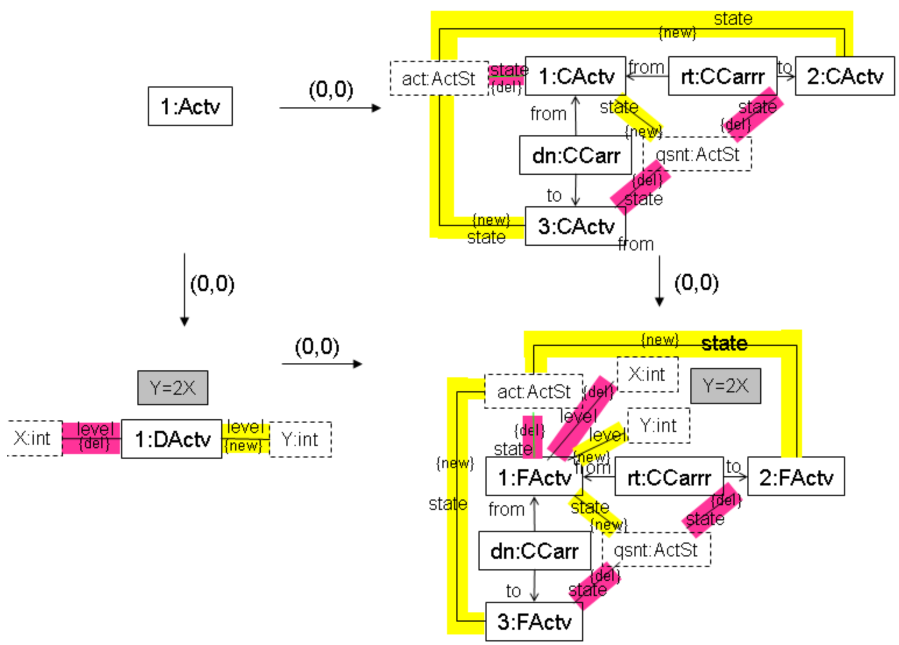


Figure 12: The resulting rule specifying the condition of application.

relations can also be exploited to generate different versions of flows and formulas.

Future work will explore other types of spatial structures, typically trees and pyramids, to define adequate *cf*-rules, also considering the distinction between formula evaluation on control flows reaching or leaving the involved cells, and develop ways of reasoning about the compatibility of independent *cf*-rules (e.g. one for reading and one for writing).

Bibliography

[AFGK02] L. F. Andrade, J. L. Fiadeiro, J. Gouveia, G. Koutsoukos. Separating computation, coordination and configuration. *J. of Software Maintenance* 14(5):353–369, 2002.

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The design of a language for model transformations. *Software and Systems Modeling* 5(3):261–288, 2006.

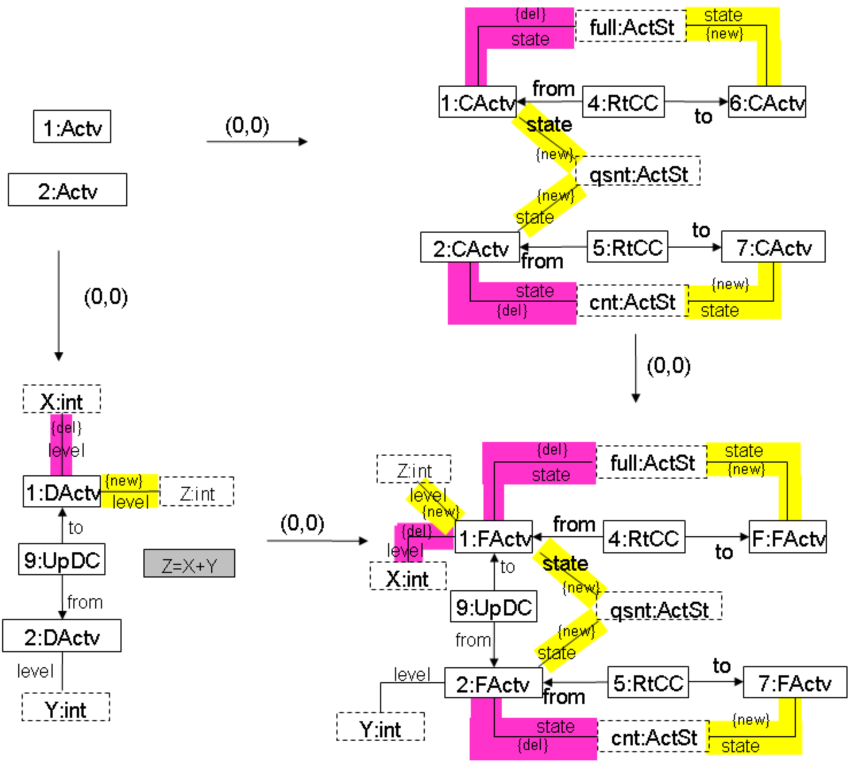


Figure 13: The rule resulting from the coordination of movements of full and contour flashing.

[BG04] P. Bottoni, A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. In *Proc. VL/HCC 2004*. Pp. 83–90. 2004.

[BKPT00] P. Bottoni, M. Koch, F. Parisi Presicce, G. Taentzer. Automatic Consistency Checking and Visualization of OCL Constraints. In *Proc. UML 2000*. Pp. 294–308. 2000.

[BL07] P. Bottoni, A. Labella. Pointed pictures. *Journal of Visual Languages and Computing* 18:523–536, 2007.

[BLG07] P. Bottoni, J. de Lara, E. Guerra. Action Patterns for Incremental Specification of Execution Semantics of Visual Languages. In *Proc. VL/HCC 2007*. Pp. 163–170. 2007.

[CMR96] A. Corradini, U. Montanari, F. Rossi. Graph processes. *Fundamenta Informaticae* 26(34):241–265, 1996.

[dBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed graph transformation with node type inheritance. *TCS* 376:139–163, 2007.

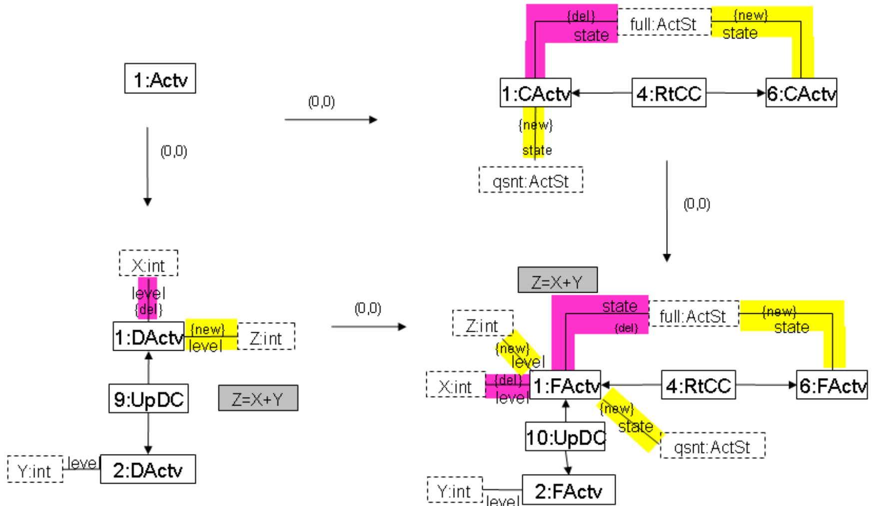


Figure 14: Coordinating movement of the full flashing flow with the formula in Figure 13.

[dGB07] J. de Lara, E. Guerra, P. Bottoni. Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules. In *Proc. GT-VMT'07*. 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[GEMT00] M. Goedicke, B. Enders, T. Meyer, G. Taentzer. Towards integration of multiple perspectives by distributed graph transformation. In Nagl et al. (eds.), *Proc. AGTIVE 1999*. Pp. 369–377. 2000.

[GR97] D. Giammarresi, A. Restivo. Two-dimensional languages. In *Handbook of Formal Languages*. Volume III, pp. 215–267. Springer, 1997.

[HKT02] R. Heckel, J. Küster, G. Taentzer. Confluence of Typed Attributed Graph Transformation with Constraints. In *Proc. ICGT 2002*. LNCS 2505, pp. 161–176. 2002.

[HP95] A. Habel, D. Plump. Unification, rewriting, and narrowing on term graphs. *Electr. Notes Theor. Comput. Sci.* 2, 1995.

[IPS82] A. Itai, C. H. Papadimitriou, J. L. Szwarcfiter. Hamilton Paths in Grid Graphs. *SIAM J. Comput.* 11(4):676–686, 1982.

[KK99] H. Kreowski, S. Kuske. Graph Transformation Units with Interleaving Semantics. *Formal Aspects of Computing* 11:690–723, 1999.

[LS04] S. Lack, P. Sobocinski. Adhesive Categories. In Ehrig et al. (eds.), *Proc. FOSSACS 2004*. Pp. 273–288. Springer, 2004.

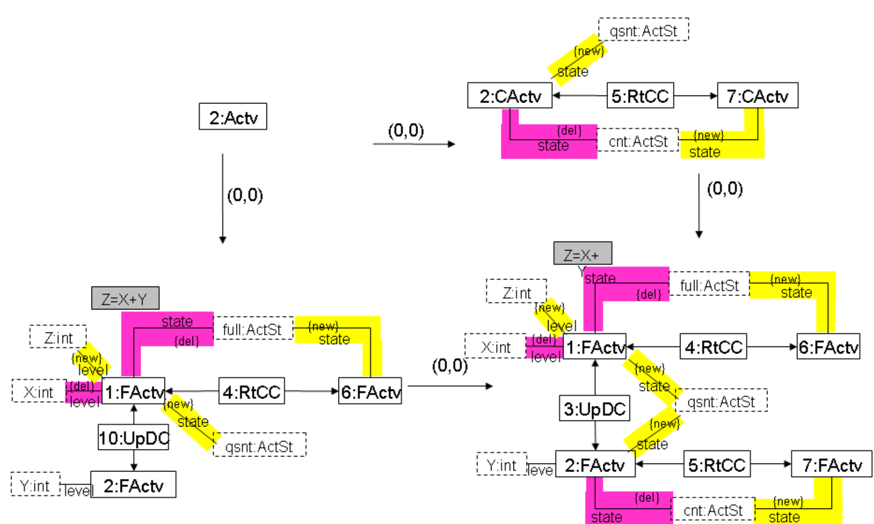


Figure 15: Coordinating movement of the contour flashing flow with the rule of Figure 14.

[Mos94] P. Mosses. *Recent Trends in Data Type Specification*. Chapter Unified algebras and abstract syntax, pp. 280–294. Springer, 1994.

[MW93] M. K. M Löwe, A. Wagner. *Term Graph Rewriting: Theory and Practice*. Chapter An Algebraic Framework for the Transformation of Attributed Graphs, pp. 185–199. John Wiley and Sons Ltd, 1993.

[Par94] F. Parisi Presicce. Transformations of Graph Grammars. In *TAGT*. LNCS 1073, pp. 428–442. 1994.

[SK74] R. Siromoney, K. Krithivasan. Parallel context-free grammars. *Information and Control* 24:155–162, 1974.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES-Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2*. Pp. 487–550. World Scientific, 1999.

[TB94] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG. In *Dagstuhl Seminar on Graph Transformations in Computer Science*. LNCS 776, pp. 380–394. Springer, 1994.

[WMYM08] Y. Watanobe, N. N. Mirenkov, R. Yoshioka, O. Monakhov. Filmification of methods: A visual language for graph algorithms. *JVLC* 19(1):123–150, 2008.

[YM02] R. Yoshioka, N. N. Mirenkov. Visual computing within environment of self-explanatory components. *Soft Computing* 7(1):20–32, 2002.

On a Graph-Based Semantics for UML Class and Object Diagrams

Anneke Kleppe, Arend Rensink

a.kleppe@utwente.nl, a.rensink@utwente.nl
Department of Computer Science
University of Twente, Enschede, The Netherlands

Abstract: In this paper we propose a formal extension of type graphs with notions that are commonplace in the UML and have long proven their worth in that context: namely, inheritance, multiplicity, containment and the like. We believe the absence of a comprehensive and commonly agreed upon formalisation of these notions to be an important and, unfortunately, often ignored omission. Since our eventual aim (shared by many researchers) is to give unambiguous, formal semantics to the UML using the theory of graphs and graph transformation, in this paper we propose a set of definitions to repair this omission. With respect to previous work in this direction, our aim is to arrive at more comprehensive and at the same time simpler definitions.

Keywords: UML, Class Diagram, Type Graph, Instance Graph, Graph Constraint

1 Introduction

Software industry is showing an increasing interest in model-driven development. Indeed, we have little doubt that the future lies in higher-level models to take the place of code, in all but the most performance critical domains. With this trend, however, the *quality* of those models is of increasing importance. By this we do not mean the quality of the product being modelled (which obviously is the final consideration) but rather of the modelling paradigm. Good models may not guarantee good software, but on the other hand, a bad (ambiguous, inconsistent or unclear) model can never be expected to yield a good end product, in particular if the transformation from model to software is largely automatic.

The quality of models is determined by many aspects, among we believe *precision*, *consistency* and *completeness* to be paramount. The precision of a model corresponds to the lack of ambiguity, or in other words, the degree to which the model will be understood in exactly the same way by different persons and tools during the software development process. Consistency formally means the existence of an (i.e., at least one) instance, or implementation, of the model, whereas completeness means the inclusion of all relevant aspects, or (in other words) the ability to predict the behaviour of the system under all circumstances.

The above "quality criteria" have a clear, universally agreed-upon interpretation in the world of mathematics. To make the benefits of the mathematical interpretation available for everyday use in the world of software modelling, however, it is imperative that there be a translation from the latter to the former; in other words, a formal semantics of the modelling language. For instance, it is commonly agreed that a natural interpretation of (UML-type) diagrams is in terms of *graphs* — essentially, just nodes with connecting edges. Indeed, many authors use UML class (and object) diagrams claiming that they are representations of type graphs. Unfortunately, few

provide an actual formal underpinning of this claim, or when they do, the semantics covers only a relatively small part of UML; for instance, [BELT04, LBE+07, KGKK02, TR05]. The most comprehensive is [VFV06], but even there such basic notions as multiplicities are missing. We see the absence of a more complete semantics as an important and regrettable omission, although from a purely formal standpoint, there is little challenge in providing the necessary definitions. The aim of this work is to bridge the gap between pure formalism and practicality.

Like the papers cited above, in this paper we distinguish the type and instance levels, or in other words, type graphs and instance graphs. We see a type graph as an intensional definition of a set of instance graphs, namely, those instance graphs for which it is a correct type. Type graphs are then enriched with constraints that capture UML concepts such as bi-directional associations, multiplicities, collection types, inheritance, redefinition of associations, and composition relationships. In this, we have based ourselves on the (verbal) descriptions in the UML 2.0 specification [OMG05].

In searching for the aforementioned balance between simplicity and expressiveness of the semantics, we have used the following guidelines:

- Instance graphs should be as simple and straightforward as we can make them, if necessary at the price of increasing their sizes. In other words, where there is a choice between enriching the formalism (resulting in more concise but more complex graphs) or using larger (sub-)graphs to encode complexity, we have tended to choose in favour of the latter.

- Type graphs should be as close to instance graphs as we can make them; the number of special features or decorations should be minimised.

  We have achieved this by using the concept of a *graph constraint*, which is essentially a template for a logical formula on top of an ordinary (type) graph.

The remainder of this paper is structured as follows: after providing the basic definitions to set the stage in Section 2, we discuss the graph constraints in Section 3. We consider these to be the heart of our contribution. In Section 4 we relate our constraints to the standardised UML concepts. Finally, in the conclusion (Section 5) we come back to the above considerations and re-evaluate our choices.

Unfortunately, it is not possible to include the full set of definitions into this paper. A complete version can be found in [KR08].

2 Basic concepts

Names and namespaces. UML is a visual language; its "sentences" are diagrams. However, a major part of any diagram is still text, and so we need conventions for visualising text inside diagrams. For this purpose, we define a set of *identifiers* ID, consisting of a *name* from a predefined universe *Name*, and a *namespace* from a set NS, defined as follows.

- An *identifier* is a pair $\langle ns, name\rangle$ of a namespace *ns* and a name *name*;
- There is a *root* or *top* namespace $\top \in NS$;
- For every $ns \in NS$ and $name \in Name$, the identifier $\langle ns, name\rangle$ is again a namespace.

To *visualise* an identifier we use a well-known notation, which is less cumbersome than the angular brackets: the name space and name are separated by a dot, and the top namespace is omitted altogether. Thus, $\langle ns, name\rangle$ is actually written $ns.name$.

For instance, the identifier `a.name.space` consists of the name `space` in the namespace `a.name`, which itself is an identifier with namespace `a` and name `name`. The identifier `a`, finally, consists of the name `a` in the top name space.

Signatures and algebras. For our definition of model we use the notion of attributed type graph, as defined in [EPT04]. The ingredients of this definition that are important here are:

- A collection of *data sorts* $Sort$, which are in fact identifiers (hence $Sort \subseteq ID$)
- A collection of *carrier sets* $Data$, partitioned into subsets for each of the sorts in $Sort$.

Graphs. One of the core concepts of this paper is that of *graphs*. We start by repeating the usual definition of a directed, multi-sorted graph.

Definition 1 (graph) A *graph* is a tuple $G = \langle Node, Edge, src, tgt\rangle$ where $Node$ is a set of nodes, $Edge$ a set of (directed) edges, and $src, tgt\colon Edge \to Node$ are source and target functions, respectively.

Note that although this definition does not yet specify node or edge *labels*, the nodes and edges do have *identities*. In some circumstances it will be the case that $Node, Edge \subseteq ID$ and the identities are actually meaningful to the reader; it then makes sense to include them in a visualisation of the graph. In particular, this is the case for *type graphs* — see below.

We will use two kinds of graph: instance graphs and type graphs. Both extend the notion of graph with some further structure. To start with instance graphs: these have an additional *labelling function* that associates an identifier with every node and edge. Furthermore, edges have *indices*, which are chosen from the set of natural numbers in such a way that the combination of source node, index and label together completely determine the edge.

Definition 2 (labelled graph) A *labelled graph* is a tuple $IG = \langle Node, Edge, src, tgt, ix, lab\rangle$ where $\langle Node, Edge, src, tgt\rangle$ is a graph and

- $ix\colon Edge \to \mathsf{Nat}$ is an *indexing function* assigning a natural number to every edge;
- $lab\colon (Node \cup Edge) \to ID$ is a *labelling* of nodes and edges;
- For $e_1, e_2 \in Edge$, if $src(e_1) = src(e_2)$, $ix(e_1) = ix(e_2)$ and $lab(e_1) = lab(e_2)$, then $e_1 = e_2$.

For a given node $n \in Node$ and label $a \in ID$, the set of outgoing edges is defined by

$$out(n, a) \;=\; \{e \in Edge \mid src(e) = n, lab(e) = a\} \;\; .$$

The indices assigned by the function ix are used for two purposes:

- To *distinguish* edges. Graphs may have distinct edges going out of the same node and bearing the same label, and even going to the same node (sometimes called parallel edges). These are useful to represent some UML concepts; in particular, ordered associations and bags. The indices serve to distinguish such edges, i.e., give them their own identity.

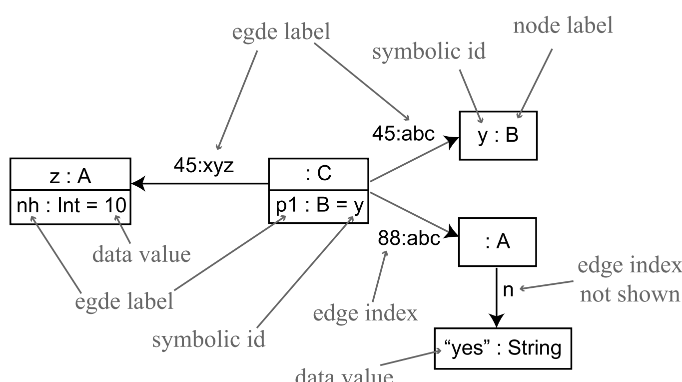


Figure 1: Example graphical representation of a labelled graph

- To *order* edges. One of the more powerful UML concepts is that of an ordered association; this does not only define a one-to-many relation between objects of one type to objects of another, but also establishes a local ordering over the set of (target) objects related to a single (source) object.

In contrast to edges, the encoding of node identities is not fixed by the above definition. It should, however, be understood that there is indeed some distinguishing mechanism, apart from the labelling function, that tells nodes apart. On the implementation level, for instance, this mechanism is typically based on memory addresses, or, for nodes in $Node \cap Data$, by the data value. On the modelling level, the position within a diagram in principle suffices as the distinguishing mechanism. On the other hand, for ease of reference it is very common to use symbolic *names* for nodes. Thus, we arrive at a graphical representation of labelled graphs based on the following conventions:

- Nodes are drawn as boxes with inscribed labels. The labels are preceded by a colon (':'). In front of a colon, there may either be a symbolic name, which is in fact itself an element of *Name*, but which plays no role in the formal meaning of the graph and in fact has no counterpart in Definition 2; or, in the case of nodes that are actually data values, the string representation of the data value may be displayed. (We will see below that the label is typically the type, which for data values $v \in Data$ is given implicitly by $type(v)$.)

- Edges are drawn as arrows with superimposed labels. The labels may be preceded by a number representing the edge index, separated from the label by a colon; in particular, this is necessary if there is more than one outgoing edge with that label and the numbering is needed to determine an ordering.

- As an important special case, edges pointing to nodes that are explicitly identified, either by data values or by symbolic names, may be represented by inscribed equations of the form "label = id" or "label:Type = id" instead of arrows.

Labelled graphs are used to represent concrete systems; in other words, they are on the level of individual programs or object diagrams. An example showing all of the graphical representation features is given in Figure 1. Here, y and z are symbolic names having no formal meaning within

the graph, whereas 10 and "yes" are data values of type Int and String, respectively, and 88, 45 etc. are edge indices.

Graph morphisms. With respect to our aim of providing a sound and comprehensive formalisation of UML concepts, one aspect is not yet completely covered, namely the fact that node identities and edge indices are not uniquely determined by the diagrams. In this sense, the formal interpretation of the diagrams remains ambiguous.

The reason why we are nevertheless content with this solution is that this ambiguity is not harmful, because the choice in no way matters to the actual meaning. Put differently, it is allowed to abstract away from the precise identities, provided the nodes and edges remain distinguishable. The standard way to formalise this type of argument is by interpreting the structures under consideration — here, our graphs — *up to* or *modulo* some equivalence. In this particular case, the standard way to define an appropriate equivalence is through the notion of *graph isomorphism*.

Definition 3 (graph (iso)morphism) Given two graphs G,H, a *morphism* from G to H is a pair of mappings $f = (f_{Node}\colon Node_G \to Node_H, f_{Edge}\colon Edge_G \to Edge_H)$ such that

- Node and edge labels are preserved: $lab_H \circ (f_{Node} \cup f_{Edge}) = lab_G$;
- Sources and targets are preserved: $src_H \circ f_{Edge} = f_{Node} \circ src_G$ and $tgt_H \circ f_{Edge} = f_{Node} \circ tgt_G$

f is an *isomorphism* if f_{Node} and f_{Edge} are bijective, i.e., provide a one-to-one mapping between $Node_G$ and $Node_H$, resp. $Edge_G$ and $Edge_H$. We write $G \cong H$ (G is isomorphic to H) to denote that there is an isomorphism from G to H.

It is especially important to realise that (iso)morphisms are *not* required to either respect node identities or edge indices, symbolic names, or diagram layout.

For one particular purpose we will later on strengthen the requirements on morphisms, in such a way that the ordering on edge indices is sometimes required to be preserved; namely, when we use the index to reflect an ordering over the edges themselves.

Type graphs. For purposes of documentation, structuring and correctness, it is common to impose a discipline over labelled graphs, comparable to the grammar of programming languages, or more to the point here, comparable to a class diagram. In particular, we use a *type graph* to impose local constraints on the allowed labels and connections between edges and nodes, and associated *constraints* to impose other, more sophisticated or less local, properties.

Definition 4 (type graph) A *type graph* is a tuple $TG = \langle NType, EType, src, tgt, inh\rangle$ where

1. $NType \subseteq ID$ is a set of *node types* and $EType \subseteq ID$ a set of *edge types*;
2. $\langle NType, EType, src, tgt\rangle$ is a graph, with $NType$ as node set and $EType$ as edge set, such that $src(e) = ns(e)$ for any $e \in EType$;
3. $inh \subseteq NType \times NType$ is a reflexive partial ordering relation expressing that some node types *inherit* from others. (Reflexivity here means that $T\ inh\ T$ holds for all node types $T \in NType$.)

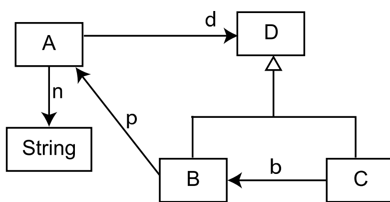


Figure 2: Example type graph

We typically use capital letters (T, E) to range over node and edge types.

The condition on the source function of edges (clause 2 in the definition) states that the source type of an edge is at the same time its name space. Since edge types are identifiers and identifiers are pairs of names and namespaces, it follows that edge types are uniquely determined by their source type and name. This setup allows us to use edge types with the same name, but only for distinct source types — which is consistent with the situation in most, if not all, object-oriented paradigms.

Also note that *inh* is a partial order, but not necessarily a forest: this implies that a node type can extend more than one other node type (in common terminology, our type graphs support multiple inheritance). At the same time, the partial order nature of *inh* implies that there can be no inheritance cycles.

For "node type" in the definition above, one may for most purposes read "class;" the only difference is that the node types typically include data sorts. We say that *TG builds on* a signature if $Sort \subseteq NType$.

A visual representation of a type graph can be given by drawing every node type as a box with the type identifier inscribed, every edge type as a "normal" arrow with the edge name as label, and every extension (i.e., from *ext*, not *inh*!) as an unlabelled arrow with triangular arrow head. Figure 2 shows an example type graph. This is very close to the traditional class diagram view, except that the data sorts are not treated as special cases (i.e., data type attributes are not distinguished from associations).

Typing and instance graphs. The meaning of a type graph is defined by the set of its (correctly typed) instances.[1] The idea is that the instances of a type graph *TG* are labelled graphs with labels chosen from the types of *TG*, and consistent with the graph structure of *TG* modulo inheritance. To formalise it, we use the following auxiliary notation for arbitrary nodes n and node types T, resp. edges e and edge types E:

$$\begin{aligned} n{:}T \quad &:\Leftrightarrow \quad lab(n)\ inh\ T \\ e{:}E \quad &:\Leftrightarrow \quad lab(e) = E \ . \end{aligned}$$

In words, $n{:}T$ expresses that the label of the node n (in the instance graph under consideration) is a node type that inherits from T. Note that it follows that, for a given n, there can easily be more than one node type T such that $n{:}T$, ranging from $T = lab(n)$ to all generalisations of T. On the other hand, in case of edges, $e{:}E$ expresses that $lab(e)$ is *exactly* the edge type E.

[1] Strictly speaking, the meaning is defined by the *category* of instances and valid morphisms: as mentioned above, in one case we need to impose additional requirements on the morphisms rather than the graphs.

Definition 5 (instance graph) Let *TG* be a type graph. A labelled graph *IG* is *typed by TG*, or an *instance graph of TG*, if for every node $n \in Node$ and every edge $e \in Edge$:

- $lab(n) \in NType$ and $lab(e) \in EType$;
- $src(e){:}src(lab(e))$ and $tgt(e){:}tgt(lab(e))$.

The set of instance graphs of *TG* is denoted $Inst[TG]$ (but see Footnote 1).

For instance, the labelled graph in Figure 1 is *not* an instance graph of the type graph in Figure 2, since it contains several edge labels that are not present in the type graph.

3 Constraints

The concepts introduced in the previous section are, in the sense of existing graph theory, straightforward; in fact, the only non-standard concepts are the structure we have chosen for identifiers, and the fact that we are using indexed edges in labelled (instance) graphs. In this section, we introduce a way to enrich type graphs, and so constrain the set of valid instance graphs, in ways that formalise the concepts found in UML.

First of all, we give a general definition of a *constraint set* over a graph; then, we define a series of special types of constraints tuned towards UML concepts.

Definition 6 (graph constraint) Let *TG* be a type graph. A *constraint set over TG* is a tuple $\langle Con, sat\rangle$ where *Con* is a set of *graph constraints*, and $sat \subseteq Inst[TG] \times Con$ is a *satisfaction relation* over the instances of *TG*. We denote *IG sat c* to denote that an instance graph *IG* satisfies a constraint *c*.

This definition only specifies that a graph constraint is something for which there exists an interpretation, expressed in terms of the graphs that satisfy the constraint. The interpretation is embodied in the satisfaction relation, *sat*. The real question is how *sat* is defined. By combining type graphs with a constraint set, we arrive at the concept of a *model*, which is our equivalent to a UML class diagram.

Definition 7 (model) A *model* is a pair $Mod = \langle TG, Con\rangle$ where *TG* is a type graph, and *Con* is a constraint set over *TG*, consisting of constraints of the types listed below.

The main contribution of this work, apart from the selection of the appropriate type and instance graph definitions, lies in the definition of a number of useful graph constraint “templates” and the corresponding satisfaction relations. The constraints can be subdivided into a number of categories, listed in Table 3. In this workshop paper, we can only discuss a few of the templates in detail; the report version [KR08] contains the complete list, in the same style as the ones reported here.

3.1 Association constraints: Bidirectionality

Associations in UML class diagrams have the property that they can (in principle) be traversed in either direction. Moreover, in general the ends of an association can have their own names. This

Table 3: A classification of constraints

Category	**Constraints**
Node type	Abstractness
Association	Bidirectionality, Multiplicities, Indexing, Uniqueness
Containment	Acyclicity, Unsharedness
Specialisation	Subsetting, Redefinition, Union
General	OCL

is in contrast to the graphs of this paper, where edges are unidirectional. To model bidirectional associations, we therefore need *two* edges, one for either direction, which *oppose* each other.

Definition 8 (bidirectionality constraint) Let TG be a type graph. A *bidirectionality constraint* over TG is a pair $\mathsf{oppose}(D,E)$ where $D,E \in EType$ are edges in TG, such that $src(D) = tgt(E)$ and $tgt(D) = src(E)$. Satisfaction is defined for all $G \in Inst[TG]$ by

$$G\ sat\ \mathsf{oppose}(D,E)\ :\Leftrightarrow\ \forall n_1{:}src(D), n_2{:}tgt(D).\ \ |\{d \in out(n_1,D) \mid tgt(d) = n_2\}| = |\{e \in out(n_2,E) \mid tgt(e) = n_1\}|\ .$$

Figure 4 gives an example of a bidirectionality constraint. The type graph (left hand side) has an associated constraint $\mathsf{oppose}(\mathsf{B.c},\mathsf{C.b})$, visualised as a two-headed arrow. The centre graph does not satisfy this constraint, as there is a C.b-typed edge without an opposing B.c-typed one. In the right hand side graph this is repaired, so that this graph is a valid instance of the (enriched) type graph.

3.2 Association constraints: Indexing

To capture the notion of an *ordered collection* from class diagrams, we need to formalise what it means for a set of graph nodes to be ordered. To capture this correctly is actually quite involved, even though it is conceptually straightforward. Here we make use of the edge indices that are part of the instance graphs (see Definition 5): if an edge type is declared as indexed, the edge indices have to be picked from a consecutive range from 1 upwards; and moreover (in fact, more importantly), morphisms are required to respect the edge indices.

Definition 9 (indexing constraint) Let TG be a type graph. An *indexing constraint* over TG is a predicate $\mathsf{indexed}(E)$, with $E \in EType$. Satisfaction is defined for all $G \in Inst[TG]$ and all

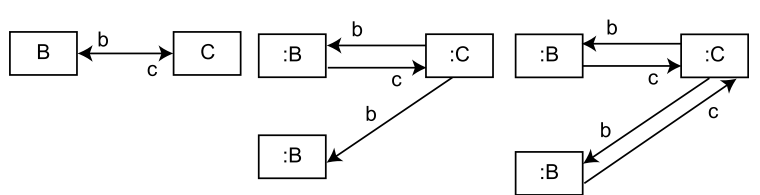


Figure 4: Example type graph modelling bidirectional edges.

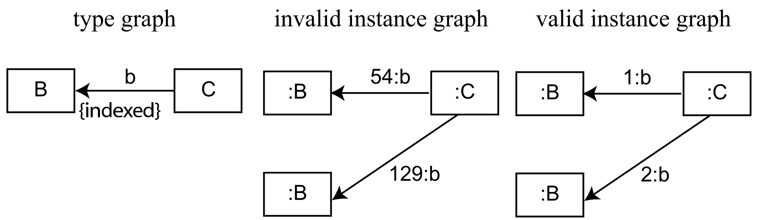


Figure 5: Example type graph with an indexing constraint

morphisms f between instance graphs $G,H \in Inst[TG]$ by

$$\begin{aligned} G\ sat\ \mathsf{indexed}(E) \quad &:\Leftrightarrow \quad \forall n{:}src(E), \forall e \in out(n,E).\ 1 \leq ix(e) \leq |out(n,E)| \\ f\ sat\ \mathsf{indexed}(E) \quad &:\Leftrightarrow \quad \forall e : E \Rightarrow ix(f_{Edge}(e)) = ix(e) \ . \end{aligned}$$

Figure 5 gives an example of an indexing constraint. The type graph (left hand side) has an associated constraint $\mathsf{indexed}(\mathsf{C.b})$, visualised by the annotation $\{\mathsf{indexed}\}$ near the arrow head. The centre graph does not satisfy this constraint, as it has two outgoing C.b-typed edges with indices $\{54,129\}$, which do not form a consecutive range. This is repaired in the right hand side graph. More importantly, where ordinarily the right hand side graph would be considered symmetric (having two interchangeable B-typed nodes), this is no longer true in the presence of the indexing constraint: the symmetry (formally, an isomorphism from the graph to itself) maps $(n,\mathsf{C.b},1)$ to $(n,\mathsf{C.b},2)$ (where n is the C-typed node in the graph) and hence does not satisfy the constraint, since it does not preserve edge indices.

3.3 Containment constraints: Acyclicity and unsharedness

Another notion from UML class diagrams that has proved to be quite useful in practice is that of *aggregation* or *containment*. Whereas ordinary edges may impose an arbitrary structure on the nodes they connect, containment is intended to reflect a hierarchy of things. Therefore, when edges in a type graph are declared to be acyclic, the intention is that the edges in the corresponding instance graphs do not form a cycle.

This type of constraint is in fact quite powerful if the edge types in the hierarchy do form a cycle *in the type graph*. In that case, there could in principle be instance graphs with arbitrarily large cycles, all of which are ruled out by a single acyclicity constraint. From this it can be seen that the acyclicity constraint is a non-local property, and hence outside the class of first-order logic.

Definition 10 (acyclicity constraint) Let TG be a type graph. An *acyclicity constraint* over TG is a tuple $\mathsf{acyclic}(E_1,\ldots,E_n)$ where $E_1,\ldots,E_n \in EType$ is a collection of edge types. Satisfaction is defined for all $G \in Inst[TG]$ by

$$G\ sat\ \mathsf{acyclic}(E_1,\ldots,E_n) \ :\Leftrightarrow\ \{e{:}E_i \mid 1 \leq i \leq n\} \text{ is cycle free.}$$

Figure 6 shows an example of an acyclicity constraint. The type graph (left hand side) has an associated constraint $\mathsf{acyclic}(\mathsf{C.b},\mathsf{B.c})$ visualised by diamond-shaped decorations at the sources of the edge types. (This visualisation always specifies a *single* acyclicity constraint, consisting

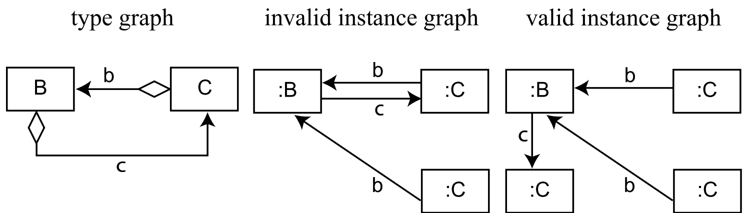


Figure 6: Example type graph showing an acyclicity constraint.

of all diamond-decorated edge types. The case where a node or edge type can in principle be part of distinct acyclic-hierarchies cannot be visualised without adding further distinguishing information to the diamonds, for instance in the form of identifiers.) The centre graph of Figure 6 shows a small instance of such a cycle; hence this graph violates the constraint. In the right hand side graph this is repaired, so that this is a valid instance of the (enriched) type graph.

The acyclicity constraint guarantees the absence of cycles (as its name suggests), but it does *not* guarantee the absence of sharing; in other words, on its own it is not certain that the structure imposed by acyclic edges is a forest. To complement this, we also introduce a constraint that specifies the absence of sharing; as will see, the UML *composite* is a combination of acyclicity and unsharedness. For an example unsharedness constraint, we refer to the technical report.

Definition 11 (unsharedness constraint) Let TG be a type graph. An *unsharedness constraint* over TG is a tuple $\mathsf{unshared}(E_1,\dots,E_n)$, where $E_1,\dots,E_n \in EType$. Satisfaction is defined for all $G \in Inst[TG]$ by:

$$G\ sat\ \mathsf{unshared}(E_1,\dots,E_n) \ :\Leftrightarrow\ \forall d{:}E_i, e{:}E_j.\ tgt(d) = tgt(e) \Rightarrow d = e \ .$$

3.4 Specialisation constraints: Redefinition

We have included node type inheritance as a basic notion in type graphs, reflecting the common concept from UML and other object-oriented settings. For edges, on the other hand, although there is likewise a notion of specialisation, but no single commonly accepted way to capture this. Instead, UML knows several ways to define specialisation-like relationships between edges, which we here formalise through edge type constraints.

These can be categorised as *subset*, *redefinition* and *union* constraints. The only type we discuss in this paper is redefinition; for the others see the technical report. Redefinition imposes a kind of “subtype” relation over edges, such that the supertype is overridden by the subtype. More precisely, if an edge type D redefines another type E, then a node of D’s source type may no longer have an outgoing E-type edge — instead, this should be a D-type edge.

Definition 12 (redefinition constraint) Let TG be a type graph. A *redefinition constraint* over TG is a pair $\mathsf{redefine}(D,E)$, where $D,E \in EType$ are edges in TG, such that $src(D)\ inh\ src(E)$ and $tgt(D)\ inh\ tgt(E)$. Satisfaction is defined for all $G \in Inst[TG]$ by:

$$G\ sat\ \mathsf{redefine}(D,E) \ :\Leftrightarrow\ \nexists e{:}E.\ src(e){:}src(D) \ .$$

Figure 7 shows an example of a redefinition constraint. The type graph (left hand side) has an associated constraint $\mathsf{redefine}(\mathsf{F.b},\mathsf{A.d})$, visualised by the annotation $\{\mathsf{redefines}\}$ at the arrow

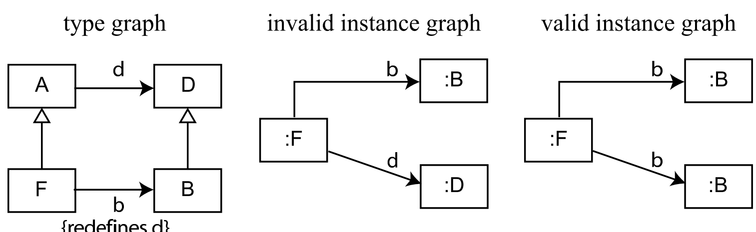


Figure 7: Example type graph with a redefinition constraint

head. The centre graph does not satisfy the constraint, since there is an A.d-type edge going out of an F-type node. In the right hand side this is repaired, by changing the offending edge into an F.b-type; as a result, this instance graph satisfies the redefinition constraint.

4 UML Semantics

In this section, we will apply the general framework introduced above to UML class and object diagrams, thus providing a formal, graph-based semantics for these diagrams.

Class and object diagrams. The formal meaning of a UML class diagram is that it is a model. An overview of the mapping of UML class diagram concepts to the concepts in our framework can be found in Table 9. The model's type graph can be easily recognized: each class in the diagram is a node and each directed association is an edge. Non-directed associations translate to pairs of edges with a bi-directionality constraint, where the edge labels correspond to the names of the association ends.

Most of the constraint types in our graph-based framework can also be easily recognized in a class diagram, for instance a bidirectionality constraint is shown in a class diagram in the same manner as we have shown in Figure 4.

Table 8: Summary of all constraints — including those that are omitted from this workshop version; see the full report [KR08]. (Notation: $\vec{E} = E_1 \cdots E_n$)

abstract(T)	$\nexists n \in Node_G.\ lab(n) = T$
oppose(D,E)	$\forall n_1{:}src(D), n_2{:}tgt(D).$
	$\lvert\{d \in out(n_1,D) \mid tgt(d) = n_2\}\rvert = \lvert\{e \in out(n_2,E) \mid tgt(e) = n_1\}\rvert$
mult(E,μ)	$\forall n{:}src(E).\ \lvert out(n,E)\rvert \in \mu$
indexed(E)	$\forall n{:}src(E), \forall e \in out(n,E).\ 1 \leq ix(e) \leq \lvert out(n,E)\rvert$
	$\forall e : E \Rightarrow ix(f_{Edge}(e)) = ix(e)$
unique(E)	$\forall n{:}src(E). \forall e_1, e_2 \in out(n,E). tgt(e_1) = tgt(e_2) \Rightarrow e_1 = e_2$
acyclic$(\vec{E})$	$\{e{:}E_i \mid 1 \leq i \leq n\}$ is cycle free
unshared$(\vec{E})$	$\forall d{:}E_i, e{:}E_j.\ tgt(d) = tgt(e) \Rightarrow d = e$
subset(D,E)	$\forall d{:}D.\ \exists e{:}E.\ src(e) = src(d) \wedge tgt(e) = tgt(d)$
redefine(D,E)	$\nexists e{:}E.\ src(e){:}src(D)$
union$(D,\vec{E})$	$\forall 1 \leq i \leq n :$ subset$(E_i, D) \wedge$
	$\forall d{:}D.\ \exists 1 \leq i \leq n, e : E_i.\ src(e) = src(d) \wedge tgt(e') = tgt(e)$
ocl(ϕ)	$G \models [\![\phi]\!]$

Table 9: Mapping of UML class diagram concepts to graphs

Category	UML class diagram	Graph model
General	class	type node
	primitive type attribute	type edge E with $tgt(E) \in Sort$
	non-primitive type attribute	type edge E with $tgt(E) \notin Sort$
Association	directed association	type edge
	non-directed/bi-directional	pair of type edges with oppose
	multiplicity	mult-constraint
	set (default for mult > 1)	unique but not indexed
	bag	neither indexed nor unique
	sequence	indexed but not unique
	ordered set	both indexed and unique
	aggregation	acyclic
	composition	both acyclic and unshared
	OCL constraint	ocl
Specialisation	inheritance	*inh*-relation on type nodes
	subset	subset constraint
	redefines	redefine constraint
	union	union constraint
	UML object diagram	**Labelled graph**
General	object	node
	object type	node label
	link	edge
	link type	edge label
	instance name	symbolic id

UML class diagrams can be accompanied by OCL constraints. In our semantics, these are also translated to graph constraints, of the type ocl(ϕ)(omitted in this workshop paper) , by relying on the existing OCL semantics (which essentially provides a translation to first order logic). In other words, a class diagram together with its OCL constraints is translated to a model, in the sense of Definition 7. This illustrates the fact that OCL constraints cannot be seen as separate from the class diagram. It will be no surprise that we define an object diagram to be a labeled graph. When a labeled graph satisfies a certain model, for instance a class diagram or a class diagram combined with constraints, it is a valid instance of that model. An overview of the mapping of UML object diagram concepts to the concepts in our framework can be found in Table 9.

Evaluation. The semantics presented here should, as any semantics, uphold the commonly known characteristics of UML diagrams, even those that have (unfortunately) not been made explicit in the UML specification. As an example, we show two such characteristics; again, we refer the reader to [KR08] for a more comprehensive discussion. Let $\langle TG, C\rangle$ be the model representing the class diagram under consideration.

- A commonly known characteristic of UML class diagrams is that if the one end of a bi-directional association is marked {bag} then the other end should be marked {bag} or {sequence}. Likewise, if one end is an (ordered) set, the other end must be a set as well.

In our semantics, this situation arises when the bi-directionality constraint is combined with the uniqueness constraint. This UML characteristics then translates to the following "law" of graph constraints:

$$\mathsf{oppose}(D,E) \Rightarrow (\mathsf{unique}(D) \Leftrightarrow \mathsf{unique}(E))$$

- According to the UML specification, the acyclicity constraint should always be combined with bi-directionality: "Only binary associations can be aggregations" ([OMG05], page 37). At the same time, only one end of this association can be marked as aggregate. This is in accordance with common sense, which says that a part cannot contain its container. In our framework this forbidden situation would occur when the acyclicity constraint is defined for both edges of a bi-directional association, i.e.:

  $$\mathsf{oppose}(D,E) \wedge \mathsf{acyclic}(D) \wedge \mathsf{acyclic}(E) \quad .$$

  In our semantics, there are no instances that would satisfy such a model; in other words, this combination of constraints is inconsistent (i.e., a contradiction).

These cases give confidence that the presented semantics really conforms to the intuition behind UML. On the other hand, our semantics does not support all UML aspects; in particular, the following are not included:

- Names of associations. Only the role names associated with the association ends are taken into account, because we consider these to be more important.
- N-ary associations, i.e. associations between more than two classes. These tend to occur very rarely in class diagrams; moreover, the UML specification itself treats them more like classes than like associations.
- Derived attributes or association ends. As the name suggests, these are derived values and need not be explicitly part of the formal representation.
- Navigability of associations. We feel that the directionality of the edges in the association pair provides enough information.
- Operations. These cannot be expressed by a static structure.

5 Conclusions

In this paper we present an elegant and simple semantics of UML class and object diagrams based on graph structures that are as close as possible to familiar notions in graph theory. The main insight used is that a UML class diagram cannot be treated as a simple type graph. It is a much richer structure, which is embodied in our framework by the use of (graph) constraints. The use of constraints also makes it possible to change the given semantics to include or exclude certain semantic elements. For instance, by disallowing the abstract class constraint type one can easily define class diagrams without abstract classes. Furthermore, our definitions do not only provide a semantics for both diagram types, but for the relationship between them as well.

As stated in the introduction, simplicity was one of our main guidelines. The only addition we made to the familiar notion of labelled graph is the edge indexing function, in order to capture ordered associations. We investigated (and rejected) several alternatives. The use of special "collection node types" makes the definition of instance graph much more complex. Another possibility is to use hypergraphs, but that in itself makes the model much more complex. A third option is to use special edges between the target nodes of an ordered association to represent the ordering, as we have done before in [KKR06]. The problem with this solution is that the ordering needs to be local not only to all edges of the given type, but also to the source node.

In the introduction we stated that the quality of models is determined by precision, consistency, and completeness. Our semantics provide a precise meaning to class and object diagrams. Furthermore, the consistency of a model, i.e. the existence of instances, can be checked using the given definitions. For instance, we can prove that a model with a bi-directional association that is an aggregate in both directions is inconsistent. Research into this "logic of UML models" has so far been scarce (see e.g. [MB07]). The completeness of a model cannot be guaranteed by our semantics. However, the semantics themselves are more complete than any other graph-based semantics that we have found in the literature. For instance, [BELT04, KGKK02, LBE+07] only visualise a type graph with inheritance as a UML class diagram, thus implying a graph-based meaning for class diagrams without actually defining an UML semantics. [VFV06] includes attributes, associations, and inheritance, but not containment, multiplicity, abstract classes, or edge specialisation.

As a next step, we intend to investigate the integration of our framework with the existing theory of graph transformation. An important issue is to reconciliate our encoding of indexed edges with the requirements of algebraic graph transformations.

A final point we would like to make goes back to the introduction, and concerns the (scientific) merit of the type of effort we have undertaken in this paper. We believe to have achieved a simple, intuitive and workable graph-based semantics. The fact that UML was conceived over a decade ago and still no graph-based semantics with this degree of completeness had been presented (in contrast to other theoretical bases, e.g., [LB98, Gei98, Öve99, Kna99, EK99, DJPV02, Ham05]), indicates that our undertaking was not trivial. Moreover, there is a great need for such semantics, if ever model-driven engineering is to become a dependable method. We know that many of the ideas brought together in this work have been presented earlier; however, the strength of this contribution lies in the particular combination of these ideas. All in all, we believe that the result should be judged not only on novelty, but on completeness, adaptability, and usability as well. If there is no well-defined forum where this type of effort can receive recognition, there will be no incentive, and the gap between theory and practice may remain with us forever.

Acknowledgements: The research in this paper was carried out in the GRASLAND project, funded by the Dutch NWO (project number 612.063.408).

Bibliography

[BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In Wer-

melinger and Margaria (eds.), *FASE*. LNCS 2984, pp. 214–228. Springer, 2004.

[DJPV02] W. Damm, B. Josko, A. Pnueli, A. Votintseva. Understanding UML: A Formal Semantics of Concurrency and Communication in Real-Time UML. In Boer et al. (eds.), *FMCO*. LNDS 2852, pp. 71–98. Springer, 2002.

[EK99] A. Evans, S. Kent. Core Meta-Modelling Semantics of UML: The pUML Approach. Pp. 140–155 in [FR99].

[EPT04] H. Ehrig, U. Prange, G. Taentzer. Fundamental Theory for Typed Attributed Graph Transformation. In Ehrig et al. (eds.), *ICGT*. LNCS 3256, pp. 161–177. Springer, 2004.

[FR99] R. B. France, B. Rumpe (eds.). *UML'99: The Unified Modeling Language - Beyond the Standard*. LNCS 1723. Springer, 1999.

[Gei98] R. Geisler. Precise UML Semantics Through Formal Metamodeling. In Andrade et al. (eds.), *Proceedings of the OOPSLA'98 Workshop on Formalizing UML. Why? How?* 1998.

[Ham05] Y. Hammal. A Formal Semantics of UML StateCharts by Means of Timed Petri Nets. In Wang (ed.), *FORTE*. LNCS 3731, pp. 38–52. Springer, 2005.

[KGKK02] S. Kuske, M. Gogolla, R. Kollmann, H.-J. Kreowski. An Integrated Semantics for UML Class, Object and State Diagrams Based on Graph Transformation. In Butler et al. (eds.), *Integrated Formal Methods (IFM)*. Lecture Notes in Computer Science 2335, pp. 11–28. Springer, 2002.

[KKR06] H. Kastenberg, A. G. Kleppe, A. Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In Gorrieri and Wehrheim (eds.), *FMOODS*. LNCS 4037, pp. 186–201. Springer Verlag, London, June 2006.

[Kna99] A. Knapp. A Formal Semantics for UML Interactions. Pp. 116–130 in [FR99].

[KR08] A. Kleppe, A. Rensink. A Graph-Based Semantics for UML Class and Object Diagrams. Ctit technical report TR-CTIT-08-06, Department of Computer Science, University of Twente, Jan. 2008.
http://eprints.eemcs.utwente.nl/11963/

[LB98] K. Lano, J. Bicarregui. Semantics and Transformations for UML Models. In Bézivin and Muller (eds.), *UML*. LNCS 1618, pp. 107–119. Springer, 1998.

[LBE+07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed graph transformation with node type inheritance. *Theoretical Computer Science* 376(3):139–163, May 2007.

[MB07] A. Maraee, M. Balaban. Efficient Reasoning About Finite Satisfiability of UML Class Diagrams with Constrained Generalization Sets. In Akehurst et al. (eds.), *ECMDA-FA*. LNCS 4530, pp. 17–31. Springer, 2007.

[OMG05] OMG. Unified Modeling Language: Superstructure. Technical report formal/05-07-04, OMG, 2005.
http://www.omg.org/cgi-bin/doc?formal/05-07-04

[Öve99] G. Övergaard. A Formal Approach to Collaborations in the Unified Modeling Language. Pp. 99–115 in [FR99].

[TR05] G. Taentzer, A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. In Cerioli (ed.), *FASE*. LNCS 3442, pp. 64–79. Springer, April 2005.

[VFV06] G. Varró, K. Friedl, D. Varró. Implementing a Graph Transformation Engine in Relational Databases. *Journal of Software and Systems Modelling* 5(3):313–341, Sept. 2006.

EASST

Graph Transformations for the Resource Description Framework

Benjamin Braatz[1] **and Christoph Brandt**[2]

[1] bbraatz@cs.tu-berlin.de
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany

[2] christoph.brandt@uni.lu
Computer Science and Communications Research Unit
Université du Luxembourg, Luxembourg

Abstract: The Resource Description Framework (RDF) is a standard developed by the World Wide Web Consortium (W3C) to facilitate the representation and exchange of structured (meta-)data in the "Semantic Web". While there is a large body of work dealing with inference on RDF, a concept for transformation and manipulation is still missing. Since RDF uses graphs as a formal basis, this paper proposes the use of algebraic graph transformations with their wealth of well-known constructions and results for this purpose. It turns out that RDF graphs are an interesting application area for graph transformation methods, where some significant differences to classical graphs yield practically relevant solutions for features like attribution, typing and globally unique nodes.

Keywords: Resource Description Framework, Algebraic Graph Transformation, Category Theory

1 Introduction

The Resource Description Framework (RDF) (see [MM04]) consists of a set of specifications, which provide an abstract syntax, semantics and several concrete representations for storage, exchange and reasoning on arbitrary (meta-)data. Its primary use case is the "Semantic Web", in which the creation of globally distributed knowledge bases is envisaged. In Section 2 we summarise the abstract syntax of RDF and provide a categorical framework for it.

The theoretical treatment of RDF mainly consists of inference mechanisms, which allow to derive information from RDF data stores by the use of inference rules like, e. g., the transitivity of a predicate. This focus is also apparent in the SPARQL Query Language for RDF (see [PS07]), the proposed standard for accessing an RDF data store, which in contrast to SQL does not contain any structures for manipulating the data themselves, but only constructs for retrieval.

At the present time, the modification of data in RDF stores is mostly implemented by adding and removing individual data items. For many use cases, such as data in web-based applications or the use of RDF as an abstract syntax for visual languages, it would, however, be desirable to restrict this to sensible rule-based transformations, which can be modelled and analysed formally.

This contribution proposes the use of algebraic graph transformation for the purpose of manipulating RDF data stores. Such a transformation approach allows for the formal treatment of reversibility, dependencies and conflicts of transformations. Moreover, it facilitates the use of grammars to restrict the possible structures and their development. In Section 3 we investigate several well-known graph transformation approaches concerning their suitability for RDF and in Section 4 we present our proposal for an RDF transformation concept.

2 The Resource Description Framework

In this section we reformulate and slightly extend the theoretical underpinning of RDF in the framework of category theory. Some of the specific phenomena of RDF theory correspond to well-known constructions in category theory and can hence be treated more elegantly in this setting, which makes this reformulation worthwhile independently of the transformation approach developed in the following sections.

2.1 RDF Graphs

The basic building blocks of RDF are Uniform Resource Identifiers (URIs) (see [BFM98]) and literals. For the purposes of this paper we assume to have a given set URI of URIs, where we will use XML Namespaces (see [BHLT06]) to shorten URIs. The namespaces $\mathsf{rdf:}$, $\mathsf{rdfs:}$ and $\mathsf{xsd:}$ are used for pre-defined URIs in the corresponding RDF and XML Schema specifications.

Literals are Unicode strings (see [Uni07]), which can be typed literals (with a URI denoting the data type of the literal) or plain literals (with an optional language tag denoting the human language of the literal). We formalise this by assuming a set String of all Unicode strings and a set Lang of all language tags (including the empty tag to support optionality). The set of all literals is then constructed by $\mathsf{Lit} := (\mathsf{URI} \times \mathsf{String}) + (\mathsf{Lang} \times \mathsf{String})$, where $\times$ denotes the (Cartesian) product and $+$ the disjoint union of sets.

The typed literals facilitate the attribution of an RDF graph by literal values from arbitrary pre- or self-defined data types, which are given by corresponding string representations. For example, the datatypes of XML Schema (see [BM04]) are recommended in the RDF specifications, such that literals like $(\mathsf{xsd{:}integer}, \mathtt{42})$ or $(\mathsf{xsd{:}date}, \mathtt{2008\text{-}03\text{-}29})$ may be used in RDF graphs. Hence, an extension of the theory by algebraic specifications or similar techniques as it is necessary for attributed graphs (see, e. g., [EEPT06]) is not needed in RDF.

RDF graphs are now given by sets of statements, where a statement is a triple consisting of a subject, a predicate and an object. In the language of graph theory subjects and objects are nodes and statements are edges labelled with predicates. Subjects and objects can be URIs, literals or "blank nodes", where blank nodes are nodes, which do not have a global identity, but are local to the graph. Predicates are always given by URIs.

Definition 1 (RDF Graph) An RDF graph $G = (G_{\text{Blank}}, G_{\text{Triple}})$ consists of a set G_{Blank} of blank nodes and a set $G_{\text{Triple}} \subseteq (G_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit}) \times \mathsf{URI} \times (G_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit})$ of triples.

Remark 1 (Differences to RDF specifications) *The formal specifications of RDF (see [KC04] and [Hay04]) assume the blank nodes to be drawn from an infinite set, which is given globally.*

Our approach is to keep the blank nodes local to the graphs and use category theoretical machinery to ensure disjointness of blank nodes from unrelated graphs in the following subsections.

Moreover, the RDF specifications only allow literals as objects, but not as subjects of triples. The relaxation of this requirement is, however, common in later literature (see e. g. [MPG07]) and eases our formal treatment significantly.

The use of global sets of URIs and literals enables the distributed creation and storage of information regarding the same entities and resources, where their connection is established by the usage of identical URIs without the need to give explicit relations or morphisms.

In order to allow the typing of nodes by classes, the declaration of domains and codomains for predicates and hierarchies of classes and predicates, RDF defines some special URIs for these concepts in "vocabularies" (see [BG04]). In [MPG07] it is shown that from the rather verbose vocabulary in [BG04] only the predicates rdf:type, rdfs:dom, rdfs:range, rdfs:subClassOf and rdfs:subPropertyOf are needed to achieve the same essential structure. A triple $(s, \text{rdf:type}, c)$ states that one of the classes of node s is represented by the node c. Triples $(p, \text{rdfs:dom}, c)$ and $(p, \text{rdfs:range}, d)$ require for a triple (s, p, o) using the URI p as a predicate that c is among the classes of the subject s and d among the classes of the object o. A triple $(c, \text{rdfs:subClassOf}, d)$ means that each node of class c is also of class d and, finally, a triple $(p, \text{rdfs:subPropertyOf}, q)$ implies that for each statement (s, p, o) also the statement (s, q, o) holds.

This concept, where schema information and typing are represented internally in the graph, is fundamentally different from the classical approach in graph transformation, where schema information is kept in a type graph (possibly with inheritance structure) and typings are given by a morphism from the graph into the type graph. The advantages of the RDF approach lie in its flexibility: Nodes may have several classes or no classes at all instead of exactly one class in the case of typed graphs. This facilitates the representation of information, for which there is no schema available yet, and the application of different schemata to the same instance information. Moreover, schemata and typing of an RDF graph may be easily modified at runtime.

Example 1 (RDF graph) *In Figure 1 a small example of an RDF graph is depicted, where some information about GT-VMT 2008 is represented. URIs are visualised by ellipses, literals by rectangles, and blank nodes by circles. The triples of the RDF graph are given by the arrows in the figure. In the upper row a simple ontology for workshops being held at conferences is introduced. This schema is instantiated in the second row to state that the workshop GT-VMT 2008 is held at the conference ETAPS 2008. We use different namespaces,* ont: *for the ontology and* ex: *for the instance information, to illustrate the possibility that the ontology namespace and its elements are defined elsewhere and just imported into this graph. The third row gives some additional information about the number of accepted contributions for GT-VMT 2008 by a typed literal, and about the venue of ETAPS 2008 by a plain literal, where the language tag states that the Hungarian name of the venue is "Budapest".*

2.2 RDF Graph Homomorphisms

To obtain a category of RDF graphs we define RDF graph homomorphisms, which capture the structural relationships between RDF graphs. Essentially these are subgraph relations modulo

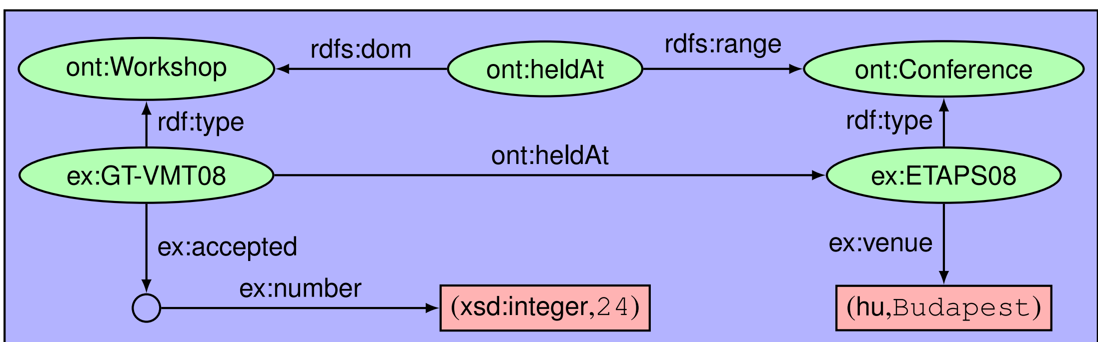


Figure 1: Example of an RDF graph

a translation of the blank nodes, which means that blank nodes can be renamed, identified and included into a larger set of blank nodes before the accordingly translated triples are included into the triples of the codomain graph.

Definition 2 (RDF Graph Homomorphism) Given two RDF graphs G and H, an RDF graph homomorphism $h\colon G \to H$ is given by a translation function $h_{\mathrm{Blank}}\colon G_{\mathrm{Blank}} \to H_{\mathrm{Blank}}$, such that $(h_{\mathrm{Blank}})^{\#}(G_{\mathrm{Triple}}) \subseteq H_{\mathrm{Triple}}$, where the extension $(h_{\mathrm{Blank}})^{\#}$ of h_{Blank} to triples is constructed by $(h_{\mathrm{Blank}})^{\#} := (h_{\mathrm{Blank}} + \mathrm{id}_{\mathsf{URI}} + \mathrm{id}_{\mathsf{Lit}}) \times \mathrm{id}_{\mathsf{URI}} \times (h_{\mathrm{Blank}} + \mathrm{id}_{\mathsf{URI}} + \mathrm{id}_{\mathsf{Lit}})$.

Remark 2 (Relationship to concepts of RDF) *RDF graph homomorphisms are not considered explicitly in the normative RDF specifications. However, a notion of graph equivalence, corresponding to a bijective homomorphism in the sense of the above definition, is defined in [KC04] and [Hay04], where there are some technical differences resulting from the different treatment of blank nodes mentioned in Remark 1.*

Example 2 (RDF graph homomorphism) *In Figure 2 an example is shown, which abstractly illustrates the possibilities of an RDF graph homomorphism. The blank nodes* 1 *and* 2 *of graph G are identified to the blank node* 3 *in graph H, i. e., the homomorphism h is non-injective. In order to satisfy the required triple set inclusion, the triples $(3, p_1, uri_1)$, $(3, p_2, uri_2)$ and (uri_2, p_3, lit_1) are included in H. Moreover, H also has some additional information not in the image of h, namely the blank node* 4 *and the triples $(4, p_2, uri_2)$, $(4, p_4, uri_3)$ and (uri_3, p_3, lit_2), i. e., h is non-surjective.*

Our first result is that RDF graphs and their homomorphisms in fact establish a category. Moreover, arbitrary limits and colimits can be constructed in this category.

Proposition 1 (Category **RDFHom**) *RDF graphs and RDF graph homomorphisms constitute a category, denoted by* **RDFHom**. *This category is complete and cocomplete, i. e., it has limits and colimits over all diagrams.*

Proof sketch. Composition and identities are just the composition and identities of the underlying translations of blank nodes, i. e., composition and identities in the category **Set**, which are

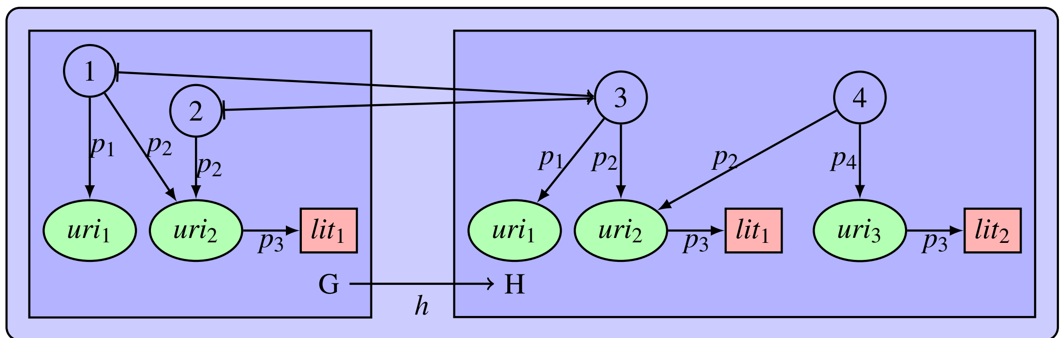


Figure 2: Example of an RDF graph homomorphism

known to satisfy associativity of composition and neutrality of identities. The required triple set inclusions for compositions follow easily from the underlying inclusions, while they are immediately obvious for identical triple sets.

Colimits can be constructed by first taking the colimit C_{Blank} of the blank nodes in **Set**. Then the triple sets in the diagram can be translated via the morphisms into the colimit to become triple sets over C_{Blank}, where we can finally take the set-theoretic union of these translated triple sets to be the triple set C_{Triple} of the colimit graph.

Limits can be constructed similarly by first taking the limit L_{Blank} of the blank nodes in **Set**. The reverse translation of the triple sets in the diagram to triple sets over L_{Blank} can be built by $\{(s,p,o) \mid (h_{\text{Blank}})^{\#}(s,p,o) \in G_{\text{Triple}}\}$ for all graphs G and corresponding functions $h_{\text{Blank}}\colon L_{\text{Blank}} \to G_{\text{Blank}}$. Finally, the triple set L_{Triple} of the limit is obtained by the set-theoretic intersection of the translated triple sets. □

Remark 3 (Interpretation of limits and colimits) *The merge of RDF graphs defined in [Hay04], which takes the union of RDF graphs, while "standardising apart" common blank nodes, is naturally obtained as the coproduct of RDF graphs in our category-theoretical setting. The more complex colimits can be used to construct merges over common blank nodes, which are protected from being standardised apart.*

Intersections of RDF graphs are not treated explicitly in the RDF specifications. The limit constructions in **RDFHom** *can be used to formalise such intersections under common blank nodes.*

2.3 RDF Graph Instantiations

In [Hay04] blank nodes are interpreted as existential variables and an instance of an RDF graph is defined to be a graph, where some of the blank nodes are replaced by concrete URIs or literals. This leads to the following definition of a more general kind of morphism on RDF graphs. Note that the name "instantiation" does not refer to the instantiation of a schema or ontology, but to the instantiation of a blank node.

Definition 3 (RDF Graph Instantiation) Given RDF graphs G and H, an RDF graph instan-

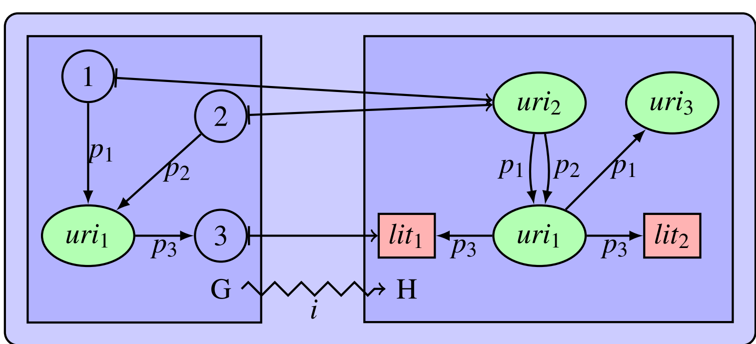


Figure 3: Example of an RDF graph instantiation

tiation $i\colon G \to H$ is given by an assignment function $i_{\text{Blank}}\colon G_{\text{Blank}} \to H_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit}$, such that $(i_{\text{Blank}})^{\#}(G_{\text{Triple}}) \subseteq H_{\text{Triple}}$, where the extension $(i_{\text{Blank}})^{\#}$ of i_{Blank} to triples is constructed by $(i_{\text{Blank}})^{\#} := (i_{\text{Blank}} + \text{id}_{\mathsf{URI}} + \text{id}_{\mathsf{Lit}}) \times \text{id}_{\mathsf{URI}} \times (i_{\text{Blank}} + \text{id}_{\mathsf{URI}} + \text{id}_{\mathsf{Lit}})$.

Remark 4 (Related concepts) *RDF graph instantiations are a combination of instances and subgraph relationships in the sense of [Hay04]. This is particularly interesting, because the Interpolation Lemma of [Hay04] states: "S entails a graph E if and only if a subgraph of S is an instance of E." Using the notion of RDF graph instantiation this is simplified to: "S entails E if and only if there is an instantiation $i\colon E \to S$." The characterisation of entailment by graph homomorphisms is also examined in [Bag05], where RDF graphs are translated into directed, labelled multigraphs and RDF entailment is shown to correspond to their morphisms.*

Instantiations may also be used to formalise queries on an RDF data store. A data store, given by an RDF graph D, is queried using a pattern, given by another RDF graph P. The result of the query should be the set of all possible instantiations $i\colon P \to D$. In [CF07] a related but more complex approach is taken, where SPARQL queries and RDF data sets are translated to conceptual graphs and the results are computed by finding conceptual graph homomorphisms.

Example 3 (RDF graph instantiation) *The RDF graph instantiation $i\colon G \to H$ in Figure 3 identifies the blank nodes 1 and 2 to the URI uri_2 and maps the blank node 3 to the literal lit_1. In terms of entailment this means that the triple (uri_2, p_1, uri_1) in H entails the triple $(1, p_1, uri_1)$ in G, (uri_2, p_2, uri_1) entails $(2, p_2, uri_1)$ and (uri_1, p_3, lit_1) entails $(uri_1, p_3, 3)$. The non-surjectivity of the instantiation i means that not all information in H is used to entail G. In this and following figures we depict instantiations by jagged arrows, while plain homomorphisms are shown as normal arrows.*

In the following proposition we show that RDF graph instantiations give rise to another category, which comprises **RDFHom** as a subcategory. This category is neither complete nor cocomplete. In Subsection 4.1 we will, however, examine circumstances under which pushouts of instantiations exist.

Proposition 2 (Category **RDFInst**) *RDF graphs and RDF graph instantiations constitute a*

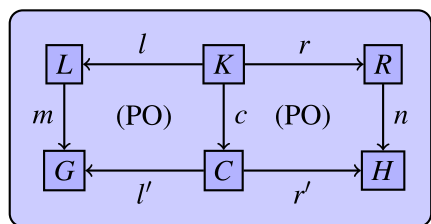


Figure 4: Double pushout transformation

category, denoted by **RDFInst***, with* **RDFHom** $\subseteq$ **RDFInst***. This category does not have limits and colimits in general.*

Proof sketch. The composition of instantiations $i\colon G \to H$ and $j\colon H \to K$ can be obtained by $(j \circ i)_{\text{Blank}} := (j_{\text{Blank}} + \text{id}_{\mathsf{URI}} + \text{id}_{\mathsf{Lit}}) \circ i_{\text{Blank}}$. Identities id_G are given by embeddings $\text{id}_{G,\text{Blank}}$ of the blank node set G_{Blank} into the coproduct $G_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit}$. Associativity of composition and neutrality of identities follow from the corresponding properties of **Set**. The required triple set inclusions are again direct consequences of the underlying inclusions in the composed instantiations. The inclusion of **RDFHom** into **RDFInst** is obvious, since each blank node translation $h_{\text{Blank}}\colon G_{\text{Blank}} \to H_{\text{Blank}}$ is also an assignment $h_{\text{Blank}}\colon G_{\text{Blank}} \to H_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit}$, which simply does not use the possibilities of the extended codomain.

The existence of colimits is impeded by instantiations of the same blank node to different URIs or literals, which cannot be reconciled. The existence of limits is on the other hand inhibited by the fact that unique instantiations into the limit are in general impossible: A blank node without statements can be instantiated to arbitrary URIs or literals. □

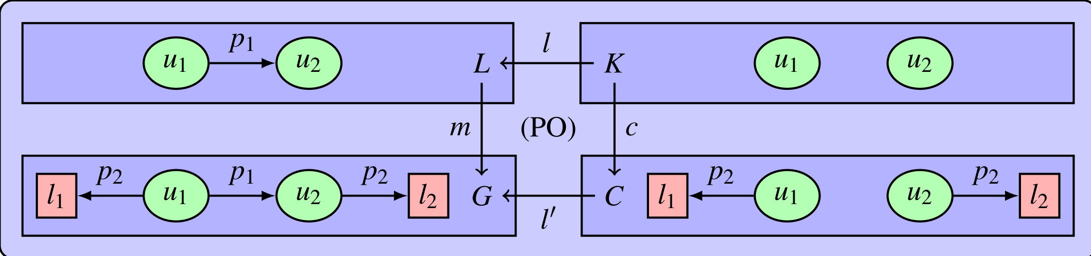
3 Which Transformation Approach?

In this section we will try to find an algebraic graph transformation approach suitable for transforming RDF graphs, which should not only provide a formal basis for the transformations themselves, but also for features like reversibility (to support undoing of editing transformations) and reasoning about dependencies (to achieve structured version histories).

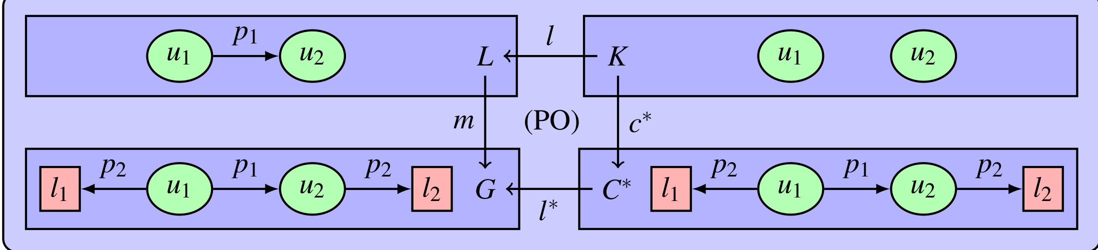
3.1 Double Pushout Approach

The Double Pushout (DPO) approach is one of the predominant approaches in algebraic graph transformation. Its theory has been reformulated in the framework of adhesive HLR categories in [EEPT06]. The main idea of DPO transformations is to split the task of transforming a graph into the deletion of graph elements by a pushout complement and the creation of new elements by a pushout. A DPO transformation is shown in Figure 4, where the rule $L \leftarrow K \to R$ is applied to the graph G via the match $m\colon L \to G$ by first constructing the context C by a pushout complement and then the resulting graph H with comatch $n\colon R \to H$ by a pushout of R and C over K.

DPO transformations are always reversible due to their symmetric structure. If a graph H is obtained from a graph G by the application of a rule $L \leftarrow K \to R$, then a graph isomorphic to G



(a) Pushout complement with deletion



(b) Pushout complement without deletion

Figure 5: Ambiguity of pushout complements in **RDFHom**

may be reconstructed from H by the inverse rule $R \leftarrow K \rightarrow L$. Moreover, there is an extensive theory about the dependencies and conflicts between DPO transformation rules.

Most instantiations of the DPO approach ensure or require unique pushout complements in order to facilitate unique transformation results. This is not possible for RDF graphs as can be seen in the example in Figure 5, where two different pushout complements for the same given situation are shown. The ambiguity results from the fact that the triple (u_1, p_1, u_2), deleted in the context graph K, can be deleted as in C in Figure 5(a) or it can be preserved as in C^* in Figure 5(b). The graph G is a pushout (set-theoretic union of the triples) in both cases.

3.2 Single and Sesqui Pushout Approach

An early alternative approach to algebraic graph transformation is the Single Pushout (SPO) approach studied in [EHK+97]. Instead of splitting deletion and creation into seperate constructions the SPO approach achieves both at the same time by pushouts in a suitable category of partial morphisms. Among the key characteristics of such partial pushouts are the deletion in unknown context (if a node is deleted all edges connected to this node are also deleted even if they are not mentioned in the rule) and the precedence of deletion over preservation (if a deleted node is identified to a preserved one by the match the node is deleted leading to a partial comatch).

A recent proposal is the Sesqui Pushout (SqPO) approach introduced in [CHHK06]. It also features deletion in unknown context but does not allow the identification of deleted and preserved nodes. This is achieved by a split into deletion and creation similar to the DPO approach, where deletion is modelled by final pullback complements instead of pushout complements.

The SPO and SqPO approaches both result in deletion in unknown context, which is not desir-

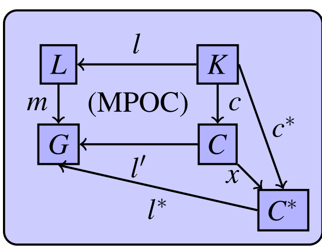


Figure 6: Minimal pushout complement

able in our case, because it impedes the reversibility of transformations. Implicitly deleted edges may not be reconstructed. While this could be resolved by using, e. g., a SqPO approach with an additional condition prohibiting dangling edges, we opt for modifying the DPO approach in the following section.

3.3 The Need for a Modified Approach

Since we want to have unique transformation results, but do not want deletion in unknown context, we cannot use one of the previous approaches unmodified. Returning to Figure 5 we observe that the result we want to achieve in this situation is the deletion of the triple. Hence, our approach, developed in the following section for RDF, is to resolve the ambiguity by canonically selecting the minimal pushout complement (MPOC), i. e., a pushout complement in which as much as possible is deleted.

Definition 4 (Minimal Pushout Complement) Given morphisms $l\colon K \to L$ and $m\colon L \to G$ in an arbitrary category, a minimal pushout complement C of l and m with morphisms $l'\colon C \to G$ and $c\colon K \to C$ is a pushout complement, i. e., G is a pushout of L and C over K, such that for each pushout complement C^* of l and m with morphisms $l^*\colon C^* \to G$ and $c^*\colon K \to C^*$ there is a unique morphism $x\colon C \to C^*$ with $l' = l^* \circ x$ and $c^* = x \circ c$ (cf. Figure 6).

This should preserve most of the well-behavedness of the DPO approach, since the resulting transformation diagram still is a double pushout with an additional side condition, which ensures uniqueness of the transformation result.

Proposition 3 (Uniqueness of Minimal Pushout Complements) *Minimal pushout complements are unique up to isomorphism.*

Proof sketch. Since two minimal pushout complements have unique morphisms in both directions and their compositions have to be the respective identities as unique endomorphisms, they are obviously isomorphic. □

Note that neither the existence of POCs nor the existence of MPOCs for non-unique POCs is guaranteed in general. In the next section we will give a construction for MPOCs in **RDFInst**, which under certain conditions also implies their existence.

4 RDF Graph Transformations and Basic Results

In this section we develop a transformation concept for RDF graphs, which uses minimal pushout complements for the deletion and pushouts for the creation of blank nodes and triple statements.

4.1 MPOC-PO Transformations for RDF

For the creation of elements we need pushouts in **RDFInst**. Sufficient conditions for their existence can be achieved by restricting one of the instantiations in the given span to be an injective homomorphism. This ensures that contradictions can arise neither due to assignments of one blank node to different URIs or literals (the homomorphism cannot assign a blank node to a URI or literal, but only to a blank node) nor due to the identification of several blank nodes assigned to different URIs or literals (injectivity prevents identifications).

Theorem 1 (Pushouts in **RDFInst**) *Given an injective RDF graph homomorphism $r\colon K \to R$ and an RDF graph instantiation $c\colon K \to C$, a pushout H with an injective RDF graph homomorphism $r'\colon C \to H$ and an RDF graph instantiation $n\colon R \to H$ can be constructed by*

- *the blank node set $H_{\text{Blank}} := C_{\text{Blank}} + (R_{\text{Blank}} \setminus r_{\text{Blank}}(K_{\text{Blank}}))$ with the injection r'_{Blank} of C_{Blank} into the coproduct and the assignment n_{Blank}, which acts on nodes from K_{Blank} as c_{Blank} does and on the new nodes as an injection into the coproduct, and*
- *the triple set $H_{\text{Triple}} := (r'_{\text{Blank}})^{\#}(C_{\text{Triple}}) \cup (n_{\text{Blank}})^{\#}(R_{\text{Triple}})$.*

Proof sketch. The asymmetric construction of H_{Blank} ensures that the assignment n_{Blank} is well-defined, because the inclusion of C_{Blank} enables its acting like c_{Blank} on the preserved nodes, while the addition of the new elements of R_{Blank} facilitates their injective mapping. The union construction of the triple set guarantees the satisfaction of the corresponding triple set inclusions, while the commutativity $n_{\text{Blank}} \circ r_{\text{Blank}} = r'_{\text{Blank}} \circ c_{\text{Blank}}$ also holds by construction.

For each other graph H^* with instantiations $n^*\colon R \to H^*$ and $r^*\colon C \to H^*$, such that $n^* \circ r = r^* \circ c$ an assignment $x_{\text{Blank}}\colon H_{\text{Blank}} \to H^*_{\text{Blank}} + \mathsf{URI} + \mathsf{Lit}$ is uniquely induced by the requirement that it acts on blank nodes from C as r^* does ($x \circ r' = r^*$) and on blank nodes from R as n^* does ($x \circ n = n^*$). □

For the deletion we will use the concept of MPOC introduced in Definition 4. In the following theorem we give sufficient conditions and a construction for MPOCs in **RDFInst**. Two of these conditions, namely the identification and the dangling condition, are well-known from the ordinary DPO approach, while the third ensures that deleted blank nodes are not assigned to URIs or literals, which cannot be deleted.

Theorem 2 (Minimal Pushout Complements in **RDFInst**) *Given an injective RDF graph homomorphism $l\colon K \to L$ and an RDF graph instantiation $m\colon L \to G$, such that the deleted blank nodes in $L_{\text{Blank}} \setminus l_{\text{Blank}}(K_{\text{Blank}})$*

- *are not identified by m_{Blank} to preserved blank nodes from $l_{\text{Blank}}(K_{\text{Blank}})$ (identification condition),*

- *are not assigned by* m_{Blank} *to URIs or literals, and*
- *are not assigned by* m_{Blank} *to blank nodes, which are used in preserved triples in* $G_{\text{Triple}} \setminus (m_{\text{Blank}})^{\#}(L_{\text{Triple}})$ *(dangling condition),*

an MPOC C *with an injective RDF graph homomorphism* $l' \colon C \to G$ *and an RDF graph instantiation* $c \colon K \to C$ *can be constructed by*

- *the blank node set* $C_{\text{Blank}} := G_{\text{Blank}} \setminus m_{\text{Blank}}(L_{\text{Blank}} \setminus l_{\text{Blank}}(K_{\text{Blank}}))$ *with the inclusion* l'_{Blank} *of this set into* G_{Blank} *and the assignment* c_{Blank}*, which behaves like* $m_{\text{Blank}} \circ l_{\text{Blank}}$*, and*
- *the triple set* $C_{\text{Triple}} := G_{\text{Triple}} \setminus (m_{\text{Blank}})^{\#}(L_{\text{Triple}} \setminus (l_{\text{Blank}})^{\#}(K_{\text{Triple}}))$.

Proof sketch. The assignment c_{Blank} is well-defined and the triple inclusion of $(c_{\text{Blank}})^{\#}(K_{\text{Triple}})$ into C_{Triple} is satisfied, because the blank nodes in the range of $m_{\text{Blank}} \circ l_{\text{Blank}}$ and the triples from K_{Triple} are explicitly not removed in C, while the commutativity $m_{\text{Blank}} \circ l_{\text{Blank}} = l'_{\text{Blank}} \circ c_{\text{Blank}}$ also holds by this construction.

The blank node set G_{Blank} may be reconstructed from L_{Blank} and C_{Blank} by the pushout construction in Theorem 1, since the removed triples $m_{\text{Blank}}(L_{\text{Blank}} \setminus l_{\text{Blank}}(K_{\text{Blank}}))$ are exactly the ones that are added by the pushout. Moreover, the pushout construction also recovers the triple set G_{Triple}. Hence, the construction in fact leads to a pushout complement.

For each other pushout complement C^* with instantiations $c^* \colon K \to C^*$ and $l^* \colon C^* \to G$ the blank node set C^*_{Blank} has to be isomorphic to C_{Blank}, because otherwise the pushout construction would also result in a non-isomorphic blank node set. The blank node bijection $x_{\text{Blank}} \colon C_{\text{Blank}} \to C^*_{\text{Blank}}$ is then already uniquely determined by the requirement $l'_{\text{Blank}} = l^*_{\text{Blank}} \circ x_{\text{Blank}}$. The triple set $(x_{\text{Blank}})^{\#}(C_{\text{Triple}})$ has to be included in C^*_{Triple}, since C^* has to contain all triples, which cannot be reconstructed from L_{Triple} by the pushout, and all triples contained in K_{Triple} in order to be a pushout complement. □

Remark 5 (Pushouts and MPOCs of proper instantiations) *The conditions given in Theorem 1 and Theorem 2 are sufficient, but not necessary, respectively. In fact, it is possible to construct pushouts of (proper) RDF graph instantiations if the assignments of blank nodes do not contradict. For our application to RDF graph transformation the given constructions over homomorphisms are, however, general enough, since we do not want to instantiate blank nodes in the rule but only in the match.*

We now have both constructions that are necessary to define RDF graph transformations as MPOC-PO transformations. This means that the deletion in the first square of Figure 7 is obtained by an MPOC, while the creation in the second square is done by a pushout.

Definition 5 (RDF Graph Transformation) An RDF graph transformation rule is given by a span $L \leftarrow K \to R$ of injective RDF graph homomorphisms. An application of this rule to an RDF graph G via an RDF graph instantiation $m \colon L \to G$ is given by the diagram in Figure 7 resulting in the RDF graph H with the RDF graph instantiation $n \colon R \to H$.

Example 4 (RDF graph transformation) *In Figure 8 an example of an RDF graph transformation is depicted, where the rule replaces a sequential occurrence of predicates* p_1 *and* p_2 *by a*

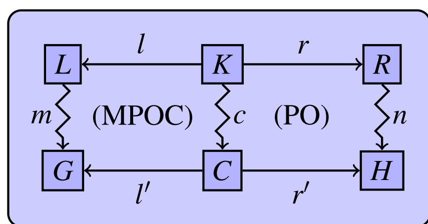


Figure 7: RDF graph transformation

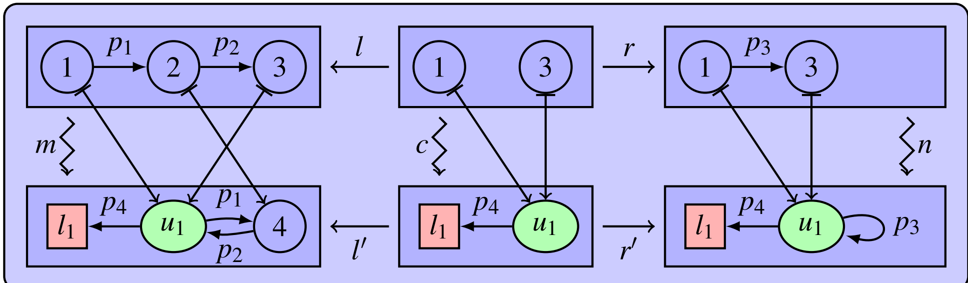


Figure 8: Example of an RDF graph transformation

single occurrence of p_3*, while deleting the intermediate blank node. Such a rule is not typical* for reasoning *in RDF, since inference only yields additional triples, but does not delete anything. We claim, however, that such deleting transformations are useful for* editing *RDF data, e. g., if the predicates* p_1 *and* p_2 *are deprecated and shall be replaced by* p_3*.*

The following corollary summarises the existence and uniqueness properties of RDF graph transformations.

Corollary 1 (Applicability and Uniqueness of RDF Transformations) *An RDF graph transformation rule* $L \leftarrow K \rightarrow R$ *is applicable via a match instantiation* $m\colon L \rightarrow G$ *if* m *does not identify deleted and non-deleted blank nodes, does not assign deleted blank nodes to URIs or literals and does not assign deleted blank nodes to blank nodes occuring in triples not in the range of* m*. In this case the resulting RDF graph* H *with comatch* $n\colon R \rightarrow H$ *is unique up to isomorphism.*

Proof. This corollary follows directly from Theorem 2 and Theorem 1 regarding the existence of the result and from Proposition 3 and the uniqueness of pushouts in any category regarding its uniqueness. □

4.2 Reversible Transformations

While DPO transformations are always reversible due to their symmetric structure, MPOC-PO transformations are only reversible if the creation pushout is also an MPOC. In RDF graph transformations this is the case if none of the additional triples in R is already present in C.

Theorem 3 (Reversibility of RDF Graph Transformations) *Given the RDF graph transformation in Figure 7, the application of the inverse rule* $R \leftarrow K \rightarrow L$ *to the graph* H *via the match* $n\colon R \rightarrow H$ *is possible and leads to a graph* G' *isomorphic to* G *provided that* $(r'_{\text{Blank}})^{\#}(C_{\text{Triple}}) \cap (n_{\text{Blank}})^{\#}(R_{\text{Triple}}) = (n_{\text{Blank}} \circ r_{\text{Blank}})^{\#}(K_{\text{Triple}}) = (r'_{\text{Blank}} \circ c_{\text{Blank}})^{\#}(K_{\text{Triple}})$.

Proof sketch. The given condition ensures that the right square is not only a pushout, but that C is also an MPOC in this square, since all common triples of C_{Triple} and R_{Triple} are also in the interface K_{Triple} and will therefore not be removed by the MPOC construction. □

In order to constrain RDF graph transformations to reversible cases negative application conditions could be used, which disallow all triples added by the rule from being already present in the host graph. To achieve the same expressibility additional rules could be added, which have these triples in their left-hand sides and just perform the other changes of the rule. A complete discussion is, however, outside the scope of this paper.

5 Summary and Future Work

In this contribution we have formalised RDF in a category theoretical framework, provided a transformation approach for RDF graphs and shown under which circumstances transformations are applicable and reversible. These results can provide a useful basis for the rule-based modification and creation of RDF data.

Future work should enhance the theoretical results for RDF graph transformations by concepts like negative application conditions and analysis techniques for dependencies and conflicts. Moreover, the relationships to the formal semantics of RDF and its inference mechanisms should be examined.

Bibliography

[Bag05] J.-F. Baget. RDF Entailment as a Graph Homomorphism. In Gil et al. (eds.), *The Semantic Web – ISWC 2005*. LNCS 3729, pp. 82–96. Springer, 2005.
doi:10.1007/11574620_9
ftp://ftp.inrialpes.fr/pub/exmo/publications/baget2005a.pdf

[BFM98] T. Berners-Lee, R. T. Fielding, L. Masinter. RFC 2396 – Uniform Resource Identifiers (URI): Generic Syntax. Internet Engineering Task Force (IETF), Aug. 1998.
http://tools.ietf.org/html/rfc2396

[BG04] D. Brickley, R. V. Guha (eds.). *RDF Vocabulary Description Language 1.0: RDF Schema*. World Wide Web Consortium (W3C), Feb. 2004.
http://www.w3.org/TR/2004/REC-rdf-schema-20040210/

[BHLT06] T. Bray, D. Hollander, A. Layman, R. Tobin (eds.). *Namespaces in XML 1.0 (Second Edition)*. World Wide Web Consortium (W3C), Aug. 2006.
http://www.w3.org/TR/2006/REC-xml-names-20060816/

[BM04] P. V. Biron, A. Malhotra (eds.). *XML Schema Part 2: Datatypes (Second Edition)*. World Wide Web Consortium (W3C), Oct. 2004.
http://www.w3.org/TR/2004/REC-xmlschema-2-20041028/

[CF07] O. Corby, C. Faron-Zucker. Implementation of SPARQL Query Language Based on Graph Homomorphism. In Priss et al. (eds.), *Conceptual Structures: Knowledge Architectures for Smart Applications. Proc. ICCS 2007*. LNCS/LNAI 4604, pp. 472–475. Springer, 2007.
doi:10.1007/978-3-540-73681-3_37

[CHHK06] A. Corradini, T. Heindel, F. Hermann, B. König. Sesqui-Pushout Rewriting. In Corradini et al. (eds.), *Graph Transformations. Proc. ICGT 2006*. LNCS 4178, pp. 30–45. Springer, 2006.
doi:10.1007/11841883_4

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

[EHK+97] H. Ehrig, R. Heckel, M. Korff, 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 Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. 1: Foundations*. Chapter 4. World Scientific, 1997.

[Hay04] P. Hayes (ed.). *RDF Semantics*. World Wide Web Consortium (W3C), Feb. 2004.
http://www.w3.org/TR/2004/REC-rdf-mt-20040210/

[KC04] G. Klyne, J. J. Carroll (eds.). *Resource Description Framework (RDF): Concepts and Abstract Syntax*. World Wide Web Consortium (W3C), Feb. 2004.
http://www.w3.org/TR/2004/REC-rdf-concepts-20040210/

[MM04] F. Manola, E. Miller (eds.). *RDF Primer*. World Wide Web Consortium (W3C), Feb. 2004.
http://www.w3.org/TR/2004/REC-rdf-primer-20040210/

[MPG07] S. Muñoz, J. Pérez, C. Gutierrez. Minimal Deductive Systems for RDF. In Franconi et al. (eds.), *The Semantic Web: Research and Applications. Proc. ESWC 2007*. LNCS 4519, pp. 53–67. Springer, 2007.
doi:10.1007/978-3-540-72667-8_6
http://www2.ing.puc.cl/~jperez/papers/minimal-rdf-camera-ready-ext.pdf

[PS07] E. Prud'hommeaux, A. Seaborne (eds.). *SPARQL Query Language for RDF*. World Wide Web Consortium (W3C), Nov. 2007.
http://www.w3.org/TR/2007/PR-rdf-sparql-query-20071112/

[Uni07] The Unicode Consortium. The Unicode Standard, Version 5.0.0. 2007.
http://www.unicode.org/versions/Unicode5.0.0/

Controlling resource access in Directed Bigraphs

Davide Grohmann[1], **Marino Miculan**[2]

[1] grohmann@dimi.uniud.it, [2] miculan@dimi.uniud.it
Department of Mathematics and Computer Science, University of Udine, Italy

Abstract: We study *directed bigraph with negative ports*, a bigraphical framework for representing models for distributed, concurrent and ubiquitous computing. With respect to previous versions, we add the possibility that components may *govern* the access to resources, like (web) servers control requests from clients. This framework encompasses many common computational aspects, such as name or channel creation, references, client/server connections, localities, etc, still allowing to derive systematically labelled transition systems whose bisimilarities are congruences.

In order to illustrate the expressivity of this framework, we give the encodings of client/server communications through firewalls, of (compositional) Petri nets and of chemical reactions.

Keywords: Bigraphs, reactive systems, Petri nets, graph-based approaches to service-oriented applications.

1 Introduction

Bigraphical reactive systems (BRSs) are an emerging graphical framework proposed by Milner and others [Mil01, Mil06] as a unifying theory of process models for distributed, concurrent and ubiquitous computing. A bigraphical reactive system consists of a category of *bigraphs* (usually generated over a given *signature of controls*) and a set of *reaction rules*. Bigraphs can be seen as representations of the possible configurations of the system, and the reaction rules specify how these configuration can evolve, i.e., the reaction relation between bigraphs. Often, bigraphs represent terms up-to structural congruence and reaction rules represent term rewrite rules.

Many process calculi have successfully represented as bigraphical reactive systems: λ-calculus [Mil07], CCS [Mil06], π-calculus [BS06, JM04], Mobile Ambients [Jen08], Homer [BH06], Fusion [GM07c], Petri nets [LM06], and context-aware systems [BDE$^+$06]. The advantage of using bigraphical reactive systems is that they provide powerful general results for deriving a labelled transition system *automatically* from the reaction rules, via the so-called *IPO construction*. Notably, the bisimulation on this transition system is always a congruence; thus, bigraphical reactive systems provide general tools for compositional reasoning about concurrent, distributed systems.

Bigraphs are the key structures supporting these results. A bigraph is a set of nodes (the *controls*), endowed with two independent graph structures, the *place graph* and the *link graph* (Figure 1). The place graph is a tree over the nodes, representing the spatial arrangement (i.e., nesting) of the various components of the system. The link graph represents the communication connections between the components, possibly traversing the place structure. A bigraph may be "not ground", in the sense that it may have one or more "holes", or *sites* (the gray boxes) to

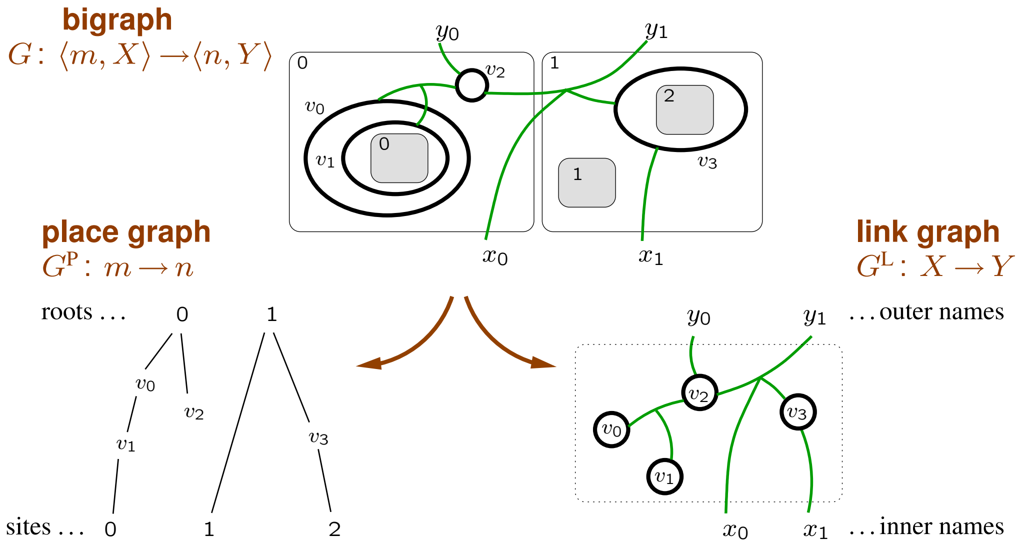


Figure 1: Example of pure bigraph (from [Mil06]).

be instantiated; these holes are specific leaves of the place graph, where other bigraphs can be grafted, respecting the connection links. This operation gives rise to a notion of composition between bigraphs, and hence to a categorical structure.

In Milner's "pure bigraphs" [Mil06], connections are represented by hyper-arcs between nodes (Figure 1). This model has been successfully used to represent many calculi, such as CCS, and (with a small variant) λ-calculus, π-calculus. Nevertheless, other calculi, such as Fusion [PV98], seem to escape this framework. Aiming to a more expressive framework, in previous work [GM07b, GM07c], we have introduced *directed bigraphs*. Pure and directed bigraphs differ only on the link structure: in the directed variant, we distinguish "edges" from "connections". Intuitively, edges represent *(delocalized) resources*, or *knowledge tokens*, which can be *accessed* by controls. Arcs are arrows from ports of controls to edges (possibly through names on the interfaces of bigraphs); moreover, in the version considered in the present paper, we allow arcs to point to other control's ports (Figure 2). Outward ports on a control represent the capability of the control to access to (external) resources; insted, inward ports represent the capability of the control to "stop" or "govern" other node's requests. The presence of both kinds of capabilities is common in distributed scenarios, such as client/server communications, firewalls, web services etc; for instance a system may ask to access to some data, but this attempt may be blocked, checked and possibly redirected by a guarding mechanism. Moreover, controls with inward ports can represent *localized* resources, that is, resources with a position within the place hierarchy; this cannot be represented easily by edges, which do not appear in the place graph.

Notably, these extended have RPO and IPO constructions, there is a notion of normal form, and a sound a complete axiomatization can be given. Therefore, these bigraphs can be conveniently used for building wide reaction systems from which we can synthesize labelled transition systems via the IPO construction, and whose bisimilarity is still a congruence.

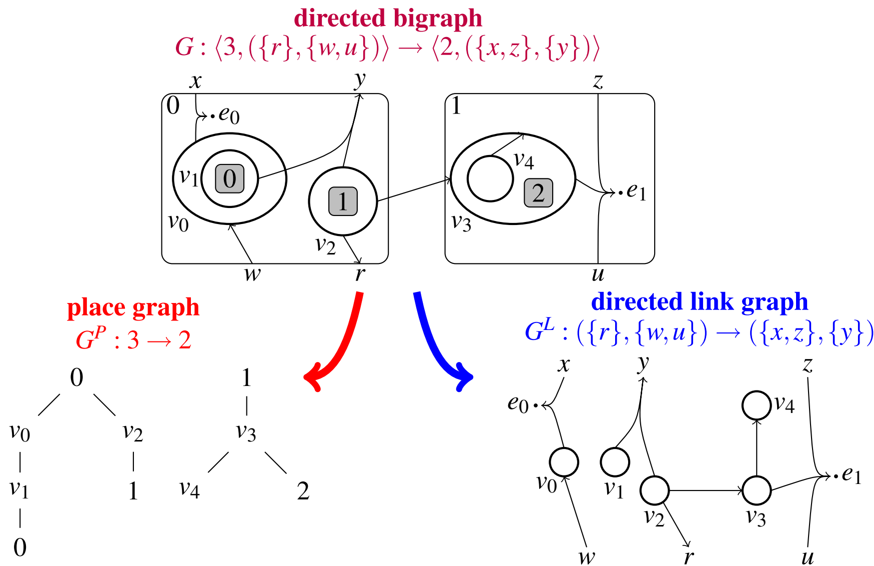


Figure 2: An example of directed bigraph, with negative ports.

Due to lack of space, in this paper we can only skim over these theoretical results; we prefer to focus on some important applications of this framework. In Section 2 we give the basic definitions about directed bigraphs. In Sections 3 we present the elementary bigraphs, which are enough to generate all possible bigraphs. Section 4 is devoted to example applications, highlighting the expressive power of this framework: we show how distributed services and protocols can be represented, by describing a three-tier architecture with a firewall; we will present an encoding of Petri nets, and finally we apply this framework to the representation of chemical reactions. Conclusions and direction for future work are in Section 6. Constructions of RPOs and IPOs, a notion of normal form and a complete axiomatization appear in [GM08].

2 Directed bigraphs over polarized signatures

In this section we introduce directed bigraphs, with inward ("negative") ports on controls, extending [GM07b]. Following previous developments about pure and directed bigraphs, we work in *supported monoidal precategories*; we refer to [JM04, §3] for an introduction.

A *polarized signature* is a signature of controls, which may have two kind of ports: *negative* and *positive*. Let $\mathscr{K}$ be a polarized signature; we denote with $ar^n, ar^p : \mathscr{K} \to \mathbb{N}$ the arity functions of the negative and positive ports, respectively. Thus, for $k \in \mathscr{K}$, the arity function is $ar(k) \triangleq (ar^n(k), ar^p(k))$. A control k is *positive* if it has only positive ports (i.e., $ar^n(k) = 0$); it is *negative* if it has only negative ports (i.e., $ar^p(k) = 0$).

Definition 1 A *polarized interface* X is a pair of sets of names $X = (X^-, X^+)$; the two components are called *downward* and *upward* interfaces, respectively.

A *directed link graph* $A : X \to Y$ is $A = (V, E, ctrl, link)$ where X, Y are the *inner* and *outer* interfaces, V is the set of *nodes*, E is the set of *edges*, $ctrl : V \to \mathcal{K}$ is the *control map*, and $link : \mathsf{Pnt}(A) \to \mathsf{Lnk}(A)$ is the *link map*, where *ports*, *points* and *links* of A are defined as follows:

$$\mathsf{Prt}^n(A) \triangleq \sum_{v \in V} ar^n(ctrl(v)) \quad \mathsf{Prt}^p(A) \triangleq \sum_{v \in V} ar^p(ctrl(v)) \quad \mathsf{Prt}(A) \triangleq \mathsf{Prt}^n(A) \cup \mathsf{Prt}^p(A)$$

$$\mathsf{Pnt}(A) \triangleq X^+ \uplus Y^- \uplus \mathsf{Prt}^p(A) \qquad \mathsf{Lnk}(A) \triangleq X^- \uplus Y^+ \uplus \mathsf{Prt}^n(A) \uplus E$$

The link map cannot connect downward and upward names of the same interface, i.e., the following condition must hold: $(link(X^+) \cap X^-) \cup (link(Y^-) \cap Y^+) = \emptyset$; moreover the link map cannot connect positive and negative ports of the same node.

Directed link graphs are graphically depicted much like ordinary link graphs, with the difference that edges are explicit objects, and not hyper-arcs connecting points and names; points and names are associated to links (that is edges or negative ports) or other names by (simple, non hyper) directed arcs. An example are given in Figure 2. This notation aims to make explicit the "resource request flow": positive ports and names in the interfaces can be associated either to internal or to external resources. In the first case, positive ports and names are connected to an edge or a negative port; these names are "inward" because they offer to the context the access to an internal resource. In the second case, the positive ports and names are connected to an "outward" name, which is waiting to be plugged by the context into a resource.

In the following, by "signature", "interface" and "link graphs" we will intend "polarized signature", "polarized interface" and "directed link graphs" respectively, unless otherwise noted.

Definition 2 The precategory of *directed link graphs* has polarized interfaces as objects, and directed link graphs as morphisms.

Given two directed link graphs $A_i = (V_i, E_i, ctrl_i, link_i) : X_i \to X_{i+1}$ $(i = 0, 1)$, the composition $A_1 \circ A_0 : X_0 \to X_2$ is defined when the two link graphs have disjoint nodes and edges. In this case, $A_1 \circ A_0 \triangleq (V, E, ctrl, link)$, where $V \triangleq V_0 \uplus V_1$, $ctrl \triangleq ctrl_0 \uplus ctrl_1$, $E \triangleq E_0 \uplus E_1$ and

$$link : X_0^+ \uplus X_2^- \uplus \mathsf{Prt}^p(A_0) \uplus \mathsf{Prt}^p(A_1) \to X_0^- \uplus X_2^+ \uplus E \uplus \mathsf{Prt}^n(A_0) \uplus \mathsf{Prt}^n(A_1)$$

is defined as follows:

$$link(p) \triangleq \begin{cases} link_0(p) & \text{if } p \in X_0^+ \uplus \mathsf{Prt}^p(A_0) \text{ and } link_0(p) \in X_0^- \uplus E_0 \uplus \mathsf{Prt}^n(A_0) \\ link_1(x) & \text{if } p \in X_0^+ \uplus \mathsf{Prt}^p(A_0) \text{ and } link_0(p) = x \in X_1^+ \\ link_1(p) & \text{if } p \in X_2^- \uplus \mathsf{Prt}^p(A_1) \text{ and } link_1(p) \in X_2^+ \uplus E_1 \uplus \mathsf{Prt}^n(A_1) \\ link_0(x) & \text{if } p \in X_2^- \uplus \mathsf{Prt}^p(A_1) \text{ and } link_1(p) = x \in X_1^-. \end{cases}$$

The identity link graph of X is $id_X \triangleq (\emptyset, \emptyset, \emptyset_{\mathcal{K}}, id_{X^-} \cup id_{X^+}) : X \to X$.

It is easy to check that composition is associative, and that given a link graph $A : X \to Y$, the compositions $A \circ id_X$ and $id_Y \circ A$ are defined and equal to A.

Definition 1 forbids connections between names of the same interface in order to avoid undefined link maps after compositions. Similarly, links between ports on the same node are forbidden, because these graphs cannot be obtained by composing an "unlinked" node and a context.

It is easy to see that the precategory 'DLG is self-dual, that is $'\mathrm{DLG} \cong {'\mathrm{DLG}}^{op}$.

The notions of openness, closeness, leanness, etc. defined in [GM07b] can be easily extended to the new framework, considering negative ports as a new kind of resources. Moreover, the definition of tensor product can be derived extending to negative ports the one given in [GM07b],

Finally, we can define the *(extended) directed bigraphs* as the composition of standard place graphs (see [JM04, §7] for definitions) and directed link graphs.

Definition 3 A *directed bigraph* with signature $\mathscr{K}$ is $G = (V, E, ctrl, prnt, link) : I \to J$, where $I = \langle m, X\rangle$ and $J = \langle n, Y\rangle$ are its inner and outer interfaces respectively. An interface is composed by a *width* (a finite ordinal) and by a pair of finite sets of names. V and E are the sets of nodes and edges respectively, and *prnt*, *ctrl* and *link* are the parent, control and link maps, such that $G^P \triangleq (V, ctrl, prnt) : m \to n$ is a place graph and $G^L \triangleq (V, E, ctrl, link) : X \to Y$ is a directed link graph.

We denote G as combination of G^P and G^L by $G = \langle G^P, G^L\rangle$. In this notation, a place graph and a (directed) link graph can be put together iff they have the same sets of nodes.

Definition 4 The precategory 'DBIG of directed bigraph with signature $\mathscr{K}$ has interfaces $I = \langle m, X\rangle$ as objects and directed bigraphs $G = \langle G^P, G^L\rangle : I \to J$ as morphisms. If $H : J \to K$ is another directed bigraph with sets of nodes and edges disjoint from the respectively ones of G, then their composition is defined by composing their components, i.e.:

$$H \circ G \triangleq \langle H^P \circ G^P, H^L \circ G^L\rangle : I \to K.$$

The identity directed bigraph of $I = \langle m, X\rangle$ is $\langle id_m, id_X\rangle : I \to I$.

Analogously, the tensor product of two bigraphs can be defined tensoring their components.

It is easy to check that for every signature $\mathscr{K}$, the precategory 'DBIG is wide monoidal; the origin is $\varepsilon = \langle 0, (\emptyset, \emptyset)\rangle$ and the interface $\langle n, X\rangle$ has width n. Hence, 'DBIG can be used for applying the theory of *wide reaction systems* and *wide transition systems* as developed by Jensen and Milner; [JM04, §4, §5]. To this end, we need to show that 'DBIG has RPOs and IPOs. Since place graphs are as usual, it suffices to show that directed link graphs have RPOs and IPOs.

Theorem 1 *If a pair $\vec{A}$ of link graphs has a bound $\vec{D}$, there exists an RPO $(\vec{B}, B)$ for $\vec{A}$ to $\vec{D}$.*

As a consequence, 'DLG has IPOs too. See [GM08] for the constructions for RPOs and IPOs in directed bigraphs with negative ports, extending the construction given in [GM07b].

Actually, often we do not want to distinguish bigraphs differing only on the identity of nodes and edges. To this end, we introduce the category DBIG of *abstract directed bigraphs*, which is constructed from 'DBIG forgetting the identity of nodes and edges and any idle edge. More precisely, abstract bigraphs are bigraphs taken up-to an equivalence $\eqsim$ (see [JM04] for details).

Definition 5 Two concrete directed bigraphs G and H are *lean-support equivalent*, written $G \eqsim H$, if there exists an iso between their nodes and edges sets after removing any idle edges.

The category DBIG of abstract directed bigraphs has the same objects as 'DBIG, and its arrows are lean-support equivalence classes of directed bigraphs.

3 Algebra and Axiomatization

As for directed bigraphs, also in the case of polarized signature it is possible to give a sound and complete axiomatization. In this section, due to lack of space, we describe only the main classes of bigraphs and the elementary bigraphs which can generate all bigraphs according to a well-defined normal form. Due to lack of space, the definition of normal form and the normalization theorem is given in [GM08]. We refer the reader to [GM07a] for a complete presentation of the notation used here.

First, we introduce two distinct and complementary subclasses of bigraphs: *wirings* and *discrete bigraphs*. that are strongly used in defining the normal form and the axiomatization.

Definition 6 A *wiring* is a bigraph whose interfaces have zero width (and hence has no nodes). The wirings ω are generated by the composition or tensor product of three elements: substitutions $\sigma : (\emptyset, X^+) \to (\emptyset, Y^+)$, fusions $\delta : (Y^-, \emptyset) \to (X^-, \emptyset)$, and closures $\rotatebox{90}{$\bowtie$}^x_y : (\emptyset, y) \to (x, \emptyset)$.

Definition 7 An interface is *prime* if its width is 1. Often we abbreviate a prime interface $\langle 1, (X^-, X^+)\rangle$ with $\langle (X^-, X^+)\rangle$, in particular $1 = \langle (\emptyset, \emptyset)\rangle$. A prime bigraph $P : \langle m, (Y^-, Y^+)\rangle \to \langle (X^-, X^+)\rangle$ has a prime outer interface and the names in Y^+, X^- are linked to negative ports of P.

An important prime bigraph is $merge_m : m \to 1$, it has no nodes and maps m sites to one root.

Definition 8 A bigraph is *discrete* if it has no edges and every open link has exactly one point.

The discreteness is well-behaved, and preserved by composition and tensor. It is easy to see that discrete bigraphs form a monoidal sub-precategory of $'$DBIG.

Definition 9 Let K be any non atomic control with arity (k^-, k^+), let $\vec{x}^-, \vec{x}^+$ be two sequences of distinct names, and let $\vec{Y}^+, \vec{Y}^-$ be two sequences of (possibly empty) sets of distinct names, such that: $|\vec{x}^-| + |\vec{x}^+| = k^+$ and $|\vec{Y}^-| = |\vec{Y}^+| = k^-$. For a K-node v, we define the discrete *ion*

$$K(v,l) : \langle (\vec{x}^-, \vec{Y}^+)\rangle \to \langle (\vec{Y}^-, \vec{x}^+)\rangle$$

as the bigraph with exactly a node v and l is a pair of maps: an iso map $l^p : \vec{x}^- \cup \vec{x}^+ \to \mathsf{Prt}^p(v)$ describing the linking among positive ports and names in $\vec{x}^-$ or $\vec{x}^+$, and another iso map $l^n : \vec{Y}^- \cup \vec{Y}^+ \to \mathsf{Prt}^n(v)$ describing the linking among negative ports and sets of upward inner names (in $\vec{Y}^+$) and sets of downward outer names (in $\vec{Y}^-$). We omit v when it can be understood.

For a prime discrete bigraph P with outer names in (Z^-, Z^+), we define a discrete *molecule* as:

$$(K(l) \otimes id_{(Z^- \setminus \vec{x}^-, Z^+ \setminus \vec{Y}^+)}) \circ P.$$

If K is atomic, we define the discrete *atom*, as an ion without sites:

$$K(l) : (\vec{x}^-, \vec{Y}^+) \to \langle (\vec{Y}^-, \vec{x}^+)\rangle.$$

An arbitrary (non-discrete) ion, molecule or atom is formed by the composition of $\omega \otimes id_1$ with a discrete one. Often we omit $\ldots \otimes id_I$ in the compositions, when there is no ambiguity.

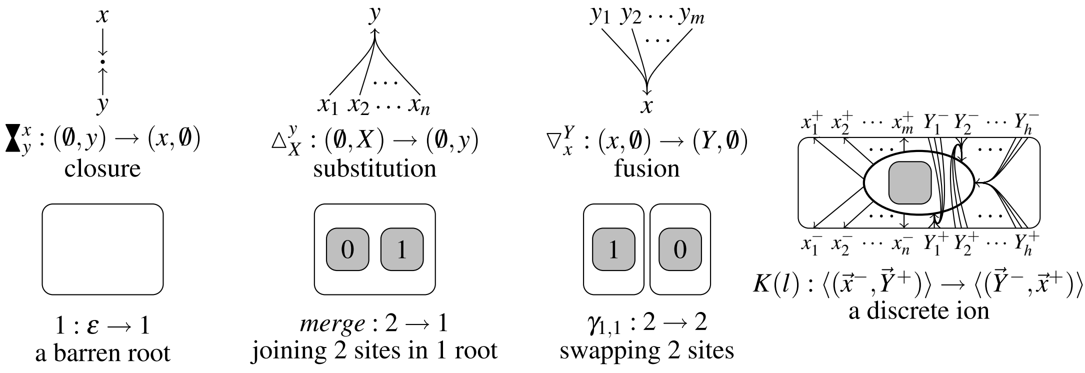


Figure 3: Elementary Bigraphs over polarized signatures.

Figure 3 shows the *algebraic signature*, that is a set of elementary bigraphs able to define any other bigraph using composition and tensor product. The various *sharing products* are the intuitive generalization of the ones defined in [GM07a]; see [GM08] for a detailed description.

4 Applications

4.1 Three-tier interaction with access control

As mentioned before, directed bigraphs over polarized signatures allow to represent resource access control, by means of negative ports. This is particularly useful for representing access policies between systems, possibly in different locations; the edges can represent access tokens (or keys), which are global (although known possibly to only some controls). An example and quite common scenario is a client-server connection, where the access to the server is subject to authentication; after the request has been accepted, the server can route it to a back-end service (e.g., a DBMS); see Figure 4. The security policy is implemented by the firewall control, which allows a query to reach the server only if the client knows the correct key (rule AUTH). The server routes the query to the correct back-end service using rules like ROUTE; finally the back-end service provides the data (rule GET). An example computation is shown in Figure 5.

4.2 Compositional Petri Nets

In this section we recall briefly what a Petri net is and we give an encoding of these nets as directed bigraphs; to this end it is preferable to work with sorted links, as in [LM06]. Notice that this encoding yields naturally a notion of composition between Petri nets.

Definition 10 A *place transition net* (*P/T net*) is a 5-tuple (P,T,F,M_i) $(P \cap T = \emptyset)$, where:

- P is the set of *places; T* is the set of *transitions;*

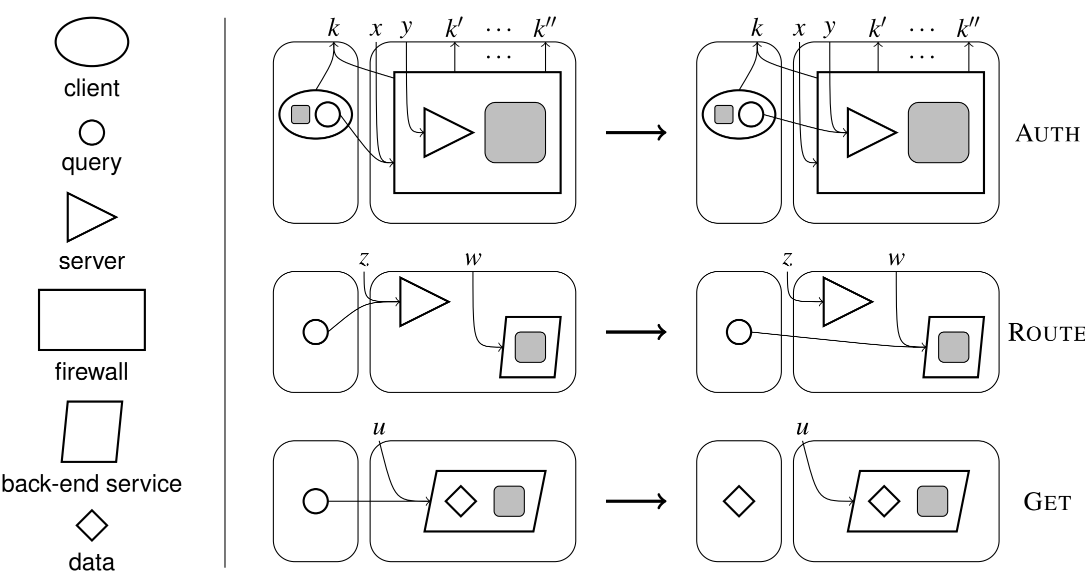


Figure 4: Signatures and rules for three-tier architecture services through a firewall.

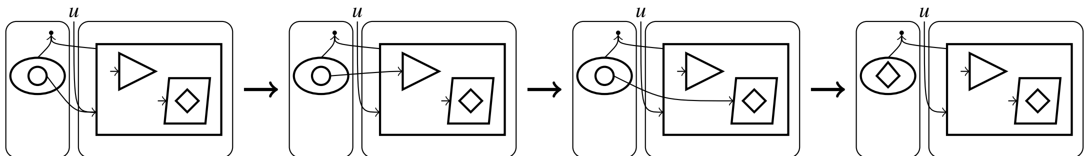


Figure 5: An example of client-server interaction through a firewall.

- F is the multiset of *arcs,* linking places to transitions and vice versa: $F \triangleq \langle (P \times T) \cup (T \times P), f : (P \times T) \cup (T \times P) \rightarrow \mathbb{N} \rangle$, with the constrain $\forall t \in T.\ \exists p,q \in P.\ (p,t) \in F \wedge (t,q) \in F$;
- $M : P \rightarrow \mathbb{N}$ is a *marking,* giving to each place a number of tokens, a place p is *marked* by M if $M(p) > 0$ and *unmarked* if $M(p) = 0$; M_i is the *initial marking.*

We define ${}^{\bullet}t \triangleq \{p \mid (p,t) \in F\}$ to be the *pre-multiset* of the transition t, and $t^{\bullet} \triangleq \{p \mid (t,p) \in F\}$ the *post-multiset* of the transition t.

A transition t is *enabled* by a marking M if M marks every place in ${}^{\bullet}t$; a transition *fires* from a marking M to a marking M', written $M \xrightarrow{t} M'$, iff for all $p \in P : M'(p) = M(p) - \sharp({}^{\bullet}p) + \sharp(p^{\bullet})$, where $\sharp({}^{\bullet}p)$ and $\sharp(p^{\bullet})$ are the number of occurrences of p in ${}^{\bullet}t$, $t^{\bullet}$, respectively.

Notice that we allow multiple connections between a place and a transition, that is analogous to assign a weight to an arc representing the token that have to be consumed to fire the reaction.

Definition 11 Let $N = (P,T,F,M)$ and $N' = (P',T',F',M')$ be two P/T nets, we say that N and N' are *isomorphic*, if there exist two bijections $\alpha : P \rightarrow P'$ and $\beta : T \rightarrow T'$, such that:

- $(p,t) \in F$ iff $(\alpha(p), \beta(t)) \in F'$;

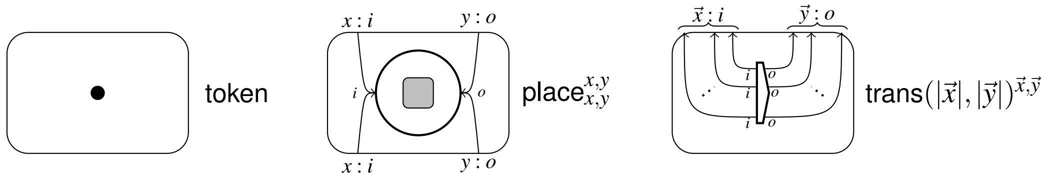


Figure 6: Signature for the encoding of compositional Petri nets.

- $(t,p) \in F$ iff $(\beta(t),\alpha(p)) \in F'$;
- $M = M' \circ \alpha$.

We recall, as defined in [LM06], the definition of link sorting.

Definition 12 A *link sorting* is a triple $\Sigma = (\Theta, \mathcal{K}, \Phi)$, where Φ is a set of sorts, and $\mathcal{K}$ is a sorted signature (that is, a signature enriched with a sort to ports of each control). Furthermore, each name in the interface (X^-, X^+) is given a sort, so the interfaces take the form $(\{x_1^- : \theta_1^-, \ldots, x_n^- : \theta_n^-\}, \{x_1^+ : \theta_1^+, \ldots, x_m^+ : \theta_m^+\})$. Finally, Φ is a rule on such enriched bigraphs, that is preserved by identities, composition and tensor product.

We denote the precategory and category of, respectively, concrete and abstract Σ-sorted directed bigraphs with $'\mathrm{DBIG}(\Sigma)$ and $\mathrm{DBIG}(\Sigma)$.

Definition 13 A *positive-negative sorting* $\Sigma = (\Theta, \mathcal{K}, \Phi)$ has sorts: $\Theta = \{\theta_1, \ldots, \theta_n\}$. The signature $\mathcal{K}$ assigns sorts to ports arbitrarily. The unique Φ rule is: a point and a link (except of edges) can be connected if they are equally sorted.

In order to define an encoding for compositional Petri nets, we introduce a positive-negative sorting Σ_{petri}, having sort $\Theta_{petri} \triangleq \{i, o\}$ and sorted signature:

$$\mathcal{K}_{petri} \triangleq \{\mathsf{token} : (0,0),\ \mathsf{place} : (\{1 : i, 1 : o\}, 0),\ \mathsf{trans}(h,k) : (0, \{h : i, k : o\})\} \qquad \text{where } h,k > 0$$

where the controls token and trans are both atomic, while the control place is passive. Finally, the Φ rule ensures that the linking is allowed only among ports having the same sort. An example of use of this sorted signature is shown in Figure 6. The encoding function $[\![\cdot]\!]$ is defined as follows:

$$[\![(P,T,F,M)]\!] = merge_{(|P|+|T|)} \circ \left(id_{|P|} \curlyvee id_{(P\times\{i,o\},\emptyset)} \curlyvee \left(\curlyvee_{t\in T} trans(|^\bullet t|,\ |t^\bullet|)_{(^\bullet t\times\{i\},\ t^\bullet\times\{o\})} \right) \right) \circ \left(\sum_{p\in P} place^{(p,i),\ (p,o)}_{(p,i),\ (p,o)} \circ (merge_{(|M(p)|+1)} \circ (\triangle^{\{(p,i),\ (p,o)\}} \otimes (\sum_{i=0}^{M(p)} token) \otimes 1)) \right).$$

where, with an abuse of notation, $trans(|^\bullet t|,\ |t^\bullet|)_{(^\bullet t\times\{i\},\ t^\bullet\times\{o\})}$ means that if in the multisets there are some repetitions of places then the ports of *trans* are linked to the same downward inner name (i.e., (p,i) or (p,o)), an alternative definition is to link every port of *trans* to a different downward inner name and then (eventually) "equate" these names using fusions.

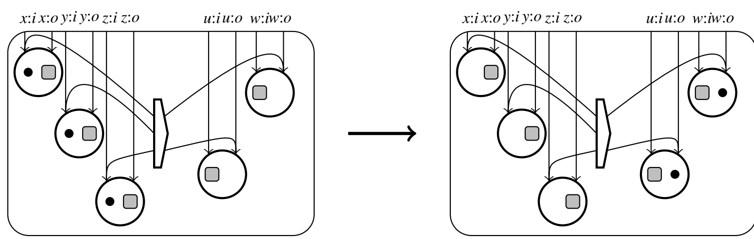


Figure 7: Example of reaction rule in the case of 3 input and 2 output places.

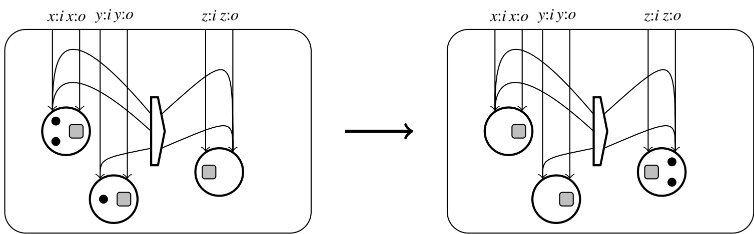


Figure 8: Example of reaction rule in the case of 2 input and 1 output places (with multiple arcs).

Proposition 1 *Let N, N' be two P/T nets, N is isomorphic to N' iff $[\![N]\!] = [\![N']\!]$ up to iso.*

We have a different reaction rule for any pair (h,k) associated to the control trans, in Figure 7 we show the reaction rule for the pair $(3,2)$, that is a transition having 3 inputs and 2 outputs. Moreover, we allow multiple connections between places and transitions, as in Figure 8, and we can have transitions using some places as inputs and outputs, see Figure 9.

Now we can show that the given translation is adequate.

Theorem 2 *Let (P,T,F,M_i) be a P/T net, $M \xrightarrow{t} M'$ iff $[\![(P,T,F,M)]\!] \longrightarrow [\![(P,T,F,M')]\!]$.*

Proof. $(\Rightarrow)$ Suppose $M \xrightarrow{t} M'$, so M enable the transition t, then there exists a trans-node in $[\![(P,T,F,M)]\!]$ encoding the transition t, and the corresponding place-node of $^\bullet t$ contain the necessary tokens to fire the transition (by translation of M), then we can apply the appropriate rule to perform the reaction reaching the configuration $[\![(P,T,F,M')]\!]$.

$(\Leftarrow)$ If $[\![(P,T,F,M)]\!] \longrightarrow [\![(P,T,F,M')]\!]$, there exists a matching of a rule with a sub-bigraph of $[\![(P,T,F,M)]\!]$, in particular the matched nodes have a counter part into the P/T net (P,T,F,M), so the marking M enables a transition t (corresponding to the trans-node), and then $M \xrightarrow{t} M'$. $\square$

An interesting future work is to study the bisimulation induced by the IPO LTS over these compositional Petri nets. We remark however, that this notion of composition is different from that in Open Petri nets, since in the latter the interfaces express also behavioural properties, while in the bigraphical encoding the interfaces express resource requests and offerings.

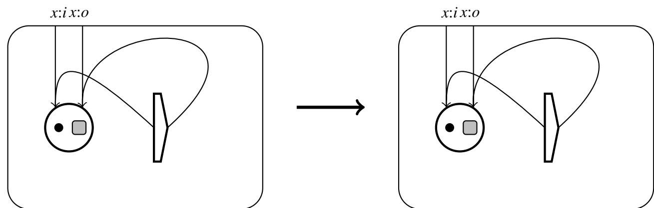


Figure 9: Example of reaction rule in the case of 1 place used as input and output.

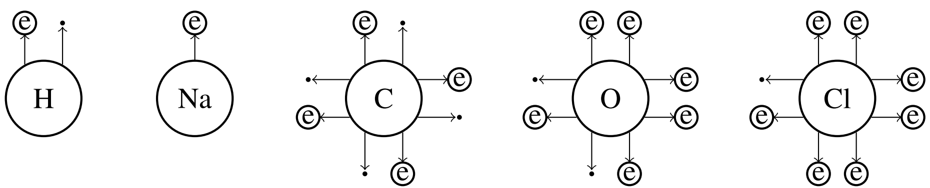


Figure 10: Example of atom encodings in directed link graphs.

5 Chemical Reactions

A chemical reactions is a process describing the conversions of a chemical compositions. Always, the chemical changes caused by a reaction involve the motion of electrons in the forming or breaking of *chemical bonds*. For example, the *octet rule* says that atoms tend to gain, lose or share electrons so as to have eight electrons in their outer electron shell.

In this section, we give an encoding of atoms into directed link graphs, as shown in Figure 10, inspired by the well-known *Lewis structures*. We describe the atoms as nodes, and those nodes have a number of positive ports equal to the number of valence electrons. Each of these ports are linked to an electron, represented as a node having a negative port (accepting incoming connections; for sake of simplicity we identify the node representing the electron with its port, that is, we do not force all incoming connections to be linked to a precise point of the node). Moreover, some nodes can have extra ports, that are initially linked to edges, hydrogen and oxygen can be two examples, the idea is that such a configuration describes the aim of the atom to “capture” electrons to complete its external shell; e.g. an oxygen atom has two missing electrons, so it tries to share these two electrons with a pair of hydrogen atoms forming the water molecule.

We apply this model describing the forming and breaking of bonds among atoms, here we deal with *strong bonds*, that is *covalent* and *ionic bonds*.

Some examples of covalent bonds are shown in Figure 11, the first shows how to hydrogen atoms can share their electron. The second one is well-know and describes the generation of a water molecule from two hydrogen atoms and an oxygen one: the oxygen shares two electrons: one with each hydrogen, in this way it gets the two missing electrons in its external orbit, conversely each hydrogen atom completes its orbit sharing an electron with the oxygen. The latter describes a more complicate situation, where the two carbon atoms (each needing four electron

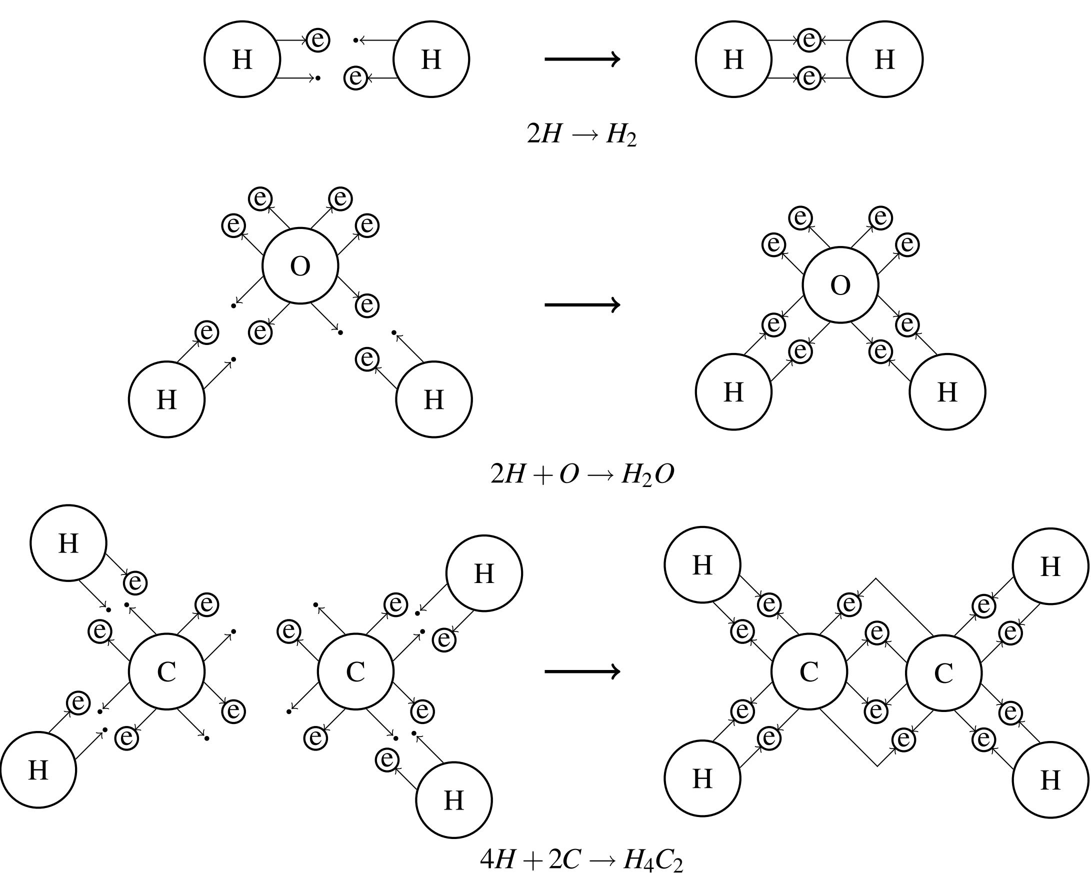


Figure 11: Example of covalent bonds among atoms.

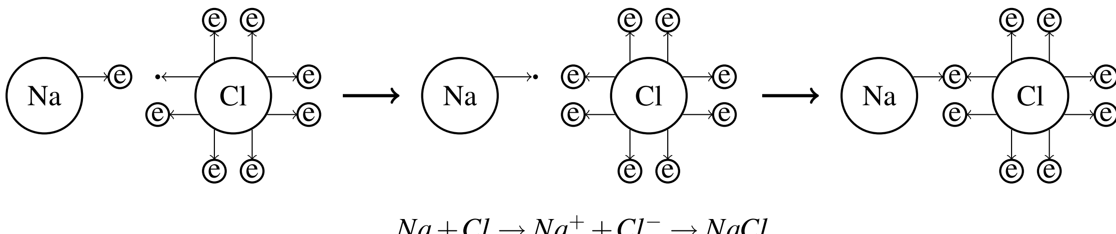
to complete its orbit) share two electron with the other carbon atom, and the remaining two missing electrons are provided by a pair of hydrogen.

In Figure 12, we show an example of ionic bond: given an atom of sodium and a chlorine one, it may happen (by octet rule) that the external electron of sodium is lost by the atom and "captured" by the chlorine, forming a sodium (positive) ion and a chlorine (negative) ion. These two ions attract each other by the electrostatic force caused by the electron exchange. Finally the ions can be composed to form sodium-chloride molecule, that is the common salt.

An interesting future work concern to represent the *weak bonds*, i.e. *hydrogen bonds* and *van der Waals bonds*, using the same representation as much as possible.

6 Conclusions

In this paper, we have considered directed bigraphs over *polarized signatures*, a bigraphical model for concurrent, distributed system with resources and controls. The main difference with previous versions of bigraphs is the capability of nodes (i.e., systems) to ask for resource access



$$Na + Cl \rightarrow Na^{+} + Cl^{-} \rightarrow NaCl$$

Figure 12: Examples of ion bonds among atoms.

(via the "positive ports") and to control other's requests, providing access to own resources (via the negative ports). These bigraphs have RPO and IPO constructions, thus allowing to derive systematically labelled transition systems from reactive systems, as in [JM03, GM07c]; notably the bisimilarities induced by these labelled transition systems are always congruences. These directed bigraphs admit also a notion of normal form, and a complete axiomatization.

We have exhibited the expressive power of this framework, by applying it some interesting cases: a three-tier interaction between client, server and back-end service through a firewall, the Petri nets, and chemical reactions. All these cases are faithfully encoded as directed bigraphs with polarized signatures (possibly with sorting).

An interesting future work is to develop properly the treatment of web service interactions, extending the ideas shown in Section 4.1. In particular, we would like to give a bigraphical semantics of some formal calculus for web services, such as SCC or CC-Pi [BBC+06, BM07].

Another future development is to use this kind of bigraphs as a general framework for systems biology. Some preliminary experiment about the representation of biochemical reactions, not shown in this paper due to lack of space, are promising: ions, electrons, chemical links can be represented as controls and arcs, and the place structure can be fruitfully used to represent nesting of chemical compounds. It would be interesting to encode in directed bigraphs some important formalism for systems biology, such as the κ-calculus [DL04]. Along this line, also the possibility of adding quantitative aspects (i.e., reaction rates) sounds very promising.

Bibliography

[BBC+06] M. Boreale, R. Bruni, L. Caires, R. D. Nicola, I. Lanese, M. Loreti, F. Martins, U. Montanari, A. Ravara, D. Sangiorgi, V. T. Vasconcelos, G. Zavattaro. SCC: A Service Centered Calculus. In Bravetti et al. (eds.), *Proc. WS-FM*. Lecture Notes in Computer Science 4184, pp. 38–57. Springer, 2006.

[BDE+06] L. Birkedal, S. Debois, E. Elsborg, T. Hildebrandt, H. Niss. Bigraphical Models of Context-Aware Systems. In Aceto and Ingólfsdóttir (eds.), *Proc. FoSSaCS*. Lecture Notes in Computer Science 3921, pp. 187–201. Springer, 2006.

[BH06] M. Bundgaard, T. T. Hildebrandt. Bigraphical Semantics of Higher-Order Mobile Embedded Resources with Local Names. *Electr. Notes Theor. Comput. Sci.* 154(2):7–29, 2006.

[BM07] M. G. Buscemi, U. Montanari. CC-Pi: A Constraint-Based Language for Specifying Service Level Agreements. In Nicola (ed.), *Proc. ESOP*. Lecture Notes in Computer Science 4421, pp. 18–32. Springer, 2007.

[BS06] M. Bundgaard, V. Sassone. Typed polyadic pi-calculus in bigraphs. In Bossi and Maher (eds.), *Proc. PPDP*. Pp. 1–12. ACM, 2006.

[DL04] V. Danos, C. Laneve. Formal molecular biology. *Theoretical Computer Science* 325, 2004.

[GM07a] D. Grohmann, M. Miculan. An Algebra for Directed Bigraphs. In Mackie and Plump (eds.), *Pre-proceedings of TERMGRAPH 2007*. Electronic Notes in Theoretical Computer Science. Elsevier, 2007.

[GM07b] D. Grohmann, M. Miculan. Directed bigraphs. In *Proc. XXIII MFPS*. Electronic Notes in Theoretical Computer Science 173, pp. 121–137. Elsevier, 2007.

[GM07c] D. Grohmann, M. Miculan. Reactive Systems over Directed Bigraphs. In Caires and Vasconcelos (eds.), *Proc. CONCUR 2007*. Lecture Notes in Computer Science 4703, pp. 380–394. Springer-Verlag, 2007.

[GM08] D. Grohmann, M. Miculan. Controlling resource access in Directed Bigraphs. Technical report, Department of Mathematics and Computer Science, University of Udine, 2008. Available at http://www.dimi.uniud.it/miculan/Papers/.

[Jen08] O. H. Jensen. *Mobile Processes in Bigraphs*. PhD thesis, University of Aalborg, 2008. To appear.

[JM03] O. H. Jensen, R. Milner. Bigraphs and transitions. In *Proc. POPL*. Pp. 38–49. 2003.

[JM04] O. H. Jensen, R. Milner. Bigraphs and mobile processes (revised). Technical report UCAM-CL-TR-580, Computer Laboratory, University of Cambridge, 2004.

[LM06] J. J. Leifer, R. Milner. Transition systems, link graphs and Petri nets. *Mathematical Structures in Computer Science* 16(6):989–1047, 2006.

[Mil01] R. Milner. Bigraphical Reactive Systems. In Larsen and Nielsen (eds.), *Proc. 12th CONCUR*. Lecture Notes in Computer Science 2154, pp. 16–35. Springer, 2001.

[Mil06] R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation* 204(1):60–122, 2006.

[Mil07] R. Milner. Local Bigraphs and Confluence: Two Conjectures. In *Proc. EXPRESS 2006*. Electronic Notes in Theoretical Computer Science 175(3), pp. 65–73. Elsevier, 2007.

[PV98] J. Parrow, B. Victor. The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes. In *Proceedings of LICS '98*. Pp. 176–185. Computer Society Press, July 1998.
http://www.docs.uu.se/~victor/tr/fusion.shtml

Interaction nets: programming language design and implementation

Abubakar Hassan[1], **Ian Mackie**[2] **and Shinya Sato**[3]

[1] Department of Informatics, University of Sussex, Falmer, Brighton BN1 9QJ, UK

[2] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France

[3] Himeji Dokkyo University, Faculty of Econoinformatics, 7-2-1 Kamiohno, Himeji-shi, Hyogo 670-8524, Japan

Abstract: This paper presents a compiler for interaction nets, which, just like term rewriting systems, are user-definable rewrite systems which offer the ability to specify and program. In the same way that the λ-calculus is the foundation for functional programming, or horn clauses are the foundation for logic programming, we give in this paper an overview of a substantial software system that is currently under development to support interaction based computation, and in particular the compilation of interaction nets.

Keywords: Interaction Nets, programming languages

1 Introduction

Interaction nets [3] are a graphical—visual—programming language. Programs are expressed as graphs, and computation is graph reduction. From another perspective, interaction nets are also a low-level implementation language: we can define systems of interaction nets that are instructions for the target of compilation schemes of other programming languages (typically functional languages, based on the λ-calculus). To facilitate both these uses of interaction nets, we need high quality, robust implementations.

In this paper we focus on using interaction nets as a programming language. Although we can already program in interaction nets (they are after all Turing complete) they lack what programming languages should offer: features such as input/output, etc. However, more important to this paper is that they lack the structure that we expect from modern programming languages: a module system for instance. Thus the first contribution in this paper is a programming language for interaction nets, which lifts them from the "pure" world to allow them to be used in practice. To give some analogies as to what we are doing, consider the following where we give for a particular formalism (or model), some examples of languages that have been created for it:

- λ-calculus: functional programming languages, such as Haskell [8], Standard ML [7], OCaml, etc.
- Horn clauses: logic programming languages, such as Prolog.
- Term rewriting systems: OBJ, Elan, Maude [1].
- π-calculus: PICT [9]

The first goal of this paper is to add interaction nets to this list by providing a corresponding programming language that we call Pin. In the list above, the programming language on the right is there to provide not only some syntactical sugar, but also to provide features that the theory does not offer. For instance, if we look in detail at the analogy with the λ-calculus and functional programming languages, functional languages allow the definition of functions such as: `twice f x = f(f x)`, which is a significant improvement over $\lambda fx.f(fx)$ as a programming language, as programs can be reused for instance. In addition languages provide a module system, data-types (built-in and a mechanism for user-defined data-types), input/output, etc.

In [6] we made a first attempt to build a programming language for interaction nets. Here we take that language as a starting point, and in addition investigate how to compile it. This gives the second contribution to this paper: we provide, for the first time, a compilation of interaction nets. The motivation for this line of compilation is that interaction nets have been implemented very efficiently, and also in parallel: by providing a compiler to interaction nets we potentially obtain parallel implementations of programming languages (even sequential ones) for free. However, this is not reported in the current paper: we are simply setting up the foundations and the framework for this development.

To summarise, the paper contains three main contributions: we define a programming language for interaction nets, we define an abstract machine for interaction nets, and we define a compiler for interaction nets.

We have implemented this language, and we report on this at the end of the paper. In particular, we show that the new techniques developed for implementing interaction nets improve upon existing implementations. We also remark that the ideas used in this paper could be extended to work with other rewriting systems through translations to interaction nets. Finally, we remark that this is the first attempt in this area, there is now a need to develop tools and optimisation techniques for this paradigm.

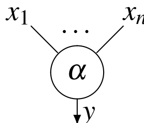
Related work. There are several implementations of interaction nets in the literature: a graphical one [4] and a textual, parallel one [11]. The goals of those works was to investigate interaction nets: our focus is on explicitly extending interaction nets to a rich programming language.

Structure. The rest of this paper is structured as follows. In the next section we recall some background information about interaction nets. In Section 3 we define a programming language for interaction nets. In Section 4 we define an abstract machine for interaction nets (IAM) and give the compilation schemes in Section 5. Section 6 gives some properties of the compilation. Section 6 gives details of the implementation. Finally, we conclude the paper in Section 7.

2 Background

An interaction net system is a set Σ of symbols, and a set $\mathscr{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ is called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port*

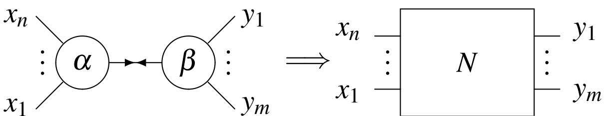
depicted by an arrow, and n *auxiliary ports* labelled $x_1,\ldots,x_n$ corresponding to the arity of the symbol. We represent an agent graphically in the following way:



If $n = 0$ then the agent has no auxiliary ports, but it will always have a principal port. We represent agents textually as: $\mathtt{y} \sim \alpha(\mathtt{x_1},\ldots,\mathtt{x_n})$, and we omit the brackets if the arity of an agent is zero.

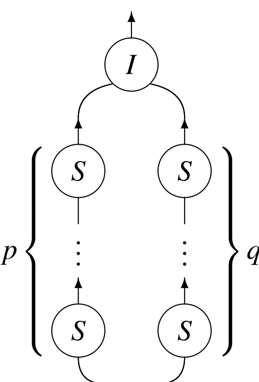
A net N built on Σ is a graph (not necessarily connected) with agents at the vertices. The edges of the net connect agents together at the ports such that there is only one edge at every port, although edges may connect two ports of the same agent. The ports of an agent that are not connected to another agent are called the free ports of the net. There are two special instances of a net: a wiring (a net with no agents) and the empty net.

A pair of agents $(\alpha,\beta) \in \Sigma\times\Sigma$ connected together on their principal ports is called an *active pair*, which is the interaction net analogue of a redex. An interaction rule $((\alpha,\beta) \Longrightarrow N) \in \mathscr{R}$ replaces an occurrence of the active pair (α,β) by a net N. The rule has to satisfy a very strong condition: all the free ports are preserved during reduction, and moreover there is at most one rule for each pair of agents. The following diagram illustrates the idea, where N is any net built from Σ.

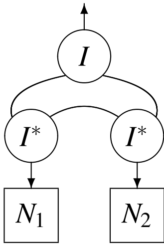


An implementation of this rewriting process has to create the right-hand side of the net, and make all the connections (re-wirings). Although it may not be the most trivial computation step, it is a known, constant time operation. It is for this reason that interaction nets lend themselves to the study of cost models of computation. *All* aspects of a computation are captured by the rewriting rules—no external machinery such as copying a chunk of memory, or a garbage collector, are needed. Interaction nets are amongst the few formalisms which model computation where this is the case, and consequently they can serve as both a low level operational semantics and an object language for compilation, in addition to being well suited as a basis for a high-level programming language. We refer the reader to other papers on interaction nets for properties and additional theory.

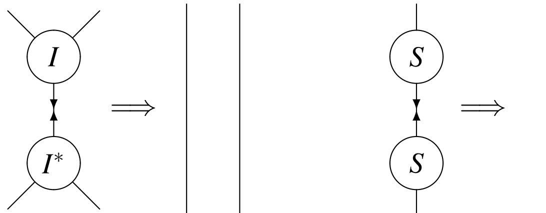
An example: encoding the integers. Many simple examples of systems of interaction nets can be found in the literature, for instance the encoding of arithmetic expressions, etc. However, interaction nets are well suited for alternative representations of data-structures and algorithms. Any integer n can be represented (non-uniquely) as a difference of two natural numbers: $p-q$. Using this idea, we can represent integers in the following way:



Here, the agent S (of arity 1) is interpreted as *successor*. A linear chain of S agents of length n is used to give a representation of a natural number n. The representation of an integer simply takes two chains of length p and q, and connects them together as shown, using the agent I (of arity 2). Although this representation is not unique, we can talk about *canonical forms* when $p = 0$ or $q = 0$ (or both), and there are interaction rules that can compute this. The integer zero is a net where $p = q$, in particular when $p = q = 0$: we can encode addition, subtraction and negation, for which we need several interaction rules. We just detail the encoding of addition. If N_1 and N_2 are the net representations of $z_1 = (p_1 - q_1)$ and $z_2 = (p_2 - q_2)$ respectively, then we can use the following configuration to encode the addition:



The interaction rules for the agents are:



To see how this works, there will be two interactions between the agents I (of nets N_1 and N_2) and I^* which concatenate the two chains. If z_1 and z_2 are both in canonical form and moreover both positive (or both negative) then the addition is completed with the two interactions. Otherwise there will be an additional $\min\{q_1, p_2\}$ interactions between two S agents to obtain a net in normal form.

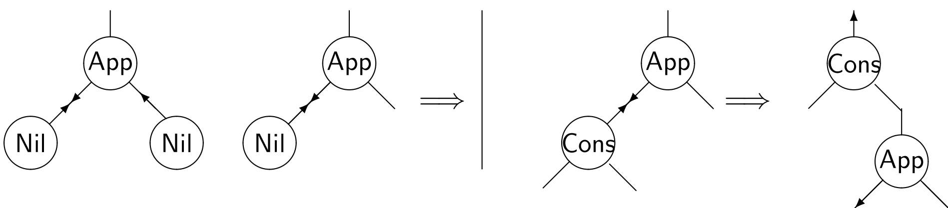


Figure 1: Example net and rules

3 Programming Language

Following [2], an interaction net system can be described as a configuration $c = (\Sigma, \Delta, \mathscr{R})$, where Σ is a set of symbols, Δ is a multiset of active pairs, and $\mathscr{R}$ is a set of rules. A language for interaction nets needs to capture each component of the configuration, and provide ways to structure and organise the components. Starting from a calculus for interaction nets we build a core language. A core language can be seen both as a programming language and as a target language where we can compile high-level constructs. Drawing an analogy with functional programming, we can write programs in the pure λ-calculus and can also use it as a target language to map high-level constructs. In this way, complex high-level languages can be obtained which by their definition automatically get a formal semantics based on the core language.

Nets are written as a comma separated list of agents, corresponding to a flattening of the graph. There are many different (equivalent) ways we could do this depending on the order we choose to enumerate the agents. Using the net on the left-most side of Figure 1 as an example, we can generate a representation as the following list:

$$\mathtt{x} \sim \mathtt{App(r,a)},\ \mathtt{a} \sim \mathtt{Nil},\ \mathtt{x} \sim \mathtt{Nil}$$

This can be simplified by eliminating some names:

$$\mathtt{App(r,Nil)} \sim \mathtt{Nil}$$

In this notation the general form of an active pair is $\alpha(\ldots) \sim \beta(\ldots)$. We assume that all variable names occur at most twice. If a name occurs once, then it corresponds to one of the free ports of the net (r is free in the above). If a name occurs twice, then this represents an edge between two ports. In this latter case, we say that a variable is *bound*. The limitation on at most two occurrences corresponds to the requirement that it is not possible to have two edges connected to the same port in the graphical setting.

We represent rules by writing $l \Longrightarrow r$, where l consists of two agents connected at their principal ports. Therefore rules can be written as $\alpha(..) \sim \beta(..) \Longrightarrow N$, and as such we replace the '$\sim$' by '><' so that we can distinguish an occurrence of a rule from an occurrence of an active pair. The two rules on the right of Figure 1 (append) can be represented as:

$$\begin{array}{lll} \mathtt{App(r,y)} \ \mathtt{><} \ \mathtt{Nil} & \Longrightarrow & \mathtt{r} \sim \mathtt{y} \\ \mathtt{App(r,y)} \ \mathtt{><} \ \mathtt{Cons(v,u)} & \Longrightarrow & \mathtt{r} \sim \mathtt{Cons(v,z)}, \mathtt{u} \sim \mathtt{App(z,y)} \end{array}$$

$\langle Pin\rangle$ ::= { $\langle ruleDef\rangle$ | $\langle netDef\rangle$ }

$\langle ruleDef\rangle$ ::= $\langle agent\rangle$ '><' $\langle agent\rangle$ ['=>' $\langle netDef\rangle$]

$\langle netDef\rangle$ ::= $\langle equation\rangle$ { ';' $\langle equation\rangle$}

$\langle equation\rangle$ ::= $\langle term\rangle \sim \langle term\rangle$

$\langle term\rangle$ ::= $\langle agent\rangle$ | $\langle var\rangle$

$\langle agent\rangle$ ::= $\langle agentName\rangle$ [$\langle ports\rangle$]

$\langle ports\rangle$ ::= '(' $\langle term\rangle$ [',' $\langle term\rangle$] ')'

$\langle agentName\rangle$ ::= $\langle Name\rangle$

$\langle var\rangle$ ::= $\langle Name\rangle$

Figure 2: Syntax of the core language

The names of the bound variables in the two nets must be disjoint, and the free variables must coincide, which corresponds to the condition that the free variables must be preserved under reduction. Under this assumption, these two rules can be simplified to:

```
App(y,y) >< Nil                      ==>
App(Cons(v,z),y) >< Cons(v,App(z,y)) ==>
```

In this notation we observe that the right-hand side of the rule can always be empty. In this case we will omit the '$\Longrightarrow$' symbol. This notation therefore allows sufficient flexibility so that we can either write nets in a compact way (without any net on the right of the '$\Longrightarrow$') or in a more intuitive way where the right-hand side of the net is written out in full to the right of the arrow.

In Figure 2, we give the syntax for the core language discussed above.

Language extensions. The core language allows the representation of the three main components of an interaction net system: agents, rules and nets. The language is very attractive for theoretical investigations: it is minimal yet computationally complete. The price for this simplicity is programming comfort. Here, we give an overview of some practical extensions that enhance the usability of interaction nets. Details and examples of these extensions can be found at http://www.informatics.sussex.ac.uk/users/ah291/ [1].

Modular construction of nets. We allow nets to be named so that they can be built in a modular way. Named nets contain as their parameters a list of the net's free ports. Using this mechanism, we can build a new net by *instantiating* a named net with some argument nets. As an

[1] we refer to this url as the *project's web page* in this paper.

example, we can represent the example net in Figure 1 as:

```
Append x,r,a : App(r,a) ~ x
Applist r : Append Nil,r,Nil
```

Agent Variables. Some rules have the same structure and only differ in the agent names used. Agent variables act as a place holder for agents in the rules. An instance of a rule with a set of agent variables A will create a new rule with elements of A replaced by the actual agents. In the example below, `Templ` has agent variables `A1,A2`. The instance of this rule creates `App(x,x) >< Nil` where the agent variables have been replaced with `App, Nil` appropriately.

```
Templ A1,A2 : A1(x,x) >< A2
Templ (App,Nil)
```

Further extensions. We have designed a module system and a set of built-in agents and rules that perform input/output and arithmetic operations—these are reported on the project's web page.

4 The Abstract Machine

Here we describe the interaction net abstract machine (IAM) by giving the instructions operating on a machine state. We first set up some notation used in the description of the machine.

Definition 1 (Memory model) Let $Adr \subseteq \mathbb{N}$ be a set of memory locations. Agents of an interaction net system are $agent = name \times arity \times W$, where $arity \in \mathbb{N}$, *name* is an identifier, and W models connectivity between *agent* ports: $W = \{((l,p),(l',p')) \mid l,l' \in Adr, p,p' \in \mathbb{N}\}$. An element $((l_1,p_1),(l_2,p_2))$ in the set W is ordered if either $l_1 < l_2$ or $l_1 = l_2,\ p_1 < p_2$. If an agent stored at location l_k has it's auxiliary port p_i connected to the port p_j of some agent at location l_m, then $((l_k,p_i),(l_m,p_j)) \in W$.

We next define two functions to operate on the machine state:

Definition 2

- $\mathscr{L}_a : Adr \times \mathbb{N} \to Adr$ returns the location $l \in Adr$ pointed to by a given port of some *agent*: $\mathscr{L}_a(l_k,p_i) = l_m$ such that $((l_k,p_i),(l_m,p_j)) \in W$.
- $\mathscr{L}_p : Adr \times \mathbb{N} \to \mathbb{N}$ returns a port number that is connected to a port of some *agent* node: $\mathscr{L}_p(l_k,p_i) = p_j$ such that $((l_k,p_i),(l_m,p_j)) \in W$.

We define the function $\Upsilon : name \times name \to Inst$ that given a pair of *names* for an active pair (α,β) of a rule $(\alpha,\beta) \Longrightarrow N$, returns a sequence of IAM instructions that will build and rewire the net N. The mappings in Υ are constructed during the compilation of a rule.

Machine configuration components. The IAM machine consists of rules that transform configurations. A configuration is given by a 5-tuple $\langle C, \varsigma, F, A, H\rangle$ where C is a sequence of instructions, ς is a stack of frames. Each frame has an array of local variables and an operand stack. F

is a set of pointers to the variable agents of the net, A is a stack of active pair agents, and H is the heap.

An IAM program C is a sequence of instructions that we summarise in Figure 3. We write '$-$' for the empty sequence and $i, v, p, ar \in \mathbb{N}$.

The component ς is a stack of frames. Each frame $f = (L, S)$ where L is a partial function with a finite domain of definition, mapping memory locations to their contents. If L is defined, then $L[i \mapsto l]$ means that $L(i) = l$. S is an operand stack produced by the grammar: $S := - \mid v : S$ where v is any value representing a constant or a memory location, and $-$ is the empty stack.

The component F is a mapping from identifiers to a pair of natural numbers defined by: $F(x) = (l, p)$. Intuitively, it is used to hold the interface of the net.

The heap $H : Adr \to agent$ returns an *agent* node given some location $l \in Adr$. The special token *next* is used to hold the next free location $l \in dom(H)$. Intuitively, H is a memory area filled with agent nodes. Whenever a new node is put into the heap, the unused area marked by *next* is updated.

Figure 4 gives the IAM instructions as a set of transition rules. Each transition rule takes the form:

$$\Upsilon \vdash \langle C, \varsigma, F, A, H \rangle \Rightarrow \Upsilon \vdash \left\langle C', \varsigma', F', A', H' \right\rangle$$

which indicate how the components of the machine are transformed. We abbreviate $(L, S) : \varsigma$ to (L, S) in a configuration with only one frame in ς.

Initial and final states. The machine initialises the components C and Υ giving the initial configuration: $\Upsilon \vdash \langle C, -, [], -, [] \rangle$. The machine stops *successfully* when the instruction `halt` is executed with the configuration $\Upsilon \vdash \langle -, -, F, -, H \rangle$ or prematurely if a pre-condition of an instruction is not satisfied. In this case, the final configuration is the one obtained after the last instruction that has been executed successfully.

The evaluation mechanism. The evaluation of the net is started by executing the `eval` instruction. This instruction is appended at the end of the code sequence for the start active pair or *initial expression*. Thus, before the evaluation of the net, there is at least one active pair in the machine's active pair stack A. The pair in the stack A is examined and the code sequence of the rule for the pair is appended to the code component C of the machine (see semantics of `eval` in Figure 4).

The code for a rule will load one of the active agents into the stack S using the instruction `loadActive`, then start to build and rewire the right hand side net of the rule to the auxiliary agents connected to the interacting agent in the stack. The instruction `pop` pops the active agent from the component A. Evaluation is terminated when A is empty and execution jumps to the instruction sequence after `eval`.

We remark that this version of the machine treats only acyclic nets.

Instruction	*Description*
`enter`	push a new frame into the stack ς
`return`	remove the top frame from ς
`dup`	duplicate the top element on the stack S
`pop`	remove the top element on the active pair stack A
`load` i	push the element at index i of L onto the stack S.
`store` i	remove the top element of S and store it in L at index i.
`ldc` v	push the value v onto the stack S.
`fstore` x	store the top 2 elements at the top of S onto index x in F.
`fload` x	push the elements at index x of F onto the stack S.
`mkAgent` $ar\ \alpha$	allocate (unused) memory for an agent node of arity ar and name α in the heap H.
`mkVar` x	allocate memory for a variable node of arity 2 and name x in the heap
`getConnection` $p\ i$	push the agent a and the port number of a that connects at the auxiliary port p of the agent stored in local variable i
`loadActive` α	push the active agent α from active pair stack
`connectPorts`	pop two agents and two port numbers and connects the ports of the agents. If both ports are 0 (active pair) push an agent to A
`eval`	evaluate the active pair on top of the active stack.
`halt`	stop execution.

Figure 3: Summary of IAM instructions

5 Compilation

The compilation of Pin into IAM instructions is governed by the schemes: $\mathscr{C}_{pin}$ compiles a program, $\mathscr{C}_a$ compiles an agent, $\mathscr{C}_t$ compiles a term, $\mathscr{C}_n$ compiles a net and $\mathscr{C}_r$ compiles a rule. The compilation of a program generates the following code:

$$\mathscr{C}_{pin}[\![(\Sigma,\langle u_1 \sim v_1,\dots u_n \sim v_n\rangle,\mathscr{R})]\!] = \\ \mathscr{C}_n[\![u_1 \sim v_1,\dots,u_n \sim v_n]\!];\texttt{eval};\texttt{halt};\mathscr{C}_r[\![r_1]\!];\dots\mathscr{C}_r[\![r_n]\!];$$

where $r_1,...,r_n = \mathscr{R}$ are instances of rules. Σ is a set of symbols and each $u_i \sim v_i$ is an active pair. The compilation scheme $\mathscr{C}_n[\![u \sim v,\dots,u_n \sim v_n]\!]$ compiles a sequence of active pairs. We use the scheme $\mathscr{C}_r[\![r_i]\!]$ to compile a rule $r_i \in \mathscr{R}$:

$$\mathscr{C}_r[\![r_i]\!] = Inst = \mathscr{C}_r[\![\alpha(t_1,\dots,t_n)\ \texttt{><}\ \beta(u_1,\dots,u_n)\texttt{=>}u_1 \sim s_1,\dots,u_n \sim s_n]\!], \\ \Upsilon[(\alpha,\beta) \mapsto Inst, (\beta,\alpha) \mapsto Inst].$$

Compilation of a rule creates a mapping from active agent names to the instruction sequence *Inst* generated in the rule table Υ.

Figure 5 collects together the compilation schemes $\mathscr{C}_n$ and $\mathscr{C}_r$, that generate code for the input source text. The schemes use a set $\mathscr{N}$ of identifiers to hold all free variables of the net. Compiling a variable that already exists in $\mathscr{N}$ means that the variable is bound. The auxiliary function $ar(\alpha)$ returns the arity of the agent $\alpha \in \Sigma$.

$$
\begin{array}{l}
\Upsilon \vdash \langle \texttt{enter}: C, \varsigma, F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, ([], -) : \varsigma, F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{return}: C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, \varsigma, F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{dup}: C, (L, v : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, v : v : S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{pop}: C, (L,S), F, l : A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L,S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{load}\, i: C, (L[i \mapsto v], S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L[i \mapsto v], v : S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{store}\, i: C, (L, v : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L[i \mapsto v], S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{ldc}\, v: C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, v : S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{fstore}\, x: C, (L, p : l : S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, p : l : S), F[x \mapsto (l,p)], A, H\rangle \\
\Upsilon \vdash \langle \texttt{fload}\, x: C, (L,S), F[x \mapsto (l,p)], A, H[l_a \mapsto (n, a, \{((l_a, p_l), (l,p))\} \cup w)]\rangle \Rightarrow \\
\quad \Upsilon \vdash \langle C, (L, p : l : S), F, A, H[l_a \mapsto (n,a,w)]\rangle \\
\Upsilon \vdash \langle \texttt{mkAgent}\, ar\, \alpha: C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, l : S), F, A, H[l \mapsto (\alpha, a, \emptyset)]\rangle \\
\qquad \text{where } l = next \\
\Upsilon \vdash \langle \texttt{mkVar}\, x\ : C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L, 1 : l : S), F, A, H[l \mapsto (x, 2, \emptyset)]\rangle \\
\qquad \text{where } l = next \\
\Upsilon \vdash \langle \texttt{getConnection}\, p\, i: C, (L[i \mapsto l], S), F, A, H\rangle \Rightarrow \\
\quad \Upsilon \vdash \langle C, (L[i \mapsto l], \mathscr{L}_p(l,p) : \mathscr{L}_a(l,p) : S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{loadActive}\, \alpha: C, (L,S), F, l : A, H[l \mapsto (n,a,w)]\rangle \Rightarrow \\
\quad if(n\ =\ \alpha) \\
\qquad \Upsilon \vdash \langle C, (L, l : S), F, \mathscr{L}_a(l,0) : A, H[l \mapsto (n,a,w)]\rangle \\
\quad else \\
\qquad \Upsilon \vdash \langle C, (L, \mathscr{L}_a(l,0) : S), F, l : A, H[l \mapsto (n,a,w)]\rangle \\
\Upsilon \vdash \langle \texttt{connectports}: C, (L, p_1 : l_1 : p_2 : l_2 : S), F, A, H[l_1 \mapsto (n_1, a_1, w_1), \\
\qquad l_2 \mapsto (n_2, a_2, w_2)]\rangle \Rightarrow \\
\qquad if(p_1 + p_2 = 0) \\
\qquad\quad \Upsilon \vdash \langle C, (L,S), F, l_2 : A, \\
\qquad\qquad H[l_1 \mapsto (n_1, a_1, w_1 \cup \{((l_1,0),(l_2,0))\}), \\
\qquad\qquad l_2 \mapsto (n_2, a_2, w_2 \cup \{((l_1,0),(l_2,0))\})]\rangle \\
\qquad else \\
\qquad\quad \Upsilon \vdash \langle C, (L,S), F, A, \\
\qquad\qquad H[l_1 \mapsto (n_1, a_1, w_1 \cup \{((l_1,p_1),(l_2,p_2))\}), \\
\qquad\qquad l_2 \mapsto (n_2, a_2, w_2 \cup \{((l_1,p_1),(l_2,p_2))\})]\rangle \\
\Upsilon[(\alpha,\beta) \mapsto c] \vdash \langle \texttt{eval}: C, (L,S), F, l_1 : A, H[l_1 \mapsto (\alpha, a_1, w_1 \cup \{((l_1,0),(l_2,0))\}), \\
\quad l_2 \mapsto (\beta, a_2, w_2 \cup \{((l_1,0),(l_2,0))\})]\rangle \Rightarrow \\
\qquad \Upsilon[(\alpha,\beta) \mapsto c] \vdash \langle c : \texttt{eval}: C, (L,S), F, l_1 : A, H[l_1 \mapsto (\alpha, a_1, w_1), l_2 \mapsto (\beta, a_2, w_2)]\rangle \\
\Upsilon \vdash \langle \texttt{eval}: C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle C, (L,S), F, A, H\rangle \\
\Upsilon \vdash \langle \texttt{halt}: C, (L,S), F, A, H\rangle \Rightarrow \Upsilon \vdash \langle -, -, F, -, H\rangle
\end{array}
$$

Figure 4: IAM instructions

$$\mathscr{C}_t[\![x]\!] = \begin{cases} \textit{if } (x \in \mathscr{N}) \\ \quad \texttt{fload}\ x; \\ \quad \mathscr{N} \setminus \{x\} \\ \textit{else} \\ \quad \texttt{fstore}\ x; \\ \quad \texttt{mkVar}\ x; \\ \quad \mathscr{N} \cup \{x\} \end{cases} \qquad \mathscr{C}_e[\![t \sim s]\!] = \begin{cases} \mathscr{C}_t[\![t]\!]; \\ \mathscr{C}_t[\![s]\!]; \\ \texttt{connectPorts}; \end{cases}$$

$$\mathscr{C}_t[\![\alpha(t_1,\ldots,t_n)]\!] = \begin{cases} \texttt{mkAgent}\ ar(\alpha)\ \alpha; \\ \text{for } 1 \le i \le n \\ \quad \mathscr{C}_p\ [\![t_i]\!]\ i; \\ \texttt{ldc}\ 0; \end{cases} \qquad \mathscr{C}_{rr}[\![t]\!]\ j\ i = \begin{cases} \texttt{getConnection}\ j\ i; \\ \mathscr{C}_t[\![t]\!]; \\ \texttt{connectPorts}; \end{cases}$$

$$\begin{array}{l}\mathscr{C}_r[\![\alpha(t_1,\ldots,t_n) \\ \quad \texttt{><}\ \ \beta(v_1,\ldots,v_k) \\ \texttt{=>} u_1 \sim s_1,\ldots, \\ \qquad u_m \sim s_m]\!] = \end{array} \begin{cases} \texttt{enter}; \\ \mathscr{C}_{rt}[\![\alpha(t_1,\ldots,t_n)]\!]; \\ \mathscr{C}_{rt}[\![\beta(v_1,\ldots,v_k)]\!]; \\ \text{for } 1 \le i \le m \\ \quad \mathscr{C}_e[\![u_i \sim s_i]\!]; \\ \texttt{pop}; \\ \texttt{return}; \end{cases} \mathscr{C}_n[\![u \sim v,\ldots,u_n {\sim} v_n]\!] = \begin{cases} \texttt{enter}; \\ \mathscr{C}_e[\![u {\sim} v]\!]; \\ \vdots \\ \mathscr{C}_e[\![u_n {\sim} v_n]\!]; \end{cases}$$

$$\mathscr{C}_{rt}[\![\alpha(t_1,\ldots,t_n)]\!] = \begin{cases} \texttt{loadActive}\ \alpha; \\ \texttt{store}\ 0; \\ \text{for } 1 \le i \le n \\ \quad \mathscr{C}_{rr}[\![t_i]\!]\ i\ 0; \end{cases} \qquad \mathscr{C}_p[\![t]\!]\ j = \begin{cases} \texttt{dup}; \\ \texttt{ldc}\ j; \\ C_t[\![t]\!]; \\ \texttt{connectPorts}; \end{cases}$$

Figure 5: Compilation schemes

We end this section with a concrete example of compilation, and give the code generated for the simple system given below:

```
Eps >< Eps
Eps ~ Eps
```

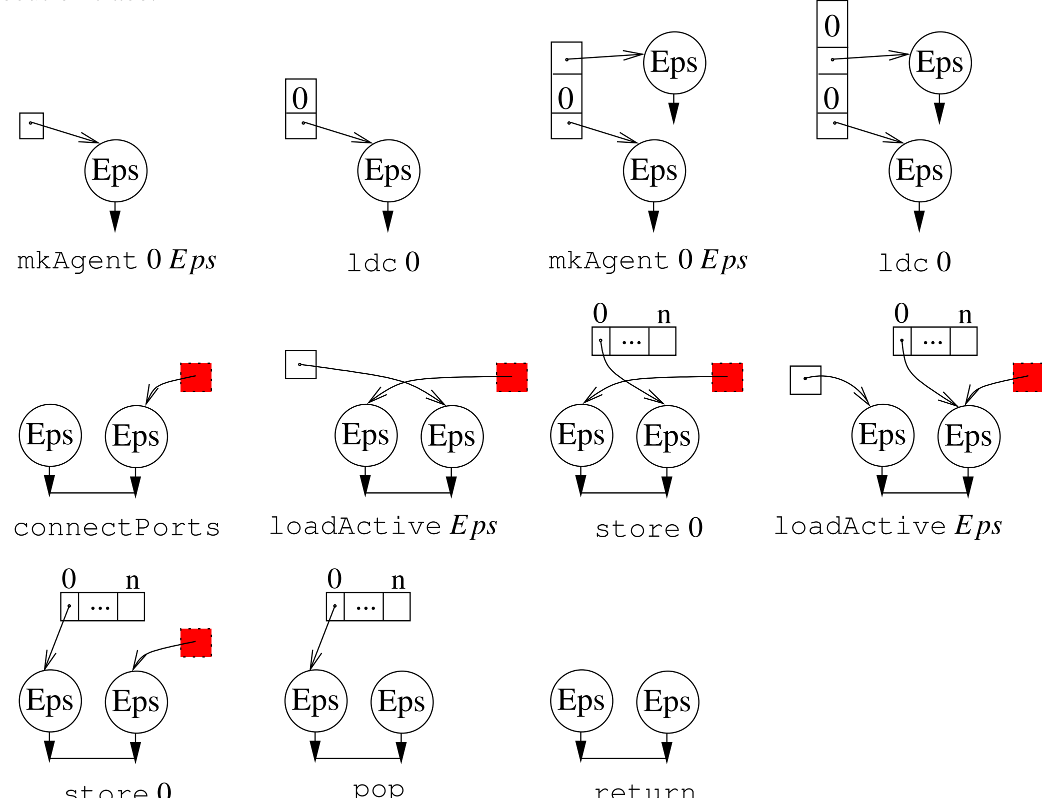
This system contains just one agent and one rule, and the net to be compiled is an instance of that rule. The output of the compiler is given below.

```
enter
mkAgent 0 Eps       enter
ldc 0               loadActive Eps
mkAgent 0 Eps       store 0
ldc 0               loadActive Eps
connectPorts        store 0
eval                pop
halt                return
```

The abstract machine loads the program into memory then sequentially executes the byte codes for the active pairs. The instruction `eval` calls the execution of the code block for the corresponding rule. Below we give a snap shot of the execution trace.

- represents the active pair stack A of the machine.
- top
  represents the stack S.
- 0 n
  ⋯ represents the local variable array L.

The state after execution of each instruction is shown. Components that do not contain any value are omitted. Note that this net contains no interface, thus the interface list F does not appear in the execution trace.



Observe that after the execution of `connectPorts`, a pointer to the newly created active pair is pushed into the stack A. Since the rule for this active pair contains an empty right hand side net, there is no re-wiring that is performed. After evaluation, the active pair stack becomes empty. After the last instruction `return` of the rule, remaining active pair agents in the heap are unreachable from any of the machine's components, and can be garbage collected or reused. We do not address the issues of heap garbage collection or agent reuse in this paper.

6 The implementation

Here we give a brief overview of the pragmatics of the language. We have implemented the compiler, and here we show example programs, the use of the system, and also some benchmark

results comparing with other implementations of interaction nets.

The prototype implementation of the compiler and abstract machine can be downloaded from the project's web page. The compiler reads a source program and outputs an executable with the extension '.pin'. The pin file can then be executed by the abstract machine. Various examples and instructions on how to compile and execute a program are provided on the webpage.

The table below shows some benchmark results that we have obtained. We compare the execution time in seconds of our implementation (Pin) with Amine [10] - an interaction net interpreter, and SML [7] - a fully developed implementation. The last column gives the number of interactions performed by both Pin and Amine. The first two input programs are applications of church numerals where $n = \lambda f.\lambda x.f^n x$ and $I = \lambda x.x$. The encodings of these terms into interaction nets are given in [5]. The next programs compute the Ackermann function defined by:

$$A(m,n) = \begin{cases} n+1 \text{ if } m = 0 \\ A(m-1,1) \text{ if } n = 0 \text{ and } m > 0 \\ A(m-1,A(m,n-1)) \text{ if } m > 0 \text{ and } n > 0 \end{cases}$$

The following rules are the interaction net encoding of the Ackermann function:

```
Pred(Z) >< Z,                Dup(Z,Z) >< Z,
Pred(x) >< S(x),             Dup(S(a),S(b)) >< S(Dup(a,b)),
A(r,S(r)) >< Z,              A1(Pred(A(S(Z),r)),r) >< Z,
A(A1(S(x),r),r) >< S(x),     A1(Dup(Pred(A(r1,r)),A(y,r1)),r) >< S(y),
```

and A(3,8) means computation of `A(S(S(S(S(S(S(S(S(Z)))))))),r)` $\sim$ `S(S(S(Z)))`.

Input	Pin	Amine	SML	Interactions
322II	0.002	0.006	2.09	383
245II	0.016	0.088	1355	20211
A(3,8)	3	16	0.04	8360028
A(3,10)	51	265	0.95	134103148

We can see from the table that the ratio of the average number of interactions/sec of Pin to Amine is approximately 3 : 1. Interaction nets are by definition very good in sharing computation thus more efficient than SML in the first two programs. However, interaction nets do not perform well in computations that benefit from sharing data - interacting agents are consumed. Our short term goal is to extend interaction nets with data sharing mechanisms.

7 Conclusions

In this paper we have given an overview of a programming language design and compilation for interaction nets. Experience with the compiler indicates that the system can be used for small programming activities, and we are investigating building a programming environment around this language, specifically containing tools for visualising interaction nets and editing and debugging tools.

Current research in this area is focused on richer programming constructs and higher level languages of interaction that do not burden the programmer with some of the linearity and pattern

matching constrains. The compiler presented in this paper is a first attempt to compile interaction nets, and issues such as compiler optimisations are very much the subject of current research.

There are well-known translations of other rewriting formalisms into interaction nets: the compiler presented in this paper can consequently be used for these systems. Current work is investigating the usefulness of this approach.

Bibliography

[1] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. *A Maude Tutorial*. SRI International, 2000.

[2] M. Fernández and I. Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Proceedings of the International Conference on Principles and Practice of Declarative Programming (PPDP'99)*, volume 1702 of *Lecture Notes in Computer Science*, pages 170–187. Springer-Verlag, September 1999.

[3] Y. Lafont. Interaction nets. In *Proceedings of the 17th ACM Symposium on Principles of Programming Languages (POPL'90)*, pages 95–108. ACM Press, Jan. 1990.

[4] S. Lippi. in^2 : A graphical interpreter for interaction nets. In S. Tison, editor, *Rewriting Techniques and Applications (RTA'02)*, volume 2378 of *Lecture Notes in Computer Science*, pages 380–386. Springer, 2002.

[5] I. Mackie. YALE: Yet another lambda evaluator based on interaction nets. In *Proceedings of the 3rd International Conference on Functional Programming (ICFP'98)*, pages 117–128. ACM Press, 1998.

[6] I. Mackie. Towards a programming language for interaction nets. *Electronic Notes in Theoretical Computer Science*, 127(5):133–151, May 2005.

[7] R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[8] S. Peyton Jones. *Haskell 98 Language and Libraries*. Cambridge University Press, 2003.

[9] B. C. Pierce and D. N. Turner. Pict: A programming language based on the pi-calculus. Technical Report 476, Indiana, 1997.

[10] J. S. Pinto. Sequential and concurrent abstract machines for interaction nets. In J. Tiuryn, editor, *Proceedings of Foundations of Software Science and Computation Structures (FOSSACS)*, volume 1784 of *Lecture Notes in Computer Science*, pages 267–282. Springer-Verlag, 2000.

[11] J. S. Pinto. Parallel evaluation of interaction nets with mpine. In A. Middeldorp, editor, *RTA*, volume 2051 of *Lecture Notes in Computer Science*, pages 353–356. Springer, 2001.

Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences

Leen Lambers[1], **Hartmut Ehrig**[2], **Gabriele Taentzer**[3]

[1] leen@cs.tu-berlin.de, [2] ehrig@cs.tu-berlin.de
Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
[3] taentzer@mathematik.uni-marburg.de
Fachbereich Mathematik und Informatik
Philipps-Universität Marburg, Germany

Abstract: In several rule-based applications using graph transformation as underlying modeling technique the following questions arise: How can one be sure that a specific sequence of rules is applicable (resp. not applicable) on a given graph? Of course, it is possible to use a trial and error strategy to find out the answer to these questions. In this paper however, we will formulate suitable sufficient criteria for applicability and other ones for non-applicability. These criteria can be checked in a static way i.e. without trying to apply the whole rule sequence explicitly. Moreover if a certain criterion is not satisfied, then this is an indication for reasons why rule sequences may or may not be applicable. Consequently it is easier to rephrase critical rule sequences. The results are formulated within the framework of double pushout (DPO) graph transformations with negative application conditions (NACs).

Keywords: graph transformation, analysis, applicability of rules

1 Introduction

When analyzing integrated specifications of rules with control structures [LEMP07, MMT06] for consistency, a considerable amount of rule sequences is to be checked for applicability (resp. non-applicability). Hence, it is not appropriate to check the applicability or non-applicability of each rule sequence explicitly. Therefore static analysis techniques are desirable. Statically means that it is not necessary to use a trial and error strategy and therefore backtracking can be avoided. In the following, we present sufficient criteria for the applicability and for the non-applicability of a rule sequence to a graph. These criteria can be checked in a static way, since they are based mainly on the dependency or independency of rules. Moreover the non-satisfaction of one of the criteria gives a hint to the reason for a rule sequence to be applicable or inapplicable. Applicability criteria have been studied also in [VL07] for simple digraphs using matrix graph grammars.

As example we consider a simple Mutual Exclusion Algorithm implemented by graph transformation rules with a start graph G and type graph presented in Fig. 1. The algorithm enables two processes to access a resource according to the mutual exclusion principle. The purpose of this algorithm is to control the usage of the resource such that at most one process holds the

resource at time (safety property). Furthermore, if a process demonstrates a request for the resource it should be served eventually (liveness property). In the following, we will check safety (resp. liveness) by checking if certain rule sequences are not applicable (resp. applicable) on the start graph.

The paper is structured as follows: Section 2 presents preliminaries. At first we repeat the main definitions for rule-based transformations with means of double pushout (DPO) graph transformations [CMR+97] with negative application conditions (NACs) [HHT96]. Then we define asymmetric dependency and independency on the level of transformations. Therefrom we can deduce the concept of asymmetric independency for rules. In Section 3 and 4 we define sufficient criteria for applicability and non-applicability of rule sequences. Section 5 takes up the example again and shows how to check the criteria presented in Section 3 and 4 with some support of the graph transformation tool AGG [Tae04].

2 Independency and Dependency of Rules

2.1 Graph Transformation with NACs

We repeat the basic definitions for double pushout graph transformation with negative application conditions (NACs). A graph rule holding a NAC n can be applied on a graph G only if the forbidden structure expressed by n is not present in G.

Definition 1 (graph, graph morphism, rule) A *graph* $G = (G_E, G_V, s, t)$ consists of a set G_E of edges, a set G_V of vertices and two mappings $s, t : G_E \to G_V$, assigning to each edge $e \in G_E$ a source $q = s(e) \in G_V$ and target $z = t(e) \in G_V$. A *graph morphism* (short *morphism*) $f : G_1 \to G_2$ between two graphs $G_i = (G_{i,E}, G_{i,V}, s_i, t_i)$, $(i = 1, 2)$ is a pair $f = (f_E : G_{E,1} \to G_{E,2}, f_V : G_{V,1} \to G_{V,2})$ of mappings, such that $f_V \circ s_1 = s_2 \circ f_E$ and $f_V \circ t_1 = t_2 \circ f_E$. A morphism $f : G_1 \to G_2$ is injective (resp.surjective) if f_V and f_E are injective (resp. surjective) mappings. A graph transformation *rule* $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ consists of a rule name p and a pair of injective morphisms $l : K \to L$ and $r : K \to R$. The graphs L, K and R are called the left-hand side (LHS), the interface, and the right-hand side (RHS) of p, respectively.

Definition 2 (rule and transformation with NACs, applicability of rule with NACs)

- A *negative application condition* or $NAC(n)$ *on* p for a rule $p : L \overset{l}{\leftarrow} K \overset{r}{\rightarrow} R$ (l,r injective) is an arbitrary morphism $n : L \to N$. A morphism $g : L \to G$ *satisfies* $NAC(n)$ on L, written $g \models NAC(n)$, if and only if $\nexists\, q : N \to G$ injective such that $q \circ n = g$.

  $$\begin{array}{ccc} L & \xrightarrow{n} & N \\ {\scriptstyle g}\downarrow & \overset{X}{\swarrow}{\scriptstyle q} & \\ G & & \end{array}$$

  A set of NACs on p is denoted by $NAC_p = \{NAC(n_i) | i \in I\}$. A morphism $g : L \to G$ *satisfies* NAC_p if and only if g satisfies all single NACs on p i.e. $g \models NAC(n_i)\ \forall i \in I$.

Minimal Conflicts

first \ second	1: req...	2: tak...	3: rele...	4: giv...	5: nex...
1: requestRule	1	0	0	0	1
2: takeRule	0	3	0	0	0
3: releaseRule	0	0	1	0	0
4: giveRule	0	0	0	3	0
5: nextProc	0	0	0	0	3

Minimal Dependencies

first \ second	1: req...	2: tak...	3: rele...	4: giv...	5: nex...
1: requestRule	0	1	0	0	0
2: takeRule	0	0	1	0	1
3: releaseRule	1	0	0	1	0
4: giveRule	0	1	0	0	2
5: nextProc	0	1	0	0	2

Figure 1: Start graph G, graph H and type graph of Mutual Exclusion - Rules *requestRes*,*takeRes*,*releaseRes*,*giveRes* and *nextProc* - Dependency and Conflict Matrix in AGG

- A *rule* (p, NAC_p) *with NACs* is a rule with a set of NACs on p.
- A *direct transformation* $G \overset{p,g}{\Rightarrow} H$ via a rule $p : L \leftarrow K \rightarrow R$ with NAC_p and a match $g : L \rightarrow G$ consists of the double pushout [CMR+97] (DPO)

$$\begin{array}{ccccc} L & \leftarrow & K & \rightarrow & R \\ {\scriptstyle g}\downarrow & & \downarrow & & \downarrow{\scriptstyle h} \\ G & \leftarrow & D & \rightarrow & H \end{array}$$

  where g satisfies NAC_p, written $g \models NAC_p$. Since pushouts along injective morphisms always exist, the DPO can be constructed if the pushout complement of $K \rightarrow L \rightarrow G$ exists. If so, we say that the match g satisfies the gluing condition of rule p. If there exists a morphism $g : L \rightarrow G$ which satisfies the gluing condition and $g \models NAC_p$ we say that rule p is *applicable* on G via the match g.

Example 1 *Consider the graph transformation rules of the Mutual Exclusion Algorithm presented in Fig. 1. Rule* requestRes *expresses a request of a process to access the resource if this process has not requested the resource yet or is using the resource already. Rule* takeRes *enables a process to start using the resource if this process possesses a token.* releaseRes *can release the resource again and* giveRes *passes the token to the other process after releasing. Finally rule* nextProc *can pass a token from one process to the other one as long as the first one has not expressed a request.*

In [LEOP07] it is proven that to each rule p with NAC_p there exists an inverse rule p^{-1} with $NAC_{p^{-1}}$ such that each transformation can be inverted. The following definition shows how to construct from NAC_p on rule p equivalent NACs $NAC_{p^{-1}}$ on the inverse rule p^{-1} [EEHP04]:

Definition 3 (construction of NACs on inverse rule) For each $NAC(n_i)$ with $n_i : L \rightarrow N_i$ on $p = (L \leftarrow K \rightarrow R)$, the equivalent NAC $R_p(NAC(n_i))$ on $p^{-1} = (R \leftarrow K \rightarrow L)$ is defined in the following way:

$$\begin{array}{ccccc} L & \leftarrow & K & \rightarrow & R \\ {\scriptstyle n_i}\downarrow & (1) & \downarrow & (2) & \downarrow{\scriptstyle n'_i} \\ N_i & \leftarrow & Z & \rightarrow & N'_i \end{array}$$

- If the pair $(K \rightarrow L, L \rightarrow N_i)$ has a pushout complement, we construct $(K \rightarrow Z, Z \rightarrow N_i)$ as the pushout complement (1). Then we construct pushout (2) with the morphism $n'_i : R \rightarrow N'_i$. Now we define $R_p(NAC(n_i)) = NAC(n'_i)$.
- If the pair $(K \rightarrow L, L \rightarrow N_i)$ does not have a pushout complement, we define $R_p(NAC(n_i)) = true$.

For each set of NACs on p, $NAC_p = \cup_{i \in I} NAC(n_i)$ we define the following set of NACs on p^{-1}: $NAC_{p^{-1}} = R_p(NAC_p) = \cup_{i \in I'} R_p(NAC(n_i))$ with $i \in I'$ if and only if the pair $(K \rightarrow L, L \rightarrow N_i)$ has a pushout complement.

With means of this construction we can formulate the following fact [EEHP04, LEOP07]:

Fact 1 (inverse direct transformation with NACs) *For each direct transformation with NACs $G \Rightarrow H$ via a rule $p : L \leftarrow K \rightarrow R$ with NAC_p a set of NACs on p, $H \Rightarrow G$ is a direct transformation with NACs via the inverse rule p^{-1} with $NAC_{p^{-1}}$.*

Example 2 *Consider rule* requestRes. *The inverse rule of* requestRes *deletes the* request *edge between a process and the resource if there is no other* request *edge nor a* heldby *edge between this process and the resource.*

2.2 Independency and Dependency of Rules

The (non-)applicability of a rule sequence on a graph G is affected by dependencies and independencies between the rules in the rule sequence. In particular it is affected by asymmetric dependencies as explained in this section.

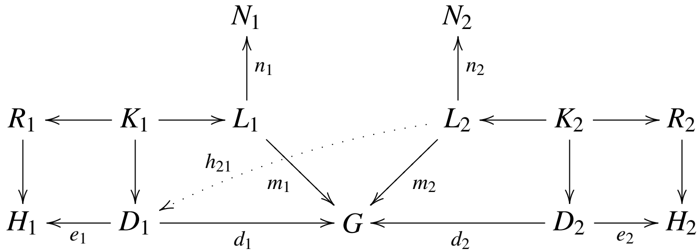
Two direct transformations are in conflict if they are not parallel independent. In [LEO06] a Conflict Characterization is given in which different reasons for conflicting transformations become clear. It is possible that one transformation deletes (resp. produces) a structure which is used (resp. forbidden) by the other one and the other way round. In particular we can deduce that two direct transformations are in conflict if *at least* one transformation depends on the other one. This asymmetric parallel dependency and on the contrary asymmetric parallel independency are expressed by the following definition.

Definition 4 (asymmetrically parallel dependent and independent transformations) A direct transformation $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ with NAC_{r_2} is *asymmetrically parallel dependent* on $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ with NAC_{r_1} if:

1. $\nexists h_{21} : L_2 \rightarrow D_1 : d_1 \circ h_{21} = m_2$ *(delete-use-conflict)*
   OR

2. there exists a unique $h_{21} : L_2 \rightarrow D_1 : d_1 \circ h_{21} = m_2$, but $e_1 \circ h_{21} \not\models NAC_{r_2}$ *(produce-forbid-conflict)*.

A direct transformation $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ with NAC_{r_2} is *asymmetrically parallel independent* on $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ with NAC_{r_1} if $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ is not asymmetrically parallel dependent on $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$. This means in particular that $\exists h_{21} : L_2 \rightarrow D_1 : d_1 \circ h_{21} = m_2$ such that $e_1 \circ h_{21} \models NAC_{r_2}$.



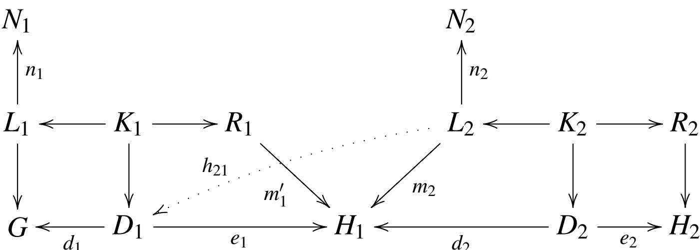
If a sequence of two direct transformations is not sequentially independent this is because either the second transformation depends on the first one or the other way round. The case in which the second transformation depends (resp. is independent) on the first one is described by asymmetric sequential dependency (resp. independency) as in the following definition. This case occurs in particular when the first transformation produces a structure which is used by the second one or the first transformation deletes a structure forbidden by the second one. In other words the first transformation triggers the second one.

Definition 5 (asymmetrically sequential dependent and independent transformations) A direct transformation $H_1 \stackrel{(r_2,m_2)}{\Longrightarrow} H_2$ with NAC_{r_2} is *asymmetrically sequential dependent* on $G \stackrel{(r_1,m_1)}{\Longrightarrow} H_1$ with NAC_{r_1} if

1. $\nexists h_{21} : L_2 \to D_1 : e_1 \circ h_{21} = m_2$ *(produce-use-dependency)*
   OR

2. there exists a unique $h_{21} : L_2 \to D_1 : e_1 \circ h_{21} = m_2$, but $d_1 \circ h_{21} \not\models NAC_{r_2}$ *(delete-forbid-dependency).*

A direct transformation $H_1 \stackrel{(r_2,m_2)}{\Longrightarrow} H_2$ with NAC_{r_2} is *asymmetrically sequential independent* on $G \stackrel{(r_1,m_1)}{\Longrightarrow} H_1$ with NAC_{r_1} if $H_1 \stackrel{(r_2,m_2)}{\Longrightarrow} H_2$ with NAC_{r_2} is not asymmetrically sequential dependent on $G \stackrel{(r_1,m_1)}{\Longrightarrow} H_1$ with NAC_{r_1}. This means in particular that $\exists h_{21} : L_2 \to D_1 : e_1 \circ h_{21} = m_2$ such that $d_1 \circ h_{21} \models NAC_{r_2}$.



Now we can define asymmetric parallel (resp. sequential) independency for rules by demanding that the corresponding transformations are asymmetrically independent. Analogously it is possible to define asymmetric parallel (resp. sequential) dependency for rules.

Definition 6 (asymmetrically parallel independent rules) The rule r_2 is *asymmetrically parallel independent* on r_1 if every transformation $G \stackrel{(r_2,m_2)}{\Longrightarrow} H_2$ via r_2 with NAC_{r_2} is asymmetrically parallel independent on any other transformation $G \stackrel{(r_1,m_1)}{\Longrightarrow} H_1$ via r_1 with NAC_{r_1}.

Definition 7 (asymmetrically sequential independent rules) A rule r_2 is *asymmetrically sequential independent* on r_1 if every transformation $H_1 \stackrel{(r_2,m_2)}{\Longrightarrow} H_2$ via r_2 with NAC_{r_2} is asymmetrically sequential independent on any other transformation $G \stackrel{(r_1,m_1)}{\Longrightarrow} H_1$ via r_1 with NAC_{r_1} .

Example 3 *Rule* nextProc *is asymmetric parallel dependent on* requestRes, *since rule* requestRes *produces a* request *edge which is forbidden by rule* nextProc. *On the contrary, rule* requestRes *is asymmetric parallel independent on* nextProc, *since rule* nextProc *neither deletes anything what can be used by* requestRes *nor produces anything forbidden by* requestRes. *Rule* requestRes *is asymmetric sequentially dependent on rule* releaseRes, *since* releaseRes *deletes a* heldby *edge which is forbidden by* requestRes. *On the contrary, rule* releaseRes *is asymmetric sequentially independent on rule* requestRes, *since* requestRes *neither produces anything what can be used by* releaseRes *nor deletes anything forbidden by* requestRes.

For the criteria defined in the next section we need a special case of asymmetric sequential dependency for the case without NACs. It is possible namely that a rule r_1 produces everything which is needed by r_2 regardless of what is already present in the corresponding transformations.

Definition 8 (purely sequential dependent rules) A rule $r_2 : L_2 \leftarrow K_2 \rightarrow R_2$ is *purely sequential dependent* on $r_1 : L_1 \leftarrow K_1 \rightarrow R_1$ if r_2 is a rule without NACs and there exists an injective morphism $l_{21} : L_2 \rightarrow R_1$.

Remark 1 *r_2 is asymmetrically sequential dependent on r_1, if r_2 is purely sequential dependent on r_1 and the following mild assumptions are satisfied: a morphism $k_{21} : L_2 \rightarrow K_1$ does not exist such that $r_1 \circ k_{21} = l_{21}$, r_2 is non-deleting on nodes and $id : R_1 \rightarrow R_1 \models NAC_{r_1^{-1}}$.*

Example 4 *Rule* releaseRes *is purely sequential dependent on rule* takeRes, *because its LHS can be embedded completely into the RHS of* takeRes. *Rule* takeRes *is asymmetric sequentially, but not purely dependent on rule* requestRes, *since* requestRes *does not produce a* token *edge.*

We can analogously derive from the usual definition of sequential (resp. parallel) independence of transformations with NACs [LEO06] the definition for sequential (resp. parallel) independency on rules by demanding that all existing transformations via these rules are sequentially (resp. parallel) independent. Recall that for each pair of parallel independent transformations with NACs $H_1 \overset{r_1,m_1}{\Leftarrow} G \overset{r_2,m_2}{\Rightarrow} H_2$, there are an object G' and direct transformations $H_1 \overset{r_2,m_2'}{\Rightarrow} G'$ and $H_2 \overset{r_1,m_1'}{\Rightarrow} G'$ such that $G \overset{r_1,m_1}{\Rightarrow} H_1 \overset{r_2,m_2'}{\Rightarrow} G'$ and $G \overset{r_2,m_2}{\Rightarrow} H_2 \overset{r_1,m_1'}{\Rightarrow} G'$ are sequentially independent. The transformations can thus be performed in any order with the same result (Local Church-Rosser Theorem with NACs [LEO06]). Note that we have the following relationship between parallel and sequential independency: $G \overset{r_1}{\Rightarrow} H_1 \overset{r_2}{\Rightarrow} H_2$ are sequentially independent iff $G \overset{r_1^{-1}}{\Leftarrow} H_1 \overset{r_2}{\Rightarrow} H_2$ are parallel independent. In the next section we need sequential independency of a rule pair in order to be able to switch adjacent rules in a rule sequence.

Definition 9 (parallel and sequential independent rules) Rules r_1 and r_2 are *parallel independent* if each pair of transformations $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ via r_1 with NAC_{r_1} and $G \overset{(r_2,m_2)}{\Longrightarrow} H_2$ via r_2 with NAC_{r_2} is parallel independent. The pair of rules (r_1, r_2) is *sequentially independent* if each sequence of transformations $G \overset{(r_1,m_1)}{\Longrightarrow} H_1$ via r_1 with NAC_{r_1} and $H_1 \overset{(r_2,m_2)}{\Longrightarrow} G'$ via r_2 with NAC_{r_2} is sequentially independent.

Remark 2 *Note that the following correspondences between asymmetric parallel (resp. sequential) independency and parallel (resp. sequential) independency exists. Rules r_1 and r_2 are parallel independent if and only if r_1 is asymmetrically parallel independent of r_2 and r_2 is asymmetrically parallel independent of r_1. The rule pair (r_1, r_2) is sequentially independent if and only if r_2 is asymmetrically sequential independent on r_1 and r_1^{-1} is asymmetrically sequential independent on r_2^{-1}.*

Example 5 *The rule pair* (requestRes,nextProc) *is sequentially independent. Note that the NAC of rule* nextProc *forbids a process to shift the token if this process expressed a request. Thus whenever* (requestRes,nextProc) *can be applied in this order the request is expressed by the process to which the token is shifted. This is equivalent to first shifting the token to this process which then expresses a request.*

3 Applicability of Rule Sequences

3.1 Applicability Criteria

Let $s : r_1 r_2 \dots r_n$ be a sequence of n rules and G_0 a graph on which this sequence should be applied. The criteria defined in the following definition guarantee this applicability. The initialization criterion is trivial, since it just requires the first rule being applicable to graph G_0. The no node-deleting rules criterion avoids dangling edges. The third criterion ensures that the applicability of a rule r_i is not impeded by one of the predecessor rules r_j of r_i. Criterion 4a will be satisfied if rule r_i is purely sequential dependent from a rule r_j occurring somewhere before r_i in the sequence s. In this case r_j triggers the applicability of r_i regardless of what is present already in the start graph G_0. As soon as the sequential dependencies are not pure though this criterion is not satisfiable. Therefore we have also a more general criterion 4b. It ensures the applicability of a rule r_i needing some subgraph of a direct predecessor rule together with parts of the start graph G_0. This is expressed by the fact that a concurrent rule r_c of r_{i-1} and r_i exists which is applicable on the start graph G_0. The construction of a concurrent rule with NACs is explained in [LEOP07]. The correctness of the criteria is described in Theorem 1 which is proven in [LET08].

Definition 10 (applicability criteria) Given a sequence $s : r_1 r_2 \dots r_n$ of n rules and a graph G_0. Then we define the following applicability criteria for s on G_0:

1. r_1 is applicable on G_0 via the injective match $m_1 : L_1 \to G_0$ *(initialization)*
2. each rule occurring in s is non-deleting on nodes *(no node-deleting rules)*
3. $\forall r_i, r_j$ in s with $1 \leq j < i \leq n$, r_i is asymmetrically parallel independent on r_j *(no impeding predecessors)*
4. $\forall r_i$ in s with $1 < i \leq n$ which are not applicable on G_0 via an injective match
   (a) there exists a rule r_j in s which is applicable via an injective match on G_0 with $1 \leq j < i \leq n$ and r_i is purely sequential dependent on r_j *(pure enabling predecessor)*,

which especially means that r_i has no NACs
OR

(b) there exists a concurrent rule r_c of r_{i-1} and r_i such that r_c is applicable via an injective match on G_0 and r_c is asymmetrically parallel independent on r_j for all $j < (i-1)$ and r_j is asymmetrically parallel independent on r_c for all $i < j \leq n$. *(direct enabling predecessor)*

Theorem 1 (correctness of applicability criteria) *Given $s : r_1 r_2 \ldots r_n$ a sequence of n rules and a graph G_0. If the criteria in Def. 10 are satisfied for rule sequence s and graph G_0, then this rule sequence is applicable on G_0 with injective matching i.e. there exists a graph transformation $G_0 \overset{r_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ with injective matching.*

3.2 Applicability of Summarized Rule Sequences

The 4th criterion in Def. 10 will be satisfied only, if the rule r_i is purely sequential dependent from one single rule r_j occurring before r_i in the sequence or if it is asymmetrically sequential dependent on the rule r_{i-1}. In some transformations though a rule needs not only a subgraph from one single predecessor, but from several ones. For these cases the criteria could then be satisfied by a sequence in which exactly these rules are summarized to a concurrent rule. Note that the correctness of the following theorem is proven in [LET08].

Theorem 2 (summarized rule sequences) *Given $s : r_1 r_2 \ldots r_n$ a sequence of n rules and a graph G_0. If the criteria in Def. 10 are satisfied for a rule sequence $s' : r'_1 r'_2 \ldots r'_m$ with $m < n$ in which neighbored rules in s can be summarized by a concurrent rule, then the original rule sequence s is applicable on G_0 with injective matching i.e. there exists a graph transformation $G_0 \overset{r_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{r_n}{\Rightarrow} G_n$ with injective matching.*

3.3 Applicability of Shift-Equivalent Rule Sequences

If it is not possible to satisfy criterion 4b in Def. 10 for a rule sequence $s : r_1 r_2 \ldots r_n$, then it is still possible to exploit shift-equivalence. This is because criterion 4b could be satisfiable for a rule sequence s' in which a normal predecessor is shifted to be a direct one. Note that a rule r_j in s can be switched with a rule r_{j+1} only if the pairs of rules (r_j, r_{j+1}) and (r_{j+1}, r_j) is sequentially independent. If the criteria then hold though for rule sequence s' in which some rules have been shifted, they hold also for the original rule sequence s.

Definition 11 (shift-equivalent rule sequences) A rule sequence s' is *shift-equivalent* with a rule sequence $s : r_1 r_2 \ldots r_m$ if s' can be obtained by switching rules r_j with r_{j+1} and the switching is allowed only if (r_j, r_{j+1}) and (r_{j+1}, r_j) are sequentially independent according to Def. 9.

Theorem 3 (checking shift-equivalent rule sequences) *If the criteria in Def. 10 are satisfied for a rule sequence $s : r_1 r_2 \ldots r_n$ and a graph G_0, then all shift-equivalent rule sequences as defined in Def. 11 are applicable on G_0 with injective matching as well with the same result.*

Proof. This follows directly from Def. 11, the Local Church-Rosser Theorem with NACs [LEO06],

Def. 9 and Theorem 1. □

Remark 3 *Note that a somewhat weaker version of this theorem holds as well. Namely, suppose that we want to prove applicability of a rule sequence $s : r_1r_2 \ldots r_n$ to a graph G_0. Then it is sufficient to show the satisfaction of the criteria in Def. 10 for a rule sequence s' which can be deduced from s by switching forward rule r_{i+1} with r_i only if rule pair (r_{i+1}, r_i) is sequentially independent.*

4 Non-Applicability of Rule Sequences

Let $s : r_1r_2 \ldots r_n$ be a sequence of n rules and G_0 a graph. The satisfaction of the following criteria for s and G_0 guarantee that the sequence s will not be applicable on G_0. Criterion 1 is trivial, since it just requires the first rule being non-applicable to graph G_0. Criterion 2 checks if predecessors for a rule r_i which is not applicable already on G_0 are present in the sequence such that they can trigger the applicability of r_i. If not, r_i will not be applicable and therefore neither the rule sequence will be applicable. Note that the correctness of the following criteria is proven in [LET08].

Definition 12 (non-applicability criteria) Given $s : r_1r_2 \ldots r_n$ a sequence of n rules and a graph G_0. Then we define the following non-applicability criteria for s on G_0:

1. r_1 is not applicable on G_0 *(initialization error)*
   OR

2. $\exists r_i$ in s with $1 < i \leq n$ such that r_i is not applicable on G_0 but for all rules r_j in s with $1 \leq j < i \leq n$, r_i is asymmetrically sequential independent on r_j *no enabling predecessor*.

Theorem 4 (correctness of non-applicability criteria) *Given a sequence $s : r_1r_2 \ldots r_n$ of n rules and a graph G_0. If the criteria in Def. 12 are satisfied for rule sequence s and graph G_0, then this rule sequence is not applicable on G_0 i.e. there exists no graph transformation $G_0 \overset{r_1}{\Rightarrow} G_1 \ldots G_{n-1} \overset{r_n}{\Rightarrow} G_n$.*

Analogously to Theorem 3 we can formulate the following theorem expressing that shift-equivalent-rule sequences are not applicable to a graph G_0 if one of the sequences satisfies the non-applicability criteria.

Theorem 5 (checking shift-equivalent rule sequences) *If the criteria in Def. 12 are satisfied for a rule sequence $s : r_1r_2 \ldots r_n$ and a graph G_0, then all shift-equivalent rule sequences as defined in Def. 11 are not applicable on G_0 either.*

Proof. This follows directly from Def. 11, the Local Church-Rosser Theorem with NACs [LEO06], Def. 9 and Theorem 4. □

Remark 4 *Note that a somewhat weaker version of this theorem holds as well. Namely, suppose that we want to prove non-applicability of a rule sequence $s : r_1r_2 \ldots r_n$ to a graph G_0. Then it is*

sufficient to show the satisfaction of the criteria in Def. 12 for a rule sequence s' which can be deduced from s by switching forward rule r_{i+1} with r_i only if rule pair (r_i, r_{i+1}) is sequentially independent.

5 Checking the Criteria

5.1 Checking for Sequential and Parallel Dependency of Rules in AGG

To check asymmetric sequential and parallel dependency of rules, we compute all corresponding critical pairs by AGG [Tae04]. A critical pair represents the parallel (resp. sequential) dependency of rules in a minimal context. The minimal conflicts (i.e. asymmetric parallel dependencies) are represented in a conflict matrix. The minimal dependencies (i.e. asymmetric sequential dependencies) are represented in a dependency matrix. The entry numbers within these matrices indicate how many minimal conflicts and dependencies, resp. have been found. For the example they are shown in Fig. 1.More precisely, entry (r_j, r_i) (row, column) in the conflict matrix in AGG describes all $G \overset{(r_i,m_i)}{\Rightarrow} H_i$ which are asymmetrically parallel dependent on $G \overset{(r_j,m_j)}{\Rightarrow} H_j$ in a minimal context. Entry (r_j, r_i) in the dependency matrix in AGG describes all $G \overset{(r_j,m_j)}{\Rightarrow} H_j \overset{(r_i,m_i)}{\Rightarrow} G'$ such that the second transformation is asymmetrically sequential dependent on the first one in a minimal context. Note that it is not possible yet to check with AGG if a pair of rules (r_j, r_i) is sequentially independent as defined in Def.9. It is part of current work though to enrich the dependency matrix with more information and thus enable this possibility.

5.2 Checking Applicability Criteria on Mutual Exclusion

We check applicability (i.e. liveness) of the following rule sequences:

requestRes, takeRes, releaseRes should be applicable to the right process in the graph G shown in Figure 1. We thus check for this rule sequence and graph G that the applicability criteria in Def. 10 are satisfied.

1. Rule *requestRes* is applicable to G.
2. Rule *takeRes* is asymmetrically parallel independent of rule *requestRes*. Furthermore, rule *releaseRes* is asymmetrically parallel independent of *requestRes* as well as *takeRes*.
3. Rules *takeRes, releaseRes, giveRes* are not applicable to G. Thus, we have to show that their application is enabled by the rule applications performed before. Rule *takeRes* is asymmetrically sequential dependent on *requestRes*, and the concurrent rule of *takeRes* and *requestRes* is applicable on G. It expresses how a resource can be requested and taken in one step. *requestRes* is thus a direct enabling predecessor for *takeRes*. Moreover rule *releaseRes* is purely sequential dependent on rule *takeRes* and therefore *takeRes* is a pure enabling predecessor for *releaseRes*.

requestRes, takeRes, releaseRes, giveRes is a slightly longer sequence and should still be applicable to G. It is not possible though to fulfill in particular criterion 4 in Def. 10 for this

sequence. However, it is still possible to fulfill the applicability criteria for a summarized sequence as proven in Theorem 2. In this case, it is possible to satisfy the criteria for the summarized sequence only. It consists of one concurrent rule *requestTakeReleaseGiveRes* which is equal to the rule *nextProc*. *nextProc* is applicable on G and therefore the original sequence is applicable on G as well. Thus sometimes an applicable rule sequence as given in this example might be hard to detect. This is because *giveRes* needs some subgraph of the start graph G which is not needed by the other rules and in addition all rules (except for the first one) are asymmetrically sequential dependent of the previous rule.

5.3 Checking Non-Applicability Criteria on Mutual Exclusion

We check non-applicability (i.e. safety) of the following rule sequences:

requestRes, requestRes, nextProc specifies a request of both processes and then a token transfer. This sequence should not be applicable on the start graph G in Fig. 1, since before transferring a token one of the requests should be processed. We check the non-applicability criteria in Def. 12 for this rule sequence on G.

1. *requestRes* is applicable on the start graph G.
2. *nextProc* is not applicable on the start graph G and is sequentially independent on *requestRes*. Therefore the second criterion is fulfilled and there is no enabling predecessor.

requestRes, giveRes specifies a request and then a token shift because the resource has been released. This sequence should not be applicable to the start graph G in Fig. 1, since the resource is still unused. It is easy to verify that the criteria in Def. 12 are fulfilled. Moreover rule pair *(requestRes,giveRes)* and *(giveRes,requestRes)* are sequentially independent. Therefore due to Theorem 5 we can conclude directly that sequence *giveRes,requestRes* is not applicable either on G.

takeRes, takeRes specifies a take of the resource by both processes simultaneously. This sequence should not be applicable on the graph H in Figure 1 in which both processes request the resource. Thus we check the non-applicability criteria in Def. 12 for this sequence and graph H.

1. The first rule *takeRes* is applicable to graph H on the right process.
2. The second rule is again *takeRes* and we just mentioned that it is applicable on the right process in H. We want to check though for safety reasons that it is impossible to apply the second *takeRes* on the other (i.e. *left*) process simultaneously. A matching of *takeRes* on the left process into H does not exist. For this kind of matching criterion 2 holds since in addition rule *takeRes* is asymmetrically sequentially independent from itself. This means in particular that the application of the first *takeRes* on the right process does not enable the application of the second *takeRes* on the left process and hence such a rule application on H is not possible.

This example demonstrates that sometimes it is necessary to include information about the matches for the rule sequence to be checked for safety. Note moreover that if we would have taken a longer sequence consisting of *requestRes, requestRes, takeRes* and *takeRes*, we could not have checked the non-applicability of that sequence on the start graph G by Def. 12, since rule *takeRes* is asymmetrically sequentially dependent on rule *requestRes*. Thus it is crucial in this case to select the kernel sequence where the problem is conjectured and in particular this shows us that our criteria are sufficient but not necessary.

6 Conclusion and Future Work

In this paper sufficient criteria are formulated for the applicability and non-applicability of rule sequences. These criteria can be checked in a static way, i.e. without applying the rule sequences. Future work is concerned with formulating criteria for rules with attributes, further optimization of the criteria, efficiently checking the satisfaction of the criteria and implementing applicability checks in AGG. Furthermore, we like to evaluate the criteria in larger case studies such as [LEMP07, MMT06].

Bibliography

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. Chapter 3, pp. 163–245. World Scientific, 1997.

[EEHP04] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann. Constraints and Application Conditions: From Graphs to High-Level Structures. In Parisi-Presicce et al. (eds.), *Proc. 2nd Int. Conference on Graph Transformation (ICGT'04)*. LNCS 3256, pp. 287–303. Springer, Rome, Italy, October 2004.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph Grammars with Negative Application Conditions. *Special issue of Fundamenta Informaticae* 26(3,4):287–313, 1996.

[LEMP07] L. Lambers, H. Ehrig, L. Mariani, M. Pezzè. Iterative Model-driven Development of Adaptable Service-Based Applications. In *proceedings of the International Conference on Automated Software Engineering*. 2007.

[LEO06] L. Lambers, H. Ehrig, F. Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Proc. Third International Conference on Graph Transformation (ICGT'06)*. LNCS 4178, pp. 61–76. Springer, Natal, Brazil, September 2006.

[LEOP07] L. Lambers, H. Ehrig, F. Orejas, U. Prange. Parallelism and Concurrency in Adhesive High-Level Replacement Systems with Negative Application Conditions. In Ehrig et al. (eds.), *Workshop on Applied and Computational Category Theory (ACCAT'07)*. Elsevier Science, 2007.

[LET08] L. Lambers, H. Ehrig, G. Taentzer. Sufficient Criteria for Applicability and Non-Applicability of Rule Sequences. Technical report, TU Berlin, 2008.

[MMT06] K. Mehner, M. Monga, G. Taentzer. Interaction Analysis in Aspect-Oriented Models. In *Proc. 14th IEEE International Requirements Engineering Conference*. Pp. 66–75. IEEE Computer Society, Minneapolis, Minnesota, USA, September 2006.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*. Pp. 446 – 456. LNCS 3062, Springer, 2004. AGG-Homepage:http://tfs.cs.tu-berlin.de/agg

[VL07] P. Velasco, J. de Lara. Using Matrix Graph Grammars for the Analysis of Behavioural Specifications: Sequential and Parallel Independence. In *Jornadas de Programacion y Lenguajes (PROLE2007)*. 2007.

Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars

Florian Brieler[†] **and Mark Minas**[†]

[†] Institute for Software Technology
Computer Science Department
Universität der Bundeswehr München
85577 Neubiberg, Germany
{florian.brieler|mark.minas}@unibw.de

Abstract: Sketching, i.e., drawing diagrams by hand and directly on the screen, is gaining popularity, as it is a comfortable and natural way to create and edit diagrams. Hand drawing is inherently imprecise, and often sloppy. As a consequence, when processing hand drawn diagrams with a computer, ambiguities arise: it is not always clear what part of the drawing is meant to represent what component. Resolution of these ambiguities is the main issue of sketching. Ambiguity can only be solved by exploring the context of ambiguous components. This paper describes ambiguity resolution by syntax analysis in DiaGen, a generic framework for generating diagram editors. Such editors support free-hand editing (which is closely related to sketching), and allow for analyzing the created diagrams based on a hypergraph grammar. Our approach adds support for sketching to the generated editors. In order to resolve the ambiguities in sketched diagrams, DiaGen's diagram analysis based on graph parsing is used. The necessary modifications to DiaGen and its graph parser in particular are discussed.

Keywords: Sketching, Ambiguity Resolution, Hypergraph, Parser

1 Introduction

Nowadays diagram languages like the UML are very popular, and there is a lot of work going on to model applications and systems (or, at least, part of them) using diagram languages, instead of coding them traditionally with textual languages. Diagrams are more expressive in terms of exposing structure and coherence of the modeled system; the used diagram language can be domain-dependent, thus better focusing on the problem in question; and diagrams are – for some problems – much more suited, e.g., for expressing graph-like structures like Petri nets or class diagrams.

Tool support for processing of diagrams has evolved in the last years, with many approaches available, e.g., *Fujaba* [FNTZ00], *AToM*^{3} [LV02], DiaMeta [Min06] and DiaGen [Min02]. Among these, the specification of syntax and semantics of a diagram language is either given by metamodels or by graph grammars.

However, creating and editing of diagrams using such tools is not very natural. Diagram components have to be selected from some graphical widget like a list, and placed on the canvas

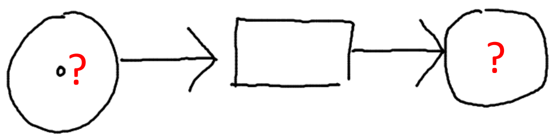


Figure 1: A simple Petri net with two ambiguities, indicated by question marks.

by one or more mouse clicks. Instead, sketching, i.e., drawing diagrams by hand on the screen, is similar to pen and paper and is much more natural, because it does not require complex user interfaces.

This paper is concerned with ambiguity resolution, which is the main issue of sketching. Ambiguities arise because drawing by hand is inherently imprecise and sloppy. Fig. 1 shows a practical example of a simple Petri net, which is also used as running example in this paper. Places and tokens are drawn as circles, transitions are drawn as rectangles. The two question marks indicate ambiguities. On the left it is unclear for the two circles whether they represent places or tokens. The component on the right could be a circle or a very deformed rectangle; possible interpretations are place, token, or transition, resp. By looking at the context of the ambiguous components, it becomes obvious that we have a place containing a token on the left (a place containing a place is not meaningful, and so is a token containing another component), and another place on the right (as the arrow connects a transition and a place, but not a transition and a token or another transition). Apparently, context of an ambiguous component has to be exploited to decide for the correct interpretation. This requires diagram analysis.

In [BM08] we present a comprehensive approach to sketching enabled diagram editors, including user interface, recognition of components in the drawing, support for text, and basic ideas for diagram analysis. There are many approaches to sketching, but most of them do not exploit the power of grammar-based approaches for ambiguity resolution. For diagram analysis we have decided for DIAGEN,

- because it supports *free-hand editing*, which is the basis of sketching,
- because it uses graph grammars for its visual language parser, which are very powerful for ambiguity resolution (discussed in this paper),
- and because DIAGEN is generic (it can be customized to any diagram language by a specification of the language). The approach in [BM08] is designed to be generic as well.

In the present paper we describe in detail how we employ DIAGEN for diagram analysis in order to resolve ambiguities. The main idea is that for each component it must be decided which of its possible interpretations fits best to the other components.

This paper is organized as follows. Sec. 2 explains DIAGEN by the example of Petri nets, and outlines the basic idea to support sketching. Sec. 3 and Sec. 4 describe the necessary modifications to DIAGEN. Sec. 5 discusses related work. Sec. 6 gives a brief summary and describes further work.

2 Hypergraph Grammars and Parsing in DiaGen

A *diagram* is a set of diagram *components*. Each component has one or more *attachment areas*, i.e., areas where the component can be related to other components. Relationships between attachment areas depend on spatial placement. A relation is detected if two attachment areas overlap or are close to each other (since sketching is imprecise, it is not meaningful to require precise spatial placement of components). For example, places and transitions in Petri nets have one attachment area each (their full shape). Arrows have two attachment areas (their head and their tail), and can be related to places and transitions if its head or tail is close. There may be relations which are not required for a diagram type, e.g., overlapping arrow heads in Petri nets.

DiaGen [Min02] is a generic editor generator that generates diagram editors from language-dependent specifications. Each specification describes one diagram language, and defines aspects like diagram components and attachments areas, desired relationships, reduction rules, grammar rules, and attributes for parsing (see below). Hypergraphs are used as internal models required for diagram processing. Each component is represented as a single hyperedge visiting as many unique nodes as the component has attachment areas. If a component visits more than one node, its tentacles (connecting an edge with its visited nodes) are numbered in order to be identifiable. Hyperedges are labeled; the label depends on the type of the respective diagram component. For Petri nets, we have four different types of components: places, transitions, arrows and tokens. Hence, hyperedges are labeled with `c_place`, `c_trans`, `c_token`, or `c_arrow`, resp. We call such hyperedges *component edges* in the following. Additional information about a component is stored in attributes of the representing component edge, for example, the position and radius of a place. Relationships between diagram components are binary hyperedges visiting the two nodes representing the related attachment areas. For Petri nets, we have a relationship that relates an arrow head or tail to a transition (`at_trans`), a relationship that relates an arrow head or tail to a place (`at_place`), a relationship that relates overlapping places or transitions (`touch`), and a relationship that relates a token to the place it is contained in (`inside`). Hyperedges representing relationships are called *relation edges*.

The overall system architecture of a diagram editor generated by DiaGen is shown in Fig. 2. Rounded boxes depict data structures, rectangles depict processing units. The figure shows an editor without sketching support, called *regular editor* in the following. The *layouter* and the *transformer* are not relevant for this paper. Also, we neglect *attribute evaluation* as the final step of processing a diagram.

The *drawing tool* provides the user with a GUI, it is the actual diagram editor. As mentioned before, a hyperedge is created for each component placed on the canvas by the user. We change this behavior for sketching and create a hyperedge for each component that can be recognized in the hand-drawn diagram. Therefore we have replaced the original DiaGen editor by another editor that allows for drawing by hand. The process of *recognition* of components from the hand drawing is described in [BM08]. Result of the recognition is a set of components. *How* these components were drawn is neither relevant nor visible to the approach shown here, but completely handled by the recognition process.

In the next step of the processing chain, the *modeler* identifies relationships between components and creates respective hyperedges. No user input or user interaction is required for this process. Relations cannot be restricted, e.g., by a condition, but depend solely on the spatial

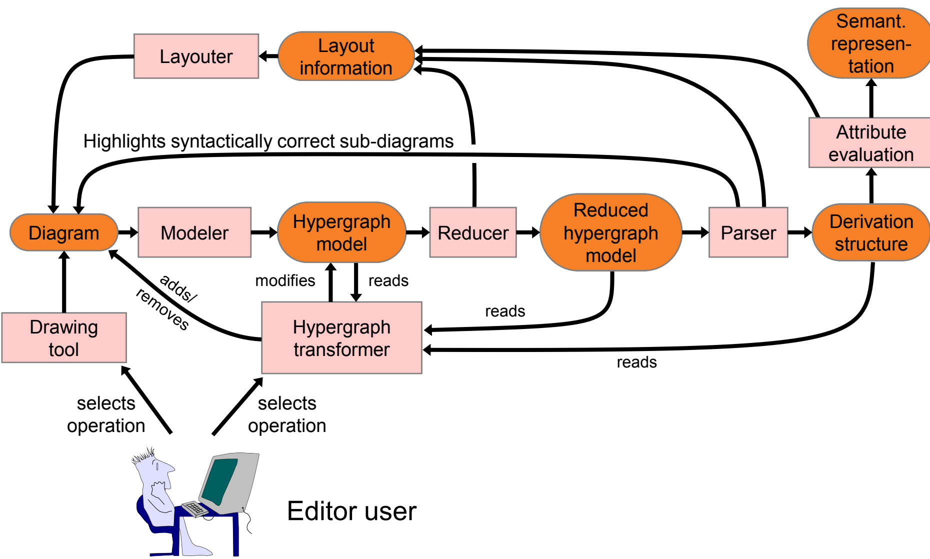


Figure 2: Architecture of a diagram editor generated with DIAGEN.

placement of the components. The result of the modeler is a hypergraph (the *hypergraph model*, or HM) containing all component edges and all respective relation edges. In case of ambiguity, a component edge is created for each possible interpretation of the ambiguous component. Such component edges are independent from each other, although they were recognized from the same strokes in the hand drawing. No information is stored that these edges actually represent the same component. For Petri nets, it is clear that only one of these edges can be valid at the same time, but for other diagram languages the situation may be different. The mechanisms of the subsequent reducer and parser are employed accordingly to account for such component edges.

The HM for the Petri net shown in Fig. 1 is depicted in Fig. 3. Relation edges are shown as arrows. As `touch` is a symmetric relation, each of the respective arrows has two arrow heads. The two ambiguities identified in the drawing are highlighted in gray: for each of the two circles on the left in Fig. 1, two component edges are created (`c_place` and `c_token`). For the single component on the right, three component edges are created. The only non-ambiguous components are the two arrows and the transition in the center. Because all component edges are independent of each other, each of the three `c_token`-edges in Fig. 3 is also related to that `c_place`-edge which represents the same circle in the drawing (the two vertically displayed `inside`-relations on the left, and the `inside`-relation on the right).

Even the `c_token`-edge representing the large circle is identified to be *inside* the `c_place`-edge representing the small circle, because their attachment areas overlap, hence an `inside` relationship can be found (the name `inside` is misleading in this case). Furthermore, all identified places and tokens in in this example overlap and have their full shapes as attachment areas, so the distance of the respective attachment areas is always 0. The same is true for the `touch`-relation between the `c_place` and the `c_trans` on the right.

Both components and relationships are rated by a positive real number. For components, the

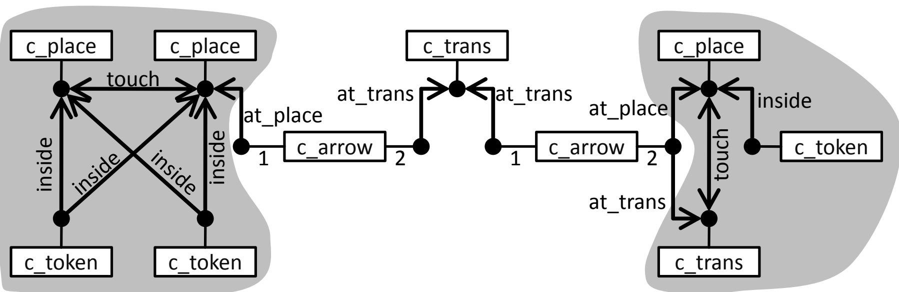


Figure 3: Hypergraph model of the Petri net shown in Fig. 1. Ambiguities are highlighted.

rating depends on the complexity of the component, and on the precision of how it has been drawn. Components being more complex, or being drawn more precisely, gain a higher rating. For relationships, the rating depends on the distance of the two respective attachment areas. A smaller distance means a higher rating. Each component edge and relation edge contains the rating of the represented component or relation in an attribute. The rating will be used by the parser.

Next, the *reducer* applies a set of reduction rules, i.e., graph transformation rules, to the HM. Such rules consist of an LHS and an RHS, each are hypergraphs. The result of the reducer is the *reduced hypergraph model* (RHM). In the first place, the RHM is newly created and therefore empty. Then, for each match of the LHS of a reduction rule in the HM, a respective match for the RHS of the matched rule is added to the RHM. The HM is not changed by the reducer.

The reducer serves two tasks: first, the RHM usually contains less hyperedges than the HM; as the RHM is the input for the parser, a smaller model containing less edges improves processing time. Second, invalid configurations which may occur in the HM due to misplaced components are not transformed, i.e., they do not occur in the RHM. Application of reduction rules can be restricted by conditions, and by negative application conditions (NACs).

The reduction rules for Petri nets are shown in Fig. 4. Corresponding nodes on the LHS and the RHS are labeled with the same letter. NACs are highlighted in gray and crossed out. The two upper rules transform places and transitions, as long as these do not overlap with any other place or transition. The third rule transforms tokens inside places, ignoring tokens not inside places. Here, a condition is applied: a token is only reduced if its radius is smaller than half the radius of the containing place. The last two rules transform arrows between places and transitions. As arrows have two attachment areas (head and tail), the attachment areas are distinguished by numbers. Note that there are no reduction rules for arrows between two places or between two transitions, as Petri nets do not allow such arrows.

Fig. 5 shows the RHM created by the modified reducer. The original reducer for regular editors would not produce the subgraphs highlighted in gray, due to the NACs. Except for the transition in the center, no transition or place would be reduced, as each of them touches another place or transition because of ambiguous interpretation of the corresponding components. Of the three tokens, only one would be reduced (the one that actually *is* a token), the other two would not, due to the condition in the third rule.

Obviously, ambiguity cannot be resolved for those components not reduced. In the depicted

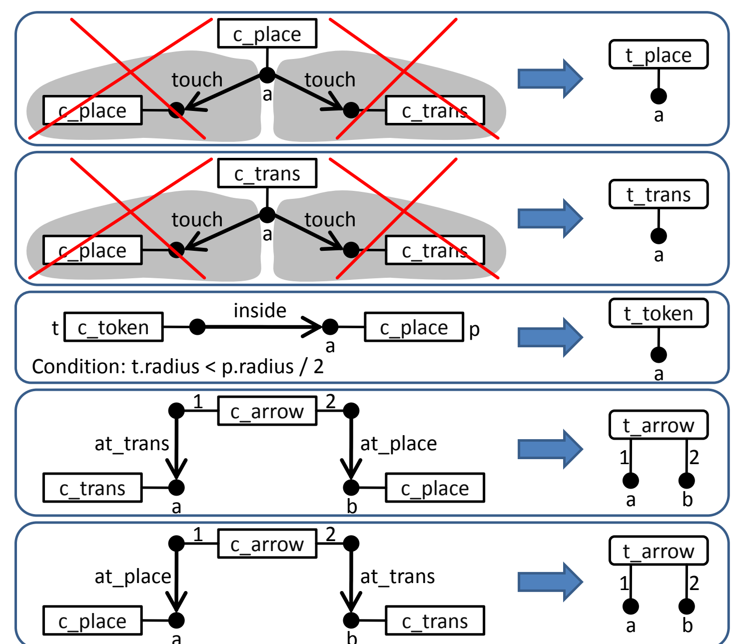


Figure 4: Reduction rules for the language of Petri nets.

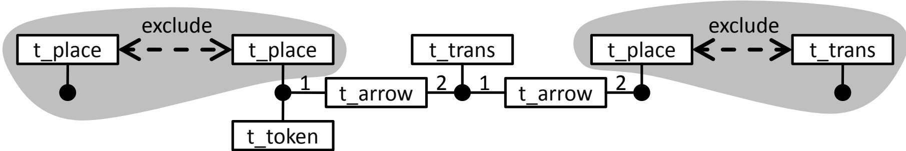


Figure 5: Reduced hypergraph model of the hypergraph model shown in Fig. 3.

case, the original reducer is too strict and discards almost all ambiguous components. We rather have to postpone ambiguity resolution to the parser, i.e., the RHM must contain complete information, even on the ambiguous components. Therefore, the highlighted subgraphs are added to the RHM in Fig. 5, together with the information about which components *exclude* each other. This information is based on the NACs. Note that the conditions (like the one for tokens) are not affected, and have to be met anyway.

Finally, the *parser* uses the hyperedges from the RHM as terminal edges and attempts to deduce the start symbol in a bottom up fashion. The production rules for Petri nets are shown in Fig. 6. Terminals from the RHM are depicted as rounded rectangles with a white background, while nonterminals have a gray background. There are two types of production rules unique in DIAGEN. The one are *set productions* and the other are *embedding productions*.

Set productions are used to *collect* all edges with the same label, not regarding any order or any subset, thus improving performance of the parsing process. In Fig. 6, set productions are depicted with a stack of edges on the RHS (the two productions with `Transitions` and `Places` on the LHS, and a stack of `Trans` and `Place` on the RHS). An alternative specification would

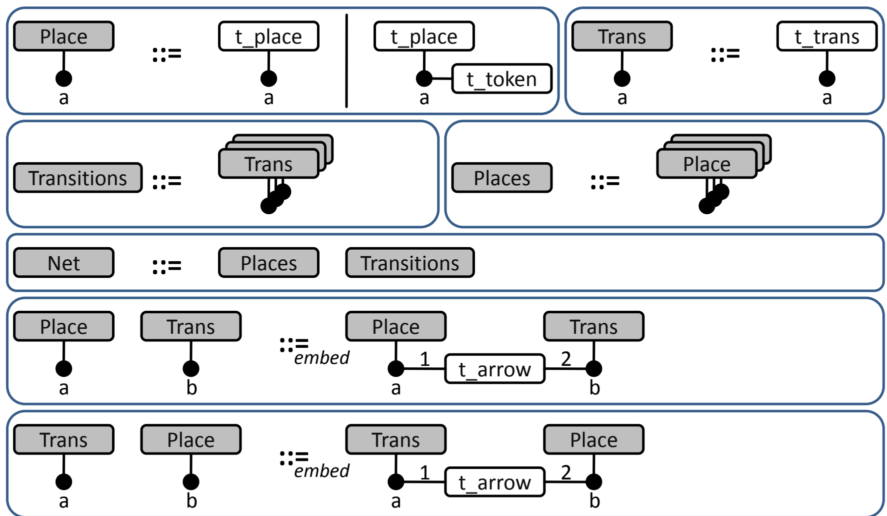


Figure 6: Production rules for the language of Petri nets.

use recursive productions like `Places ::= Place | Places Place`. However, that would require the parser to deduce any possible subset of all places, and in any possible order, which leads to a combinatorial explosion. The two set productions for Petri nets express that `Transitions` is a nonterminal edge representing the non-empty set of all `Trans` edges, and `Places` is a nonterminal edge representing the non-empty set of all `Place` edges. The start symbol for Petri nets is `Net`. The shown grammar therefore accepts a diagram as a correct Petri net if it contains at least one place and one transition (a modified grammar could also accept Petri nets without places or transitions).

Embedding productions are the other kind of production rule unique to DIAGEN. For efficiency reasons, the used graph grammar is context-free. Embedding productions are used to embed (nonterminal or terminal) edges into a context which has been derived by context-free productions. An example are arrows in Petri nets. With embedding productions they can be added to a derivation tree, resulting in a *direct acyclic graph* (DAG) as derivation structure. In the following we will always use the term *DAG*, not distinguishing between a DAG and a tree. Embedding productions consist of the same graphs on the LHS and the RHS, but with one additional edge on the RHS (the edge that is embedded). The two productions at the bottom of Fig. 6 are the two embedding productions required for Petri nets.

The parser must use the exclusion information provided by the reducer in order to deduce only those start symbols which do not contain terminal edges excluding each other in their DAG. This is described in Sec. 4, but first, it is explained how the reducer can use the NACs to collect the necessary information for the exclusion of components.

Note that the modifications to the reducer and the parser do not require the reduction rules or production rules to be modified. They are left unchanged, but are applied differently, as it is described next. Also, the architecture shown in Fig. 2 is preserved. The information necessary for ambiguity resolution is stored in additional attributes of the terminal and nonterminal edges.

3 Sketching-related Modifications to the Reducer

We require the reducer to apply the reduction rules even if they are prohibited by matched NACs, and create an exclusion relation *exclude* for the hyperedges in the RHM, i.e., the terminal edges for the subsequent parsing step. The simple example of Petri nets discussed in the previous section only shows terminal edges which mutually exclude each other. The general case is more sophisticated, as shown in the following. Such complicated situations occur when a NAC consists of more than only a single component hyperedge.

In the following we call hyperedges simply *edges*, and edges from the RHM *terminal edges* or simply *terminals*. A *match* of some pattern graph P in a host graph H is an occurrence of P in H. We regard a (sub)graph as set of edges. For a terminal t, where t is in the occurrence of the RHS of some reduction rule r, we denote by $model(t)$ the corresponding occurrence of the LHS of r, and by $nacs(t)$ a set of all matches for NACs that would have prohibited the creation of t. In the following, we omit all relation edges in $nacs(t)$ and keep only the component edges, as relation edges completely depend on component edges. We can then define a relation *exclude* between a set of terminals T and a single terminal t:

$$T\ exclude\ t :\Leftrightarrow \exists N \in nacs(t) : N \subseteq \bigcup_{t' \in T} model(t')$$

T excludes t if the union of all $model(t'), t' \in T$ contains all edges from a NAC N in the NACs from t, $N \in nac(t)$. For example, let m_p be a component edge representing a place, and m_t be a component edge representing a transition, and both components overlap, i.e., they exclude each other (apart from the token, this is the case for the right of Fig. 3). Then, the reducer creates two terminals, t_p and t_t, with

$$model(t_p) = \{m_p\}, nacs(t_p) = \{N_1\}, N_1 = \{m_t\}$$
$$model(t_t) = \{m_t\}, nacs(t_t) = \{N_2\}, N_2 = \{m_p\}$$

Now the set with the single terminal t_t excludes t_p, because $model(t_t)$ contains all edges in N_1, which is a NAC in $nacs(t_p)$. By analogy, $\{t_p\}$ excludes t_t. If we had included the relation edges in $nacs(t_p)$ and $nacs(t_t)$, both exclusions would not hold, as the relation edges do not occur in $model(t_t)$ and $model(t_p)$.

A slight modification of this example shows that the *exclude* relation is not symmetric in general. We change the reduction rule for places, so that a place has no NACs, i.e., it can overlap with any other component (the rule for transitions is not changed). We get

$$model(t_p) = \{m_p\}, nacs(t_p) = \varnothing$$
$$model(t_t) = \{m_t\}, nacs(t_t) = \{N_2\}, N_2 = \{m_p\}$$

Here, $\{t_t\}$ still excludes t_p, but not the other way around. In the following we assume the original reduction rule for places as shown in Fig. 4.

For the general case, a NAC contains more than one component edge. Let N be a match of a NAC. By applying several reduction rules, different terminals can be reduced from the edges in N. Then, not only one, but more terminals are required to exclude a single terminal.

Based on the ratings for component edges and relation edges, a rating can be computed for a terminal t by adding up all ratings from all edges in $model(t)$. The next section describes how the parser exploits the *exclude* relation when deducing the start symbol, and how ratings are used.

4 Sketching-related Modifications to the Parser

The central data structure for the parser is the derivation DAG. Each DAG has a unique *root*: it is the node which has no incoming edges. *Leaves* of a DAG have no outgoing edges. All leaves represent terminals, all other nodes represent nonterminals. Unless embedding productions are used, each node has a unique *parent*, except for the root. Each parent node is the LHS match of a production rule, and its children are the respective match for the RHS of that rule. Nodes representing embedded edges can have more than one parent; all parents of such a node represent the context of the respective embedded edge.

The general idea for the parser is to avoid deduction of nonterminals from a set of terminals T where some terminal in T is excluded by other terminals in T. For this purpose we will define a symmetric relation *conflict*; if two nonterminals *conflict* with each other, they must not occur in the same derivation DAG.

Let nt be a nonterminal. By $term(nt)$ we denote the set of all terminals used to deduce nt, i.e., the set of all leaves in the DAG with nt as root[1]. DIAGEN applies a production rule (production, for short) if three conditions hold: (i) a match M for the RHS must be found, (ii) for all nonterminals nt in M all $term(nt)$ must be pairwise disjoint, and (iii) the condition defined for the production must hold. We leave (i)-(iii) unchanged, but add a fourth condition which regards the *exclude* relation created by the reducer. We will see in the following that this fourth condition depends on the type of the production. A *Chomsky Normal Form* can be computed for each hypergraph grammar (apart from the set productions and the embedding productions). Consequently, four different types of production rules have to be distinguished:

- *terminal productions* with exactly one terminal on the RHS.
- *nonterminal productions* with exactly two nonterminals on the RHS.
- *embedding productions* with nonterminals on both sides.
- *set productions* with an arbitrary number of nonterminals on the RHS.

Terminal productions may always be applied, no conflicts may arise here. For *nonterminal productions* we first consider a set T of terminals. We call T *conflicting* if there exists a subset $E \subset T$ and a single terminal $t \in T \setminus E$ where E excludes t. Then, two nonterminals nt_1 and nt_2 may be used on the RHS of a production rule if the union of their terminals, $term(nt_1) \cup term(nt_2)$, is not conflicting. This also implies that both $term(nt_1)$ and $term(nt_2)$ are not conflicting. We define the symmetric relation *conflict* between two nonterminals. $(nt_1, nt_2) \in conflict$ if and only if $term(nt_1) \cup term(nt_2)$ is conflicting. Then, nt_1 and nt_2 can be used on the RHS of a nonterminal production if they are not *conflicting*, i.e., $(nt_1, nt_2) \notin conflict$.

[1] $term(t)$ is only relevant during construction of the derivation *tree*, so embedded edges are never in $term(t)$.

Embedding productions are treated differently than the other types of productions. First, the DIAGEN parser identifies each match for the context of each embedding production. This is only possible if no edges of a match conflict with each other. Then, for each derivation DAG with the start symbol as root, each of the previously identified matches is checked whether all of its edges are contained in the DAG, i.e., are (direct or indirect) children of the root. Finally, for all such matches, the additional edge of the RHS of the production is embedded, i.e. added to the DAG. The edges of a match can only be contained in the DAG if none of these edges conflicts with any other edge from the DAG. Therefore, the only condition that must be checked before an edge may be embedded is whether it conflicts with the root of the DAG. If there is no conflict, there can also be no conflict with any other edge in the DAG.

The fourth type of production rule are the *set productions*. Basically, set productions can be seen as nonterminal productions with not exactly two, but one or more edges in the match of its RHS. However, the production may be applied even if nonterminals in this match are conflicting. In this case, the problem is to decide which of the conflicting edges should be omitted, because this decision may have consequences on subsequent applications of production rules and contexts for embedding productions. The former happens if the nonterminal that has not been omitted conflicts with another nonterminal in a subsequent production rule. The latter happens if the omitted nonterminal was part of a match for a context of an embedding production. Consequently, we must defer the decision, as we cannot make it when applying a set production. We temporarily ignore all conflicts, and do not omit any of the nonterminals on the RHS match. When the start symbol is reached, there can be no further productions, and we can finally decide which nonterminals to omit.

The nonterminal matching the LHS of a set production may be used in the match of a RHS of another set production itself, either directly, or as a node contained in the DAG of a nonterminal in this match. This can lead to a complex structure. An example is depicted in Fig. 7, where the top part of a derivation DAG can be seen. As before, nonterminals are depicted as rounded rectangles with a gray background. The thin arrows indicate parent-child relationship. The subtrees of nonterminals with two outgoing arrows not ending in other nonterminals are of no interest in this example. Embedding productions are not shown. A, B and C are nodes in the DAG indicating applications of set productions. The fat arrows, marked with crosses, depict conflicts between nonterminals.

When applying the nonterminal production $a_3 \rightarrow BF$ we can immediately discard b_3, as it conflicts with F. The production cannot be applied otherwise. The same is true for production $D \rightarrow EC$ and c_3. The following cases are more difficult. When applying set production $A \rightarrow a_1a_2a_3a_4$, we cannot decide for a_2 or for b_1, as we do not know about possible later consequences. There is a conflict between a_2 and c_1. The problem is that we do not know yet that A and C will be used in the same DAG. Finally, when applying $S \rightarrow AD$, we can decide for a_2 or b_1, for a_2 or c_1, for c_1 or c_2, and for a_4 or c_1, as we know that there will be no further productions.

The problem of omitting nonterminals is NP-complete. It is a slight variation of the *maximum clique problem* [Kar72]. However, we do not need the best solution; a heuristic is sufficient. In order to guide the heuristic we use the ratings assigned to each terminal. The rating $rating(nt)$ of a nonterminal nt is the sum of all ratings from the terminals in $term(nt)$. This way, the start symbol in a derivation DAG is rated. For set productions, we would like to find the non-conflicting subset of the conflicting nonterminals with the highest rating of all nonterminals. The

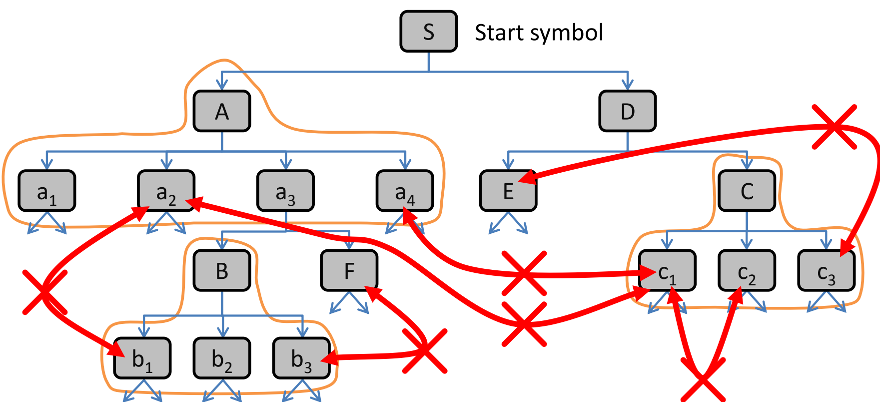


Figure 7: Top part of an exemplary derivation DAG with three set productions and root S. Conflicts between nonterminals are depicted by fat red arrows marked with crosses.

better the result is, i.e., the higher the final rating for the start symbol (which is decreased by every nonterminal omitted), the better the ambiguity resolution is, because a higher rating means more components, more complex components and more drawing precision.

The basic idea for the heuristic is to prefer nonterminals (i) with a high rating, (ii) with few conflicts, (iii) whose conflicting nodes have low ratings, and (iv) which are part of many matches of contexts for embedding productions. A ratio is calculated for each nonterminal nt. Let $embed(nt)$ be the set of all embedded edges where nt is one edge of the match of their context; the ratio r is calculated as

$$r(nt) = \frac{rating(nt) + \sum\{rating(e)|e \in embed(nt)\}}{\sum\{rating(c)|(nt,c) \in conflict\}}$$

The heuristic then works as follows: as long as there are conflicts, the nonterminal with the lowest ratio is omitted, and the ratios of its conflicting nonterminals are increased accordingly. If the start symbol cannot be deduced any more, because the last nonterminal of a set production is omitted, backtracking is applied and the next nonterminal is tried. We found that this very simple approach works well in practical cases, producing meaningful results quickly.

Therewith it is explained how the symmetric *conflict* relation must be exploited to generate derivation DAGs which are correct, i.e., which do not use terminals that must not occur in the same DAG due to the *exclude* relation introduced in the previous section. The ambiguities shown in the introductory example in Fig. 1 now can be solved in the desired manner. The RHM and a schematic representation of its DAG is shown in Fig. 8. The thin dashed arrows depict the matches for the contexts for the two arrows. Note that, due to the CNF, the one nonterminal `Place` cannot be the parent of the two terminals `t_place` and `t_token`. However, for the sake of clarity, we do not show the correct representation, which requires artificially generated nonterminal symbols.

Given that places and transitions in Petri nets are similarly rated, the decision about which components to omit depends on the embedded arrows. For Fig. 1, the leftmost place and the rightmost transition will be omitted, which is exactly the result we initially described. In general,

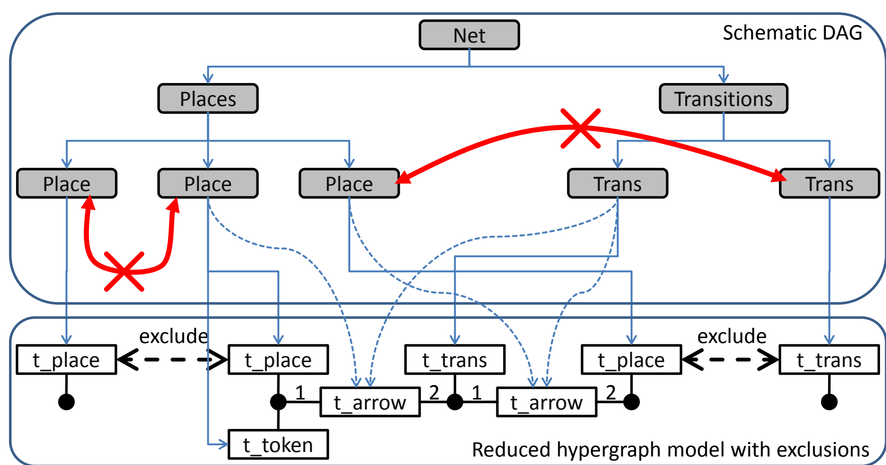


Figure 8: Reduced hypergraph model as shown in Fig. 5, and a schematic representation of the corresponding derivation DAG.

there may be a number of correct derivation DAGs. We then choose the one with the highest rated start symbol.

5 Related Work

Using context allows the machine to automatically decide for an interpretation in case of ambiguity. The alternative is to have the user explicitly make a choice. This is called *mediation* [MHA00]. Various strategies are conceivable, e.g., providing the user with a list of possible interpretations and let him decide, or requiring the user to redraw an ambiguous symbol. Especially the latter is limited in applicability. In our case, places and tokens in Petri nets are both drawn as a circle (cf. Fig. 1). Redrawing this circle obviously cannot resolve this ambiguity. Providing a list would help here, but is very uncomfortable for the user.

Some approaches decide for a possible interpretation without explicit user interaction, but neglect context information for this decision. For example, *LADDER* compares only the ambiguous components, not regarding context, and uses simple rules to prune alternatives, at the risk of preserving a wrong interpretation [HD05].

An approach by [AD06], limited to the domain of mechanical drawings, merges automatic decisions and user interaction. The system collects evidence from the drawing, based on rules. Additionally, each component is scored. Based on the evidence and the score, alternatives are pruned by a greedy algorithm. The algorithm is not able to undo its decisions, thus the result may not be optimal. In case of a wrong decision, the user can indicate that another possible interpretation is to be taken.

We are aware of only one other approach to sketching that treats ambiguity resolution with a grammar-based approach. It is based on so-called *sketch grammars* [CDPR04, CDR06]. Unlike our approach, diagram analysis is not separated into a reducer and a parser. Instead, the parser is directly applied. Parsing is directed by probabilities and rankings (similar to the ratings we

use) in order to avoid processing of unlikely interpretations. The used grammar is not based on hypergraphs, but extends positional grammars, which themselves extend traditional string grammars by more general relations than concatenation. For the actual parse process, only little detail is published. The concept of NACs is not employed. Result of the parser is a forest of ranked derivation trees, each representing a valid interpretation of the drawing. The user can then choose the desired representation.

Both recognition and ambiguity resolution works different for (handwritten) text, which we do not want to cover with our approach. Various methods are reported for this issue. For example, characters can be disambiguated by use of vocabularies, words can be disambiguated by statistical methods like Hidden Markow Models, language models, or specialized statistical grammars like PCFG (probabilistic context-free grammar) [PS00, ZCB06, HLB00].

6 Conclusion and Future Work

In this paper we explained an approach to ambiguity resolution in sketched diagrams using a hypergraph-grammar-based approach. Omission of components due to NACs is deferred as long as possible, until the parser needs a decision to go on. Using Petri nets as example we motivated the basic ideas. Applicability is restricted because of an NP-complete problem. We have implemented the presented approach as an extension of DIAGEN. Practical results suggest that the heuristic we apply works well, and quickly computes a result. We will inspect this restriction further, both from a theoretical and practical point of view. Graph grammars without set productions do not suffer from this issue.

The shown approach is incremental, i.e., modifications to the diagram do not require analysis from scratch. When components are added to the diagram, respective components edges are created, and all possible relationships of these components edges are checked. The reducer then only regards the newly created edges, and so does the parser, which modifies existing derivation DAGs. When diagram components are removed, the situation is similar.

Although we use the approach for ambiguity resolution in the context of sketching, regular graph parsers may benefit from the shown approach as well, as feedback in case of misplaced components (leading to a syntactically incorrect diagram) may be improved. The very strict behavior of the DIAGEN reducer, as shown in Sec. 2, suggests that feedback is very coarse and does not aid very much in finding misplaced components. What happens is that maximum sized subdiagrams with correct syntax are highlighted. Using our approach, the size of these subdiagrams may be increased.

Metamodel-based approaches have gained popularity in recent years. As future work, we would like to find out how DIAMETA, as a metamodel-based approach and an extension of DIAGEN, can benefit from the shown results as well, and how and to what extent they can be applied. Then we can compare the graph-based and the metamodel-based approaches.

Bibliography

[AD06] C. Alvarado, R. Davis. Resolving ambiguities to create a natural computer-based sketching environment. In *SIGGRAPH '06: ACM SIGGRAPH 2006 Courses*. P. 24.

ACM, New York, NY, USA, 2006.

[BM08] F. Brieler, M. Minas. Recognition and Processing of Hand Drawn Diagrams Using Syntactic and Semantic Analysis. Submitted to AVI '08. 2008. http://www2.cs.unibw.de/publ/brieler/avi08.pdf.

[CDPR04] G. Costagliola, V. Deufemia, G. Polese, M. Risi. A Parsing Technique for Sketch Recognition Systems. In *Proc. VL/HCC '04*. Pp. 19–26. IEEE Computer Society, Washington, DC, USA, 2004.

[CDR06] G. Costagliola, V. Deufemia, M. Risi. A Multi-layer Parsing Strategy for On-line Recognition of Hand-drawn Diagrams. In *Proc. VL/HCC '06*. Pp. 103–110. IEEE Computer Society, Washington, DC, USA, 2006.

[FNTZ00] T. Fischer, J. Niere, L. Turunski, A. Zündorf. Story Diagrams: A New Graph Grammar Language Based on the Unified Modelling Language and Java. In Ehrig et al. (eds.), *Theory and Application of Graph Transformation (TAGT'98), Selected Papers*. Volume 1764, pp. 296–309. Springer, 2000.

[HD05] T. Hammond, R. Davis. LADDER, a sketching language for user interface developers. *Computers & Graphics* 29(4):518–532, 2005.

[HLB00] J. Hu, S. G. Lim, M. K. Brown. Writer independent on-line handwriting recognition using an HMM approach. *Pattern Recognition* 33(1):133–147, 2000.

[Kar72] R. Karp. Reducibility among Combinatorial Problems. In Miller and Thatcher (eds.), *Complexity of Computer Computations*. Plenum Press, New York, 1972.

[LV02] J. de Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modelling. In *Proc. FASE '02*. Pp. 174–188. Springer-Verlag, London, UK, 2002.

[MHA00] J. Mankoff, S. E. Hudson, G. D. Abowd. Interaction techniques for ambiguity resolution in recognition-based interfaces. In *Proc. UIST '00*. Pp. 11–20. ACM Press, New York, NY, USA, 2000.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Journal of Science of Computer Programming* 44(2):157–180, 2002.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. In *Electronic Communications of the EASST, Proc. GraBaTs '06*. September 2006.

[PS00] R. Plamondon, S. N. Srihari. Online and off-line handwriting recognition: a comprehensive survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22(1):63–84, 2000.

[ZCB06] M. Zimmermann, J.-C. Chappelier, H. Bunke. Offline Grammar-Based Recognition of Handwritten Sentences. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 28(5):818–821, 2006.

A Static Layout Algorithm for DiaMeta

Sonja Maier[1] **and Mark Minas**[2]

[1] sonja.maier@unibw.de
[2] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: The diagram editor generator framework DiaMeta utilizes meta-model-based language specifications and supports *free-hand* as well as *structured editing*. In this paper we present a layouting approach that is especially well suited for a *static layout*. It is based on the layout algorithm presented in [MM07a] that uses the two concepts constraint satisfaction and attribute evaluation. This algorithm is combined with graph transformations and the result is a natural way of describing the layout of visual languages. As an example we use a simplified version of *Sugiyama's algorithm*, applied to *statechart* diagrams.

Keywords: Layout Algorithm, Static Layout, Dynamic Layout, Constraints, Attribute Evaluation, Graph Transformation

1 Introduction

Each visual editor implements a certain visual language. Several approaches and tools have been proposed to specify visual languages and to generate editors from such specifications. These attempts can be characterized by the way the diagram language is specified, and by the way the user interacts with the editor and creates respectively edits diagrams. Most visual languages as of today have a model as (abstract) syntax specification. Models are essentially class diagrams of the data structures that are visualized as diagrams.

When considering user interaction and the way how the user can create and edit diagrams, structured editing is usually distinguished from free-hand editing. Structured editors offer the user some operations that transform correct diagrams into (other) correct diagrams. Free-hand editors, on the other hand, allow to arrange diagram components from a language-specific set on the screen without any restrictions, thus giving the user more freedom. The editor has to check whether the drawing is correct and what its meaning is.

One weak point of such editors is, as always, layout. When talking about layout, we need to distinguish two terms: *Layout*, the general term, and *layout refinement*. *Layout refinement* starts with an initial layout and performs minor changes to improve it while still preserving the "feel" (or "mental map" [PHG07]) of the original layout. Especially user interaction is considered in this context. *Layout* may also position components of the diagram from scratch without an initial layout. For structured editing, *layout* is required, as newly created components need to be positioned from scratch. For free-hand editing, either *layout* or *layout refinement* may be applied. We also need to distinguish the two terms *static layout* and *dynamic layout*. When applying a static layout algorithm to a diagram, it always returns the same visual representation

of the diagram. When applying a dynamic layout algorithm, the result is influenced by the ”layout” of the initial diagram and by the user input.

In [MM07a] we presented a dynamic layout algorithm usable for model-based visual languages. It meets the demands of structured as well as free-hand editing. The algorithm combines the concepts *constraints* and *attribute evaluation* to an algorithm that is fast, flexible and behaving the desired way. This approach provides us with all we need for layout refinement. But we have recognized that *constraints* and *attribute evaluation rules* for such a specification quickly get long and complicated, especially when using a ”real world” layouting strategy.

In this paper, we extend our approach, improving this aspect. We combine graph transformation and the dynamic layout algorithm presented in [MM07a]. Graph transformations are applied, with the goal that the complexity of the *constraints* and *attribute evaluation rules* used to specify the dynamic layout algorithm is reduced. The approach also has the benefit that the layout algorithm is separated into phases, each of them treated independently. The reduction of complexity as well as the separation into phases simplifies the creation process of a new layouting strategy. The modularity of the algorithm additionally offers us perfect conditions for setting up a ”testing environment” for layout strategies.

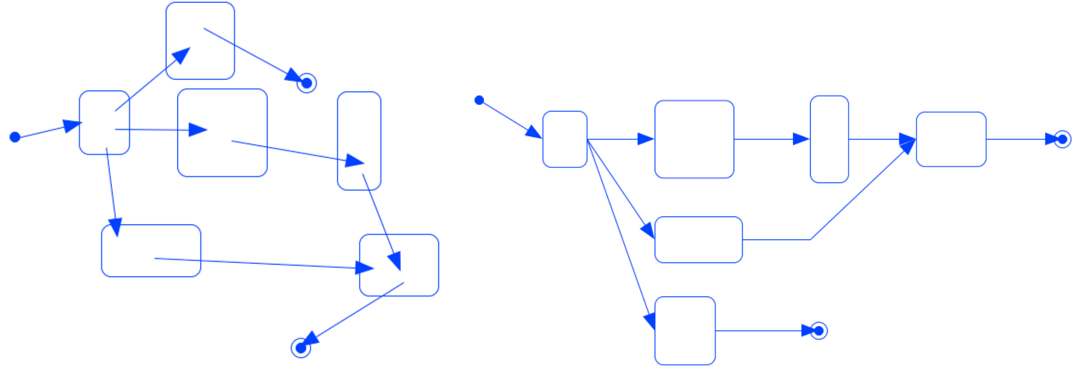


Figure 1: Sugiyama’s Algorithm applied to Statecharts

We demonstrate our approach by specifying Sugiyama’s algorithm (a simplified version), a standard layouting strategy for graphs.

We have integrated and tested our approach in DiaMeta [Min06]. DiaMeta follows the model-driven approach to specify diagram languages. From such a specification, an editor, offering structured as well as free-hand editing, can be generated. Figure 1 shows a statechart diagram before and after applying the Sugiyama’s algorithm. The diagram was created via a DiaMeta editor. The layout algorithm was implemented using the approach presented in this paper.

Section 2 introduces the model of statecharts, the visual language that is used as a running example. Section 3 gives an overview of DiaMeta, the environment in which the algorithm has been tested. Section 4 explains the proposed algorithm, and Section 5 gives a detailed example. Section 6 contains related work, and Section 7 concludes the paper.

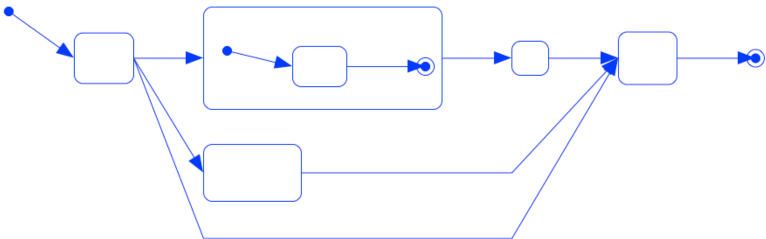


Figure 2: Statechart Diagram

2 Running Example

As a running example we use statecharts. In Figure 2 we see a layouted statechart diagram.

Figure 3 shows the meta model of simplified statecharts presented in this paper. We have ”states” (class *State*) and ”transitions” (class *Transitions*). A ”state” can either be a ”transition source” (class *TransSource*) or a ”transition target” (class *TransTraget*). A ”transition” connects one ”transition source” with one ”transition target” (roles *from* and *to*). A ”transition source” or ”transition target” may be connected with an arbitray number of ”transitions” (roles *inv_from* and *inv_to*). A ”transition source” can either be an ”initial state” (class *InitState*) or a ”labeled state” (class *LabeledState*). A ”transition target” can either be a ”labeled state” or a ”final state” (class *FinalState*). A ”labeled state” can be an ”or state” (class *OrState*), an ”and state” (class *AndState*) or a ”plain state” (class *PlainState*). An ”and state” must contain at least two ”and compartments” (class *AndCompartment*). A ”state container” (class *StateContainer*) may contain one or more ”states”. A ”state container” can either be an ”or state” or an ”and compartment”.

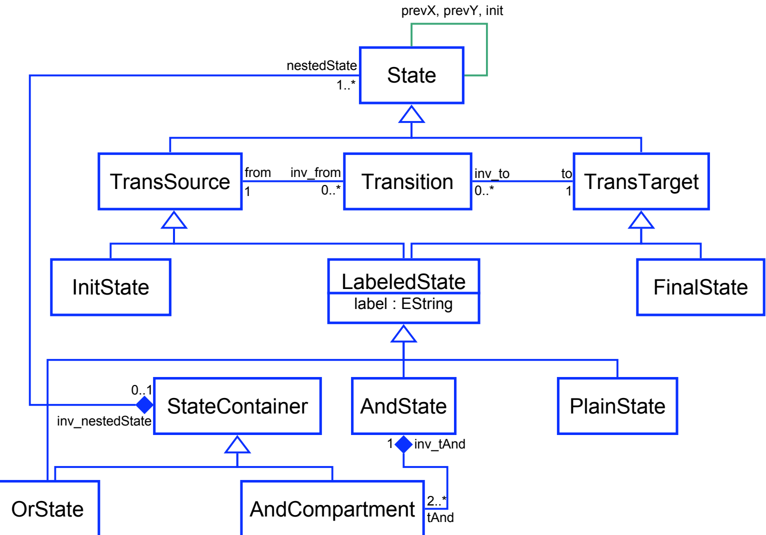


Figure 3: Meta Model of Statecharts

For the layout specification, we added three associations with the roles *prevX*, *prevY* and *init* to the meta model. Their usage will be explained in Section 5.

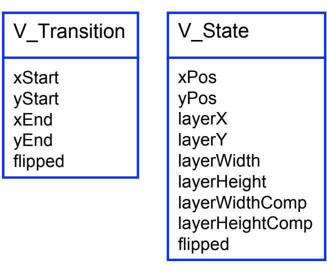


Figure 4: Visual Components

Visual components are created for ”states” and ”transitions” (Figure 4). For the visual representation, we distinguish between ”initial states”, ”labeled states” and ”final states”. A ”transition” is visualized by an arrow with the start point (*xStart,yStart*) and the end point (*xEnd,yEnd*). It also has the attribute *flipped*, that will be described later. ”Initial states” are visualized by a filled circle, ”final states” by two circles, and ”labeled states” by a rounded rectangle. Its position is described by its top left corner (*xPos,yPos*) and its size by its width (*width*) and height (*height*). A state also has the attributes *layerX*, *layerY*, *layerWidth*, *layerHeight*, *layerWidthComp*, *layerHeightComp* and *flipped*, that will be described later.

3 DiaMeta

In this section, we are going to introduce DiaMeta, the environment the algorithm was implemented in. It is needed to understand the context in which graph transformations and the dynamic layout algorithm are used. In particular it generates an overview of the implementation of graph transformations and the implementation of the dynamic layout algorithm, and how they were combined. The most important fact that is introduced is that graph transformations operate on an internal graph model, whereas the dynamic layout algorithm operates on the object model.

3.1 DiaMeta Architecture

The editor generator framework DiaMeta provides an environment for rapidly developing diagram editors based on meta-modeling. Each DiaMeta editor is based on the same editor architecture which is adjusted to the specific diagram language. This architecture is described in this paragraph. DiaMeta's tool support for specification and code generation, primarily the DiaMeta Designer are postponed to the next paragraph. Figure 5 shows the struc-

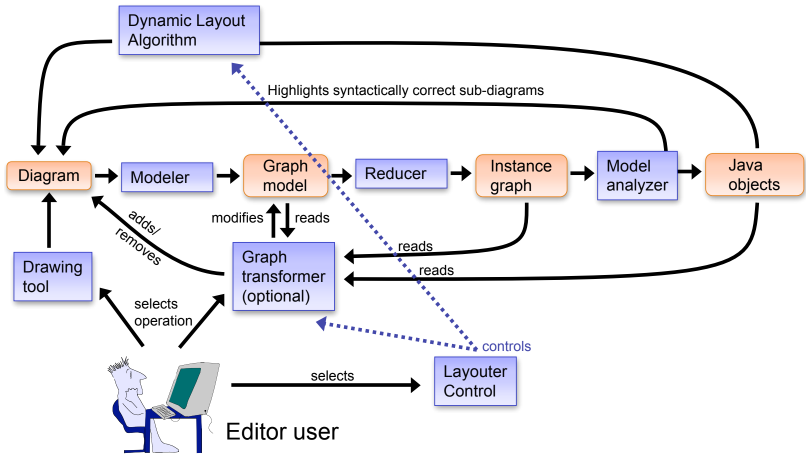


Figure 5: Architecture of DiaMeta

ture which is common to all DiaMeta editors - editors generated and based on DiaMeta. 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 DiaMeta Designer. With this drawing tool, the user is able to create, arrange and modify the diagram components of the particular diagram language. Editor specific program code, specified by the editor developer and generated by the DiaMeta Designer, is responsible for the visual representation of the language specific components. The drawing tool creates the data structure of the diagram as a set of diagram components together with their attributes (e.g., position, size). The sequence of processing steps, necessary for free-hand editing, starts with the modeler and ends with the model analyzer; the modeler first transforms the diagram into an internal model, the graph model. The reducer then creates the diagram's instance graph that is analyzed by the model analyzer. This last processing step identifies the maximal subdiagram which is (syntactically) correct and provides visual feedback to the user by drawing those diagram components in a certain color. However, the model analyzer not only checks the diagram's abstract syntax, but also creates the object structure of the diagram's correct subdiagram. The layouter modifies attributes of diagram components and thus the diagram layout is based on the object structure, which allows to access the represented diagram components.
The layouter is optional for free-hand editing, but necessary for structured editing. Structured editing operations modify the graph model by the means of the graph transformer (Sect. 3.3) and add or remove components to respectively from the diagram. The visual representation of the diagram and its layout is then computed by the layouter. For our approach, we introduced an additional component that controls the layout, which will be explained in Sect. 3.5.

3.2 DiaMeta Framework

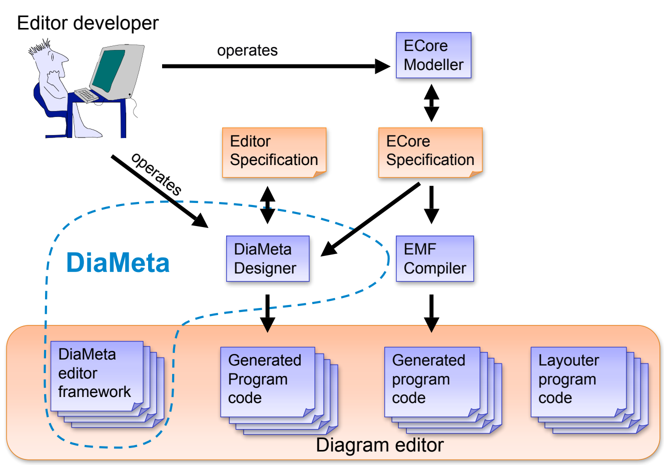


Figure 6: DiaMeta Framework

This paragraph outlines DiaMeta's environment supporting specification and code generation of diagram editors that are tailored to specific diagram languages. The DiaMeta environment shown in Figure 6 consists of an editor framework and the DiaMeta Designer.

The framework is basically a collection of Java classes and provides the dynamic editor functionality, which is necessary for editing and analyzing diagrams. In order to create an editor for a specific diagram language, the editor developer has to enter two specifications: First, the abstract syntax of the diagram language in terms of its model and second, the visual appearance of diagram components, the concrete syntax of the diagram language, the reducer rules and the interaction specification. Besides that, he may provide a layout specification, if he

wants to define a specific layouter. A language's class diagram is specified as an *EMF* model (ECore specification), created by using the *ECore* modeller. The *EMF* compiler is used to create Java code that represents this model. Figure 3 shows the class diagram of statecharts as an *EMF* model. The editor developer uses the DIAMETA Designer for specifying the concrete syntax and the visual appearance of diagram components, e.g., initial states are drawn as circles. The DIAMETA Designer generates Java code from this specification. In addition, the editor developer can provide a layouter. This Java code, together with the Java code generated by the DIAMETA Designer, the Java code created by the *EMF* compiler, and the editor framework, implement an editor for the specified diagram language.

3.3 Graph Transformer

DIAMETA offers the possibility to specify structured editing operations via graph transformations. These transformations operate on an internal graph model that is freely modifiable. They are defined in the Designer Specification, by a textual language. A graph transformation is either called by the editor user, or by the *Layouter Control*, which is described in Subsection 3.5.

3.4 Dynamic Layout Algorithm

In Figure 7 we can see a birds-eye view of the *dynamic layout algorithm* that has been presented in [MM07a]: The algorithm is based on the idea that a set of declarative constraints is given, assuring the characteristics of the *layout*. If all constraints are satisfied, the *layouter* terminates. If one or more constraints are not satisfied, the *layouter* needs to change some attributes to satisfy the constraints. Therefore it switches on one or more attribute evaluation rules. These rules are responsible for updating the attributes, i.e., to satisfy the constraints. The *layouter* is either called directly by the user or by the *Layouter Control*. All potentially violated layout constraints are checked, and the rules that were switched on are collected. Thereafter the new values for the attributes are calculated via attribute evaluation. Now, the constraints are checked again, since new constraints may have become unsatisfied due to changes performed by the *layouter*. If all constraints are satisfied, the *layouter* succeeds and reports the new attribute values. Otherwise, the *layouter* has to evaluate the rules again. If the *layouter* does not succeed after a certain number of iterations, the *layouter* stops and returns the old values as result.

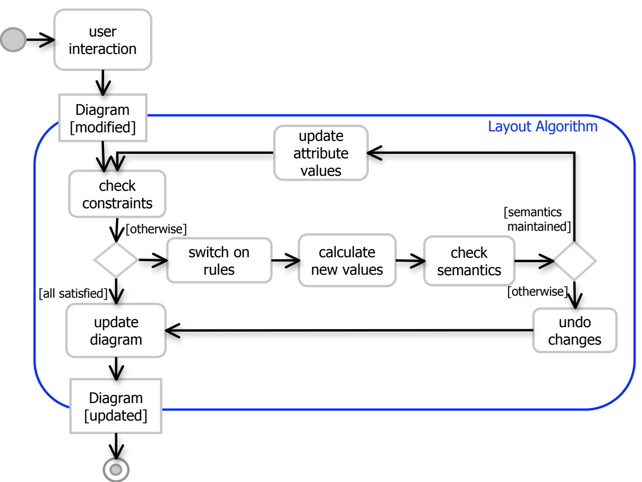


Figure 7: Dynamic Layout Algorithm

3.5 Layouter Control

We offer the editor user two possibilities to apply a layouting strategy. Either the user can choose a layouting strategy, and all phases are applied automatically. This is the ”normal” operating mode. Alternatively, the editor user may apply each phase manually by clicking a button. This is the ”test” operating mode, that turned out to be very helpful during layouter creation.

When applying a strategy automatically, as shown in Figure 5, the editor user has to select the desired layouting strategy (*layouter control*). The *layouter control* then ”controls” the graph transformer and the dynamic layout algorithm. For each phase, the *layouter control* inserts the ”right” graph transformation or dynamic layout algorithm, and initiates the process shown in Figure 5.

When applying a strategy manually, the editor user has to select the desired graph transformation or dynamic layout algorithm. The *layouter control* then inserts this graph transformation or dynamic layout algorithm, and initiates the process shown in Figure 5.

Graph transformations operate on the internal graph model and change this model. Afterwards, the processing steps that are required after a graph transformation are executed, and the diagram as well as the object model are updated. The dynamic layout algorithm operates on the object model and updates the attributes of the object model. Afterwards, the necessary steps are processed and the diagram is updated.

4 Layout Algorithm

The idea of the algorithm is combining graph transformation and the dynamic layout algorithm described in [MM07a]. Therefore, the layouting strategy is separated into different phases. In each phase, either a graph transformation or the dynamic layout algorithm is applied.

The general idea is the following: A graph transformation changes the model. Then the dynamic layout algorithm updates attribute values. Afterwards, a graph transformation undoes intermediate changes in the model. The purpose of intermediate changes in the model is reducing the complexity of the constraints and attribute evaluation rules specified in the dynamic layout algorithm. The separation into different phases splits up the dynamic layout algorithm into small pieces, and again reduces the complexity.

In Section 5 we are going to examine a concrete example, providing the reader with a better understanding of how to separate an algorithm into different phases.

5 Layout Algorithm for Statecharts

The layout algorithm we are going to specify is a simplified version of Sugiyama’s algorithm [STT81]. It has been defined by a combination of graph transformations and the dynamic layout algorithm presented in [MM07a].

Sugiyama’s algorithm is split up into different phases. We have refined this sequence of phases and specify each phase either by graph transformation (GT) or the dynamic layout algorithm (LA) to update the diagram. In the following section we are going to describe these phases in more detail.

01 (GT) Cycle Removal	07 (LA) Calculation of Layer Height and Width
02 (LA) Horizontal Layering	
03 (GT) Dummy Node Insertion	08 (LA) Update Position of States
04 (GT) Connecting of States in Vertical Layer	09 (LA) Update Position of Arrows
05 (LA) Vertical Layering	10 (GT) Dummy Node Replacement
06 (GT) Connecting of States in Horizontal Layer	11 (GT) Undo Cycle Removal

5.1 Phases

01 (GT) Cycle Removal A statechart diagram - when ignoring hierarchical states - is a directed graph that may contain cycles. In the first phase, these cycles are removed by flipping one or more edges.

Figure 8: Cycle Removal

```
forall [arrow]=getArrow_() (markUnvisitedArrow_(arrow))
forall [state]=getState_() (markUnvisitedState_(state))
( (
    forall [state]=getStateWithNoOutgoing_()
      (unmarkIngoing_(state)! unmarkState_(state))
    getStateWithNoOutgoing_()
  )!
  flipOutgoing_()
  getStateUnvisited_()
)!
```

We use the following algorithm to do this. First, all states and transitions are marked with a unary hyperedge in the graph model. The mark is removed at all states without any marked outgoing transition arrows. Their ingoing transitions are also unmarked. If all marks are removed, we are done. Otherwise, we know that the diagram contains one or more cycles. If this is the case, the algorithm arbitrarily chooses one marked state, and flips all outgoing edges by changing the value of the attribute *flipped*. Now, the algorithm continues with the first steps and proceeds till all marks were removed.

We implemented this algorithm by the graph transformation shown above, using the hypergraph transformation approach provided by DIAMETA. A statement of the form `forall [a] = getA() (do(a))` calls the rule `getA()` and stores return values in the list `[a]`. Then the rule `do(a)` is applied to all elements `a` of the list. The statement `(do())!` calls the rule `do()` as many times as applicable. Figure 8 shows a statechart before and after cycle removal. In this example, the black arrow (indicated by the ellipse) has been flipped.

02 (LA) Horizontal Layering The attribute evaluation rule that is associated with the transition *trans* takes care of computing the attribute *layerX*. If a state has no incoming edges, *layerX* has the value 1. Otherwise, *layerX* is the maximal distance of a state from the initial state. Figure 9 shows a statechart with the values of the attributes *layerX* and *layerY*[1].

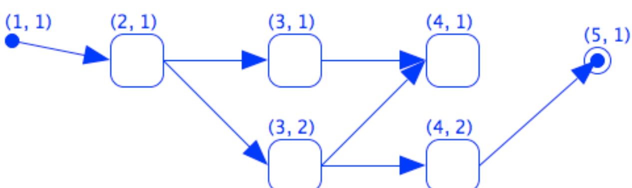


Figure 9: Statechart with Layers

```
trans.to.layerX := max(trans.to.layerX, trans.from.layerX + 1)
```

03 (GT) Dummy Node Insertion As a next step, dummy nodes are inserted, such that arrows only connect states in one layer with states in the next layer (*layerX*).

The algorithm works as follows: For each arrow, it is checked if it connects a state in one layer with a state in the next layer. When this is not the case, e.g., if it connects a state in layer n with a state in layer $n+2$, one or more dummy nodes are inserted.

We have implemented this algorithm by a graph transformation. The inserted *dummy nodes* are *plain states* that are marked as *dummy* (by changing the value of the attribute *dummy*). These *dummy* nodes also have layer attributes. Figure 10 shows a sample dummy node insertion. Here, three nodes had to be inserted.[2]

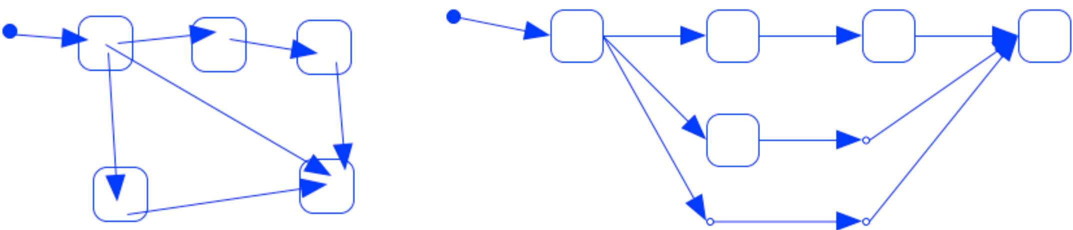


Figure 10: Dummy Node Insertion

04 (GT) Connecting of States in Vertical Layer All states in one vertical layer are connected by the link *prevY*. The elements of a layer can be identified by the attribute *layerX* that was previously set. The sorting of the elements is arbitrary. This connection is realized by a simple graph transformation. The transformation inserts for each link required in the object model an edge in the graph model, which is then translated into the link in the object model. The arbitrary sorting could be replaced by a more sophisticated strategy, e.g., by a strategy that minimizes the number of edge crossings, or by a dynamic layout algorithm enabling the user to change the sorting.

[1] The attribute *layerY* is updated in phase 05.

[2] The diagram in Figure 10 was already layouted. Otherwise, the dummy nodes would appear at the point (0,0).

05 (LA) Vertical Layering All states in one layer are connected by a link (*prevY*). The computation of *layerY* is done by the following attribute evaluation rule, that is associated with the state *state*. If *state.prevY* does not exist, *layerY* is set to 1.

```
state.layerY := state.prevY.layerY + 1
```

06 (GT) Connecting of States in Horizontal Layer After updating the attribute *layerY*, the states in each horizontal layer are connected via the link *prevX*. The elements of a layer are identified by the attribute *layerY*. The sorting of the elements is similar to the value of the attribute *layerX*.[3] This connection is again realized by a graph transformation.

07 (LA) Calculation of Layer Width and Height As stated in the last paragraph, all states in one horizontal layer are connected via the link *prevX*, and all states in one vertical layer are connected by the link *prevY*. The height of a horizontal layer is the height of the highest component in the layer. The width of a vertical layer is the width of the widest component in the layer. The computation is done by a pairwise comparison, i.e., by the following rules, associated with the state *state*. Initially, *state.layerWidth* is set to *state.width* and *state.layerHeight* is set to *state.height*. Figure 11 shows a diagram with states of variable size.

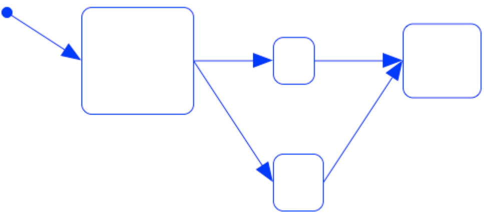


Figure 11: Variable Size

```
state.layerWidth := state.width
state.layerWidth := max(state.prevY.layerWidth, state.layerWidth)

state.layerHeight := state.height
state.layerHeight := max(state.prevX.layerHeight, state.layerHeight)
```

08 (LA) Update Position of States Firstly, the states are layouted. The position of initial states is not updated. This has the consequence that the position of initial states is variable. Labeled states and final states are updated via the following attribute evaluation rules, associated with the state *state*.

```
state.layerWidthComp := state.prevX.layerWidthComp + state.prevX.layerWidth
state.layerHeightComp := state.prevY.layerHeightComp + state.prevY.layerHeight

state.xPos := state.init.xPos + state.layerWidthComp + state.layerX*80
state.yPos := state.init.yPos + state.layerHeightComp + state.layerY*40
              + state.layerHeight/2 - state.height/2
```

state.layerWidthComp and *state.layerHeightComp* is the complete width respectively height of all previous states. *state.init* is the corresponding initial state. *state.layerX*80* and *state.layerY*40* insert spacing between the layers. Components are horizontally centered by adding + *state.layerHeight/2 - state.height/2*.[4]

[3] It might be the case that one or more numbers are missing, due to the structure of the statechart.

[4] Initial states are not layouted, and hence they are not centered, e.g., as can been seen in Figure 11.

09 (LA) Update Position of Arrows Arrows are also layouted. Attribute updating is performed via the following attribute evaluation rules, associated with the transition *trans*. The transition starts in the middle right of a state, and ends in the middle left of the next state.

```
trans.xStart := trans.from.xPos + trans.from.width
trans.yStart := trans.from.yPos + trans.from.height / 2

trans.xEnd := trans.to.xPos
trans.yEnd := trans.to.yPos + trans.to.height / 2
```

10 (GT) Dummy Node Replacement In this step, all dummy nodes are replaced by bends. This is done by a simple graph transformation. For each dummy node, the incoming and the outgoing arrow[5] are merged, and a new arrow is created. The dummy node is removed. Figure 12 shows the diagram of Figure 10 after replacing dummy nodes by bends. Here, three dummy nodes were replaced.

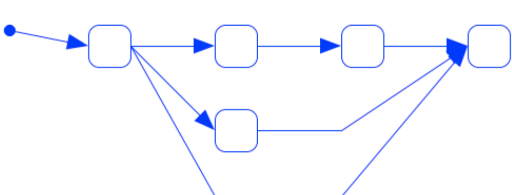


Figure 12: Dummy Node Replacement

11 (GT) Undo Cycle Removal As a last step, the previously flipped edges need to be turned around again. E.g., in Figure 8 (left diagram) we can see the result of undoing the cycle removal as it was done in Figure 8 (right diagram). In this case, the result is the initial diagram. This is not always the case, as there could have been changes in the last steps of the algorithm.

Statecharts with Nested States It is also possible to include nested states. Therefore, the two statecharts are layouted and the size of the containing state is computed from the size of the layouted contained statechart. Figure 13 shows a sample statechart with nested states. Right now, the size of the containing state and the position of the contained initial must be changed by the user, but may be specified via the layout algorithm, e.g., an additional phase.

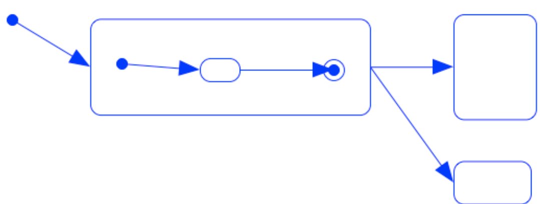


Figure 13: Nested States

General Remarks Besides automatic layouting of nested states, many other enhancements, like space saturation by floor planning, are imaginable. In our example we only used attribute evaluation rules, not the whole functionality of the dynamic layout algorithm presented in [MM07a]. Using this, a static layout algorithm is defined. A dynamic layout is also possible when using the whole functionality of the dynamic layout algorithm. E.g., one could offer the user the possibility to change the order of elements in a layer (*layerY*).

[5] A dummy node has exactly one incoming and one outgoing edge, due to the way it had been created.

6 Related Work

Many comparable tools, like AToM3, GMF, or Tiger, offer the possibility to use a standard layout algorithm, such as FlowLayout. Besides that, some tools, like DiaGen, offer the possibility to use constraints for layout specification. Most tools also allow the developer to write the layouter by hand, as only a small subset of layouter's can be realized by the mechanisms provided. With the approach presented we try to extend this subset.

Many tools support graph transformation, e.g., Fujaba, AToM3 or Tiger, but only rarely use them in the context of layout specification. Guerra et al. presented Event Driven Grammars [GL07]. Rules in these grammars may be triggered by user actions, and are combined with triple graph transformation systems. Rules may be defined especially for layout. If a rule is applicable, it is executed and attribute values are updated. In this approach, attributes are updated through graph transformations. In our approach, the graph model is changed via graph transformations, and then the attributes are updated via the dynamic layout algorithm. This gives us more freedom and reduces the size of a layout specification, especially when considering dynamic layout.

UML diagrams, such as activity and statechart diagrams are basically directed graphs. Most approaches for drawing directed graphs used in practice are based on Sugiyama's algorithm [STT81], e.g., an efficient implementation is presented in [ESK05]. Visual language specific layout algorithms based on Sugiyama's algorithm are for example described in [SK06] (activity diagrams) or in [CMT02] (statechart diagrams). Also some work had been performed to take user interaction into account, and to preserve the "mental map". When we take a look at these algorithms, we examine that they are always hand coded. With our approach, this is done by visual programming. In order to create a new layout algorithm, you have to provide graph transformation rules, constraints and attribute evaluation rules instead of plain Java code. This has the consequence that we can benefit from the advantages of visual programming: the creation and adaption of layouting strategies is easier, and hence experiments in this context are made possible.

Ware et al. state in [WPCM02] that (graph) aesthetics are taken as axiomatic, and have not been empirically tested. They argue that human pattern perception can tell us much that is relevant to the study of graph aesthetics. With our approach we created a platform for performing this kind of studies easily, not only for graphs but for all kinds of visual languages.

Purchase et al. state in [PHG07] that dynamic graph layout algorithms have only recently been developed. They anticipated that maintaining the "mental map" between time slices assists with the comprehension of the evolving graph. In DiaMeta, we do not only have automatic time-slices, but also time-slices triggered by user interaction. Besides that, freehand editing provides an initial layout that needs to be considered. In DiaMeta, many degrees of freedom are available, that may be considered when creating a new layout algorithm.

7 Conclusions and Future Work

In this paper we presented a layouting approach, that is especially well suited for a *static layout*. It is based on the layout algorithm described in [MM07a] that uses the two concepts constraint satisfaction and attribute evaluation. This algorithm is combined with graph transformations, and the result is a natural way of describing the layout of visual languages. As an example a simplified version of *Sugiyama's algorithm* is used, applied to *statechart* diagrams. We integrated and tested our approach in DIAMETA. The possibility to define a static layout with our algorithm was shown, and it turned out to be the very elegant and intuitive. When Sugiyama's algorithm is explained in literature, it is described step by step. We translated each of these steps into one or more phases. It was then possible to specify the phases independently, each of them either by a simple graph transformation or a simple dynamic layout algorithm. In contrast to other approaches, our layout specification is similar to the initial idea of the algorithm and, and hence the creation is more convenient for the developer.

In the current implementation, graph transformation operates on the graph model and attribute evaluation on the instance of the meta model. For future implementations, we will investigate the possibility to offer ”graph transformation” that operates on the instance of the meta model. E.g., we will investigate the EMF Transformer presented in [BEK+06]. This would reduce the complexity of the approach presented. In the current implementation, this is not possible, as DiaMeta is based on a hypergraph approach: Java objects can be created from a diagram, but a diagram cannot be created from Java objects directly.

Right now, only the parts of the diagram that are recognized as correct, are layouted. Other parts are not changed. Consequently, it might happen that visual components are moved by the layouter on top of other components, not recognized as correct. In future implementations we will need to consider parts of the diagram not recognized as correct in our layouting strategy.

For DIAMETA, an enhanced language for graph transformations and the dynamic layout algorithm is planned. For the dynamic layout algorithm, has been introduced already a pattern concept [MM07b] that simplifies the specification.

Future work has to investigate how layouters interfere with user interactions. When creating a diagram, we recognized that users create (one or more) states first. Then they create the transitions between them. The layouter has complicated the process of diagram creation, as it moves components away. In consequence, users turned off the layouter during creation, and turned it on again afterwards. During user interaction, a dynamic layouting strategy that only performs minor changes would be more adequate. Providing users with more freedom, e.g., offering the possibility of rearranging layers, would be an enhancement. These requirements can be realized by using the whole functionality the dynamic layout algorithm offers.

In [MM07b] we focused on dynamic layout, in this paper we focused on static layout. The most important next step will be experiments about the combination of static and dynamic layout in the context of structured and freehand editing. To identify the ”best” layouting strategy, we will need to perform empirical studies. With the algorithm presented, a testing environment was created to conduct these studies easier.

Bibliography

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. Pp. 425–439. 2006.

[CMT02] R. Castelló, R. Mili, I. G. Tollis. A Framework for the Static and Interactive Visualization of Statecharts. 2002.

[ESK05] M. Eiglsperger, M. Siebenhaller, M. Kaufmann. An Efficient Implementation of Sugiyama's Algorithm for Layered Graph Drawing. In *Graph Drawing, New York, 2005*. Springer, 2005.

[GL07] E. Guerra, J. de Lara. Event-driven grammars: relating abstract and concrete levels of visual languages. *Software and Systems Modeling* 6:317–347, 2007.

[Min06] M. Minas. Generating Meta-Model-Based Freehand Editors. Electronic Communications of the EASST, Proc. of 3rd International Workshop on Graph Based Tools, Natal, Brazil, 2006.

[MM07a] S. Maier, M. Minas. A Generic Layout Algorithm for Meta-model based Editors. In *Applications of Graph Transformation with Industrial Relevance, Proc. 3rd Intl. Workshop AGTIVE'07, Kassel, Germany*. 2007.

[MM07b] S. Maier, M. Minas. A Pattern-Based Layout Algorithm for Diagram Editors. In *Electronic Communications of the EASST, Proc. Workshop LED'07, Coeur d'Alene, Idaho, USA*. 2007.

[PHG07] H. C. Purchase, E. Hoggan, C. Görg. How Important is the "Mental Map"? – an Empirical Investigation of a Dynamic Graph Layout Algorithm. In Kaufmann and Wagner (eds.), *Graph Drawing, Karlsruhe, Germany*. Springer, 2007.

[SK06] M. Siebenhaller, M. Kaufmann. Drawing Activity Diagrams. In *Proceedings of the 2006 ACM symposium on Software visualization*. ACM, New York, USA, 2006.

[STT81] K. Sugiyama, S. Tagawa, M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man, and Cybernetics*, 1981.

[WPCM02] C. Ware, H. Purchase, L. Colpoys, M. McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 2002.

Foundations of Modelling and Simulation of Complex Systems

Hans Vangheluwe[1]

Modelling, Simulation & Design Lab
School of Computer Science
McGill University
3480 University Street
Montréal, Québec
Canada H3A 2A7
[1]hv@cs.mcgill.ca

Abstract

Modelling and simulation are becoming increasingly important enablers in the analysis and design of complex systems. In application domains such as automotive design, the notion of a "virtual experiment" is taken to the limit and complex designs are model-checked, simulated, and optimized extensively before a single realization is ever made. This "doing it right the first time" leads to tremendous cost savings and improved quality. Furthermore, with appropriate models, it is often possible to automatically synthesize (parts of) the system-to-be-built.

As a starting point, the basic concepts of modelling and simulation will be introduced. These concepts are based on general systems theory and start from the idea of a model as an abstract representation of knowledge about structure and behaviour of some system, either to be understood (in analysis) or built (in design) in a particular context (experimental frame). Typically, different formalisms are used such as Ordinary Differential Equations, Queueing Networks, and State Automata. It will be shown how these different formalisms all share a common structure and differ in the choice of time base, state space, and description of temporal evolution. This allows one to classify formalisms on the one hand and to find a common ground for implementing simulators on the other hand.

Building on these general modelling and simulation foundations, it will be shown how graph-based language engineering techniques such as meta-modelling and graph transformation can assist in the rapid development of (visual) modelling and simulation environments. This, in particular for formalisms where model structure may vary over time.

Complexity of systems is often due, not only to a large number of components, but also to the heterogeneity of these components. This leads quite naturally to the notion of multi-formalism modelling. To support the simulation of such models, both co-simulation and formalism transformation will be presented.

EASST

Dynamic Software Architectures Verification using DynAlloy

Antonio Bucchiarone[1],[3] and Juan P. Galeotti[2]

IMT of Lucca, Italy[1]
a.bucchiarone@imtlucca.it
Universidad de Buenos Aires, Argentina[2]
jgaleotti@dc.uba.ar
ISTI-CNR of Pisa, Italy[3]

Abstract: Graph Grammars have been often used for modeling dynamic changes in software architectures. In particular, we have previously characterized some classes of dynamicity in terms of particular aspects of graph grammars. Moreover we have identified classes of properties that can be naturally associated to any of such kinds of dynamicities. In this paper we approach the problem of verifying such properties over graph grammars specifications. In particular, we use DYNALLOY for attempting this task and we have concentrated on proving properties associated to a particular programmable dynamic software architecture.

Keywords: Dynamic Software Architectures, Typed Graph Grammars, Verification and DynAlloy

1 Introduction

Modern software systems have changed from isolated static devices to highly interconnected machines that execute their tasks in a cooperative and coordinate manner. Therefore, the structure and the behavior of these systems is dynamic with continuously changes. These systems are known as *Global Computing Systems (GCS)*, and have to deal with frequent changes of the network environment. The principal characteristics that these systems have are summarized in the following: *Globality:* each GCS is composed of autonomous computational entities where activities are not centrally controlled, either because global control is impossible of impractical, or because the entities are created or controlled by different owners (i.e., Global Services). *Heterogeneity:* GCSs are composed of heterogeneous devices (i.e., PDAs, laptops, mobile phones, etc..). that provide different configurations and functionalities. *Mobility:* each computational entity is mobile, due to the movement of the physical platforms or by movement of entities from one platform to another. *User-Dependent:* the end-user of a GCS is always the source of each change and a GCS must be able to adapt itself to make the user's task easier. *Fault-Tolerance:* GCSs provide mechanisms to guarantee that faults in the system do not interrupt a service delivery. The runtime behavior of the system is monitored to determine whether a change is needed. In such case, a reconfiguration is automatically performed without compromise the current system execution. *Scalability:* GCSs are able to start small and then expand over time in terms of size (i.e. more number of users, devices and connections) and functionalities (i.e., new service request) insuring the system availability.

Software architectural models are intented to describe the structure of a system in terms of computational components, their interactions, and its composition patterns [SG96], so to reason about systems at a more abstract level, disregarding implementation details. Since GCSs may change at run-time [Ore96], software architecture models for GCSs should be able to describe the changes of the system structure and to enact the modifications during the system execution. Such models are generally referred to as *Dynamic Software Architecture(*DSA*)*, to emphasize that the system architecture evolves during runtime. In this paper we select graph grammars [Roz97] as a formal framework to model DSA [BBGM07] and we approach the problem of verifying such properties over graph grammars specifications. In particular, we use DYNALLOY [FGLA05] for attempting this task proving properties associated to a particular programmable dynamic software architecture. We use graph grammars to model programmed DSA because they provide several advantages, these include: *(i)* a graphical representation of architectural styles and configurations that is in line with the usual way architectures are represented, *(ii)* independence from any particular solution technique, *(iii)* a formal basis based on the theory of formal languages [Roz97]. From the verification point of view, we might like to guarantee that a set of architectural structure changes (e.g., adding or removing components or connectors) will preserve some architectural invariants or, in other words, that a specific sequence of structural changes will result in a new Software Architecture that satisfies a particular property of interest. For this aspect we show how to use the ALLOY language to model our DSA based on graph grammars formalism and how to use DYNALLOY [FGLA05], an extension of the ALLOY language [Jac02, Jac06], to specify operations over architectures. Finally, we show how to use the ALLOY ANALYZER to check properties of interest for DSA . In Section 2 we introduce a running example that we use to introduce our idea. In Section 3 we summarize the principal elements of our formal framework that is used to represent DSA using hypergraphs. Then, in section 4, we show the way in which, using DYNALLOY structural properties of programmed dynamic SA are verified. Finally, related works are presented in Section 5, conclusions and future directions are shown in Section 6.

2 Running Example

We use as running example a simple scenario (see [BLMT07]) inspired by the automotive case study of Sensoria Project [Sen]. A road assistance service platform is supported by a wireless network of ad hoc stations that are situated along a road. Bikes equipped with electronic devices can access the service as they move along the road, e.g. to request a taxi in case of breakdowns. The graph in Figure 1 depicts a simple configuration of such a system. Each bike (🚲) is connected to the service access point (○) of a station (📡) which is possibly shared with other bikes. A station and its accessing bikes form a *cell*. Stations, in addition to the service access point, use two other communication points that we call chaining point (•). Such points are used to link cells in larger cell-chains. Bikes can move away from the range of the station of their current cell and enter the range of another cell. A handover protocol supports the migration of bikes to adjacent cells as in standard cellular networks. Stations can shut down, in which case their *orphan* bikes call for a repairing reconfiguration.

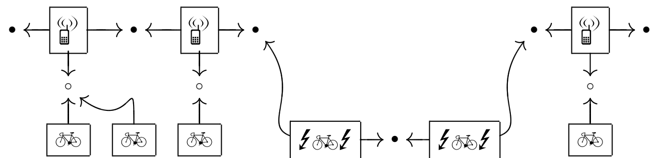


Figure 1: The road assistance scenario.

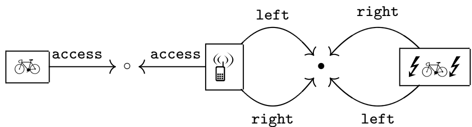


Figure 2: Type Graph T of the running example

3 Formal Model: A Typed Graph Grammar Approach

We model Software Architecture (SA) configurations using typed graph grammars [BBGM07]. Each SA is represented by an hypergraph where components (connectors) are modeled using hyperedges and their ports (roles) by the outgoing tentacles (i.e., labels). Moreover components and connectors are attached together connecting the respective tentacles to the same node. In the following we introduce the fundamental definitions that we will use in this formalization.

Definition 1 (Hypergraph) A *(hyper)graph* is a triple $H = (N_H, E_H, \phi_H)$, where N_H is the set of nodes, E_H is the set of (hyper)edges, and $\phi_H : E_H \to N_H^+$ describes the connections of the graph, where N_H^+ stands for the set of non-empty strings of elements of N_H. We call $|\phi_H(e)|$ the *rank* of e, with $|\phi_H(e)| > 0$ for any $e \in E_H$.

The connection function ϕ_H associates each hyperedge e to the ordered, non empty sequence of nodes n is attached to. An architectural style is just a hypergraph T that describes only the types of ports, connectors, components and the allowed connections. A configuration compliant to such style is then described by the notion of a T-typed hypergraph.

Definition 2 (Typed Hypergraph) Given a hypergraph T (called the *style*), a *T-typed hypergraph* or *configuration* is a pair $\langle |G|, \tau_G \rangle$, where $|G|$ is the *underlying* graph and $\tau_G : |G| \to T$ is a total hypergraph morphism.

The graph $|G|$ defines the configuration of the system, while τ_G defines the (static) *typing* of the resources. We recall that a total hypergraph morphism $f : G \to G'$ is a couple $f = \langle f_N : N \to N', f_E : E \to E' \rangle$ such that: $f_N(\phi_G(e)) = \phi_{G'}(f_E(e))$ (we overload f_N to denote also the homomorphic extension of f_N over strings).

Figure 2 depicts the type graph of our running example. It describes the types of components,

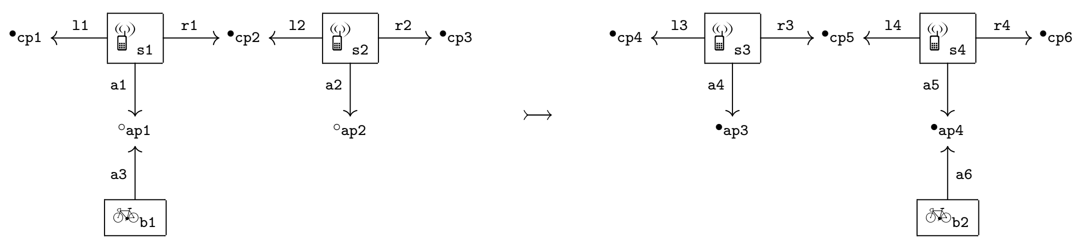


Figure 3: Reconfiguration rule that migrates a bike to the rightward station.

ports and their allowed connections. The typing morphism is defined using the τ_G that maps each element of the configuration in only one element of the type graph T.

Finally, an architecture is described by a T-typed graph grammar.

Definition 3 ((T-typed) graph grammar) A *(T-typed) graph grammar* $\mathscr{G}$ is a tuple $\langle T, G_{in}, Pr\rangle$, where G_{in} is the *initial (T-typed) graph* and Pr is a set of *productions*.

Notation. Let $\mathscr{G} = \langle T, G_{in}, Pr\rangle$ be a (T-typed) graph grammar, and G and H (T-typed) hypergraphs. We write $G \Rightarrow_p G'$ to denote that G is rewritten in one step to G' by using the production $p \in Pr$. We abbreviate the reduction sequence $G_0 \Rightarrow_{p_1} G_1 \Rightarrow_{p_2} \ldots \Rightarrow_{p_n} G_n$ with $G_0 \Rightarrow_{p_1 p_2 \ldots p_n} G_n$. We write $G \Rightarrow^* G'$ to denote that there exists a possible empty sequence $s \in Pr^*$ of derivation steps such that $G \Rightarrow_s G'$.

3.1 Software Architecture Reconfiguration

The reconfiguration of a software architecture is described by a set of rewriting productions that state the possible ways in which a SA configuration may change. Each rule is defined as a partial, injective graph morphism $p : L \rightarrow R$, where L and R are graphs, called the *left-* and *right-hand* side. Give a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation approach [EHK+97]. An application of p to a *host graph* G requires a partial graph morphism m from L to G called a *match*. A rewriting step leads to a *target graph* G'.For each node or edge x in L there exists a corresponding node or edge in G, namely $m(x)$. We have another morphism named r that maps all items from L to R, which are to remain in G during the rewriting application. Elements that are considered in the match m and that have no image under r are to be deleted. The other are preserved. Elements in R which have no pre-image under r are added to G'. r' is a partial morphism, since that elements from G may be deleted and introduced to get G'. New nodes are not in the image of r' but in the image of m'. An example of production is shown in Figure 3, it specifies the migration of a bike from one station to the right station in a chain.

3.2 Programmed Dynamism

In this section we formalize programmed DSA in terms of graph grammars that will be used later to specify our running example. Given a grammar $\mathscr{G} = \langle T, G_{in}, Pr\rangle$, we will use the following notions:

- The set $\mathscr{R}(\mathscr{G})$ of *reachable configurations*, i.e., all configurations to which the initial configuration G_{in} can evolve. Formally, $\mathscr{R}(\mathscr{G}) = \{G | G_{in} \Rightarrow^* G\}$.
- The set $\mathscr{D}_{\mathsf{P}}(\mathscr{G})$ of *acceptable configurations* of an architecture are defined as the graphs that have type T and satisfies a suitable property P.
  Formally, $\mathscr{D}_{\mathsf{P}}(\mathscr{G}) = \{G \mid G \textit{ is a } T\textit{-typed graph} \wedge \mathsf{P} \textit{ holds in } G\}$.

Programmed dynamism assumes that all architectural changes are identified at design time and triggered by the program itself [End94]. A programmed DSA $\mathscr{A}$ is associated with a grammar $\mathscr{G}_{\mathscr{A}} = \langle T, G_{in}, Pr\rangle$, where T stands for the style of the architecture, G_{in} is the initial configuration, and the set of productions Pr gives the evolution of the architecture. The grammar fixes the types of all elements in the architecture, and their possible connections, where the productions state the possible ways in which a configuration may change.

Programmed dynamism enables for the formulation of several verification questions. Consider the set of desirable configurations $\mathscr{D}_{\mathsf{P}}(\mathscr{G})$, then it should be possible (at least) to know whether:

- the specification is correct, in the sense that any reachable configuration is desirable. This reduces to prove that $\mathscr{R}(\mathscr{G}) \subseteq \mathscr{D}_{\mathsf{P}}(\mathscr{G})$, or equivalently that $\forall G \in \mathscr{R}(\mathscr{G}) : \mathsf{P} \textit{ holds in } G$.
- the specification is complete, in the sense that any desirable configuration can be reached. This corresponds to prove $\mathscr{D}_{\mathsf{P}}(\mathscr{G}) \subseteq \mathscr{R}(\mathscr{G})$, or equivalently that *if* P *holds in* G *then* $G \in \mathscr{R}(\mathscr{G})$.

Hence, programmed dynamism provides an implicit definition of desirable configurations. That is, the sets of desirable and reachable configurations should coincide, i.e., $\mathscr{D}_p(\mathscr{G}) = \mathscr{R}(\mathscr{G})$.

4 DSA Structural Verification

4.1 DynAlloy

DYNALLOY [FGLA05] is an extension of the ALLOYmodeling language. It allows us to define atomic actions that modify the state and build more complex actions (programs) from the atomic actions. Atomic actions are defined by means of preconditions and postconditions given as ALLOY formulas. DYNALLOY formulas extend ALLOY formulas with the addition of a construct for building *partial correctness assertions*. From atomic actions we can build more complex programs as follows. If α is an ALLOY formula, then α? is a test action. Operation $+$ denotes the nondeterministic choice between two programs, and ";" denotes their sequential composition. Finally, $*$ iterates a program. A partial correctness assertion of the form $\{\alpha\}p\{\beta\}$ is satisfied when no state that does not satisfy β. is reachable from $\{\alpha\}$ through program p.

One of the important features of ALLOY is the automatic analysis possibilities it provides. In effect, the ALLOY ANALYZER allows us to automatically verify if a given assertion holds in an ALLOY model. Similarly, in [FGLA05] it is showed how to translate DYNALLOY specifications to ALLOY specifications in order to achieve analyzability. This is due to imposing a bound to the depth of iterations (this is equivalent to fixing a maximum length of traces) and efficiently generating the *weakest liberal precondition* of the partial correctness assertion.

4.2 Designing Software Architectures and Styles with Alloy.

The approach described in this section follows [BBGM07] and models dynamic software architectures using typed graph grammars (TGG). The implementation of the approach is based on ALLOY [Jac06, Jac02]. ALLOY provides a logic, based on an extension of first-order logic with relational operators, to represent properties or constraints on the models. We have used this logic to implement concepts like architectural styles, graph transformation rules and architectural properties.

Each software architecture is represented by an hypergraph where components (or connectors) are modeled using hyperedges and their ports (or roles) by the outgoing tentacles (i.e. labels). Moreover, components and connectors are attached together connecting their respective tentacles to the same node. All basic elements (nodes, hyperedges and labels) are implemented in ALLOY using *signatures*. Instead, tentacles are defined as ternary relations between hyperedges, labels and nodes. All these elements are part of a graph signature definition that represents an architecture. Below we show an excerpt of the module TGG implementing the model of graphs. First we see the declaration of the basic signatures Node and Label which stand for nodes and labels. Signature Edge models hyperedges and includes a relation conn between signature labels and node. Note that in ALLOY syntax → indicates a binary relation. The keyword lone preceding Node constraints conn to relate each label to at most one element in Node. Thus, for each element of signature Edge, conn is partial function mapping labels to nodes. The signature Graph is used to define as a graph as structure composed of nodes, hyperedges and labels.

```
sig Node {}
sig Label {}
sig Edge { conn:  Label  -> lone Node }
sig Graph { n: set Node, he: set Edge, l: set Label }
```

After the definition of the signatures, we define a *predicate* (a property to be checked) to determine whether a graph g is well-formed and consistently typed over a type graph h by typing morphism t.

```
sig Tau {
  tauN: set Node -> set Node,
  tauE: set Edge -> set Edge,
  tauL: set Label -> set Label
}
pred isTG [g: Graph, h: Graph, t: Tau]{...}
```

The signature Tau is used to define mapping functions between each SA configuration and the architectural style. Architectural styles consist of a set of elements (components, connectors, ports and roles) that can constitute an architecture (e.g., *Bike_Vocabulary*) plus a set of invariant rules indicating how these elements can be legally connected [SG96]. A SA configuration compliant to such style is then described by the notion of a T-TYPED hypergraph [BBGM07]. We define an ALLOY module called BIKE-STYLE that contains all these elements. The type graph and its items are modeled as instances which we represent by using singleton extensions of signatures.

```
one sig access_Point, chain_point extends Node{}
...
```

```
one sig bike_typegraph extends Graph{}
fact Bike_Vocabulary {
  bike_typegraph.n= access_point + chain_point
  bike_typegraph.he = bike + station + bikestation
  bike_typegraph.l = access + left + right
  bike.conn =  access -> access_point
  ...
}
pred topology_linear[g: Graph, t: Tau]{ no e:g.he | e in e.^next ... }
```

For instance, the node `access_point` that stands for an access point is declared as a singleton (`one`) signature extending `extends` signature `Node`. The type graph depicted in Figure 2 is defined by a *fact*, i.e. a predicate assumed to hold in every considered model. For instance, the type graph definition states that the set of nodes of the type graph is the union (denoted by $+$) of `access_point` and `chain_point`, the signatures that stand for the access and the chain point nodes. Further properties of the style are defined via predicates. For instance, `topology_linear` is used to make sure that stations form an acyclic connected chain. The predicate make use of a negative form of quantification (`no`) and transitive closure ˆof the relation of right-neighborhood `next`. We see the first line of the predicate below where we express that no edge `e` of a graph `g` belongs to the set of edges is reachable from `e` by moving to the rightward adjacent edge.

4.3 Programmed Dynamism with Alloy

We have defined a set of rewrite rules that state the possible ways in which a software architecture configuration may change. Each rule is defined as a partial, injective graph morphism $p : L \rightarrow R$, where L and R are graphs, called the *left-* and *right-hand* side. Give a graph G and a production p, a rewriting of G using p is realised using a single-pushout graph transformation [EHK+97]. A signature `Prod` implements productions as composed of graphs `lhs` and `rhs`, standing for the left- and right-hand side graphs. A reconfiguration is implemented using two distinct predicates (i.e. `rwStepPre` and `rwStepPost`) that are used to verify conditions that must hold in target and the destination graph. For instance, below we see the code of `RwStepPre` that makes use of auxiliar predicates to check the well formedness of graph `G1` being rewritten, the left- and right-hand side graphs of production `R` and `R` itself. In addition, it checks whether there is a match of the left-hand side in `G1`.

```
pred rwStepPre[G1:Graph, R: Prod, style: Graph, t1:Tau, t2:Tau, t3: Tau, M1: Fun, P:Fun]{
  isTG[G1,style,t1]
  isTG[R.lhs,style,t2]
  isTG[R.rhs,style,t3]
  isProd[R,style,t2,t3,P]
  isMatch[R.lhs,G1,style,t2,t1,M1] }
```

An example of a production is shown in Figure 3, it specifies the migration of a bike from one station to the right station in a chain. Textually, we declare the signature of the production as a singleton extension of `Prod` and define facts that characterize the left- and right-hand side graphs:

```
one sig migrate_right extends Prod{}
```

```
fact migrate_right_lhs{
  migrate_right.lhs.n = cp1 + cp2 + cp3 + ap1 + ap2
  migrate_right.lhs.he = s1 + s2 + b1
  migrate_right.lhs.l = l1 + l2 + l3 + l4 + l5 + l6 + l7 }
```

4.4 Verification using DynAlloy

In the Alloy language, the assertions are used to check specifications. Using the Alloy analyzer, it is possible to validate assertions, by searching for possible (finite) counterexamples for them, under the constraints imposed in the specification of the system. DynAlloy, instead, has proposed the use of *actions* to model state change. An action is how it transforms the system state after its execution and moreover actions can be sequentially composed, iterated or composed by nondeterministic choice [FGLA05]. We want to specify that a given property P is invariant under sequences of applications of some operations. In our case this operation is the rewriting step that from an initial graph G and a Production P generates a new graph G'. A technique useful for stating the invariance of a property P consists of specifying that P holds in the initial Graph, and that for every non initial graph and every rewriting operation, the following holds: P(G) and $rwStep(G,G') \rightarrow P(G')$. For this objective we have defined a set o properties that each SA configuration, after a rewriting step must satisfy. In the following we describe each of them with the Alloy code related.

1. **Property 1**: Each bike can be connected to only one access point using one port of type `Access`

```
pred Property1 [tgg: TGG]{
all g: tgg.graphs |all e1: g.he
   |Type[e1,Bike] => one l1: g.l, n1:g.n
   |(Type[n1,Access_Point] and Type[l1,Access]) and
   e1.conn = l1->n1
}
```

2. **Property 2**: The system can not have bikes disconnected and each bike has at most one connection.

```
pred Property2[tgg:TGG]{
   all g: tgg.graphs |
   all e1: g.he |
   Type[e1,Bike] =>  #(e1.conn)=1
}
```

3. **Property 3**: If one bike is connected to an access point then must exist a unique station that is connected to the same access point.

```
pred connected [e1: HE, e2:HE]{
  univ.(e1.conn) = univ.(e2.conn)
}
pred Property3[tgg:TGG]{
  all g:tgg.graphs|
      all e1:g.he  |
      Type[e1,Bike]=> one e2:g.he |
      Type[e2,Station] &&  connected [e1, e2]
}
```

In order to have programmed dynamism and check that at each reconfiguration step a property P is valid, we proceed as follows. For example we want to provide that in each SA configuration *"if one bike is connected to an access point then must exist a unique station that is connected to the same access point"*. This means to provide the predicate `Property3`. For this, we have described in DynAlloy the action `rwStep` that from an initial configuration `G1` and a production `Pr` execute a single rewriting step, generating a new set of typed graphs `tgg'` with the new SA configuration `G2`. The corresponding DYNALLOY specification is:

```
act rwStep [tgg: TGG, Pr: Prod]{
   pre { rwStepPre[G1, Pr, style_bike, t1, t2, t3, M1, P]}
   post { rwStepPost[G1,G2,Pr,style_bike,t1,t2,t3,t4,P,M1,
   M2,F,tgg,tgg']}
}
```

Moreover, by using partial correctness statements on the set of regular programs generated by the set of atomic actions `rwStep[tgg,Pr1]`, `rwStep[tgg,Pr2]`, we can assert the invariance of the `Property3` under finite applications of functions `rwStep` in a simple way, as follows:

```
assert Property3InvAssertion {
  assertCorrectness[tgg:TGG]{
      pre={Property3[tgg]}
      program={(rwStep[tgg,Pr1] + rwStep[tgg,Pr2])*}
      post={Property3[tgg']}
}}
```

The DynAlloy translator allows us to produce an ALLOY model ready to be analyzed using the ALLOY ANALYZER. In order to do so, we need to provide a fixed length to bound the closure operator. In this example, if we set the bound to "n", we will analyze the following finite counterpart:

```
skip+(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])+
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2]);
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])+
...
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2]);...;
(rwStep[tgg,Pr1]+rwStep[tgg,Pr2])
/ ------------- n-times ----------------------/
```

4.5 Example of Analysis

In this section we show two examples of the analysis that we have performed using the ALLOY ANALYZER. *Model finding* is the main analysis capability offered by ALLOY. The ALLOY ANALYZER basically explores (a bounded fragment) of the state space of all possible models. For instance, we can easily use the ALLOY ANALYZER to construct initial configurations: we need to ask for a graph instance satisfying the style facts and having a certain number of bikes, stations and bikestations. In order to test this ALLOY potentiality we have created a module called `MODEL-FINDING` in which only defining elements of our initial configuration (i.e., edges, nodes and labels) we can generate the set of possible software architectures composed of a precise number of components, ports and attachments.

```
module MODEL-FINDING
...
one sig G1 extends Graph{}
fact{
   G1.he=b1+s1+bs1
   G1.n=cp1+cp2+cp3+ap1
   G1.l=a1+a2+l1+r1+l2+r2}
pred show[]{}
run show for 1 Graph, 3 Edge, 4 Node, 6 Label
```

When we run the predicate `show` the ALLOY ANALYZER generates two different SA configurations that are showed in figure 4. The difference between them is the attachments of the bikestation component (i.e., `bs1` in figure 4). In the first case it is attached in the left-side of the station, in the second in the right-side.

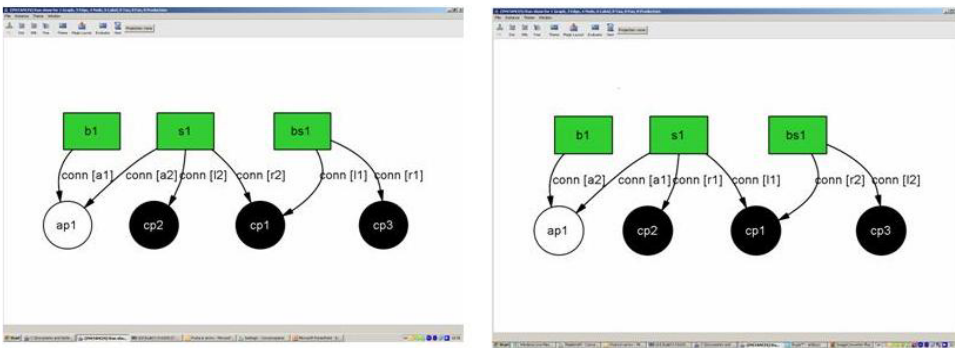


Figure 4: Software Architecture Configurations.

The second kind of analysis that we performed is called *Invariant Analysis*; its the objective to check if a property `P` is invariant under sequences of applications of reconfigurations. In other words we have executed the analysis described in 4.4. To do this we have chosen the *property3* defined before and one initial configuration generated from the previous analysis. We have defined three different modules, `PRODUCTIONS` defines the set of admissible SA reconfiguration (i.e., *Connect_Bike*, *Disconnect_bike*, etc.), `PROPERTIES` defines the set of possible properties that we want to check and `EXECUTION` is the responsible of the invariant analysis. In the last module we have defined the rewriting step action and the *assert* that we execute. The code below enables us to automatically elicit a reconfiguration trace taking the system from one configuration presented in Fig. 4 to the configuration presented in Fig. 5. Moreover, target SA configuration satisfies `Property3`.

```
module EXECUTION
....
assert Property3InvAssertion {
   assertCorrectness[tgg:TGG]
   {
      pre={Property3[tgg]}
      program={(rwStep[tgg,Disconnect_Bike])*}
      post={Property3[tgg']}
}}

check Property3InvAssertion
```

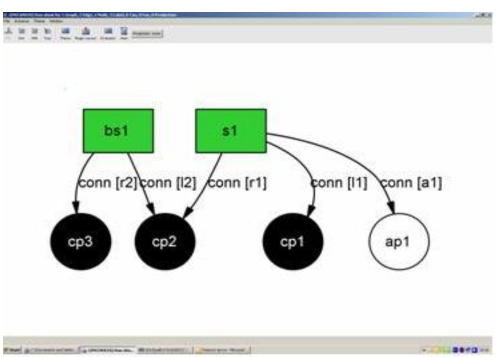


Figure 5: Reconfiguration Result.

5 Related Work

Many research works are focusing on Dynamic Software Architecture specification and validation, they can be sub-divided into three categories. The first, uses formal techniques [BCDW04] such as graph based techniques [Mét98, HIM99, BHTV04, TGM98, WLF01], logic based [End94, AM02] and process algebra based [MK96, ADG, CPT99a]. We use graph grammars and this representation is borrowed from Le Métayer approach [Mét98]. Our scope of this paper are the programmed DSA but different kind of dynamisms as Self-repairing, Ad-Hoc and Constructible have been proposed in [ADG, GS02, GMK02, SG02, Ore96, BCDW04]. The main difference w.r.t. our work is that they are aimed at providing real specification languages while we are aimed at giving an abstract representation of programmed DSA.

The second uses Architectural Description Languages (ADLs) to model adaptive SA with connection and disconnection of components and connectors [OGT+99, MT00, MK96, CPT99b, ADG] by textual or a graphical notation and using also some specific tools to verify them. On one hand, some of the mentioned ADL like Darwin [MK96], and Dynamic Wright [ADG] models DSA and verifies the models with formal techniques. However, these latter did not offer any graphical tools to display these models. On the other hand, some others such as ACME [GMW97], AADL[1],DAOP-ADL[2] and Fractal[3] provides graphical tools for modeling DSA but do not give mechanisms for validating these models. The third focuses on UML architecture modeling extending UML metamodels [KMJ+05, KKJD06] to ensure dynamic architecture and also providing UML diagram translation into formal notation in order to be analyzed[BA05]. An other analysis formal technique that we are thinking to use is Model-Checking. More research works have been done in this direction in the last few years. For example Heckel at al. in [Hec98] proposed an approach where graphs transformations systems are verified using model checking where graphs are interpreted as states and transformation rules as transitions. Rensink in [Ren03] presents a model-checking approach (i.e., GROOVE), based on a graph-specific model checking algorithm, for object-oriented systems. Our idea is to integrate the ”*Model Finding*” aspect of Alloy with Model-Checking in order to have a complete analysis framework for DSA.

[1] http://www.aadl.info/
[2] http://caosd.lcc.uma.es/CAM-DAOP/DAOP-ADL.htm
[3] http://fractal.objectweb.org/

6 Conclusion and Future Work

We have presented an approach to verify Dynamic Software Architectures modeled using Typed Graph Grammars. We have considered the programmed dynamicity of a SA in which all admissible changes (e.g., adding and removing of components, connectors and connections) are defined prior to run-time and are triggered by the system itself. The verification consists of representing this formalism in the ALLOY relational language and since that this kind of dynamicity requires the analysis of situations in which architectural state changes at run-time, we have considered the use of DYNALLOY language, an extension of ALLOY with actions, in order to represent state changes via actions and programs. This work represents our first step towards analysis of DSA. Three important future directions are needed. The first is to extend the approach, proving properties associated to each kind of DSA formalized in [BBGM07]. The second is to consider the verification of behavioral properties, using model-checking, of SAs that change not only the structure at run-time. Finally, in order to have an automatic and extensible framework to model and analyze DSA we are developing an Eclipse-based framework named ARMADA (**A**utomated **R**e**M**orphing **A**mbient for **D**ynamic **A**rchitectures) with the objective to facilitate DSA modeling and analysis within the software development life cycle.

Acknowledgements: Authors thank Hernán Melgratti for his valuable insights about this problem. This work has been partly supported by EU within the FET-GC2 IST-2005-16004 Integrated Project SENSORIA (*Software Engineering for Service-Oriented Overlay Computers*) and by the Italian FIRB Project TOCAI.IT.

Bibliography

[ADG] R. Allen, R. Douence, D. Garlan. Specifying and Analyzing Dynamic Software Architectures. In *Proceedings of FASE'98*.

[AM02] N. Aguirre, T. Maibaum. A Temporal Logic Approach to the Specification of Reconfigurable Component-Based Systems. In *ASE '02*. P. 271. IEEE Computer Society, Washington, DC, USA, 2002.

[BA05] B. Bordbar, K. Anastasakis. UML2ALLOY: A tool for lightweight modelling of discrete event systems. In Guimaraes and Isaeas (eds.), *IADIS AC*. Pp. 209–216. IADIS, 2005.

[BBGM07] R. Bruni, A. Bucchiarone, S. Gnesi, H. Melgratti. Modelling Dynamic Software Architectures using Typed Graph Grammars. In *GTVC'07*. 2007. To appear.

[BCDW04] J. S. Bradbury, J. R. Cordy, J. Dingel, M. Wermelinger. A survey of self-management in dynamic software architecture specifications. In *WOSS*. Pp. 28–33. 2004.

[BHTV04] L. Baresi, R. Heckel, S. Thöne, D. Varró. Style-Based Refinement of Dynamic Software Architectures. In *WICSA*. Pp. 155–166. 2004.

[BLMT07] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto. Style Based Reconfigurations of Software Architectures. Technical report TR-07-17, Dipartimento di Informatica, Università di Pisa, 2007.

[CPT99a] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *Proceedings of the TC2 First Working IFIP Conference on Software Architecture (WICSA1)*. Pp. 107–126. Kluwer, B.V., Deventer, The Netherlands, The Netherlands, 1999.

[CPT99b] C. Canal, E. Pimentel, J. M. Troya. Specification and Refinement of Dynamic Software Architectures. In *WICSA*. Pp. 107–126. 1999.

[EHK+97] H. Ehrig, R. Heckel, M. Korff, 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*. Pp. 247–312. 1997.

[End94] M. Endler. A Language for implementing generic dynamic reconfigurations of distributed programs. In *In Proceedings of BSCN 94*. Pp. 175–187. 1994.

[FGLA05] M. Frias, J. Galeotti, C. Lopez Pombo, N. Aguirre. DynAlloy: Upgrading Alloy with Actions. In *in Proceedings of the 27th. ACM-IEEE International Conference on Software Engineering (ICSE) 2005*. Pp. 442–450. ACM Press, 2005.

[GMK02] I. Georgiadis, J. Magee, J. Kramer. Self-Organising software architectures for distributed systems. In *In Proceedings of the 1th Workshop on Self-Healing Systems*. Pp. 33–38. 2002.

[GMW97] D. Garlan, R. T. Monroe, D. Wile. Acme: an architecture description interchange language. In *CASCON*. P. 7. 1997.

[GS02] D. Garlan, B. R. Schmerl. Model-based adaptation for self-healing systems. In *WOSS*. Pp. 27–32. 2002.

[Hec98] R. Heckel. Compositional Verification of Reactive Systems Specified by Graph Transformation. In *FASE*. Pp. 138–153. 1998.

[HIM99] D. Hirsch, P. Inverardi, U. Montanari. Modeling Software Architecutes and Styles with Graph Grammars and Constraint Solving. In *WICSA*. Pp. 127–144. 1999.

[Jac02] D. Jackson. Alloy: a lightweight object modelling notation. In *ACM Transactions on Software Engineering and Methodology*. 2002.

[Jac06] D. Jackson. *Software Abstractions: Logic, Language, and Analysis.* The MIT Press, 2006.

[KKJD06] M. H. Kacem, A. H. Kacem, M. Jmaiel, K. Drira. Describing dynamic software architectures using an extended UML model. In *SAC*. Pp. 1245–1249. 2006.

[KMJ+05] M. H. Kacem, M. N. Miladi, M. Jmaiel, A. H. Kacem, K. Drira. Towards a UML profile for the description of dynamic software architectures. In *COEA*. Pp. 25–39. 2005.

[Mét98] D. L. Métayer. Describing Software Architecture Styles Using Graph Grammars. *IEEE TSE* 24(7):521–533, 1998.

[MK96] J. Magee, J. Kramer. Dynamic structure in software architectures. In *SIGSOFT '96: Proceedings of the 4th ACM SIGSOFT symposium on Foundations of software engineering*. ACM, New York, NY, USA, 1996.

[MT00] N. Medvidovic, R. N. Taylor. A Classification and Comparison Framework for Software Architecture Description Languages. *IEEE TSE* 26(1):70–93, 2000.

[OGT+99] P. Oreizy, M. Gorlick, R. Taylor, D. Heimhigner, G. Johnson, N. Medvidovic, A. Quilici, D. Rosenblum, A. Wolf. An Architecture-based approach to self-adaptive software. In *Intelligent Systems, IEEE*. Volume 14(3), pp. 54–62. 1999.

[Ore96] P. Oreizy. Issues in the runtime modification of software architecture. *Elettronics Letters* UCI-ICS-96-35, 1996.

[Ren03] A. Rensink. The GROOVE simulator: A tool for state space generation. 2003.

[Roz97] Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. In Rozenberg (ed.), *Handbook of Graph Grammars*. World Scientific, 1997.

[Sen] Sensoria Project. Public Web Site. `http://sensoria.fast.de/`.

[SG96] M. Shaw, D. Garlan. Software Architecture: Perspectives on An emerging Discipline. In *Prentice Hall, NJ. USA*. 1996.

[SG02] B. R. Schmerl, D. Garlan. Exploiting architectural design knowledge to support self-repairing systems. In *SEKE*. Pp. 241–248. 2002.

[TGM98] G. Taentzer, M. Goedicke, T. Meyer. Dynamic Change Management by Distributed Graph Transformation: Towards Configurable Distributed Systems. In *TAGT*. Pp. 179–193. 1998.

[WLF01] M. Wermelinger, A. Lopes, J. L. Fiadeiro. A graph based architectural (Re)configuration language. In *ESEC / SIGSOFT FSE*. Pp. 21–32. 2001.

Reconfiguration of Reo Connectors Triggered by Dataflow

Christian Koehler, David Costa, José Proença, Farhad Arbab

CWI, Amsterdam

Abstract: Reo is a language for coordinating autonomous components in distributed environments. Coordination in Reo is performed by circuit-like connectors, which are constructed from primitive, mobile channels with well-defined behaviour. While the structure of a connector can be modeled as a graph, its behaviour is compositionally defined using that of its primitive constituents. In previous work, we showed that graph transformation techniques are well-suited to model reconfigurations of connectors. In this paper, we investigate how the connector colouring semantics can be used to perform dynamic reconfigurations. Dynamic reconfigurations are triggered by dataflow in the connector at runtime, when certain structural patterns enriched with dataflow annotations occur. For instance we are able to elegantly model dynamic Reo circuits, such as just-in-time augmentation of single-buffered channels to a circuit that models a channel with an unbounded buffer. Additionally we extend Reo's visual notation and the Reo animation language to describe and animate dynamically reconfiguring connectors.

Keywords: Coordination, reconfiguration, graph transformation, animation.

1 Introduction

The coordination paradigm provides models for describing the communication among the components in a composed system. Coordination languages, such as Reo [Arb04], describe the 'gluing' of loosely coupled components, such that a desired system behaviour emerges. The achieved separation of business logic and coordination of the active entities leads to a much cleaner design and helps to handle the greater complexity of large applications. Reo can be applied in various distributed scenarios—from service-oriented to grid computing—as the coordination model is exogenous and independent from the actual component implementation and infrastructure.

In a Reo network, software components are autonomous, self-contained entities that communicate with the outside world via a published interface. To avoid dependencies and to achieve a truly modular, distributed system, Reo proposes the notion of connectors which are used to coordinate the components without their knowledge. By this, the system is divided into two orthogonal aspects: 1) the computation, performed by the components and 2) the coordination of these independent components, performed by the connectors. A major advantage of this design is the ability of changing the topology of the connector, and thereby the behaviour of the system.

The configuration of a Reo connector consists on the interconnection between the structural elements of the connector, together with their states. Communication with the components may change the state of the connector, but not its topology. In this paper, we consider *reconfigurations* of a connector as high-level transformations of its underlying graph structure. Using the theory

of typed, attributed graph transformation, we can directly apply many useful results, such as termination and confluence theorems [KLA07].

In this paper, we explore the interplay between the data flow in a connector and its reconfigurations. For this, we include the connector colouring semantics into the patterns of transformation rules. Transformations are automatically applied depending on the structure, the state and the context of a connector. Connectors are reconfigured at run-time based on this information. This leads to a powerful notion of dynamic connectors. We illustrate the principles through the example of a dynamically growing buffer.

Related Work The logic *ReCTL** was introduced in [Cla08] to reason about connector reconfiguration in Reo. The reconfigurations are performed using the basic primitive operations of the Reo API, e.g. channel creation or node splitting. Up to now we have not provided a technique to reason about our dynamic high-level reconfigurations. However, we plan to look at model checking techniques for graph transformation as proposed in [Ren03a].

Architectural Design Rewriting (ADR) [BLMT07] is a framework for general reconfigurable software architectures. As in our approach, reconfigurations are modeled using graph transformation rules. Reconfigurations occur at run-time whenever the system evolves to a configuration that violates the architectural style of the system. In our case we do not fix the use of the reconfiguration to any particular purpose. The dynamic $FIFO$ introduced in this paper can be seen as an example of the use of reconfiguration to guarantee dataflow. If a write operation is performed while the buffer is full, the buffer is reconfigured to allow data to be stored and dataflow to occur. ADR is not tied to any architectural style (e.g. client-server, peer-to-peer) while in our case, Reo determines the architecture.

A systematic introduction to animations based on graph transformation concepts was given in [Erm06]. The animation language for Reo that we use in this paper was introduced in [CP07]. It is important to note that the animation language for Reo is not based on graph transformation. Instead the authors introduce an abstract animation language that can be used to compute animation descriptions for a connector compositionally out of the descriptions of its constituent primitives.

Structure of the paper This paper is organised as follows. Section 2 gives a general introduction to Reo by introducing the notions of channels, nodes and connectors. An overview of the colouring semantics for Reo is given in Section 3. We recall the concepts of graph-based reconfigurations and provide our contributions to dynamic reconfigurations in Section 4. We discuss the proposed model in Section 5. The status of the current implementation and plans for future work are given in Sections 6 and 7, respectively.

2 Reo Overview

Reo is an exogenous coordination language where complex connectors are compositionally built out of simpler ones. The simplest (atomic) connectors in Reo consist of a user defined set of channels, each of which with its particular constraint policy.

A channel is a medium of communication with exactly two directed ends. There are two types of channel ends: *source* and *sink*. A source channel end accepts data into its channel. A sink channel end dispenses data out of its channel.

A channel can connect two components or be composed with other channels using Reo nodes to build more complex connectors. Reo nodes are logical places where channel ends coincide. A node with only source channel ends is a source node; a node with only sink channel ends is a sink node; and finally a node with both source and sink channel ends is a mixed node. We use the term *boundary nodes* to refer to indistinguishably to source and sink nodes. Boundary nodes define the interface of a connector. Components connect to and interact anonymously with each other through the interface of the connector by performing I/O operations on its boundary nodes: *take* and *read* operations on sink nodes, and *write* operations on source nodes.

Reo fixes the constraint policy for the dataflow in Reo nodes. Data flows through a node only if at least one sink channel end is pushing data and all the source channel ends can *accept* a copy of the data. In case more than one sink channel end is pushing data, one is picked non-deterministically and all the others are excluded. Data cannot be stored in a node, hence its constraints on dataflow and exclusion must propagate through the connector.

Resolving the composition of the constraint policies of a connector consisting of several channels and nodes is a non-trivial task. In Figure 1 we present two examples of Reo connectors that illustrate how non-trivial dataflow behaviour emerges from composing channels using Reo nodes. The constraints propagate through the (synchronous regions of the) connector to the boundary nodes. The propagation enables a certain context-awareness in connectors. A detailed discussion of this can be found in [CCA07].

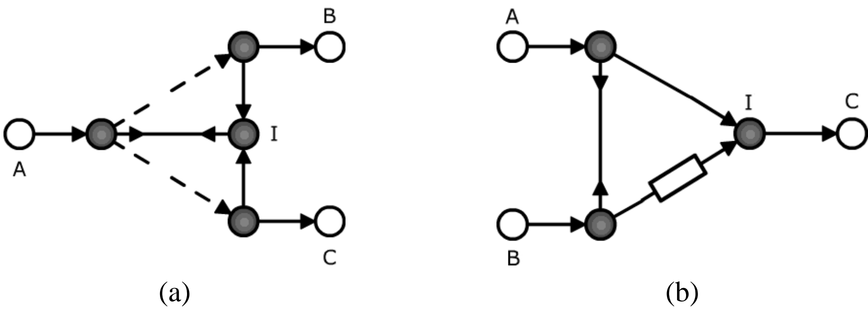


Figure 1: (a) exclusive router, (b) ordering connector.

The two connectors in Figure 1 involve, in total, four different types of channels. We represent mixed nodes as filled circles (●), and boundary nodes as empty circles (○). The *Sync* channel (⟶) synchronously takes a data item from its source end and makes it available at its sink end. This transfer can succeed only if both ends are ready to communicate. The *LossySync* (- - - - ►) has the same behavior, except that it does not block if the receiver cannot accept data. In this case, the written data item is accepted and destroyed by the channel. The $FIFO_1$ (—▭→) is an asynchronous channel that has a buffer of size one. Unlike the prior channels, $FIFO_1$ is a stateful channel. The *SyncDrain* channel (→——←) has two source ends through which it can only consume data, and no sink ends. Its behavior can be described as follows: if there are data

items available at both ends, it consumes (and looses) both of them atomically.

The exclusive router, shown in Figure 1a, routes data from A to either B or C. The connector can accept data only if there is a write operation at the source node A, and there is at least one component attached to the sink nodes B or C, which is performing a take operation. If both B and C have a take operation, the choice of whether data is routed to B or C is made non-deterministically by the mixed node I. Node I can accept data only from one of its sink ends. To the other end it gives an exclusion reason for data not to flow, which forces the *LossySync* to lose the data.

The second connector, shown in Figure 1b, imposes an ordering on the flow of the data from the input nodes A and B to the output node C. The *SyncDrain* enforces that data flows through A and B synchronously. The empty buffer together with the *SyncDrain* guarantee that the data item obtained from A is delivered to C whereas the data item obtained from B is stored in the $FIFO_1$ buffer. At this moment the buffer of the $FIFO_1$ is full and data cannot flow in through either A or B, but C can obtain the data stored in the buffer. The buffer is then empty again.

These informal descriptions of the behavior of connectors can be formalised using the connector colouring semantics, introduced in [CCA07]. The colouring semantics is used to generate animations and to implement Reo, and we discuss it in Section 3.

Reo offers a number of operations to reconfigure and change the topology of a connector at run-time. Operations that enable the dynamic creation of channels, splitting and joining of nodes, hiding internal nodes and more. The hiding of internal nodes is important concerning reconfiguration, because it allows to fix permanently the topology of a connector, such that only its boundary nodes are visible and available. The resulting connector can be viewed as a new primitive connector, or primitive for short, since its internal structure is hidden and its behaviour is fixed. Reconfiguration is impossible on a primitive. We have the basic primitives that include the user defined channels, the Reo nodes, and the I/O operations. Plus the non-basic primitives constructed through the use of the hiding operation.

3 Connector Colouring Semantics

Connector Colouring semantics is based on the idea of colouring a connector using a set of colours $\mathscr{C}olour$. We consider a set $\mathscr{C}olour$ with three colours as in Clarke et al. [CCA07]. One dataflow colour (—) to mark places in the connector where data flows and two colours for no-dataflow (−▷−, −◁−) to mark the absence of dataflow. The reason for having two distinct no-dataflow colours is to be able to trace the exclusion constraints responsible for the no-flow back to their origins. Graphically, the arrow indicates the direction of exclusion, i.e. it points away from the exclusion reason and in the direction that the exclusion propagates.

Colouring a Reo connector in a specific state with given boundary conditions (I/O operations) provides a means to determine the route alternatives for dataflow. Each colouring of a connector is a solution to the synchronization and exclusion constraints imposed by its channels and nodes.

The dataflow allowed by a connector is collected in a *colouring table* whose elements—*colourings*—are functions mapping each node of the connector to a *colour*. The different colourings present in a colouring table of a connector correspond to the alternative ways that the connector can behave in the different contexts where it can be used.

We recall some essential definitions from [CCA07] that formalise the notion of a colouring and of a colouring table. Let $\mathcal{N}ode$ be a finite set of node names.

Definition 1 (colouring) A *colouring* $c : N \to \mathcal{C}olour$ for $N \subseteq \mathcal{N}ode$ is a function that assigns a colour to every node of a connector.

Definition 2 (colouring table) A *colouring table* T over nodes $N \subseteq \mathcal{N}ode$ is a set of colourings with domain N.

To give semantics to a Reo connector using connector colouring one must provide the colouring tables for the user defined primitives, the channels, used in the construction of that connector. Table 1 shows the channels we use in this paper and their respective colouring tables. We highlight a few points of interest in this table, focusing only on reasons to exclude dataflow. No dataflow at one end of a *Sync* or *SyncDrain*, is enough to prevent dataflow in the all channel. The reason is propagated to the other end. An empty $FIFO_1$ buffer does not enable data flow on its output end, giving a reason for no dataflow. Dually, a full $FIFO_1$ buffer gives a reason for having no dataflow on its input end. The second entry of the table for a *LossySync* states that it will lose the data only when a reason for no dataflow is propagated into its output end, which amounts to saying that the channel is unable to transfer the data.

Reo fixes the colouring tables for the other primitives: nodes and the I/O operations. The Table 2 gives a brief account of the connector colouring semantics for these primitives. To comply with the page limit we omit the general colouring table of Reo mixed nodes. We give an example of one possible colouring for a Reo mixed node with 3 source ends and 2 sink ends. For the purpose of this paper that should suffice without compromising the understanding of what follows. For a full description we refer to [CCA07].

Definition 3 (primitive) A labelled tuple $(n_1^{j_1},\ldots,n_k^{j_k})_c$ represents a primitive connector, c, where for $0 < \ell \leq k$, $n_\ell \in \mathcal{N}ode$, $j_\ell \in \{i,o\}$, $k \geq 1$ is the arity of the primitive, and the labels i and o indicate a source node or a sink node respectively, such that a node n appears at most as n^i and/or n^o in $(n_1^{j_1},\cdots,n_k^{j_k})_c$. A primitive with colouring is a pair of a primitive with a colouring table T whose domain ranges over the nodes of the primitive.

$(n_1^i,n_2^o)_{Sync}$	$(n_1^i,n_2^i)_{SyncDrain}$	$(n_1^i,n_2^o)_{LossySync}$	$(n_1^i,n_2^o)_{FIFO_1}$	$(n_1^i,n_2^o)_{FIFO_1[x]}$
				x x

Table 1: User defined channels, and their colouring tables.

A connector is a collection of primitives composed together, satisfying some well-formedness conditions. As such, the colouring table of a connector is computed from the colouring tables of its constituents.

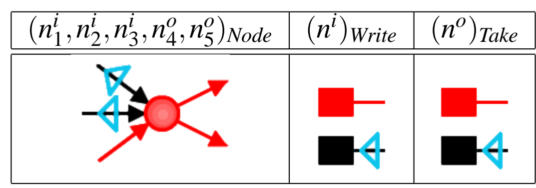


Table 2: Reo primitives, and their colouring tables.

Definition 4 (connector) A connector C is a tuple $\langle N, B, E, T\rangle$ where, N is the set of nodes that appear in E; $B \subseteq N$ is the set of boundary nodes; E is a set of primitives; T is a colouring table over N; such that (1) $n \in B$ if and only if n appears exactly once in E, and (2) $n \in N \setminus B$ if and only if n occurs exactly once as n^o and as n^i in E.

A primitive with a colouring table can straightforwardly be considered as a connector. A connector's semantics is computed by joining the tables of its constituents. Two colourings can only be composed if the common nodes in their domains are coloured with the same colour.

Definition 5 (join) Let $C_k = \langle N_k, B_k, E_k, T_k\rangle$ with $k \in \{1,2\}$ be connectors such that $(N_1 \setminus B_1) \cap (N_2 \setminus B_2) = \emptyset$, and for each $n \in B_1 \cap B_2$, n^i appears in E_1 and n^o appears in E_2, or vice versa. The join of C_1 and C_2, is given by: $C_1 \odot C_2 \doteq \langle N_1 \cup N_2, (B_1 \cup B_2) \setminus (B_1 \cap B_2), E_1 \cup E_2, T_1 \cdot T_2\rangle$, where $\cdot$ is the join operator for two colouring tables defined as:

$$T_1 \cdot T_2 \doteq \{c_1 \cup c_2 \mid c_1 \in T_1, c_2 \in T_2, n \in (dom(c_1) \cap dom(c_2)) \Rightarrow c_1(n) = c_2(n)\}.$$

Figure 2 depicts two colourings of the ordering connector of Figure 1(b). In both colourings, a component is connected to each boundary node and performs an I/O operation: *Write* on both source nodes and *Take* on the sink node. The colouring in (a) describes the dataflow behaviour

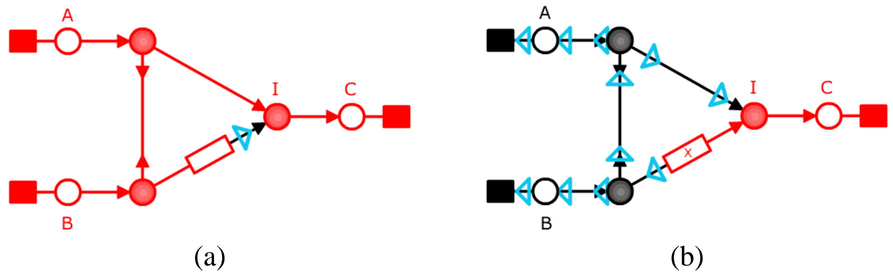


Figure 2: (a) and (b) are two possible colourings of the ordering connector.

of the connector when the buffer is empty, indicating that data flows through the entire connector except for the sink end of the $FIFO_1$ channel, data is stored in the buffer, and all three I/O operations succeed. This dataflow changes the state of the $FIFO_1$ channel, changing also its colouring table. The colouring in (b) describes the dataflow behaviour of the connector when the buffer is full. This colouring states that data flows in the connector only at the sink end of the

FIFO$_1[x]$ channel, the buffer is emptied, the take operation on node C succeeds, and the write operations on nodes A and B are delayed.

Figure 3 depicts the two colourings of the exclusive router that are the valid behaviour alternatives when a component is connected to each boundary node and performs an I/O operation: *Write* on the source node and *Take* on both the sink nodes. The colouring in (a) describes the

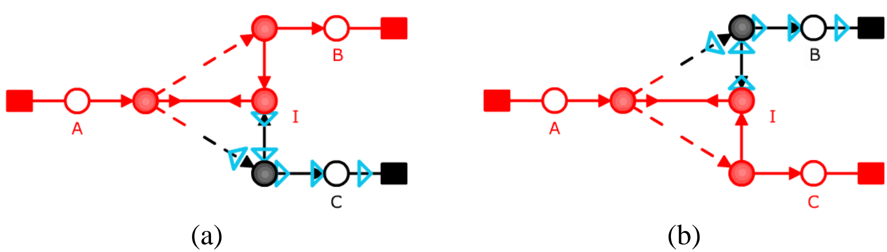


Figure 3: (a) and (b) are two possible colourings of the exclusive router.

dataflow behaviour of the connector when the mixed node I picks, non-deterministically, node B to route the data to. Alternatively the colouring in (b) describes the dataflow behaviour of the connector when the mixed node I picks, non-deterministically, node A to route the data to.

4 Connector Reconfiguration

It has been shown in [KLA07] that the theory of graph transformation can be applied to model connector reconfigurations as high-level transformations of their underlying graph structures. The approach allows to define reconfigurations at a high level of abstraction and therefore can be used to model complex reconfigurations, e.g. refactorings. While in the previous work, we considered these techniques in the context of business process customisation, we combine now these transformations with the connector colouring semantics, described in Section 3.

4.1 Reconfiguration Triggered by Dataflow

Dynamic reconfigurations are transformations of connectors at run-time. In the following, we present a framework that allows to define such dynamic reconfigurations by annotating transformation rules with colourings, which leads to a notion of dynamic connectors.

To use graph transformation for connector reconfiguration we make the following assumptions. Connectors are considered as typed, attributed graphs in the following way: i) Reo nodes are vertices of the graph and ii) channels are its edges. The typegraph in this scenario consists of a single node and one edge for each channel type. Edge attributes are used to model channel properties, e.g. the content of a full *FIFO*$_1$. Since channels in Reo are not necessarily directed (cf. the *SyncDrain* channel) we simply assert an underlying direction of the channel to fit the formal model of directed graphs that is usually assumed. Note also that we have given a formal definition of connectors in [KLA07] and showed that it indeed forms an adhesive High-Level Replacement category [EEPT06].

We use the Double-Pushout (DPO) approach [EEPT06] for our connector reconfigurations. Our transformation rules are an extended version of the usual spans of morphisms in the DPO approach. We write a connector reconfiguration rule as

$$p = (\mathscr{C}olour \xleftarrow{c} L \xleftarrow{l} K \xrightarrow{r} R)$$

where L, K and R are connectors (typed, attributed graphs), l, r are connector (typed, attributed graph) homomorphisms and c is a (potentially partial) colouring for the left-hand side of the rule. The rationale behind this extension is that we do not just want to match the structure of a particular connector part, but also its state and the current execution strategy. Note that this introduces a certain asymmetry to the rules, caused by the fact that only the left-hand side is coloured. This is also the reason why we do not model the colouring as attributes of the graphs. Such an extended rule can be applied with respect to a given match $L \xrightarrow{m} M$ and a current colouring $M \xrightarrow{k} \mathscr{C}olour$ iff

1. the gluing condition holds (see [EEPT06] for more details); and
2. the rule colouring matches the current colouring: $c = k \circ m$.

With the latter constraint we extend the pattern of a rule, in the way that a specific colouring has to be matched as well. A transformation rule can and will be applied only if the structure can be matched and a specific behaviour occurs. The extended version of the DPO approach can be summarised as shown in the diagram

$$\begin{array}{ccccccc} \mathscr{C}olour & \xleftarrow{c} & L & \xleftarrow{l} & K & \xrightarrow{r} & R \\ & = \;\nwarrow^{k} & \downarrow m & (PO) & \downarrow & (PO) & \downarrow \\ & & M & \longleftarrow & C & \longrightarrow & N \end{array}$$

where $\mathscr{C}olour$ is a fixed set of possible flow-colours and c and k are colourings. Including the colourings of a connector part makes the pattern matching much more restrictive. In fact, it is so restrictive now that the reconfiguration rules can be invoked by the Reo engine without 'supervision'. The transformation system becomes in some respect autonomous. The connector is transformed when necessary, without the need for an external party to trigger the reconfiguration.

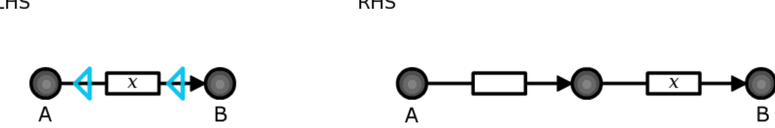


Figure 4: reconfiguration rule for a dynamic $FIFO$.

Figure 4 shows a reconfiguration rule for a dynamic (unbounded) $FIFO$. The matches from the LHS to the RHS are indicated by using the same node labels. The reconfiguration rule gives rise to a dynamic connector, which we call $FIFO_\infty$. The $FIFO_\infty$ consists of a sequence of $FIFO_1$ channels and a reconfiguration rule. The reconfiguration adds a new $FIFO_1$ in the

beginning whenever the $FIFO_\infty$ is full and someone tries to write to it assuring that way that the write can succeed and the data can flow and be stored in the buffer. The left-hand side matches an arbitrary full $FIFO_1$ with content x where someone tries to write to. The reason for the no-flow is the fact that the $FIFO_1$ is full already, not that there is no data available. If this pattern can be matched, the rule states that the original channel is destroyed and two new $FIFO_1$ channels are created in its place. The second $FIFO_1$ is filled with the original content x.

However, the rule as it is now does not reflect our initial requirement that only the first $FIFO_1$ in the sequence is replaced by two new ones. This can achieved by adding negative application conditions [EEPT06] as shown in Figure 5. We need two extra NACs for the dynamic $FIFO_\infty$

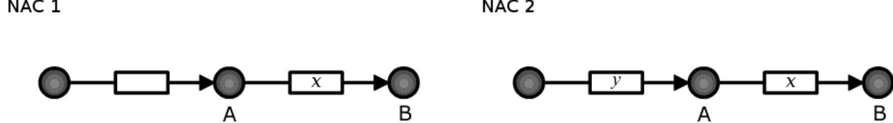


Figure 5: negative application conditions for the $FIFO_\infty$.

that restrict *where* the rule can be applied. An empty $FIFO_1$ should always be added at the very beginning of the sequence. Expressed as negative application conditions, this means that there must not be an empty or a full $FIFO_1$ in front of the one where the original rule (Figure 4) applies.

These additional restrictions allow us to apply the transformation rule automatically at runtime when the colouring occurs. For completeness we would have to define also an inverse rule that shrinks the connector again. We omit this here.

4.2 Run of a connector

Connectors are executed in an abstract Reo engine. The Reo engine includes two independent components, one to compute colouring tables and to perform the dataflow, and one for computing reconfiguration matches and executing the transformations. We refer to these components as **Df** and **Tr** respectively. Reconfiguration rules are applied locally only in specific regions of the connector. These regions can be formally viewed as disjoint sub-graphs that restrict the domain of the transformations. These regions are the reconfigurable parts of the connector. Each of these regions has a number of reconfiguration rules attached to it, such as the one in Figure 4.

The Reo engine utilises **Df** and **Tr** to execute dynamic connectors. In this scenario a run of the engine consists of performing the following actions:

1. Invoke **Df** to compute the colouring table CT of the connector for the actual boundary conditions.

2. Choose non-deterministically a colouring k from the colouring table CT.

3. For each reconfiguration region, invoke **Tr** to find pattern matches $m_1, \ldots, m_n$ for the colouring k.

4. Invoke **Df** to execute the dataflow according to k. The state of the connector is updated.

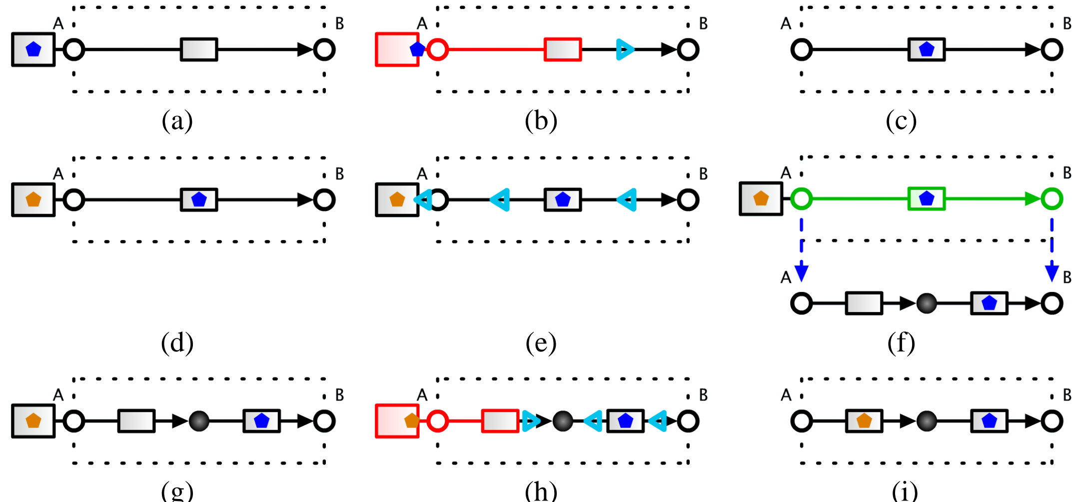


Figure 6: run of the $FIFO_\infty$.

5. For each pattern match in $m_1, \dots, m_n$, invoke **Tr** to perform the transformations.

Since the steps 3 and 4 are independent from each other, they can be performed in parallel. Note also that in general, multiple transformation policies could be supported, e.g. apply at most once or until it is not applicable anymore.

Figure 6 depicts a run according to the descriptions above, for the $FIFO_\infty$. Part (a) shows the basic connector that we want to reconfigure. It consists of a single empty $FIFO_1$ with a $Write$ operation. The region where the transformation rule from Figure 4 should be applied is delimited by the dashed box. Part (b) shows the colouring k_1 and the corresponding dataflow. The transformation engine is invoked with a snapshot of the reconfiguration region and the colouring k_1. Since the rule does not match colouring k_1 no transformation is performed. In part (c) we see how the dataflow has changed the state of the $FIFO_1$ and that the write operation disappeared after succeeding. In part (d) a new write operation is attached to the connector. At this point a new run of the engine starts. Figure 6e shows that the new write operation cannot succeed because the $FIFO_1$ is already full. This event is formally observed by the no-flow colouring k_2. The transformation engine is invoked again and returns a match m this time, because k_2 matches the colouring of the reconfiguration rule. Since no dataflow has to be performed, the transformation can immediately be applied, as shown in Figure 6f. The reconfigured connector is shown in part (g). At this point, the write operation succeeds, and the corresponding colouring k_3 is depicted in part (h). There is no rule that matches k_3, since the only possible match with the full $FIFO_1$ is restricted by NAC 1, defined in Figure 5. In Figure 6i we show the state of the connector after the dataflow, whereas the write operation is removed and the left $FIFO_1$ becomes full.

The visual representation we use is based on an extension of the animation language introduced in [CP07]. We use the colour green to highlight the part of the connector that matches the left-hand side of the rule, as depicted in the region inside the dashed box in Figure 6f, and

the connector that appears below corresponds to the right-hand side of the rule. The dashed blue arrows in Figure 6f, pointing from the match to the substitute connector, indicate that the nodes A and B are preserved by the transformation. Furthermore, the colour of the new data token matches the colour of the data token in the matched part of the connector, expressing that the argument of the full $FIFO_1$ channel is the same as the argument of the new full $FIFO_1$.

5 Discussion

Up to now we have shown that by applying graph transformation techniques to Reo, we are able to describe complex connector reconfigurations using a very compact notation. Transformation rules are extended with colouring information so that they can be applied when a certain behaviour occurs. External invocation of the transformation is not required.

In the current setup, the transformation engine is invoked in each run of the execution of the connector. Even if the required colouring is not present, the engine first tries to match the structural pattern of the current connector and then validates its colouring. If the reconfiguration regions become larger, the pattern matching can cause a significant degradation of performance. Note that general pattern matching for graphs is a problem that is known to be NP-complete.

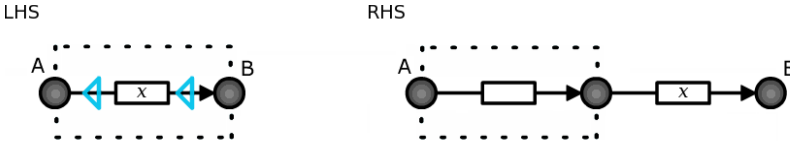


Figure 7: rule for a $FIFO_\infty$ with an evolving reconfiguration region.

To improve the performance of dynamically reconfiguring connectors, we suggest the following optimisation. We augment the transformation rules with information about the reconfiguration regions. For our example of the dynamic $FIFO$ we extend the rule as shown in Figure 7. The dashed boxes mark the reconfiguration regions. With this extension we can state that the rule is always applied to the first $FIFO_1$. While before the reconfiguration region was growing with each application of the rule, it will now remain as consisting of exactly two nodes and one $FIFO_1$ only, all the time. This way, the pattern matching is much faster and (in the ideal case) always uniquely determined.

6 Implementation

Reo as a modelling framework The Reo modelling framework [ECT] consists of a set of plug-ins for the Eclipse[1] development environment such as: a graphical editor, a model checker and a Flash-based animation tool.

Executable instances of connectors can be derived by generating code from Reo specifications or by interpreting these specifications. We generate code from abstract animation descriptions

[1] http://www.eclipse.org

when producing the Flash animations [CP07]. We have implemented an interpreter[2], based on the colouring semantics, for the application domain of web services. Currently we are integrating the dynamic reconfiguration scheme into the interpreter.

Reo as a runtime architecture We are also developing a distributed Reo engine [Pro07], where each primitive is deployed and executed as an independent block running in parallel with the other primitives. There is no centralized entity computing the colouring table of the composite connector at each step, but instead the colouring table is obtained by distributed agreement. The engine has a mechanism to pause regions of the connector, making them suitable to be reconfigured. However, since the development is still in an early stage we are not able to integrate the dynamic reconfiguration yet.

7 Future work

The example of the dynamic $FIFO$ is certainly very basic. In general, there may be more than one reconfiguration rule (a graph grammar) attached to a region of a connector. The role that the colouring extension in our rules plays in these more complex scenarios must be investigated. Furthermore, we need to make the notion of the evolving reconfiguration regions, as suggested in Section 5, more specific. Additional partial mappings from the LHS to RHS may be a possible approach to model this.

We are interested in preservation of behavioural properties by transformations. Two different classes of transformations are interesting in this setting: transformations that preserve the behaviour and transformations that change it. On the one hand, reconfigurations that do not change the behaviour are interesting in the area of automated refactoring and optimisation. On the other, reconfigurations that change the behaviour can be used to implement new sub-circuits that adapt the behaviour of a circuit based on dataflow. In this context, it will be interesting to do static analysis of reconfiguration rules to reason about behaviour preservation.

Finally, we want to integrate the dynamic reconfiguration triggered by dataflow into the Reo tools. For this purpose, we will extend both the Eclipse based development tools and the Reo runtime engine for web services. Since we model reconfiguration through graph transformation, and due to the fact that our implementation of the Reo development tools is based on the Eclipse Modeling Framework (EMF)[3], we plan to implement the reconfiguration extensions using the Tiger EMF Transformation tools [EMT].

Bibliography

[Arb04] F. Arbab. Reo: a Channel-based Coordination Model for Component Composition. *Mathematical Structures in Computer Science* 14:329–366, 2004.

[2] http://www.cwi.nl/~koehler/services
[3] http://www.eclipse.org/emf

[BEK+06] E. Biermann, K. Ehrig, C. Koehler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Model Driven Engineering Languages and Systems (MoDELS'06)*. 2006.

[BLMT07] R. Bruni, A. Lluch-Lafuente, U. Montanari, E. Tuosto. Style-based architectural reconfigurations. Technical report TR-07-17, Computer Science Department, University of Pisa, 2007.

[CCA07] D. Clarke, D. Costa, F. Arbab. Connector Colouring I: Synchronization and Context Dependency. *Sci. Comput. Program.* 66(3):205–22, 2007. doi:http://dx.doi.org/10.1016/j.scico.2007.01.009

[Cla08] D. Clarke. A Basic Logic for Reasoning about Connector Reconfiguration. *Fundamenta Informaticæ* 82:1–30, 2008.

[CP07] D. Costa, J. Proença. Connector Animation: A Compositional Framework to Analyse Reo Connectors. December 2007. Submitted. Available online: http://www.cwi.nl/~costa/publications/publications.htm.

[ECT] Eclipse Coordination Tools. http://homepages.cwi.nl/~koehler/ect.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EMT] Tiger EMF Transformation Project. http://tfs.cs.tu-berlin.de/emftrans.

[Erm06] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, 2006.

[KLA07] C. Koehler, A. Lazovik, F. Arbab. Connector Rewriting with High-Level Replacement Systems. In *Proceedings of FOCLASA 2007, to be published in Electronic Notes in Theoretical Computer Science*. 2007.

[Pro07] J. Proença. Towards Distributed Reo. Talk presented at CIC workshop, 2007.

[Ren03a] A. Rensink. GROOVE: A Graph Transformation Tool Set for the Simulation and Analysis of Graph Grammars. http://www.cs.utwente.nl/~groove, 2003.

[Ren03b] A. Rensink. Towards Model Checking Graph Grammars. In Leuschel et al. (eds.), *Workshop on Automated Verification of Critical Systems (AVoCS)*. Technical Report DSSE–TR–2003–2, pp. 150–160. University of Southampton, 2003.

Negative Application Conditions for Reconfigurable Place/Transition Systems

A. Rein, U. Prange, L. Lambers, K. Hoffmann, J. Padberg

Technische Universität Berlin
Institut für Softwaretechnik und Theoretische Informatik

Abstract: This paper introduces negative application conditions for reconfigurable place/transition nets. These are Petri nets together with a set of rules that allow changing the net and its marking dynamically. Negative application conditions are a control structure that prohibits the application of a rule if certain structures are already existent. We motivate the use of negative application conditions in a short example. Subsequently the underlying theory is sketched and the results – concerning parallelism, concurrency and confluence – are presented. Then we resume the example and explicitly discuss the main results and their usefulness within the example.

Keywords: Petri net, net transformation, control structure, negative application condition

1 Introduction

As the adaptation of a system to a changing environment gets more and more important, Petri nets that can be transformed during runtime have become a significant topic in recent years. Application areas cover, among others, computer supported cooperative work, multi-agent systems, dynamic process mining and mobile networks. Moreover, this approach increases the expressiveness of Petri nets and allows a formal description of dynamic changes. In [HEM05], this concept of reconfigurable place/transition (P/T) systems has been introduced where the main idea is the stepwise development of P/T systems by rules where the left-hand side is replaced by the right-hand side preserving a context. We use rules and transformations for place/transition systems in the sense of the double pushout approach for graph transformation (see [EP04]). More precisely, adhesive high-level replacement (HLR) systems – a suitable categorical framework for double pushout transformations [EEPT06] – have been instantiated to P/T systems.

For the suitable application of such rules specific control structures are needed, especially the possibility to forbid certain rule applications. These are known from graph transformation systems as negative application conditions (NACs). These conditions restrict the application of a rule forbidding a certain structure to be present before or after applying a rule in a certain context. Such a constraint influences thus each rule application or transformation and therefore changes significantly the properties of the replacement system.

By proving that P/T systems are weak adhesive HLR categories with negative application conditions we can transfer well-known and important results to this case: Local Church-Rosser Theorem, Completeness Theorem for Critical Pairs, Concurrency Theorem, Embedding and Extension Theorem and Local Confluence Theorem or Critical Pair Lemma.

This paper is organized as follows: first we introduce our example and discuss the need of additional control structures for the application of rules in Section 2. Then we review the formal notions for reconfigurable P/T systems in Section 3. Based on these notions we define negative application conditions and present the main results concerning parallelism, concurrency and confluence in Section 4. We discuss some of the general results with respect to the example in Section 2. Concluding remarks concern future and related work.

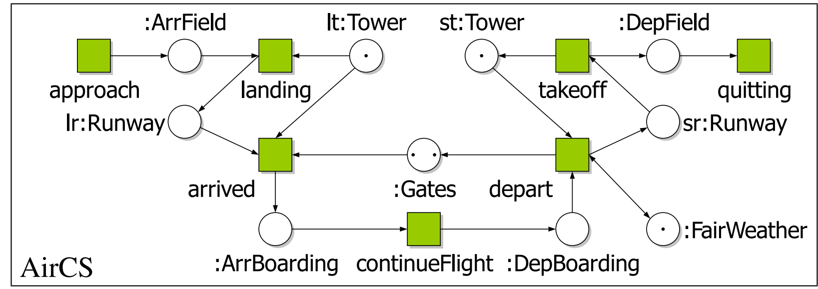
2 Example: Airport

In this section, we present an example for a reconfigurable P/T system with NACs. We model an airport control system (AirCS) that organizes the starting and landing runways of an airport. The P/T system has to ensure that certain safety properties of an airport are fulfilled, for example that some areas of the airport like the actual runways are secure, i.e. exclusively used by one airplane at the time. The AirCS is able to adapt to various changes of the airport. Changes at runtime may concern the opening or closing of a runway or its kind of use, i.e. for starting or landing. As a basic condition we require that there has to be at least one landing runway to ensure the landing of arriving airplanes, especially in emergency situations.

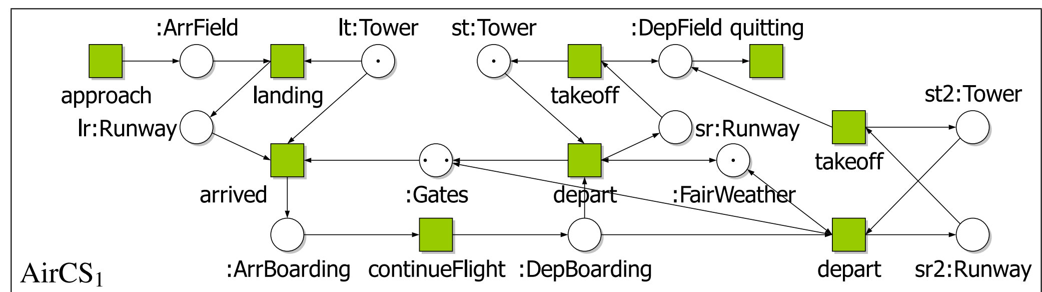
In addition, the use of the starting runways depends on the weather. Under fair weather conditions, no limitations occur. But it is possible for the system to receive a storm warning from a weather information channel. In this case, no more starting runways shall be opened and when the storm arrives it shall be forbidden for airplanes to depart.

In the top of Fig. 1, the standard AirCS with one starting and one landing runway is depicted. Each runway consists of two places of type Runway and Tower, with exactly one token on either the one or the other place. A token on Runway represents an airplane on this runway, while a token on Tower means that this runway is currently not in use. In addition, a landing runway consists of transitions landing and arrived, and a starting runway of transitions depart and takeoff, respectively. By firing the transition approach an airplane appears in the airspace of the AirCS. The transition landing may fire if the runway is currently not used leading to a token representing an airplane on the runway. In the lower part of the P/T system, the gates area is modeled. The place Gates is a counter for the available gates. If a gate is available, the airplane may proceed to a gate by firing the transition arrived. If the deboarding process is completed, the firing of the transition continueFlight initiates the boarding process, and using the transitions depart and takeoff an airplane may depart over an available starting runway. Analogously to approach, the firing of the transition quitting models that the airplane leaves the airspace of the AirCS.

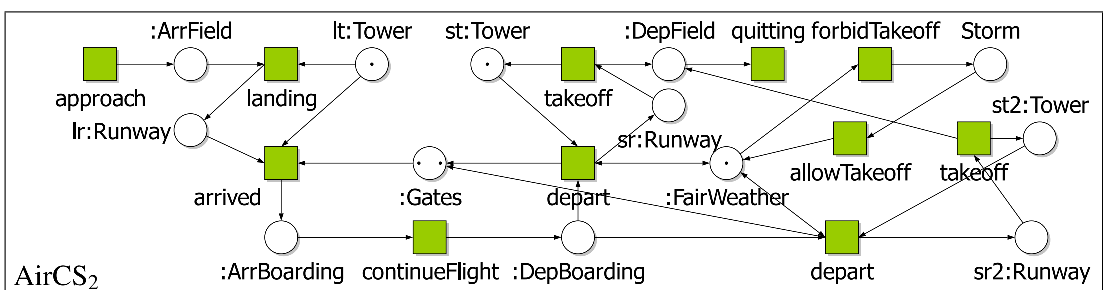
In the following, we describe the rules for changes of the airport. The rules openStartingRunway and openLandingRunway in the top of Fig. 2 are used to open a new runway. The places and transitions of the runway are inserted and connected with the already existing part of the airport. For the rule openStartingRunway, an additional NAC is needed to prevent the opening of a starting runway in case of a storm warning. To apply a rule to our P/T system, we first have to find a match of the left-hand side L to the P/T system. In case of the rule openStartingRunway, this match is unique and consists of the four places in L. If we have a NAC for the rule, we have to check if this NAC is valid, which means that we are not allowed to find a morphism from the NAC to our P/T system via the match. For the match of openStartingRunway, the NAC is ful-



⇓ openStartingRunway



⇓ incomingWarning



⇓ closeStartingRunway

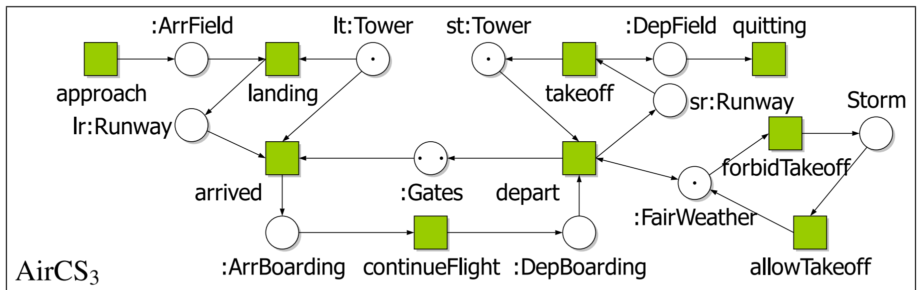


Figure 1: The standard AirCS and a transformation sequence

filled as long as no place of type Storm exists. In addition, a gluing condition has to be fulfilled to make sure the rule is applicable (see Def. 2 in Section 3). Then, we first delete those elements that are no longer needed and insert those elements that shall be created. They are glued to the already existing elements, for example when applying openStartingRunway the new transitions are connected to the places in the match. The application of the rule openStartingRunway to the AirCS from Fig. 1 is the P/T system AirCS_1 in the upper middle of Fig. 1.

The rules closeStartingRunway and closeLandingRunway in the upper middle of Fig. 2 are used to close a runway. closeStartingRunway is the inverse rule of openStartingRunway without a NAC, since closing a starting runway is always allowed. For closeLandingRunway, we have to make sure that we do not delete the last landing runway, thus another landing runway has to be present in the match.

In the lower middle of Fig. 2, the rules changeLandingToStartingRunway and changeStartingToLandingRunway for changing the kind of a runway are shown. For both rules, we have a NAC that forbids that the runway is currently used by an airplane, represented by a token on the runway which is not present in the left-hand side. This ensures that the type of the runway is not changed during its use causing strange behaviour for incoming or outgoing flights. In addition, there is a second NAC that forbids the application of changeLandingToStartingRunway in case of a storm. In the NACs, the place types are omitted if they are obvious from the left-hand side.

The arriving of a storm warning is also modeled by a rule as shown in the bottom of Fig. 2. A new place of type Storm is created and two transitions between this place and the place FairWeather. As soon as the storm warning has arrived, no starting runways can be opened. But it is still possible to take off for waiting airplanes using the already existing starting runways. The airport system itself can decide when the weather situation is that bad that no airplan shall depart. Then the transition forbidTakeoff is fired and since there is no longer a token on the place FairWeather, no more takeoffs are possible. As soon as the weather gets better, by firing the transition allowTakeoff airplanes may depart. Then the rule clearWarning deletes the storm warning and the normal airport operations may continue.

Altogether, in Fig. 1 a transformation sequence is depicted which first opens a starting runway, then receives a storm warning and afterwards, the starting runway is closed again.

3 Reconfigurable Place/Transition Nets

In this section, we formalize reconfigurable P/T systems based on our results in [HEM05]. As net formalism we use the algebraic notation of “Petri nets are Monoids” in [MM90], but extend this notation by a label function for places. So, a P/T net is given by $PN = (P,T, pre, post, label)$ with pre- and post domain functions $pre, post : T \to P^{\oplus}$ and a label function $label : P \to L$, where L is a fixed alphabet for places and $P^{\oplus}$ is the free commutative monoid over the set P of places, and a P/T system is given by (PN,M) with marking $M \in P^{\oplus}$.

In order to define rules and transformations of P/T systems we introduce P/T morphisms which preserve firing steps by Condition (1) and labels by Condition (2) below. Additionally, they require that the initial marking at corresponding places is increasing (Condition (3)). For strict morphisms, in addition injectivity and the preservation of markings is required (Condition (4)).

Rule openStartingRunway

df:DepField g:Gates :Storm w:FairWeather db:DepBoarding NAC ⟵ g:Gates df:DepField w:FairWeather db:DepBoarding L ⟹ :Tower takeoff df:DepField :Runway g:Gates depart w:FairWeather db:DepBoarding R

Rule openLandingRunway

af:ArrField g:Gates ab:ArrBoarding L ⟹ af:ArrField landing :Tower :Runway arrived g:Gates ab:ArrBoarding R

Rule closeStartingRunway

:Tower takeoff df:DepField :Runway g:Gates depart w:FairWeather db:DepBoarding L ⟹ g:Gates df:DepField w:FairWeather db:DepBoarding R

Rule closeLandingRunway

af:ArrField landing :Tower lt:Tower g:Gates :Runway arrived ab:ArrBoarding arrived lr:Runway L ⟹ af:ArrField g:Gates lt:Tower ab:ArrBoarding arrived lr:Runway R

Rule changeLandingToStartingRunway

af t df r g w :Storm ab lt NAC1 lr db | af t df r g w ab lt NAC2 lr db ⟵ af:ArrField landing t:Tower df:DepField g:Gates r:Runway arrived w:FairWeather db:DepBoarding arrived ab:ArrBoarding lt:Tower lr:Runway L ⟹ af:ArrField t:Tower takeoff df:DepField g:Gates r:Runway depart w:FairWeather db:DepBoarding arrived ab:ArrBoarding lt:Tower lr:Runway R

Rule changeStartingToLandingRunway

t df af r g ab NAC db w ⟵ af:ArrField t:Tower takeoff df:DepField g:Gates r:Runway depart w:FairWeather ab:ArrBoarding db:DepBoarding L ⟹ af:ArrField landing t:Tower df:DepField r:Runway g:Gates arrived w:FairWeather ab:ArrBoarding db:DepBoarding R

Rule incomingWarning

NAC=R ⟵ :FairWeather L ⟹ forbidTakeoff :FairWeather :Storm allowTakeoff R

Rule clearWarning

forbidTakeoff :FairWeather :Storm allowTakeoff L ⟹ :FairWeather R

Figure 2: The rules for changing the AirCS

Definition 1 (P/T Morphism) Given P/T systems $PS_i = (P_i, T_i, pre_i, post_i, label_i, M_i)$ for $i = 1,2$, a P/T morphism $f : PS_1 \to PS_2$ is given by $f = (f_P, f_T)$ with functions $f_P : P_1 \to P_2$ and $f_T : T_1 \to T_2$ satisfying

(1) $f_P^{\oplus} \circ pre_1 = pre_2 \circ f_T$ and $f_P^{\oplus} \circ post_1 = post_2 \circ f_T$,
(2) $f_P \circ label_1 = label_2 \circ f_P$ and
(3) $M_1(p) \leq M_2(f_P(p))$ for all $p \in P_1$.

Moreover, the P/T morphism f is called strict if **(4)** f_P and f_T are injective and $M_1(p) = M_2(f_P(p))$ for all $p \in P_1$.
The category defined by P/T systems and P/T morphisms is denoted by **PTS** where the composition of P/T morphisms is defined component-wise for places and transitions. The class of all strict P/T morphisms is denoted by $\mathcal{M}$.

Next we define the gluing condition which has to be satisfied in order to apply a rule at a given match. The characterization of specific points is a sufficient condition for the existence and uniqueness of the so-called pushout complement which is needed for the first step in a transformation.

Definition 2 (Gluing Condition) Given P/T systems $PS_i = (P_i, T_i, pre_i, post_i, label_i, M_i)$ for $i \in \{L, K, 1\}$, and let $PS_L \xrightarrow{m} PS_1$ be a P/T morphism and $PS_K \xrightarrow{l} PS_L$ a strict morphism, then the gluing points GP, the dangling points DP and the identification points IP of PS_L are defined by

$$
\begin{aligned}
GP = \; & l(P_K \cup T_K) \\
DP = \; & \{p \in P_L | \exists t \in (T_1 \setminus m_T(T_L)) : m_P(p) \in pre_1(t) \oplus post_1(t)\} \\
IP = \; & \{p \in P_L | \exists p' \in P_L : p \neq p' \wedge m_P(p) = m_P(p')\} \\
& \cup \{t \in T_L | \exists t' \in T_L : t \neq t' \wedge m_T(t) = m_T(t')\}
\end{aligned}
$$

The P/T morphisms m and l with l strict satisfy the gluing condition, if all dangling and identification points are gluing points, i.e $DP \cup IP \subseteq GP$, and m is strict on places to be deleted, i.e. $\forall p \in P_L \setminus l(P_K) : M_L(p) = M_1(m(p))$.

Next we present rule-based transformations of P/T systems following the double-pushout (DPO) approach of graph transformations in the sense of [Roz97, EEPT06].

Definition 3 (P/T System Rule) Given P/T systems $PS_i = (P_i, T_i, pre_i, post_i, label_i, M_i)$ for $i \in \{L, K, R, 1\}$, then a rule $rule = (PS_L \xleftarrow{l} PS_K \xrightarrow{r} PS_R)$ consists of P/T systems PS_L, PS_K, and PS_R, called the left-hand side, interface, and right-hand side of *rule*, respectively, and two strict P/T morphisms $PS_K \xrightarrow{l} PS_L$ and $PS_K \xrightarrow{r} PS_R$.
The rule *rule* is applicable at the match $PS_L \xrightarrow{m} PS_1$ if the gluing condition is satisfied for l and m. In this case, we obtain a P/T system PS_0 leading to a transformation step $PS_1 \overset{rule,m}{\Longrightarrow} PS_2$ consisting of the following pushout diagrams (1) and (2). The P/T morphism $n : PS_R \to PS_2$ is called comatch of the transformation step.

$$\begin{array}{ccccc} PS_L & \xleftarrow{l} & PS_K & \xrightarrow{r} & PS_R \\ m\downarrow & (\mathbf{1}) & \downarrow c & (\mathbf{2}) & \downarrow n \\ PS_1 & \xleftarrow[l^*]{} & PS_0 & \xrightarrow[r^*]{} & PS_2 \end{array}$$

Now we are able to define reconfigurable P/T systems, which allow the modification of the net structure using rules and transformations of P/T systems.

Definition 4 (Reconfigurable P/T Systems) Given a P/T system PS and a set of rules $RULES$, a reconfigurable P/T system is defined by $(PS, RULES)$.

In the example in Section 2 the reconfigurable P/T system consists of the P/T system in the top of Fig. 1 and the set of rules depicted in Fig. 2. Note, that the application of some of these rules is restricted by NACs and we will present the notion of reconfigurable P/T systems with NACs in Section 4.

4 Negative Application Conditions

In this section, we first state the main technical result that P/T systems are a weak adhesive HLR category with NACs. As a consequence we can define NACs for P/T system rules and transformations. Afterwards we summarize the main results available for reconfigurable P/T systems with NACs.

In addition to the class $\mathscr{M}$ in Section 3 we need two other classes of morphisms. The class $\mathscr{Q}$ denotes those morphisms that connect the NAC to the source net, which is the class of injective P/T system morphisms. Note that morphisms in this class do not need to be marking strict. The class $\mathscr{E}$ is a class of minimal jointly surjective morphism pairs, where minimal means that the markings in the codomain are as small as possible, i.e. for $e_1 : PS_1 \to PS_3$, $e_2 : PS_2 \to PS_3$ with $(e_1, e_2) \in \mathscr{E}$ we have that e_1, e_2 are jointly surjective and $M_3(p) = \max(\{M_1(p') \mid p' \in e_1^{-1}(p)\} \cup \{M_2(p') \| \ p' \in e_2^{-1}(p)\})$. This class is mainly used for constructions and proofs.

Definition 5 (Morphism classes in **PTS**) Given the category **PTS** of P/T systems and P/T morphisms, then the following morphism classes are defined:

$\mathscr{M}$: strict **PTS** morphisms (injective and marking strict **PTS** morphisms)
$\mathscr{Q}$: injective **PTS** morphisms (monomorphisms in the category **PTS**)
$\mathscr{E}$: minimal jointly surjective **PTS** morphisms

Theorem 1 (**PTS** is weak adhesive HLR category with NACs) *Given the weak adhesive HLR category* **PTS** *and the morphism classes $\mathscr{M}$, $\mathscr{Q}$ and $\mathscr{E}$ as defined above then we have*

1. *unique $\mathscr{E}$-$\mathscr{Q}$ pair factorization,*
2. *unique epi-$\mathscr{M}$ factorization,*
3. *$\mathscr{M}$-$\mathscr{Q}$ pushout-pullback decomposition property,*

4. *initial pushouts over $\mathcal{Q}$-morphisms,*
5. *$\mathcal{Q}$ is closed under pushouts and pullbacks along $\mathcal{M}$-morphisms,*
6. *induced pullback-pushout property for $\mathcal{M}$ and $\mathcal{Q}$ and*
7. *$\mathcal{Q}$ is closed under composition and decomposition.*

Altogether, $(\mathbf{PTS}, \mathcal{M}, \mathcal{E}, \mathcal{Q})$ *is a weak adhesive HLR category with NACs .*

Proof. In [EEH+07], it has been shown that $(\mathbf{PTS}, \mathcal{M})$ is a weak adhesive HLR category.

According to [LEOP08], for a weak adhesive HLR category with NACs Items 1–7 have to be proven additionally. Note, that in [LEOP08] an additional morphism class $\mathcal{M}'$ is used and some more properties have to be checked. In the case of **PTS**, $\mathcal{Q}$ and $\mathcal{M}'$ coincide, which reduces the effort for the proof. Here we only explain the properties and give proof ideas, the detailed proof can be found in [Rei08].

1. unique $\mathcal{E}$-$\mathcal{Q}$ pair factorization: For a morphism pair $f_1 : L_1 \to P$, $f_2 : L_2 \to P$, an $\mathcal{E}$-$\mathcal{Q}$ pair factorization is a pair $e_1 : L_1 \to K$, $e_2 : L_2 \to K$ with $(e_1, e_2) \in \mathcal{E}$ and $m : K \to P \in \mathcal{Q}$ such that $m \circ e_1 = f_1$ and $m \circ e_2 = f_2$. Uniqueness means that two $\mathcal{E}$-$\mathcal{Q}$ pair factorizations of f_1 and f_2 are isomorphic.

   For the construction of this $\mathcal{E}$-$\mathcal{Q}$ pair factorization in **PTS**, we first construct the coproduct L of L_1 and L_2 with coproduct inclusions i_1 and i_2, obtain a morphism $f : L \to P$ and construct an epi-mono factorization $(e : L \to K, m : K \to P)$ of f in P/T nets. Defining $M_K(p) = \max(\{M_L(p') \mid p' \in e^{-1}(p)\})$ and $e_1 = e \circ i_1, e_2 = e \circ i_2$ leads to a unique $\mathcal{E}$-$\mathcal{Q}$ pair factorization. This property is needed mainly for the embedding and extension of transformation pairs.

2. unique epi-$\mathcal{M}$ factorization: For a morphism $f : L \to P$, an epi-$\mathcal{M}$ factorization is an epimorphism $e : L \to K$ and a morphism $m : K \to P \in \mathcal{M}$ such that $m \circ e = f$. Uniqueness means that two epi-$\mathcal{M}$ factorizations of f are isomorphic.

   For the construction, we use the epi-mono factorization of f in P/T nets, and obtain the marking from the marking of P leading to a strict morphism $m \in \mathcal{M}$. Uniqueness follows directly from uniqueness of the epi-mono factorization in P/T nets and strictness of $\mathcal{M}$. This property is needed for the translation of NACs over a morphism.

3. $\mathcal{M}$-$\mathcal{Q}$ pushout-pullback decomposition property: Given the following commutative diagram with $l \in \mathcal{M}$ and $w \in \mathcal{Q}$, where $(1+2)$ is a pushout and (2) is a pullback, then (1) and (2) are both pushouts.

   In P/T nets, this property holds for injective l and w, thus we obtain pushouts (1) and (2) in P/T nets. It remains to show the additional pushout properties in **PTS**, which can be verified. This property is needed for the embedding and extension of transformation pairs and for the equivalence of left and right NACs.

4. initial pushouts over $\mathcal{Q}$-morphisms: An initial pushout over a morphism $f \in \mathcal{Q}$ represents the boundary and context of f. The construction is similar to that in P/T nets, but also

includes all places where f is not marking-strict. This property is needed for the extension of transformations.

5. $\mathscr{Q}$ is closed under pushouts and pullbacks along $\mathscr{M}$-morphisms: In a pushout or pullback square along $\mathscr{M}$, if the other given morphism is in $\mathscr{Q}$ then the opposite morphism is also a $\mathscr{Q}$-morphism.

   This property follows directly from the corresponding properties in P/T nets and is used for the translation of NACs as well as for the embedding and extension of transformations.

6. induced pullback-pushout property for $\mathscr{M}$ and $\mathscr{Q}$: Given the following pushout (PO) and the pullback (PB) then the induced morphism $x : PS_3 \to PS_4$ is a $\mathscr{Q}$-morphism.

   This property can be shown using the fact that f' and g' are jointly surjective, which can be shown for the pushout construction. It is needed for the translation of NACs over a morphism.

7. $\mathscr{Q}$ is closed under composition and decomposition: This is a standard result for monomorphisms from category theory. This property is needed for the translation of NACs and for the completeness of critical pairs.

$$
\begin{array}{ccccc}
A & \xrightarrow{k} & B & \xrightarrow{r} & E \\
{\scriptstyle l}\downarrow & (1) & \downarrow{\scriptstyle s} & (2) & \downarrow{\scriptstyle v} \\
C & \xrightarrow{u} & D & \xrightarrow{w} & F
\end{array}
\qquad
\begin{array}{ccc}
PS_0 & \xrightarrow{f} & PS_1 \\
{\scriptstyle g}\downarrow & (PB) & \downarrow{\scriptstyle h} \\
PS_2 & \xrightarrow{h'} & PS_4
\end{array}
\qquad
\begin{array}{ccc}
PS_0 & \xrightarrow{f} & PS_1 \\
{\scriptstyle g}\downarrow & (PO) & \downarrow{\scriptstyle g'} \\
PS_2 & \xrightarrow{f'} & PS_3
\end{array}
$$

□

Now, we can state negative application conditions for P/T system transformation in the following sense:

Definition 6 ((Left) Negative Application Condition) A (left) negative application condition of a rule $rule = (PS_L \stackrel{l}{\leftarrow} PS_K \stackrel{r}{\rightarrow} PS_R)$ in the weak adhesive HLR category with NACs $(C, \mathscr{M}, \mathscr{E}, \mathscr{Q})$ is of the form $NAC(n)$, where $n : PS_L \to PS_N$ is a P/T morphism.
A morphism $m : PS_L \to PS_1$ satisfies $NAC(n)$, written $m \models NAC(n)$, if there does not exist a morphism $q : PS_N \to PS_1 \in \mathscr{Q}$ with $q \circ n = m$.

Definition 7 (Rule with NACs) A rule in a weak adhesive HLR category with NACs $(C, \mathscr{M}, \mathscr{E}, \mathscr{Q})$ with a set of negative application conditions $NACS$ is called rule with NACs.

Remark 1 *Analogously to left NACs we can define right NACs on the right-hand side of a rule which have to be satisfied by the comatch of the transformation. In this paper, we only consider rules with an empty set of right NACs. This is without loss of generality since each right NAC can be translated into an equivalent left NAC as shown in [EEPT06, LEOP08].*

Definition 8 (Applicability of a Rule with NACs) Given a rule $rule = (PS_L \stackrel{l}{\leftarrow} PS_K \stackrel{r}{\rightarrow} PS_R)$ with a set of negative application conditions $NACS$ and a match $m : PS_L \to PS_1$ such that $rule$ without NACs is applicable at m, then the rule $rule$ with NACs is applicable if and only if m satisfies all NACs of the set $NACS$.

For these new rules and their restricted application we obtain the same results as known for net transformations in general. These results have been shown for NACs at the level of weak adhesive HLR categories in [LEOP08, LEO06, Lam07]. Their instantiation to P/T systems requires Theorem 1 above.

Results (For Reconfigurable P/T Systems with NACs)

1. **Local Church-Rosser and Parallelism:** The local Church-Rosser property for transformations with NACs states that for two rules with two matches, the application of the rules at the matches to the same P/T system in any order (sequentially independence) yields the same result if and only if the transformations are parallel independent. For parallel independence of two transformations with NACs it has to be checked in particular if one transformation does not delete anything the other transformation needs, and, in addition, that one transformation does not produce any structure that is forbidden by the other one. For such independent transformations a parallel transformation with NACs can be built obtaining the same result in one transformation step.

   In our airport example it makes e.g. no difference if a starting or a landing runway is opened first. After opening a starting runway it is still possible to open a landing runway, since nothing is deleted and there is no NAC on the rule openLandingRunway. After opening a landing runway a starting one can be opened: Nothing is deleted and the NAC of the rule openStartingRunway that forbids the existence of a place of type Storm remains satisfied. Moreover now both runways can be constructed in parallel. This construction leads to the same result consisting of an airport with one more starting and one more landing runway.

2. **Conflicts and Critical Pairs:** If two transformations are not parallel independent as described in Item 1 then they are in conflict. This means in particular that one of the transformations deletes some structure which is needed by the other one, or it produces a structure which is forbidden by the other one. A critical pair describes such a conflict between two transformations in a minimal context. Critical pairs are proven to be complete [LEO06], i.e. each conflict occurring in the system between two transformations is represented by a critical pair expressing the same conflict in a minimal context. The morphism class $\mathscr{E}$ is required to express this minimal context.

   In our example a transformation adding a warning (i.e. applying the rule incomingWarning) is in conflict with a transformation which opens a starting runway (i.e. applying rule openStartingRunway). This conflict is caused by the NAC of the rule openStartingRunway as it cannot be applied if a place of type Storm is present. This conflict occurs regardless of the number of runways already present in the airport and is expressed in a minimal context by the critical pair shown in Figure 3.

3. **Concurrency:** As explained in Item 1 sequentially independent transformations can be put into one parallel transformation step having the same effect. But if sequential dependencies occur between direct transformations in a transformation sequence the Parallelism Theorem cannot be applied. In this case a so-called concurrent rule with NACs can be constructed establishing the same effect in one transformation step with NACs as the whole

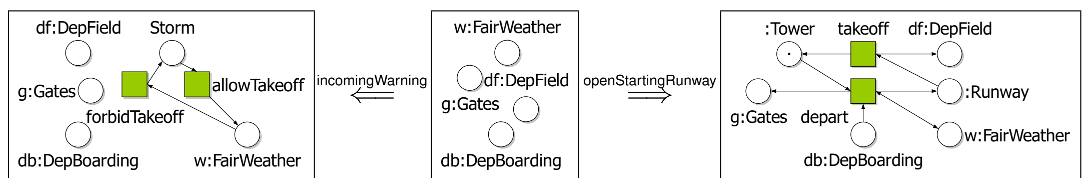


Figure 3: The critical pair for the rules incomingWarning and openStartingRunway

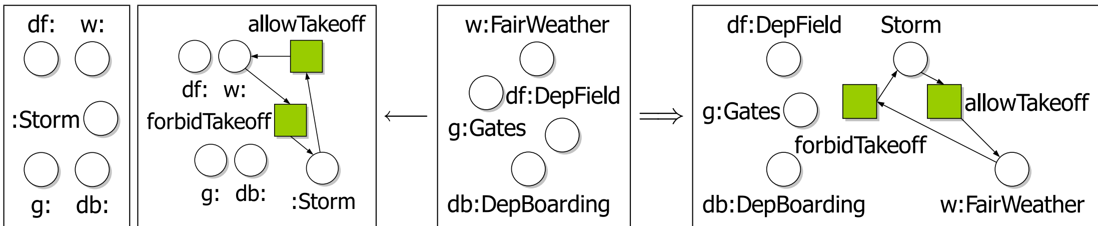


Figure 4: The concurrent rule with NACs

transformation sequence. The Concurrency Theorem states that a concurrent rule with NACs equivalent to a sequence of rules with NACs is applicable with the same result if and only if the rule sequence with NACs is applicable. The construction of the concurrent rule is analogous to the case without NACs. Additionally, all NACs occurring in the rule sequence need to be translated into equivalent NACs for the concurrent rule. The construction of such a concurrent rule with NACs is explained in [LEOP08]. A concurrent rule summarizes in one rule which parts of the net should be present, preserved, deleted and produced when applying the corresponding rule sequence to this net. Moreover we have a summarized set of NACs on the concurrent rule expressing which net parts are forbidden when applying the corresponding rule sequence with NACs to the net.

Consider for our example the transformation sequence in Fig. 1. First, a starting runway is opened followed by an incoming warning after which the starting runway is closed again. This transformation sequence can be summarized to one transformation step via a new concurrent rule with concurrent NACs as depicted in Fig. 4. Note that this concurrent rule now holds two single NACs, one originating from the first rule openStartingRunway and the other one originating from the second rule incomingWarning in the sequence. Note, moreover, that this rule adds no new behavior to our system, but merely adds the possibility of performing these three transformations in one step with the same result.

4. **Embedding and Extension:** Consider a transformation $t : N_0 \overset{*}{\Rightarrow} N_n$ and a morphism $k_0 : N_0 \rightarrow N_0'$, then the transformation t can be embedded into the larger context N_0' if and only if the extension morphism $k_0 : N_0 \rightarrow N_0'$ satisfies two consistency conditions. First, it has to be boundary consistent. This means intuitively that the extension morphism cannot embed places which are deleted by the transformation t into places connected with new transitions in the bigger P/T system N_0'. Otherwise, dangling edges will occur during the embedding. Moreover, the extension morphism k_0 should satisfy NAC-consistency

[LEO06]. Intuitively, the transformation t can be summarized into one step $t_c : N_0 \Rightarrow N_n$ using the new concurrent rule with concurrent NACs. Now the extension morphism k_0 may not map to a larger P/T system N_0' with added structures which are forbidden by this concurrent NAC. Note that boundary consistency and NAC-consistency are not only sufficient, but also necessary conditions for the construction of extended transformations with NACs. So, whenever it is possible to repeat a transformation t into a bigger context N_0' the extension morphism was boundary and NAC-consistent.

In our example, we can embed the transformation described in the last item into the P/T system AirCS_1 from Fig. 1 which is not the initial one, but already contains two starting and one landing runway. On the contrary, this transformation cannot be embedded into the airport AirCS_2 from Fig. 1 which already contains a storm warning. This is because the extension morphism k_0 adds a place of type Storm which is forbidden by the concurrent NACs as depicted in Fig. 4.

5. **Confluence:** Critical pairs as described in Item 2 are not only complete, their confluence behavior has an impact on the confluence behavior of the whole system. Intuitively this means that if a conflict can be resolved in a certain way in its minimal context, the same conflict is resolvable as well if it occurs in a larger context. A solution of a conflict in a minimal context (or critical pair) $P_1 \leftarrow K \rightarrow P_2$ is a pair of transformation sequences t_1 and t_2 such that t_1 transforms P_1 into a certain net X and t_2 transforms it into the same net X. Thus the same system state can be reached again if a conflict occurred. The solution of the critical pair needs to be strict. Intuitively speaking, this means that t_1 and t_2 preserve everything which is preserved in common by the critical pair itself. Moreover, whenever it is possible to embed the critical pair into some larger context the extension morphism should be NAC-consistent with respect to the critical pair solution (NAC-confluence [Lam07]). This means in particular that all NACs occurring in the solution of the critical pair are still satisfied by the embedding into the larger context. Under these conditions, this conflict can be resolved in the same way also in a larger context. Otherwise, the solution of the critical pair is no solution for the larger context, and no prediction for confluence can be infered. In particular, if all critical pairs of the reconfiguration system are strictly NAC-confluent then the system is locally confluent.

   Consider in our example the critical pair depicted in Fig. 3. The solution of the critical pair is depicted in Fig. 5 for the right-hand side of the critical pair. Whenever a starting runway has been opened, a warning can come in and the starting runway can be closed again. The result is a P/T system containing no runway, but a storm warning. This is already our solution because this P/T system is identical to the the first P/T system in the critical pair. This solution is moreover strictly confluent because the places g, df, fw and db are preserved by both the critical pair and our solution. Moreover, the solution is NAC-confluent because the satisfaction of the concurrent NACs as depicted in Fig. 4 is implied already by the satisfaction of the first NAC of the critical pair. This means in effect that if the critical pair can be embedded and thus the NACs of the critical pair are satisfied by this embedding then they will also be satisfied when embedding the solution into the same bigger context. Therefore it is possible to resolve the conflict between an incoming

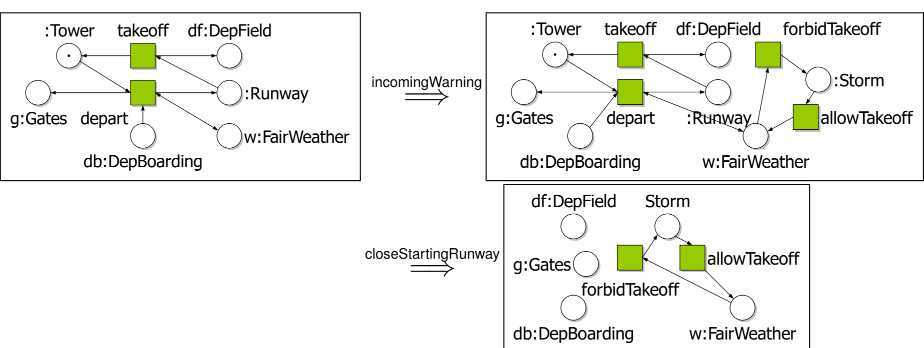


Figure 5: The solution for the critical pair

warning and opening a starting runway by having an incoming warning and closing the runway in each bigger airport as well.

5 Conclusion

We conclude with a short discussion of related and future work:
Related Work Reconfigurable nets have been defined based on net transformations that aim directly at changing the net in arbitrary ways. This approach can be restricted to transformations that preserve specific properties as safety or liveness (see [PU03]). Dynamic nets [BS01] are based on the join calculus and allow the dynamic adaption of the network configuration and are considered to be a special case of zero-safe nets [BMM04]. In a series of papers [LO04, LO06a, LO06b] rewriting of Petri nets in terms of graph grammars is used for the reconfiguration of nets as well. There marked-controlled reconfigurable nets (MCRN) are extended by some control technique that allows changes of the net for specific markings. The enabling of a rule is not only dependent on the net topology, but also dependent on the marking of specific control places. *MCReNet* [LO06a] is the corresponding tool for the modeling and verification of MCRNs.
Future Work One ongoing research tasks is the extension of this paper's results to algebraic high-level nets, a Petri net variant with additional data types in terms of algebraic specifications. Therefore, the same conditions have to be proved considering additionally the specification and algebra morphisms. Another one are algebraic higher order (AHO) nets that can be used as a controlling mechanism for reconfigurable Petri nets. There P/T systems as well as rules are the tokens of the underlying AHO net. This specification technique has been targeted at modeling workflows of mobile ad-hoc networks. Up to now we have not made use of the new feature of NACs in AHO nets. To do so we have to integrate the NACs into the algebra underlying the AHO net.

References

[BMM04] R. Bruni, H. Melgratti, U. Montanari. Extending the Zero-Safe Approach to Coloured, Reconfigurable and Dynamic Nets. In *Lectures on Concurrency and Petri*

Nets. LNCS 3098, pp. 291–327. Springer, 2004.

[BS01] M. Buscemi, V. Sassone. High-Level Petri Nets as Type Theories in the Join Calculus. In *Proceedings of FoSSaCS '01*. Springer, 2001.

[EEH+07] H. Ehrig, C. Ermel, K. Hoffmann, J. Padberg, U. Prange. Concurrency in Reconfigurable Place/Transition Systems: Independence of Net Transformations as well as Net Transformations and Token Firing. Technical report 2007-02, TU Berlin, 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer, 2006.

[EP04] H. Ehrig, J. Padberg. Graph Grammars and Petri Net Transformations. In *Lectures on Concurrency and Petri Nets*. LNCS 3098, pp. 496–536. Springer, 2004.

[HEM05] K. Hoffmann, H. Ehrig, T. Mossakowski. High-Level Nets with Nets and Rules as Tokens. In *Proceedings of ICATPN'05*. LNCS 3536, pp. 268–288. Springer, 2005.

[Lam07] L. Lambers. Adhesive High-Level Replacement Systems with Negative Application Conditions. Technical report, TU Berlin, 2007.

[LEO06] L. Lambers, H. Ehrig, F. Orejas. Conflict Detection for Graph Transformation with Negative Application Conditions. In *Proceedings of ICGT'06*. LNCS 4178, pp. 61–76. Springer, 2006.

[LEOP08] L. Lambers, H. Ehrig, F. Orejas, U. Prange. Parallelism and Concurrency in Adhesive High-Level Replacement Systems with Negative Application Conditions. *ENTCS*, 2008. to appear.

[LO04] M. Llorens, J. Oliver. Structural and Dynamic Changes in Concurrent Systems: Reconfigurable Petri Nets. *IEEE Transactions on Computers* 53(9):1147–1158, 2004.

[LO06a] M. Llorens, J. Oliver. A Basic Tool for the Modeling of Marked-Controlled Reconfigurable Petri Nets. *ECEASST* 2:13, 2006.

[LO06b] M. Llorens, J. Oliver. Marked-Controlled Reconfigurable Workflow Nets. In *SYNASC*. Pp. 407–413. IEEE Computer Society, 2006.

[MM90] J. Meseguer, U. Montanari. Petri Nets are Monoids. *Information and Computation* 88(2):105–155, 1990.

[PU03] J. Padberg, M. Urbášek. Rule-Based Refinement of Petri Nets: A Survey. In *Petri Net Technology for Communication-Based Systems*. LNCS 2472, pp. 161–196. Springer, 2003.

[Rei08] A. Rein. Reconfigurable Petri Systems with Negative Application Conditions. Technical report 2008/01, TU Berlin, 2008. Diploma Thesis, to appear.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 1: Foundations*. World Scientific, 1997.

Independence Analysis of Firing and Rule-based Net Transformations in Reconfigurable Object Nets

E. Biermann[1] **and T. Modica**[2]

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
[1]enrico@cs.tu-berlin.de, [2]modica@cs.tu-berlin.de

Abstract: The main idea behind *Reconfigurable Object Nets (RONs)* is to support the visual specification of controlled rule-based net transformations of place/transition nets (P/T nets). RONs are high-level nets with two types of tokens: object nets (place/transition nets) and net transformation rules (a dedicated type of graph transformation rules). Firing of high-level transitions may involve firing of object net transitions, transporting object net tokens through the high-level net, and applying net transformation rules to object nets, e.g. to model net reconfigurations. A visual editor and simulator for RONs has been developed as a plug-in for ECLIPSE using the ECLIPSE Modeling Framework (EMF) and Graphical Editor Framework (GEF) plug-ins.

The problem in this context is to analyze under which conditions net transformations and token firing can be executed in arbitrary order. This problem has been solved formally in a previous paper. In this contribution we present an extension of our RON tool which implements the analysis of conflicts between parallel enabled transitions, between parallel applicable net transformation rules (Church-Rosser property), and between transition firing and net transformation steps. The conflict analysis is applied to a RON simulating a distributed producer-consumer system.

Keywords: Petri nets, net transformation, graph transformation, visual editor, reconfigurable object nets, conflict analysis, independence analysis, Eclipse, GEF

1 Introduction

Modelling the adaption of a system to a changing environment has become a significant topic in recent years. Application areas cover e.g. computer supported cooperative work, multi agent systems, dynamic process mining or mobile ad-hoc networks (MANETs). Especially in the context of our project *Formal modeling and analysis of flexible processes in mobile ad-hoc networks* [PEH07, For06] we aim to develop a formal technique which on the one hand enables the modeling of flexible processes in MANETs and on the other hand supports changes of the network topology and the transformation of processes. This can be achieved by an appropriate integration of graph transformation, nets and processes in high-level net classes.

The main idea behind *Reconfigurable Object Nets (RONs)* is the integration of transition firing and rule-based net structure transformation of place/transition nets (P/T nets) during system simulation. This approach increases the expressiveness of Petri nets and allows a formal description

of dynamic system changes.

RONs are high-level nets with two types of tokens: object nets (which are P/T nets) and net transformation rules (a dedicated type of graph transformation rules). Thus, on the one hand, RONs follow the paradigm "nets as tokens", introduced by Valk in [Val98], and, on the other hand, extend this paradigm to "nets and rules as tokens" in order to allow for modelling net structure changes (reconfigurations) of object nets. The high-level net constitutes the control frame for object net behavior and rule-based reconfiguration of object nets. Firing of high-level transitions may involve firing of object net transitions, transporting object net tokens through the high-level net, and applying net transformation rules to object nets. Net transformation rules model net reconfigurations such as merging or splitting of object nets, and net refinements.

The formal basis for RONs is given in [HME05], where high-level nets with nets and rules as tokens are defined algebraically, based on algebraic high-level nets [PER95]. The basic idea behind net transformation is the stepwise development of P/T systems by given rules consisting of a left-hand side LHS related to a right-hand side RHS. Think of these rules as replacement systems where the LHS is replaced by the RHS preserving a context. Similar to the concept of graph transformations [EEPT06], each application of a rule $r = (LHS \to RHS)$ to a source net N_1 leads to a net reconfiguration step $N_1 \overset{r}{\Longrightarrow} N_2$, where in the source net N_1 the subnet corresponding to the LHS is replaced by the subnet corresponding to the RHS, yielding the target net N_2. Rule-based Petri net transformations have been treated in depth in e.g. [EP04, PU03].

The visual editor and simulator for RONs has been realized as a plug-in for ECLIPSE using the ECLIPSE Modeling Framework (EMF) [EMF06] and Graphical Editor Framework (GEF) [GEF06] plug-ins. In RONs, the algebraic operations defined for rule applications and transition firing are modeled as special RON-transition types which have a fixed firing semantics. It turned out that four RON-transition types for composition, decomposition, firing and rule-based reconfiguration are sufficient to model various interesting examples (Case studies and downloads of the RON tool are available on our RON homepage [RON07]).

Recently, work has been done to formalize independence conditions for reconfigurable P/T systems (i.e. P/T systems together with reconfiguration rules) [EHP+07, EEH+07]. While independence conditions for two firing steps of P/T-systems are well-known (transitions in conflict), independence of net reconfiguration steps is closely related to local Church-Rosser properties for graph transformations that are valid in the case of parallel and sequential independence of rule-based transformations. In [EEPT06], conditions for two transformation steps are given in the framework of high-level replacement systems with application to net transformations, so that these transformation steps applied to the same P/T-system can be executed in arbitrary order, leading to the same result. In [EHP+07] we state under which conditions a net transformation step and a firing step are independent of each other. The subject of this paper is the implementation of the different formal notions of conflict and independence analysis in the RON tool.

The paper is structured as follows: A running example (a distributed producer-consumer system) is introduced in Section 2. Section 3 introduces the RON tool, and Section 4 describes the extension of our tool with conflict and independence analysis based on the theoretical results from [EHP+07, EEH+07]. These new features of the tool are used to analyze the producer-consumer system. Section 5 concludes the paper with an outlook on future work.

2 Example: A Producer-Consumer System

In our example, RONs are applied to model a distributed system of producers and consumers where several producers and consumers may interact with each other. In the initial state of the sample RON in Fig. 1 potential producers and consumers are distributed on different Net places as independent object nets without interaction. Producer nets may fire, e.g. they can produce

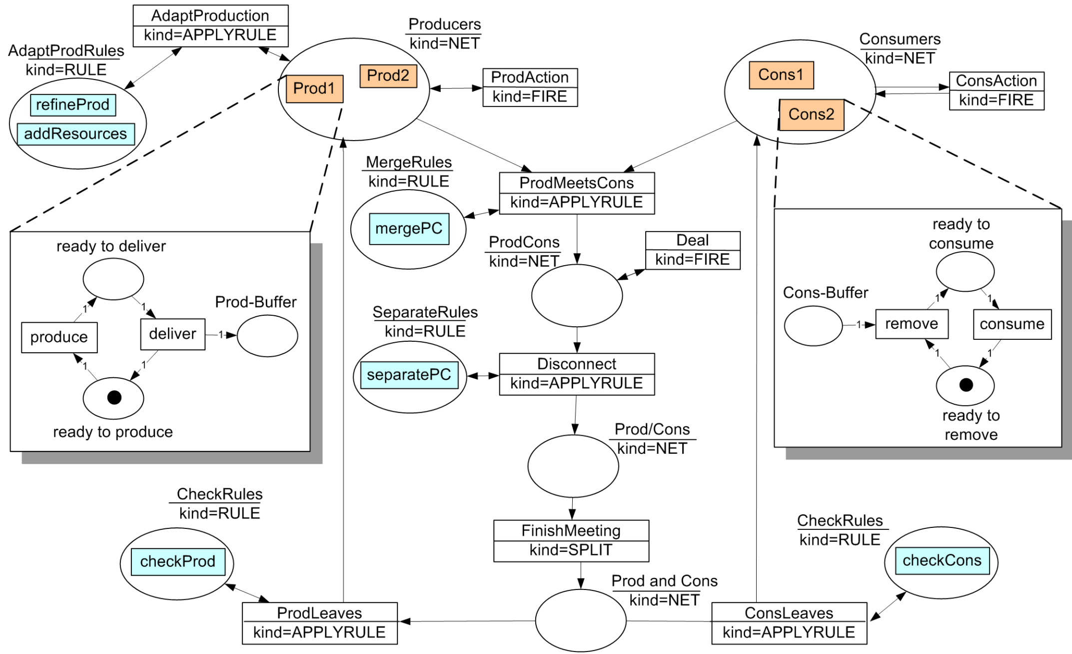


Figure 1: Distributed Producers/Consumers modelled as RON

items and place them on the buffer place. Firing in object nets is triggered by firing a RON high-level (HL) transition of type FIRE, which takes one object net with marking M from the Net place in its pre-domain and puts the same object net, now marked by one of the possible successor markings of M, into all of its post-domain places. Producer nets can also be refined by firing the RON HL-transition *AdaptProduction* of type APPLYRULE. A transition of this type takes an object net from each of the pre-domain Net places, a rule from the pre-domain Rule place, applies this rule to the disjoint union of all the taken object nets and puts the resulting net to all post-domain Net places. Note that a transition is preserved by a rule only if its pre- and postdomains are preserved as well.

Rule *refineProd*, depicted in Fig. 2, refines the transition produce by two transitions prepare production and a new transition produce. Transition produce is deleted by rule *refineProd* and generated again with a different predomain place. Rule *refineProd* can be applied only once to the same object net since the NAC forbids its application if there is already a place called production prepared in the net. Rule *addResources* (also Fig. 2) replaces the produce transition by two alternative production procedures, each using a different resource.

For producer-consumer interaction, a producer net can be merged with a consumer net by

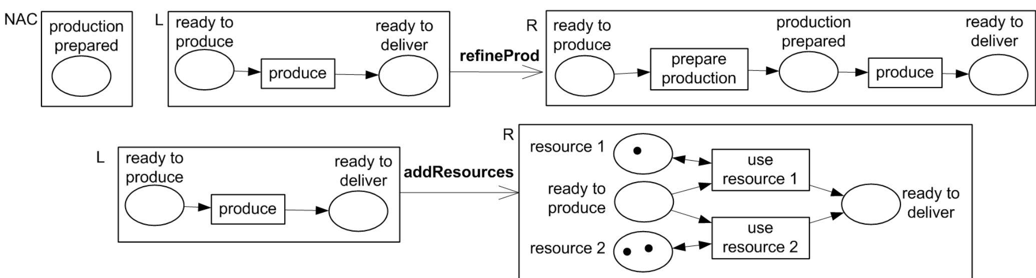


Figure 2: Rules for adapting a producer net

firing the RON HL-transition *ProdMeetsCons* of type APPLYRULE. Rule *merge-PC*, depicted in Fig. 3, glues a producer object net and a consumer object net by inserting a *connect* transition between both buffers. A so-called negative application condition (NAC) forbids the application of the rule if there already exists a *connect* transition. Note that the transition *ProdMeetsCons* controls which producer interacts with which consumer.

HL-transitions of type FIRE trigger the firing of object net transitions. Note that for firing object net transitions, no net transformation rule is applied. The firing semantics of object nets is the usual P/T net behavior. By firing the FIRE HL-transition *Deal*, in the glued net the consumer now can consume items produced by the producer as long as there are tokens on the place *Prod-Buffer*. Moreover, the producer may also produce more items and put them to the buffer. After the deal has been finished, the nets are separated again by firing the APPLYRULE HL-transition *Disconnect*. This applies rule *separate-PC* in Fig. 3 to the glued net which deletes the *connect* transition from the net.

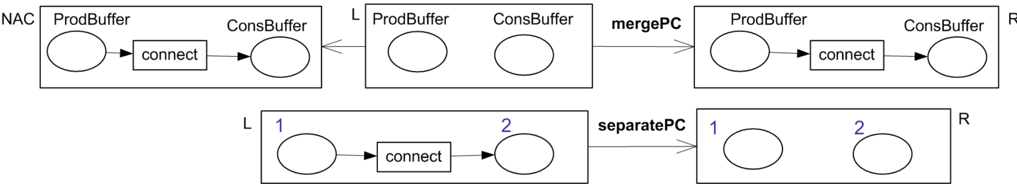


Figure 3: Rules for gluing and for separating a producer and a consumer net

Note that the resulting net, which is put on place *Prod/Cons*, is still one single object net which consists of two unconnected components. In order to split these components into two object nets, a HL-transition of type SPLIT has to be fired. Firing RON HL-transition FinishMeeting results in two separate object nets on place *Prod and Cons*. In the last step, we put the now separated producer and consumer nets back to their initial places. To prevent them to return to “wrong” initial places we again use APPLYRULE HL-transitions. These HL-transitions apply the rules *checkProd* and *checkCons*, respectively, which do not change the object nets they are applied to but simply check them for the occurrence of a *producer* or *consumer* place, respectively. Apart from HL-transitions of type FIRE, APPLYRULE and SPLIT, RONs provide a fourth HL-transition type, called STANDARD (not used in the producer-consumer example). STANDARD HL-transitions simply remove a net token from each pre-domain place and add the disjoint union of all removed

object nets to each of the post-domain Net places.

3 The RON Environment: Editor and Simulator

The RON environment [BEHM07] is divided into four main components, i.e. the RON tree view based on an EMF model for RONs, and the visual editors for object nets, for transformation rules and for high-level nets with the four HL-transition types FIRE, APPLYRULE, SPLIT and STANDARD. The visual net editors also support the simulation of an edited object net or high-level net. Fig. 4 shows a screenshot of the RON environment showing all views and editors.

RON Tree View. View 1 in Fig. 4 shows the main editor component, a tree view for the complete RON model from which the graphical views can be opened by double-click.

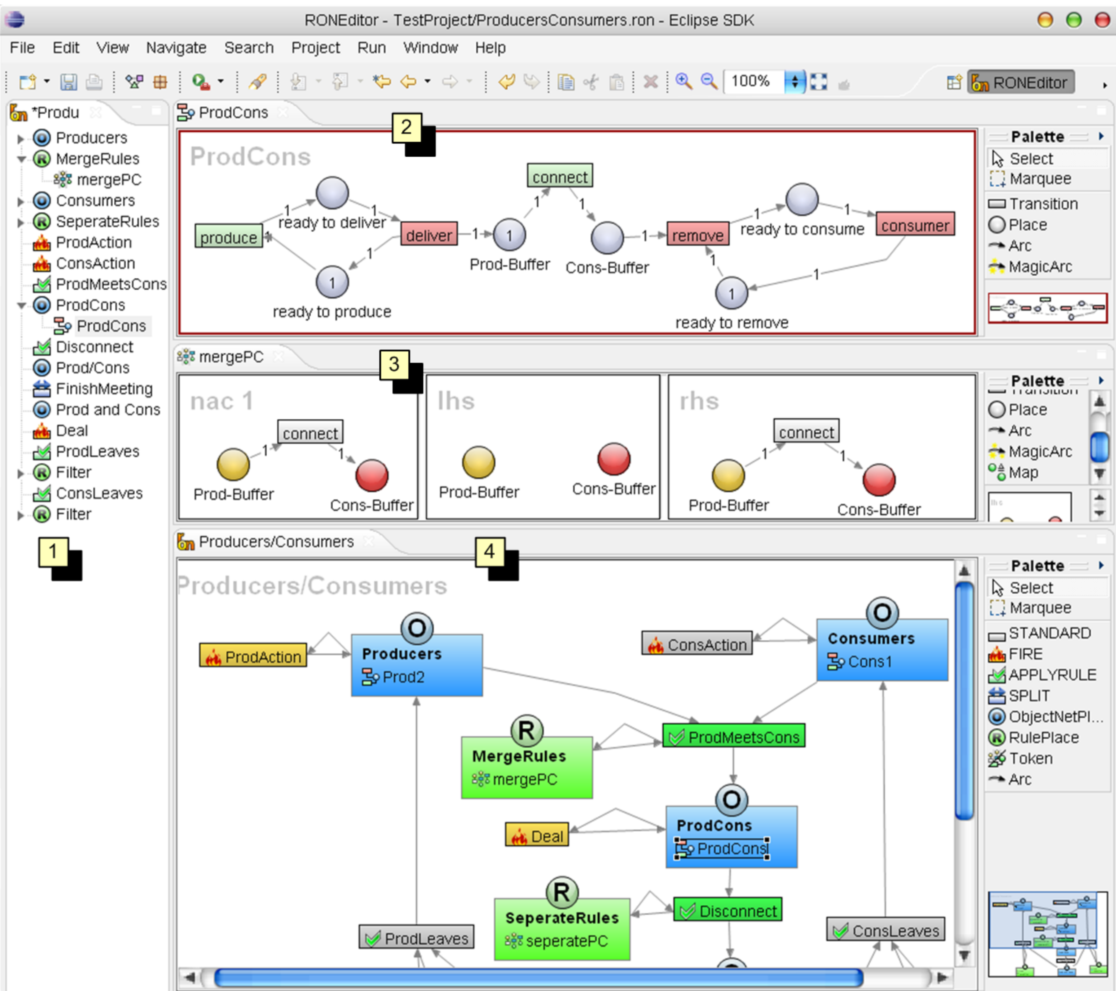


Figure 4: The RON Environment for Editing and Simulating Reconfigurable Object Nets

Object Net Editor. The kernel component is a graphical editor for object nets, i.e. the net tokens on the RON's NET-typed places. This component is actually a place/transition net editor, allowing the simulation of firing transitions. An object editor panel is shown in Fig. 4, View 2, holding the object net *ProdCons*, which models producer-consumer interaction.

Transformation Rule Editor. The editor for transformation rules mainly consists of three editor panels, one for the left-hand side (LHS), one for the right-hand side (RHS) and one for a negative application condition(NAC) (see view 3 in Fig. 4). Each editor panel is basically an object net editor itself, but with the additional possibility to relate the object nets by defining mappings on places and transitions. Mappings are realized by the *mapping tool* of the rule editor that allows the matching of LHS objects to RHS objects to define which objects are preserved by the rule, or to NAC objects to define which objects are connected to additional forbidden objects. In the editor, mappings are shown by object colouring. In order to ensure that the mapping specified by the mapping tool is also a valid Petri net morphism, it is checked for each mapped transition that all places in its pre- (post-)domain in the LHS are mapped to the corresponding places in the pre- (post-)domain of in the RHS. Another restriction is that all rules are injective, so different LHS objects must be mapped to different RHS objects. Note that the object net *ProdCons* in Fig. 4, View 2, is the result of applying rule *mergePC* to two object nets *Prod1* and *Cons2* from the places *Producers* and *Consumers*, respectively. (For the situation before the rule is applied, see Fig. 1).

High-Level Net Editor. A high-level net controls object net behaviour and rule applications to object nets. Such a high-level net is drawn in the high-level net editor panel, shown in Fig. 4, View 4. Here, NET places carrying object net tokens are blue containers marked by an "O" for *Object Nets*. RULE places carrying transformation rules are green containers marked by an "R" for *Rules*. Each transition type has a special graphical icon as visualization: for FIRE, for APPLYRULE, for SPLIT, and for STANDARD. Enabled HL-transitions are coloured, disabled ones are gray.

Simulation of RONs. A RON HL-transition is fired when double-clicked. The simulation of firing HL-transitions of kinds STANDARD, FIRE, and SPLIT has been implemented directly in the editor. In order to simulate firing of APPLYRULE HL-transitions, internally the RON editor was extended by a converter to AGG, an engine to perform and analyze algebraic graph transformations [AGG]. If the user gives the command to fire an APPLYRULE HL-transition he has to select the rule and the object net token(s) in the pre-domain the rule should be applied to. This is realized in the user interface shown in Fig. 4, View 4, by ordering the tokens in the corresponding NET and RULE containers in a way that the uppermost tokens are the ones considered by the rule application. Furthermore, the user is asked for a match defining the occurrence of the rule's left-hand side in the selected object net. Optionally, AGG can find or complete partial matches and propose them to the user in the RON editor. With the selected rule, match, and object net AGG computes the result of the transformation which is put on the post-domain places according to the firing semantics explained in Section 2.

A RON HL-transition of type FIRE triggers the firing of an object net transition for an object net

in the FIRE transition's pre-domain. We decided to implement object net transition firing directly in Java instead of modeling firing steps by graph transformation rules and using AGG to compute the successor markings. This design decision is based on the complexity of encoding P/T net behavior in graph transformation systems: there are two possibilities: 1) we could translate each single object net transition into a (model-dependent) graph rule [Erm06]; 2) we could envisage a more general encoding resulting in more than one rule for a single firing step due to the arbitrary number of input and output places for each transition. In case 1) we have the problem that our net transformation rules may delete / add / replace transitions. So, after each rule application we might be forced to adapt the set of model-dependent graph rules representing our object net. In case 2), we cannot make use of AGG's analysis features to find transitions in conflict, since the analysis of graph transformation steps is based on finding critical pairs of rules and needs one firing step to be modeled by one rule only. Thus, it proved to be more natural and straightforward to implement both the object net firing behavior and the check for conflicting object net transitions directly in Java.

4 Extending the RON Environment by Independence Analysis

4.1 Independence in Reconfigurable P/T Systems

In this section we give a brief overview on the different analysis cases and demonstrate them by conflict examples in our producer-consumer system (Fig. 1).

As object nets in RONs can evolve in two different ways (by firing object net transitions and by applying net transformation rules), the notions of conflict and concurrency become quite complex. We illustrate the situation in Fig.5, where we have in the center an object net PN_0, with marking M_0 and two transitions that are both enabled leading to firing steps $(PN_0,M_0) \xrightarrow{t_1} (PN_0,M_0')$ and $(PN_0,M_0) \xrightarrow{t_2} (PN_0,M_0'')$, and two transformations $(PN_0,M_0) \overset{prod_1,m_1}{\Longrightarrow} (PN_1,M_1)$ and $(PN_0,M_0) \overset{prod_2,m_2}{\Longrightarrow} (PN_2,M_2)$ via the corresponding rules and matches.

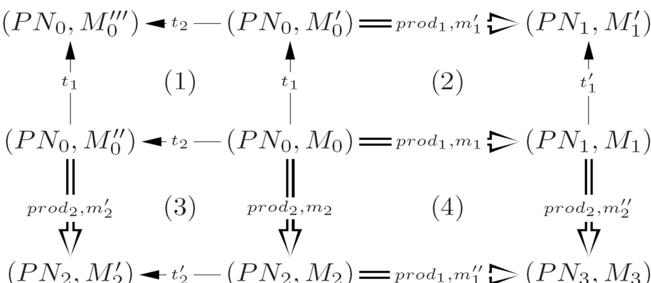


Figure 5: Concurrency in RONs

Hence, we distinguish three kinds of conflicts corresponding to the squares (1), (2/3) and (4):

Transition / Transition: The classic concurrency situation in P/T systems without place capacities regards two transitions with overlapping predomains. Such transitions are in conflict if

they are both enabled and would require more tokens for firing than are available in the current marking. Hence, for square (1), we have the usual condition that t_1 and t_2 need to be conflict free, so that both can fire in arbitrary order or in parallel, yielding the same marking.

Example: Consider the net which results from applying rule *addResource* to the *Producer* net in Fig. 1: here the two transitions *UseResource1* and *UseResource2* are in conflict because there is only one token on the place *ready to produce*.

Rule / Transition: The application of a rule might remove an enabled transition from a given net. In this case one might be able to fire the transition first and apply the rule afterwards, but not the other way round. Vice versa, a rule might require a token on a place which might have been removed by a preceding transition firing step. Hence, for squares (2) and (3), we require parallel independence which allows the execution of the transformation step and the firing step in arbitrary order leading to the same object net. Intuitive, a transition firing and a transformation are parallel independent if the transition is not deleted by the transformation and the successor marking is still sufficient for the match of the transformation (see [EHP+07]).

Example: Firing the transition *produce* in net *Prod1* on the RON place *Producers* is in conflict with applying the net transformation rule *refineProducer* to the net *Prod1*. Since the net transformation rule removes the transition *produce*, the firing step cannot take place after the net transformation step. Vice versa, the transition can fire first without disabling the application of rule *refineProducer*.

Rule / Rule: Two rules are in conflict with each other if one of them deletes certain parts of the object net which the other rule needs for its application. Another conflict possibility is the creation of net structures by the first rule which are forbidden by a NAC of the second one. Hence, for square (4), we require parallel independence of both rules. In [EHP+07, EEH+07] it has been shown that reconfigurable P/T nets fulfill the formal conditions of a weak adhesive HLR category[1]. Using such a category not only allows the notions of rules and transformations, but in addition provides a large amount of results such as:

- Parallelism: The Church-Rosser Theorem states a local confluence in the sense of formal languages. The Parallelism Theorem states that sequential or parallel independent transformations can be carried out either in arbitrary sequential order or in parallel. In the context of step-by-step development these theorems are important as they provide conditions for the independent development of different parts or views of the system.

- Concurrency and pair factorization: The Concurrency Theorem handles general transformations, which may be non-sequentially independent. Roughly speaking, for a sequence there is a concurrent rule that allows the construction of a corresponding direct transformation.

- Embedding and local confluence: Further important results for transformation systems are the Embedding, Extension and the Local Confluence Theorems [EEPT06]. The first two

[1] Adhesive High-Level Replacement (HLR) systems have been established as a suitable categorical framework for double-pushout transformations[EEPT06].

allow to embed transformations into larger contexts and with the third one we are able to show local confluence of transformation systems based on the confluence of critical pairs.

Example: Both rule *addResources* and rule *refineProducer* delete the original transition *produce*, when applied to net *Prod1*. Hence, only one of these rules can be applied, disabling the application of the other one (delete-use-conflict). Another example for a conflict would be the rule *mergePC*. The NAC of this rule forbids the application if there already exists a *connect* transition between the two buffers which results in a conflict of the rule with itself. However the structure of the high-level net will prevent two consecutive applications of this rule to the same net.

4.2 Implementation

The editor described in the previous chapter offers support to model reconfigurable object nets and to perform hand-triggered simulation steps.

We extended the basic editing and simulation features of our RON environment by the analysis techniques explained in the previous section. To perform the analysis we employ the attributed graph grammar system AGG [AGG], an environment for the execution and analysis of graph transformations. Since Petri nets and net transformation rules can be simulated as special graphs and graph transformation rules, we use AGG to compute possible rule matches, to apply rules and to perform rule/rule conflict analysis. More specifically, we implemented three methods for the three analysis cases:

1. Method `analyzeTransitionTransition(objectnet)` is implemented directly and searches the given object net for conflicting transitions. The result is a vector of transition pairs which are in conflict. We implemented this analysis directly like the simulation of object net transition firing described in Section 3.

2. Method `analyzeRuleTransition(objectnet, rule)` collects all possible matches of the given rule into the given object net and checks whether there exist enabled transitions in the object net whose firing would delete tokens which are parts of the match. Vice versa, the method checks whether the rule application would delete any of the previously enabled transitions. The resulting pairs, each consisting of an enabled transition and a match from the LHS of a rule to the object net, describe situations where the transition will no longer be enabled after applying the rule. We use AGG for match-computing and implement the formal criteria for conflicts in [EHP+07].

3. Method `analyzeRuleRule(rule1, rule2, objectnet)` computes all critical pairs for `rule1` and `rule2` and checks whether the overlapping graphs are parts of the intersection of matches from `rule1` and `rule2` into the object net. Here we use AGG critical pair analysis for this. The result of the computation will be a vector containing pairs of matches of the LHS of both rules into the object net. Semantically each pair describe a situation where the application of the first rule with the first given match will prevent the application of the second rule under the second given match.

Given a specific RON, our editor offers two modes to perform a conflict analysis: single conflict mode and local place mode.

4.3 Single conflict analysis

This mode allows to specify in detail which conflicts are interesting and should be computed. In our producer-consumer system for example, we would like to know either whether our rules for adapting a producer's production workflow might lead to object nets with transition/transition conflicts, or we might be interested in transition/rule conflicts with the transitions of the producer net itself or in possible rule/rule conflicts.

For a transition/transition analysis, the user of the RON environment selects an object net and chooses Analyze Conflicts T/T from its context menu. Window (a) in Fig. 6 shows the result of the T/T analysis of the object net Prod1, where the two transitions in conflict are highlighted.

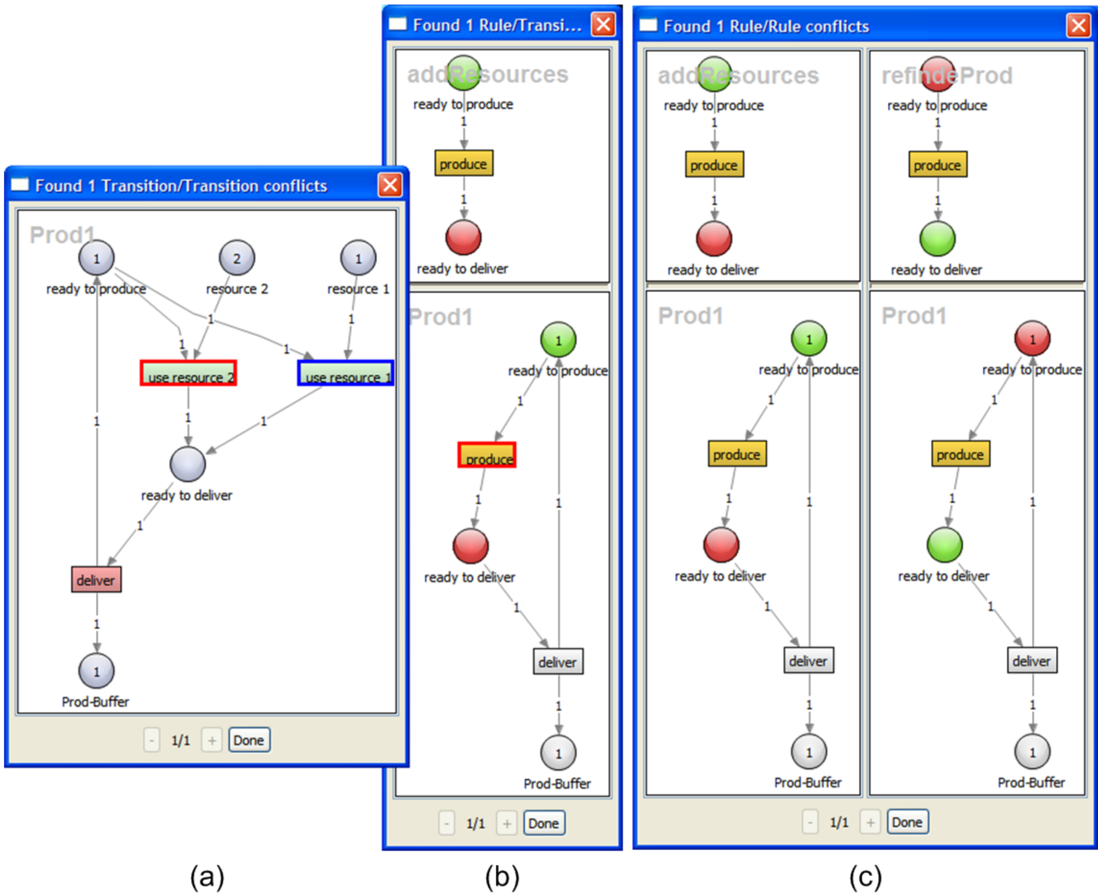


Figure 6: A Transition/Transition Conflict (a), a Rule/Transition Conflict (b), and a Rule/Rule Conflict (c) in the Producer-Consumer System

For a rule/transition analysis, the user of the RON environment selects a rule and an object net which marks a NET place connected by an APPLYRULE transition to the RULE place which contains the selected rule. Afterwards, the item Analyze Conflicts R/T can be chosen from the context menu. Window (b) in Fig. 6 shows that rule *refineProd* and transition *produce* are in R/T conflict with each other because transition *produce* is deleted by the rule. Note that the steps

still can be performed in reversed order: after firing transition *produce*, rule *refineProd* is still applicable because it does not depend on the tokens which are consumed by the transition.

For a rule/rule analysis, two rules and an object net have to be selected in order to be able to evoke the context menu item Analyze Conflicts R/R. Window (c) in Fig. 6 shows that rules *refineProd* and *addResources* (see Fig. 2) are in conflict with each other. Since the matches of the rules overlap, both rules replace the same transition *produce* by a more complex structure.

4.4 Local place conflict analysis

In this analysis mode, a comprehensive direct analysis is evoked for a selected high-level NET place. This means, all possible local conflicts are computed, taking into account the current object nets comprising the marking of the selected NET place, the marking of these object nets, and the rules on RULE places which are connected by an APPLYRULE transition to the selected NET place. Fig. 7 shows the result of a local place analysis, evoked for the NET place Producer in our Producer-Consumer-RON from Fig. 1.

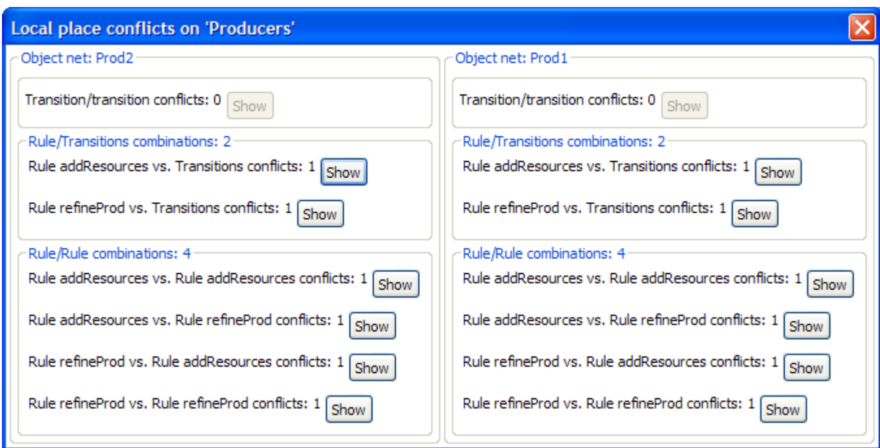


Figure 7: Local Place Analysis Results for the NET Place Producer

For all object nets being tokens on the selected high-level NET place, the calculated conflicts are displayed, sorted by conflict types transition/transition, rule/transition, and rule/rule. Pressing the corresponding Show button shows the respective conflict in detail, i.e. the conflicting transitions are highlighted (Fig. 6 (a)), or a match from a rule is shown together with a highlighted transition this rule is in conflict with (Fig. 6 (b)), or the overlapping matches of two conflicting rules being in conflict into an object net are shown (Fig. 6 (c)).

5 Conclusion

Modeling mobile and distributed systems requires a modeling language which covers both reconfiguration and coordination. RONs meet these conditions. The abstract high-level net controls the flow, selection, manipulation and behavior of object nets (P/T nets) which are tokens on the high-level net places. For object net reconfiguration at runtime, net transformation rules are

used, the application of which is also controlled by the high-level net. In this paper, the RON environment for editing and simulating RONs has been extended with conflict analysis. Apart from the classical situation of two transitions in a P/T net being in conflict, it is now also possible to analyze conflicts between parallel applicable net transformation rules, as well as between enabled transitions and net transformation rules. The knowledge about conflicts helps to detect potential problems in system behavior, e.g. "*a* produce *transition can be deleted even if there are still resources for productions available*".

To the best of our knowledge, no other Petri net tool offers the possibility to define and analyze net transformations by rules represented as tokens in a high-level net.

Work is in progress to optimize the analysis result visualization (e.g. by showing the overlapping part of two rule matches only once, and by indicating which are the critical objects causing the conflict). Furthermore, we plan to improve the conflict analysis performance for analyzing also more complex case studies.

Bibliography

[AGG] AGG Homepage. http://tfs.cs.tu-berlin.de/agg.

[BEHM07] E. Biermann, C. Ermel, F. Hermann, T. Modica. A Visual Editor for Reconfigurable Object Nets based on the ECLIPSE Graphical Editor Framework. In Juhas and Desel (eds.), *Proc. 14th Workshop on Algorithms and Tools for Petri Nets (AWPN'07)*. GI Special Interest Group on Petri Nets and Related System Models, 2007. http://tfs.cs.tu-berlin.de/publikationen/Papers07/BEHM07.pdf

[EEH+07] H. Ehrig, C. Ermel, K. Hoffmann, J. Padberg, U. Prange. Concurrency in Reconfigurable Place/Transition Systems: Independence of Net Transformations as Well as Net Transformations and Token Firing. Technical report 2007/02, TU Berlin, 2007. http://iv.tu-berlin.de/TechnBerichte/2007/2007-02.pdf

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs. Springer Verlag, 2006.

[EHP+07] H. Ehrig, K. Hoffmann, J. Padberg, U. Prange, C. Ermel. Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In Kleijn and Yakovlev (eds.), *Petri Nets and Other Models of Concurrency. Proceedings of ICATPN 2007*. LNCS 4546, pp. 104–123. Springer Verlag, 2007. http://tfs.cs.tu-berlin.de/publikationen/Papers07/EHP+07.pdf

[EMF06] Eclipse Consortium. Eclipse Modeling Framework (EMF) – Version 2.2.0. 2006. http://www.eclipse.org/emf.

[EP04] H. Ehrig, J. Padberg. Graph Grammars and Petri Net Transformations. In *Lectures on Concurrency and Petri Nets Special Issue Advanced Course PNT*. LNCS 3098, pp. 496–536. Springer Verlag, 2004.

[Erm06] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Technische Universität Berlin, Fak. IV, Books on Demand, Norderstedt, 2006.

[For06] ForMA$_l$NET. DFG Project, Technical University of Berlin. Formal Modeling and Analysis of Flexible Processes in Mobile Ad-hoc Networks. 2006. http://www.tfs.cs.tu-berlin.de/formalnet

[GEF06] Eclipse Consortium. Eclipse Graphical Editing Framework (GEF) – Version 3.2. 2006. http://www.eclipse.org/gef.

[HME05] K. Hoffmann, T. Mossakowski, H. Ehrig. High-Level Nets with Nets and Rules as Tokens. In *Proc. of 26th Intern. Conf. on Application and Theory of Petri Nets and other Models of Concurrency*. LNCS 3536, pp. 268–288. Springer Verlag, 2005. http://tfs.cs.tu-berlin.de/publikationen/Papers05/HEM05.pdf

[PEH07] J. Padberg, H. Ehrig, K. Hoffmann. Formal Modeling and Analysis of flexible Processes in Mobile Ad-Hoc Networks. *EATCS Bulletin* 91:128–132, 2007. http://tfs.cs.tu-berlin.de/publikationen/Papers07/PEH07.pdf

[PER95] J. Padberg, H. Ehrig, L. Ribeiro. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science* 5:217–256, 1995.

[PU03] J. Padberg, M. Urbášek. Rule-Based Refinement of Petri Nets: A Survey. In Ehrig et al. (eds.), *Advances in Petri Nets: Petri Net Technology for Communication Based Systems*. LNCS 2472, pp. 161–196. Springer Verlag, 2003.

[RON07] Student's Visual Language Project. TFS, TU Berlin. Reconfigurable Object Nets. 2007. http://www.tfs.cs.tu-berlin.de/roneditor

[Val98] R. Valk. Petri Nets as Token Objects: An Introduction to Elementary Object Nets. In *ICATPN '98: Proceedings of the 19th International Conference on Application and Theory of Petri Nets*. LNCS 2987, pp. 1–25. Springer, 1998.

The GP Programming System

Greg Manning and Detlef Plump

The University of York

Abstract: We describe the programming system for the graph-transformation language GP, focusing on the implementation of its compiler and abstract machine. We also compare the system's performance with other graph-transformation systems. The GP language is based on conditional rule schemata and comes with a simple formal semantics which maps input graphs to sets of output graphs. The implementation faithfully matches the semantics by using backtracking and allowing to compute all possible results for a given input.

Keywords: GP, programming system, graph transformation, non-determinism

1 Introduction

GP is a non-deterministic graph programming language based on conditional rule schemata in the double-pushout approach [PS04]. The core of GP consists of just four constructs: single-step application of a set of rule schemata, sequential composition, branching and iteration. The language is computationally complete [HP01] and comes with a formal semantics [PS08]. The current implementation of GP consists of a graphical editor for programs and graphs, a compiler and the York Abstract Machine (YAM). These components communicate as shown in Figure 1 (where YAMG is an internal graph format of the abstract machine).

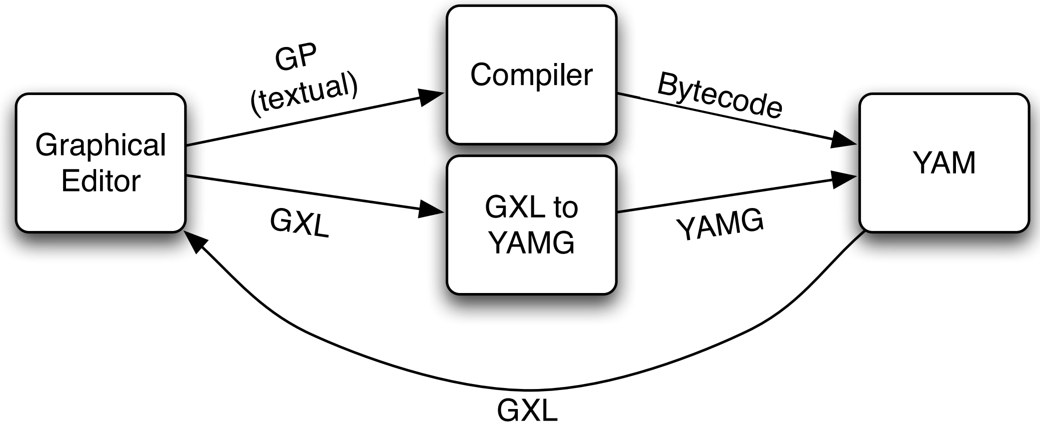


Figure 1: An overview of the GP system

We describe GP by means of an example. Consider the program `minimum_spanning_tree`

in Figure 2. This program calculates a minimum spanning tree for its input graph.[1] The program

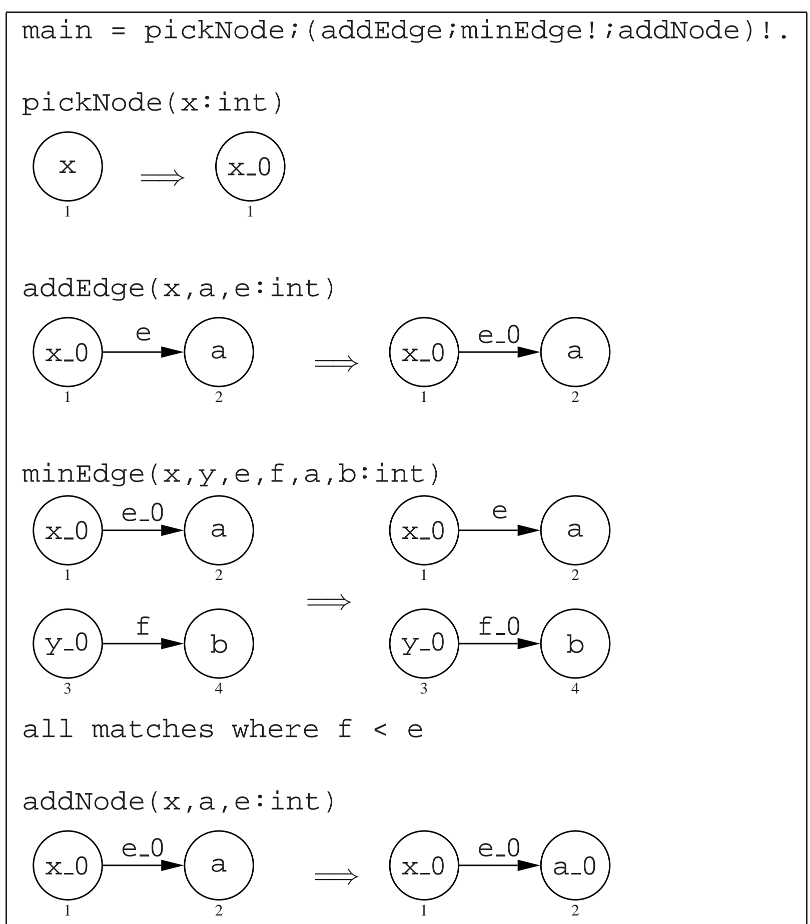


Figure 2: GP program `minimum_spanning_tree`

consists of four rule-schema declarations and the main command sequence following the key word `main`. Given an input graph whose nodes and edges are labelled with integers, the program first uses the rule schema `pickNode` to choose any node and replace its label x with x_0. The underscore operator allows to add a *tag* to a label, where in general a tagged label consists of a sequence of expressions joined by underscores. (Sequences of expressions are just ordinary labels, allowing GP's underlying theory to be based on a standard variant of the double-pushout approach rather than on some complicated model of attributed graph transformation.) This program uses the tag 0 to mark the nodes of a spanning tree. After the initial node has been marked, the iteration operator '!' executes the subprogram `(addEdge;minEdge!;addNode)` as long as possible. The subprogram first picks any edge between a marked node and an unmarked node. Then the loop `minEdge!` repeatedly swaps this edge with an edge having a smaller label, where

[1] A spanning tree for a directed graph G is a subgraph S of G such that the undirected graph underlying S is a spanning tree for the undirected graph underlying G.

the latter is checked by the condition `where f < e`. The flag `all matches` allows this rule schema to be matched non-injectively whereas the default in GP is injective matching. After the minimum edge between the current tree nodes and any unmarked node has been determined, the unmarked node of this edge is added to the spanning tree by the rule schema `addNode`. It is not difficult to see that upon termination of the outer loop, the marked nodes and edges constitute a minimum spanning tree of the input graph. (The rule schemata `addEdge`, `minEdge` and `addNode` are actually sets of rule schemata which are obtained from those depicted by reversing edges in all possible ways: `addEdge` and `addNode` consist of two rule schemata while `minEdge` contains four schemata. For readability, we have omitted these rule schemata in Figure 2.) In general, a graph can have several minimum spanning trees and the program in Figure 2 allows to compute all of them.

A leitmotiv for GP's design has been syntactic and semantic simplicity, see also [PS08]. There is only one other core construct besides those occurring in the minimum spanning tree program: a conditional statement of the form `if` C `then` P `else` Q (where C, P and Q are programs). Our programming experience so far suggests that these few constructs are sufficient and allow succinct solutions to problems. It is possible though to simulate more elaborate control mechanisms. Consider, for example, a conditional loop of the form `while` C `do` P which executes its body P as long as the program C succeeds. An equivalent GP program is (`if` C `then` P `else` `fail`)!; `if` C `then fail` (where `fail` is an always failing program such as the empty set of rules). As another example, the choice to apply a rule r either once or not at all can be simulated by the rule set $\{r, \emptyset \Rightarrow \emptyset\}$ (where $\emptyset \Rightarrow \emptyset$ has the empty graph on both sides).

The rest of this paper is organized as follows. The next section briefly adresses the graphical user interface of the GP system, Section 3 introduces the York abstract machine and Section 4 discusses the GP compiler. Section 5 compares the performance of the GP system with other graph-transformation environments. In Section 6, we conclude and give some topics for future work.

2 Graphical Editor

The GP graphical editing environment is a Java application which allows graph and program creation, loading, editing and saving, and program execution on a given graph. The outputs of executions are then available as inputs to other programs. Figure 3 shows a screenshot of the graphical editor with the rule `minEdge` of Figure 2 being edited. The editor visualises graphs using the prefuse data visualisation library [HCL05], which permits graph layout and editing. The main graph drawing algorithm used is a force-directed layout. Figure 5 shows a graph drawn by this algorithm.

3 The York Abstract Machine

The York abstract machine (YAM) is more fully described in [MP06]. Here, we give an overview highlighting the areas which have changed in the meantime.

The YAM is a backtracking graph-transformation machine which executes bytecode for low-level graph operations. It can handle nondeterministic programs and is in parts similar in de-

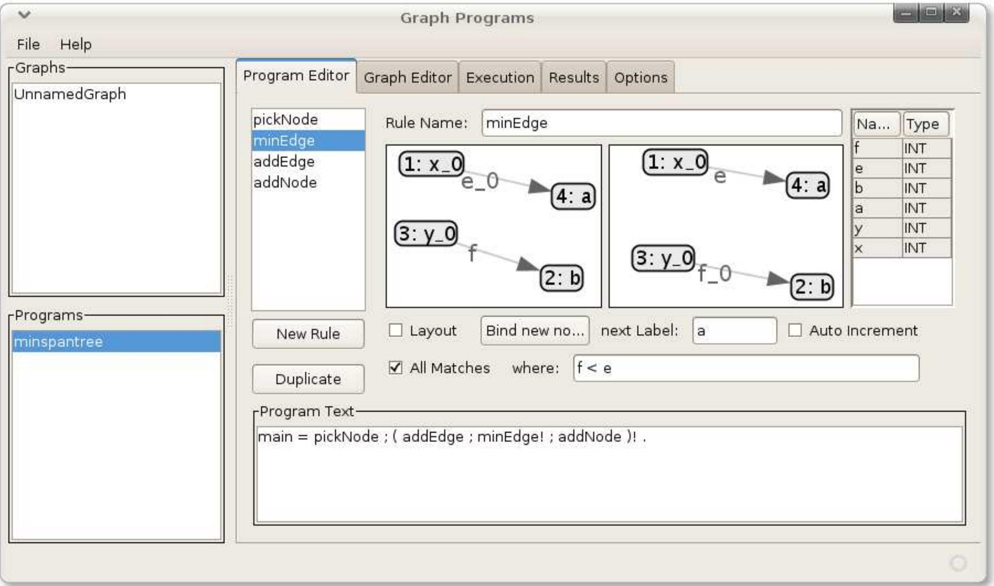


Figure 3: A screenshot of the graphical editor

sign to Warren's abstract machine for Prolog [AK91]: it manages GP's nondeterminism using a mixed stack of choice points and environment frames. The implementation of backtracking in PROGRES [Zün92] takes a WAM-like approach too, although it uses the host language's call stack rather than explicit data structures. The YAM also manages the current host graph and a (typically) small data stack.

Figure 4 shows an example state of the choice point and environment frame stack. Choice points consist of a record of the number of graph changes at their creation time, a program position to jump to if failure occurs when the choice point is the highest on the stack, and pointers to the previous choice and containing environment. The number of graph changes is recorded so that, when backtracking, the graph changes can be undone: using the stack of graph changes, the graph as it was at the choice point is recreated. Environment frames have a set of registers to store label elements or graph element identities, and an associated function and program position in the bytecode. They also show which environment and program position to return to. The number of registers each frame has is determined by the bytecode — it is fixed at compile time.

The current host graph is stored in a complex data structure, making use of the heavily optimised Judy data structures [Siv02]. The structure is designed in such a way that the graph can be interrogated easily and very quickly, at the cost of slightly slower graph updates. Typical queries to the graph structure are "edges whose target node is node n" or "nodes whose label has the value 1 in position 1". Each element (node or edge) in the graph is labelled with a list of values, each of which is of type integer or string. The YAM bytecode allows any query over the length of the list or the type or value of the list elements, such as "all nodes with a label of size two", or "all edges with an integer in the second position."

The machine as presented in [MP06] handled nondeterminism internally. At the bytecode

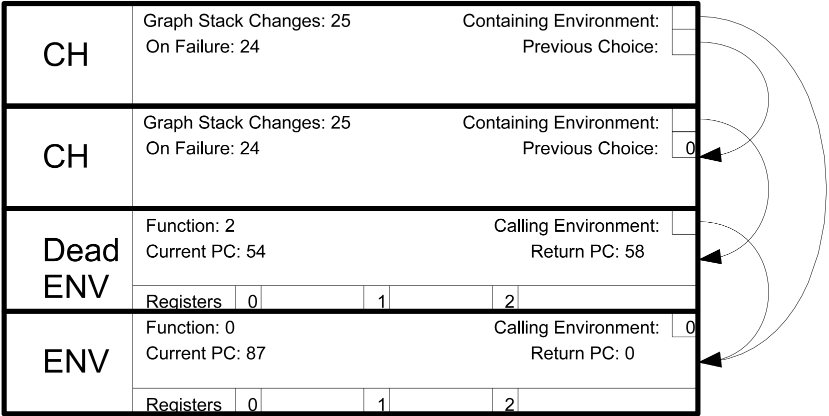


Figure 4: An example choice/environment stack

level, the instructions effectively returned a correct result: if a choice led to a failure (and such was a wrong choice), then the machine would trigger backtracking and retry the choice until a correct result was obtained. Now, however, the machine simply provides explicit instructions for handling nondeterminism such as `OnFail`[2], `UpdateFail`[3] and `Assert`[4]. This change was made to unite the failing and non-failing versions of the graph queries, and to allow more expressiveness at the bytecode level.

Using these instructions, the compiler constructs helper functions to implement backtracking. Nondeterministic choice between a set of graph rules is handled by trying them in textual order until one succeeds. Before each is tried, the failure behaviour is configured to try the next. Nondeterministic choice between graph-element candidates for a match is handled by choosing and saving the first element, and on failure, using the saved previous answer to return (and save) the next element.

Nodes and edges are identified in the structure by integers, and the graph structure contains many ordered lists of such integers. This allows complex conjunctive queries to be performed by intersecting ordered lists of integers. For example, in finding the left-hand side of the rule `addEdge` in the program of Figure 2, having found the `x_0` node, a list of potential edges is created by intersecting the list of all edges leaving this node, the list of all edges which have `e` as their first label, and the list of all edges having a label sequence of length one. The code then creates an environment to store the previous answer returned, and uses the `Next` instruction to give the first answer in the intersection which is numerically greater than the previous answer. It saves this answer to the stack and returns it. If a failure occurs whilst this choice point is on top of the stack, the code will return the next answer. When there are no more answers it will propagate the failure to a previous choice point.

---

[2] `OnFail` pops a code location and creates a new choice point which will jump to that code location on failure.

[3] `UpdateFail` pops a choice point pointer and a code location and changes the choice point so that it now goes to the new location on failure.

[4] `Assert` pops the top of stack and fails if it is zero.

Because the underlying data structure stores the node and edge references in the lists as ordered lists of integers, finding the next element in the intersection is very fast. An intersection can be done in time $O(ln)$, where l is the length of the shortest list in the intersection and n is the number of lists being intersected. Note that the entire intersection is not generated in one go, the elements of the intersection are found one at a time, as needed.

4 Compiler

The GP compiler converts textual GP programs into YAM bytecode. It does this by translating each individual rule or macro into a sequence of instructions, and composing these sequences using the YAM function calls.

4.1 Generating a searchplan for graph matching

Searchplan generation is a common technique for implementing graph matching [GBG+06, HVV07, Zün96]. The GP compiler decomposes graph rules into a static searchplan of node lookups, edge lookups (find an edge whose source and target have not been found yet) and extensions (find an edge whose source or target has been found). The choice and order of these search operations is determined using the following priorities, always preferring elements with value labels over those with variable labels:

1. Check parts of `where` clauses whose variables have all been bound, that is, all label variables have been instantiated, and all nodes or edges referred to have been found.
2. Find nodes on the ends of edges which have been found.
3. Find edges where both the start and the end node have been found.
4. Find edges where either the start or the end node has been found.
5. Find nodes where there is a negative edge condition of the form `not edge(`v`,`w`)` at the top level of the `where` clause[5] and either v or w has been found.
6. Find nodes.

The nondeterminism in this list of priorities increases: where clause checks and finding nodes of known edges are deterministic operations, finding unrestricted nodes is highly nondeterministic. There will be many different plans which satisfy these priorities, however since the compiler generates static plans (that is, the host graph is not interrogated), there is no more information to use in the generation. The choice between possible plans is made using an ordering taken from the programmer: elements mentioned first in the textual input program will be found first. For example, the first step in a searchplan is to find the first (labelled) node mentioned.

Both GrGen [GBG+06] and Fujaba [NNZ00] also make extensive use of searchplans. GrGen.NET [BG07] uses online searchplan generation, so that the searchplans can be recalculated during execution. This improves the quality of the generated searchplan significantly.

[5] That is, the negative edge condition is one of the conjuncts $C_1,...,C_n$ in `where` C_1 `and` ... `and` C_n.

Once a match has been found, and the `where` clause has passed, the remainder of the code for the graph rule handles the changes to be made to the graph. The compiler determines the changes and orders them as follows: deleted edges, deleted nodes, relabelled nodes or edges, added nodes, added edges. This order ensures that no nodes are deleted before their incident edges, and no edges are created before their incident nodes.

4.2 Compiling GP commands

With the individual graph rules compiled, they can be composed into a complete YAM program. There are several ways of joining subprograms in GP: sequential composition, macro calling, if-then-else branching, and as-long-as-possible iteration. The compilation of a sequential composition $P;Q$ is trivial: the bytecodes for P and Q are concatenated. Macro calling is achieved by using the `Call` or `TailCall`[6] bytecode instructions.

As-long-as-possible iteration, $P!$, is implemented as follows:

1. Create new failure behaviour to succeed (continue) on failure.
2. Execute P once.
3. Change failure from instruction 1 from succeed to fail. Having a failure behaviour which simply fails is the same as having no failure behaviour at all; however, failure behaviours cannot be removed since they may be referenced elsewhere in the stack.
4. Go to instruction 1.

The failure behaviour must be altered in step 3 to maintain the semantics and not an any-number-of-times semantics.

As rule sets (apply one rule from a set) are compiled, an ordering is imposed upon them. The compiled bytecode tries the first rule, and on failure will try the next rule until the end of the set is reached. If none of the rules successfully applied, then the whole rule set fails. The ordering imposed is the order in which the rules appear in the program text.

GP's branching construct `if` C `then` P `else` Q first executes the subprogram C on the input graph. If this yields a result, program P is executed *on the input graph*. Otherwise, if all executions of C end in failure, program Q is executed on the input graph. The construct is compiled in the following way:

1. Create failure behaviour to go to step 5 on failure.
2. Execute the condition C.
3. `ClearFail` the failure point created in step 1, that is, undo the graph changes, forget the choices back to that point and remove that failure frame, but leave the program pointer unchanged.
4. Execute the then-part P and succeed.
5. Execute the else-part Q and succeed.

[6] `TailCall` is equivalent to call-then-return, but is actually implemented as return-then-call because this saves space on the call stack.

5 Performance

In this section we compare the performance of the GP system with the performance of similar environments. We focus on a simple problem which has been implemented in different graph programming systems in the context of the AGTIVE 2007 tool contest [TBB+08]. The task is to generate a graph of the *n*th generation of the Sierpinski triangle, producing one generation at a time. A Sierpinski triangle is a triangle split into 4 subtriangles (made by joining the midpoints of the 3 edges), where the 3 subtriangles containing one of the original vertices are themselves Sierpinski triangles. Sierpinski triangles are represented as graphs using nodes as vertices of triangles and edges as edges of triangles. As a true Sierpinski triangle has infinite detail, we must generate an approximation. We say that the *n*th generation of a Sierpinski triangle is one which has a depth of *n*. The 0th generation Sierpinski triangle is a simple triangle. Figure 5 shows the 4th generation Sierpinski triangle.

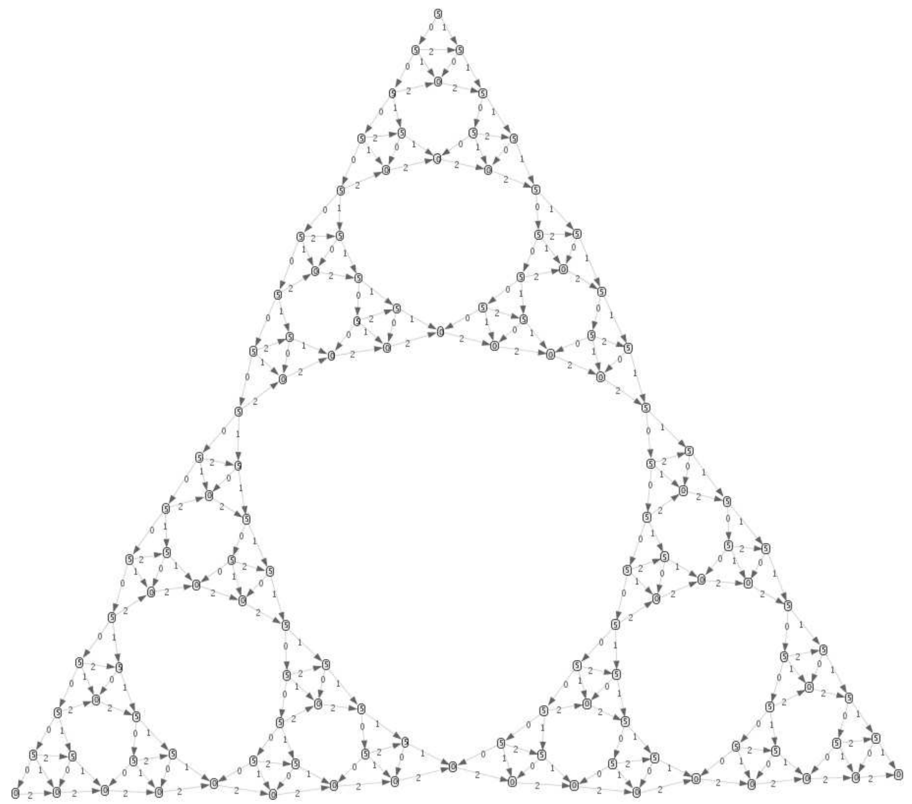


Figure 5: A 4th generation Sierpinski triangle

5.1 Generating Sierpinski triangles with GP

The GP program `sierpinski` is presented in Figure 6. It expects as input a graph consisting of a single node labelled with the generation number of the Sierpinski triangle to be produced. The rule schema `init` creates the initial Sierpinski triangle (generation 0) and turns the input node into a unique "control node" whose label is of the form `x_y`. The underscore operator is used here to hold the required generation number `x` and the current generation number `y` in a single node.

After `init` has been applied, the nested loop `(inc; expand!)!` is executed. In each iteration of the outer loop, the rule schema `inc` increases the current generation number if it is smaller than the required number. The latter is checked by the condition `where x > y`. If the test is successful, the inner loop `expand!` performs a Sierpinski step on each triangle whose root[7] is labelled with the current generation number: the triangle is replaced by four triangles such that the roots of the three outer triangles are labelled with the next higher generation number. The test `x > y` fails when the required generation number has been reached. In this case the application of `inc` fails and, as a consequence, the outer loop terminates and returns the current graph. It is not difficult to see that the resulting graph is indeed the Sierpinski triangle of the required generation.

5.2 Comparison with other systems

In Figure 7 we present the execution times for the GP system and some other graph transformation systems that participated in the Sierpinski tool contest. The times for GP were obtained on a PC with an Intel Pentium 4 processor with a clock rate of 2.8GHz and 512MB of main memory. The times for the other systems were obtained on comparable machines. Our figure includes only a subset of the tools described in [TBB+08]. We have omitted tools that are tailored for parallel rule applications in specialised areas but cannot be considered as general-purpose graph transformation tools.

As Figure 7 demonstrates, GP is faster than five other systems and is beaten only by GrGen.NET and Fujaba. GrGen.NET requires the programmer to specify types of node and edges (often hierarchical types with multiple inheritance). The information gained from these types gives more information to the graph matching algorithm and also allows better compilations. GP has very little typing, freeing the programmer from specifying these overarching types. This allows shorter, more succinct programs at the cost of some speed. However, as demonstrated by this benchmark, the speed lost is not too great.

5.3 Non-deterministic programs

Other graph programming systems do not fully exploit the non-deterministic nature of graph transformation rules. The semantics of GP programs on input graphs are *all* possible output graphs, and this is taken seriously by the implementation in that it provides users with the option to generate several or even all possible results. This mechanism is complete for terminating

[7] The *root* of a triangle is the unique node (if it exists) from which a 0-edge and a 1-edge is outgoing. Note that the inner triangle on the right-hand side of `expand` does not have a root, hence it will never be expanded.

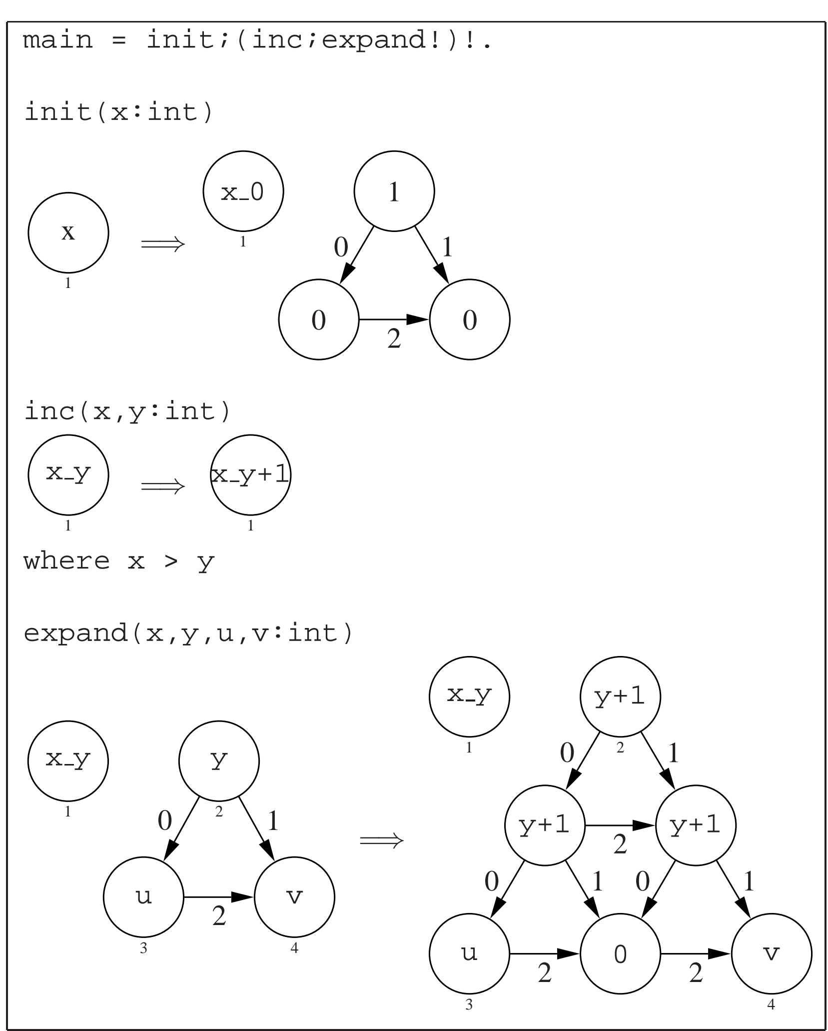


Figure 6: The Program `sierpinski`

programs. In contrast, AGG [ERT99] makes its nondeterministic choices randomly, with no backtracking. Similarly, Fujaba has no backtracking. It seems that PROGRES [SWZ99] is the only other graph transformation language in use that provides backtracking.

The Sierpinski example presented above is a deterministic problem. That is, the program `sierpinski` computes a function where each input graph produces a single output graph. Although in the GP implementation there is a choice of which order to convert subtriangles to the next generation, since they will all get done eventually this is a *confluent* program in that all output graphs are isomorphic. This is not always the case. For example, the program `minimum_spanning_tree` presented in the Introduction is non-confluent: for an input graph, there is not necessarily a unique minimum spanning tree. The implementation of GP respects the semantics, and allows computation of all possible minimum spanning trees. The use of

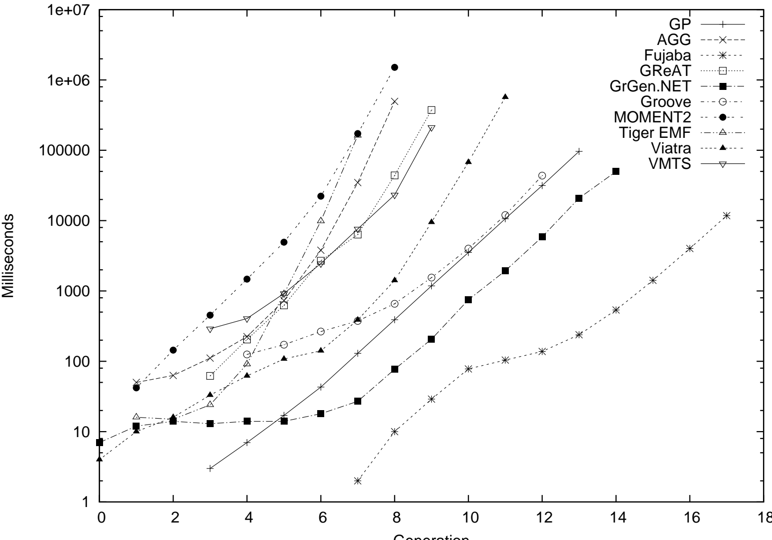


Figure 7: Execution times for the Sierpinski benchmark

this non-determinism is not limited to finding multiple answers. It is possible, and common, to write programs which rely on the backtracking behaviour to filter results, or make correct non-deterministic guesses. Most other graph transformation systems do not allow such programs. Neither AGG nor Fujaba allow backtracking over graph rules. The GrGen.NET API supplies tools necessary to perform backtracking (such as presenting all possible matches of a rule), but the GrShell example environment [BG07] does not allow backtracking in this manner.

Using nondeterminism to this extent is sometimes problematic. In the Sierpinski example, the order in which the matches of the `expand` rule are applied makes no difference, yet if later in the program there was a failure, the backtracking mechanism would try all possible different orders of the matches. It is an item of future work to develop analysis techniques to detect and disable the backtracking in such cases (see also the remarks in the next section). Since no backtracking is required in the Sierpinski example, our solution had the backtracking mechanism of the YAM disabled.

6 Conclusion and Future Work

The GP implementation matches faithfully GP's semantics and allows to compute all results of a (terminating) program. GP is a small clean language, with enough structure so that it is usable, but little enough that the semantics is understandable and useable for arguments and proofs [PS08]. The system is reasonably fast; slow execution is usually caused by a vast nondeterministic search space which can often be avoided by programming carefully.

The YAM can give more than one answer, or all answers. It provides a clearly defined separation between runtime and compile time actions. Whilst the YAM has been designed as part

of the GP system, it is by no means restricted to it; other graph systems and semantics could be realised using the bytecode provided by the YAM.

In the current implementation of GP, only one match of one rule is executed at a time. Other graph programming systems can execute multiple rules or multiple matches in parallel, which can give large speed gains in certain situations. The GP system does not currently do this because it involves a considerable amount of checking that all the matches can be successfully executed without interfering with each other, and would make the bookkeeping for backtracking very complex.

The GP system generates static searchplans at compile time, so no host-graph interrogation is possible. With runtime searchplan generation (as in GrGen.NET [GBG+06]), it is possible to always match the rarest elements first, which reduces the search space to find a match.

As GP programs get larger, it may be useful to include optional conservative static type checking. This may be implemented as graph metamodels or more complex typing systems such as the GRS types of [BPR04]. By analysing programs, it will sometimes be possible to guarantee that certain graph structures do or do not occur.

In many cases, nondeterministic (sub)programs are confluent: they cannot possibly fail, and all solutions are isomorphic. Using static analysis techniques such as critical-pair analysis [Plu05], it will sometimes be possible to detect these situations. This is useful information in itself, but can also be used to speed up the implementation, since backtracking would not be required through a confluent section of a program.

Bibliography

[AK91] H. Aït-Kaci. *Warren's Abstract Machine: A Tutorial Reconstruction*. MIT Press, 1991.

[BG07] J. Blomer, R. Geiß. The GrGen.NET User Manual. Technical report 2007-5, Universität Karlsruhe, IPD Goos, July 2007.
http://www.info.uni-karlsruhe.de/papers/TR_2007_5.pdf

[BPR04] A. Bakewell, D. Plump, C. Runciman. Specifying Pointer Structures by Graph Reduction. In *Applications of Graph Transformations With Industrial Relevance (AGTIVE 2003), Revised Selected and Invited Papers*. Lecture Notes in Computer Science 3062, pp. 30–44. Springer-Verlag, 2004.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. The AGG Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2, chapter 14, pp. 551–603. World Scientific, 1999.

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. Graph Transformations (ICGT 2006)*. Lecture Notes in Computer Science 4178, pp. 383–397. Springer-Verlag, 2006.

[HCL05] J. Heer, S. K. Card, J. A. Landay. Prefuse: A Toolkit for Interactive Information Visualization. In *Proc. SIGCHI Conference on Human Factors in Computing Systems (CHI 2005)*. Pp. 421–430. ACM Press, 2005.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. In *Proc. Foundations of Software Science and Computation Structures (FOSSACS 2001)*. Lecture Notes in Computer Science 2030, pp. 230–245. Springer-Verlag, 2001.

[HVV07] Á. Hórvath, G. Varró, D. Varró. Generic Search Plans for Matching Advanced Graph Patterns. In *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)*. Electronic Communications of the EASST 6. 2007.

[MP06] G. Manning, D. Plump. The York Abstract Machine. In *Proc. Graph Transformation and Visual Modelling Techniques (GT-VMT 2006)*. Electronic Notes in Theoretical Computer Science. Elsevier, 2006. To appear.

[NNZ00] U. Nickel, J. Niere, A. Zündorf. The FUJABA Environment. In *Proc. Software Engineering (ICSE 2000)*. Pp. 742–745. ACM Press, 2000.

[Plu05] D. Plump. Confluence of Graph Transformation Revisited. In Middeldorp et al. (eds.), *Processes, Terms and Cycles: Steps on the Road to Infinity: Essays Dedicated to Jan Willem Klop on the Occasion of His 60th Birthday*. Lecture Notes in Computer Science 3838, pp. 280–308. Springer-Verlag, 2005.

[PS04] D. Plump, S. Steinert. Towards Graph Programs for Graph Algorithms. In *Proc. International Conference on Graph Transformation (ICGT 2004)*. Lecture Notes in Computer Science 3256, pp. 128–143. Springer-Verlag, 2004.

[PS08] D. Plump, S. Steinert. A Structural Operational Semantics for GP. 2008. In preparation.

[Siv02] A. Siverstein. Judy IV Shop Manual. 2002. http://judy.sourceforge.net.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 2, chapter 13, pp. 487–550. World Scientific, 1999.

[TBB+08] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, Á. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, T. Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. In *Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007), Revised Selected and Invited Papers*. Lecture Notes in Computer Science. Springer-Verlag, 2008. To appear.

[Zün92] A. Zündorf. Implementation of the imperative/rule-based language PROGRES. Technical report 92-38, Fachgruppe Informatik, RWTH Aachen, 1992. http://citeseer.ist.psu.edu/albert92implementation.html

[Zün96] A. Zündorf. Graph Pattern Matching in PROGRES. In *Proc. Graph Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 1073, pp. 454–468. Springer-Verlag, 1996.

Type Checking C++ Template Instantiation by Graph Programs

Karl Azab and Karl-Heinz Pennemann

azab@informatik.uni-oldenburg.de, pennemann@informatik.uni-oldenburg.de
Carl v. Ossietzky Universität Oldenburg, Germany

Abstract: Templates are a language feature of C++ and can be used for metaprogramming. The metaprogram is executed by the compiler and outputs source code which is then compiled. Templates are widely used in software libraries but few tools exist for programmers developing template code. In particular, error messages are often cryptic. During template instantiation, a compiler looks up names that depend on a template's formal parameters. We use graphs to represent the relevant parts of the source code and a graph program for the name lookup and type checking for expressions involving such names. This technique provides compiler writers with a visual way of writing algorithms that generate error messages and forms the basis for a visual inspection of type problems and suggested remedies for the programmer. Our graph program terminates and emits correct error messages.

Keywords: Graph programs, Type checking, C++

1 Introduction

Templates are a feature of the C++ programming language for generic programming, i.e. programmed code generation. Generic source code is written by omitting the specific data types of variables and instead supplying those as parameters (*parameterized types*). A parameterized type and variable of that type can be used as any other type or variable, e.g. the type name can be used to resolve names and the variable's members can be accessed. This way, templates separate types from algorithms in design, and combines them into new class-types and functions at compile time. Compared to non-template code which uses a generic type like `void *`, an immediate advantage from templates is improved static type checking. Templates are used extensively in the Standard Template Library and Boost libraries [Jos99, AG04]. They have also found use in performance critical domains, such as scientific computing and embedded systems [Vel98, Str04]. An introduction to templates can be found in e.g. [Str00].

A class type or function containing generic source code is called a *template definition*. A list of type parameters for a particular template definition is called a *declaration*. For each unique declaration, the *template instantiation* mechanism generates a specialization of that template definition. A *specialization* is a copy of the definition where the parameterized types are replaced by the declaration's actual type parameters. Non-types, i.e. constants, are allowed as template parameters, allowing e.g. array sizes to be set at compile time. Templates form a computationally complete *metalanguage* [CE00], a sub-language of C++ executed during compilation.

Consider the following example: A parameterized type is used to resolve the name `size` in the template definition in Figure 1. The first specialization is for the declaration icon<char> and will not compile since the provided type `char` has no field named `size` and can therefore

not be used for the expression defining the array size. For the second specialization, if the type `resolution<128>` contains a static field named `size` of an unsigned integer type, then the second specialization will compile.

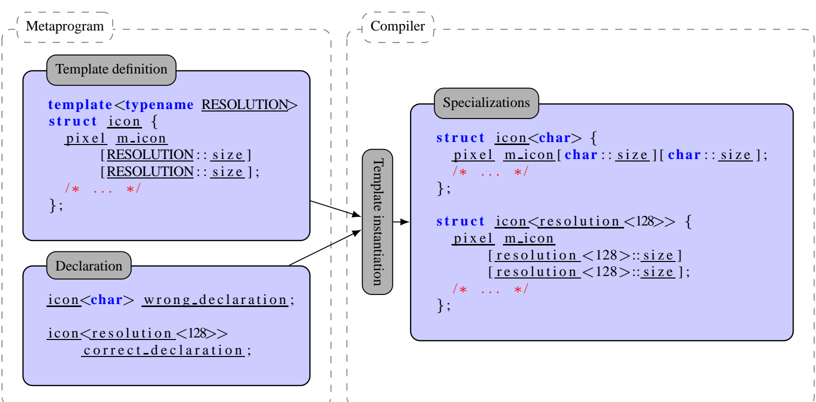


Figure 1: C++ template instantiation.

Even though templates are a useful technique, they can be complex and difficult to read and write. In particular, error messages are often cryptic. This has led to the development of methods and tools to analyze template code. The usage of specializations can be analyzed by debuggers, software patterns like tracers [VJ02], and tools like TUAnalyzer [GPG04]. For the metaprogram itself, research is being done on a debugging framework Templight [PMS06].

To improve error messages, we suggest modeling definitions and declarations by graphs, while name lookup and type checking of such graphs is made by graph programs that emit error messages as graphs instead of text. Graphs allow an abstract and visual representation of all necessary information, while graph programs provide an intuitive way of writing programs that detect problems and suggests remedies. In combination with presentation techniques such as focusing (on relevant parts) and hierarchical depiction, we believe that our model is usable as a basis for a visual inspection of type problems and suggested remedies.

Graph transformation systems is a well investigated area in theoretical computer science. An overview on the theory and applications is given in the book *Fundamentals of Algebraic Graph Transformation* [EEPT06]. Graph transformation systems rewrite graphs with (graph transformation) rules. A rule describes a left- and right-hand side. A transformation step is done by matching the left-hand side to a subgraph of the considered graph and modifying that subgraph according to the difference of the left- and right-hand side. Graph programs [HP01, PS04] provide a computationally complete programming language based on graph transformations. Graph conditions [HP05] can be used to express properties of graphs by demanding or forbidding the existence of specific structures. In a similar way, graph conditions can limit the applicability of rules in a graph program, by making demands on elements local to the subgraph matched by a rule.

In this paper, we use graphs to represent the template source code necessary for name lookup and type checking during template instantiation. We refer to those graphs as source-code graphs.

A graph program TTC (*Template-Type Checker*) looks up dependent names and detects type clashes in expressions for a subset of the C++ template features. TTC attempts to solve type clashes by implicit type casts. If such a cast loses precision, a warning message is generated. If no appropriate cast is found, an error message is generated, indicating the location of the error and suggesting a remedy for the programmer. TTC outputs a message graph, where errors and warnings are embedded. The message graph is interpreted by the programmer with the help of graph conditions. Graph conditions detect warning and error messages in graphs and when an error is present, they can determine for which declarations a definition can successfully be instantiated. Figure 2 gives an overview.

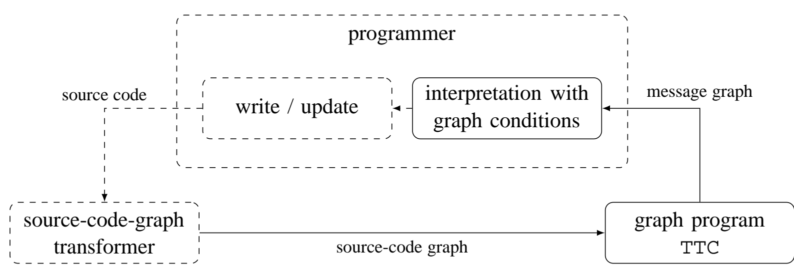


Figure 2: TTC type checks graphs and outputs error messages.

The paper is structured as follows. Graph programs are introduced in Section 2. Section 3 informally describes how C++ source code is transformed into source-code graphs and defines type safety for graphs. In Section 4 we present the graph program TTC for transforming a source-code graph into a message graph. In Section 5 we give proof ideas for how to show that TTC terminates and that the error messages generated by it correctly indicate that the input is not type safe. We conclude our results in Section 6. A long version of this paper, with complete proofs and more examples, is available as a technical report, see [AP07].

2 Graph Programs

In this section, we review conditions, rules, and programs, in the sense of [HP05] and [HP01]. In the following, we consider the category of directed labeled graphs with all injective graph morphisms. Labels distinguish different types of nodes and edges and directions model relationships between nodes. We use the standard definition of labeled graphs and labeled graph morphisms, see [EEPT06] or [AP07] for details. For expressing properties on graphs we use so-called graph conditions. The definition is based on graph morphisms.

Definition 1 (Graph conditions) A *graph condition* over an object P is of the form $\exists a$ or $\exists(a,c)$, where $a\colon P \to C$ is a morphism and c is a condition over C. Moreover, Boolean formulas over conditions (over P) are conditions (over P). A morphism $p\colon P \to G$ *satisfies* a condition $\exists a$ ($\exists(a,c)$) over P if there exists an injective morphism $q\colon C \to G$ with $q \circ a = p$ (satisfying c). An object G *satisfies* a condition $\exists a$ ($\exists(a,c)$) if all injective morphisms $p\colon P \to G$ satisfy the

condition. The satisfaction of conditions over P by objects or morphisms with domain P is extended to Boolean formulas over conditions in the usual way. We write $p \models c$ ($G \models c$) to denote that morphism p (object G) satisfies c. In the context of rules, conditions are called *application conditions*.

We rewrite graphs with rules in the double-pushout approach [EEPT06]. Application conditions specify the applicability of a rule by restricting the matching morphism.

Definition 2 (Rules) A *plain rule* $p = \langle L \leftarrow K \rightarrow R \rangle$ consists of two injective morphisms with a common domain K. L is the rule's left-hand side, and R its right-hand side. A *left application condition* ac for p is a condition over L. A *rule* $\hat{p} = \langle p, \mathrm{ac} \rangle$ consists of a plain rule p and an application condition ac for p.

$$\begin{array}{ccccc} L & \longleftarrow & K & \longrightarrow & R \\ m\downarrow & (1) & \downarrow & (2) & \downarrow m^* \\ G & \longleftarrow & D & \longrightarrow & H \end{array}$$

Given a plain rule p and injective morphism $K \rightarrow D$, a *direct derivation* consists of two pushouts (1) and (2) where the *match* m and *comatch* m^* are required to be injective. We write a direct derivation $G \Rightarrow_{p,m,m^*} H$. Given a graph G together with an injective match $m : L \rightarrow G$, the direct derivation $G \Rightarrow_{p,m,m^*} H$ can informally be described as: H is obtained by deleting the image $m(L-K)$ from G and adding $R-K$. Given a rule $\hat{p} = \langle p, \mathrm{ac} \rangle$ and a morphism $K \rightarrow D$, there is a *direct derivation* $G \Rightarrow_{\hat{p},m,m^*} H$, if $G \Rightarrow_{p,m,m^*} H$, and $m \models \mathrm{ac}$.

We now define graph programs as introduced in [HP01].

Definition 3 (Graph programs) Every rule p is a *(graph) program*. Every finite set $\mathscr{S}$ of programs is a program. If P and Q are programs, then $(P;Q)$, P^* and $P\downarrow$ are programs. The *semantics* of a program P is a binary relation $[\![P]\!] \subseteq \mathscr{G}_\mathscr{C} \times \mathscr{G}_\mathscr{C}$ on graphs: (1) For every rule p, $[\![p]\!] = \{\langle G,H \rangle \mid G \Rightarrow_p H\}$. (2) For a finite set $\mathscr{S}$ of programs, $[\![\mathscr{S}]\!] = \cup_{P \in \mathscr{S}} [\![P]\!]$. (3) For programs P and Q, $[\![(P;Q)]\!] = [\![Q]\!] \circ [\![P]\!]$, $[\![P^*]\!] = [\![P]\!]^*$ and $[\![P\downarrow]\!] = \{\langle G,H \rangle \in [\![P]\!]^* \mid \neg\exists M. \langle H,M \rangle \in [\![P]\!]\}$.

Programs according to (1) are *elementary* and a program according to (2) describes the *nondeterministic choice* of a program. The program $(P;Q)$ is the *sequential composition* of P and Q. P^* is the *reflexive, transitive closure* of P, and $P\downarrow$ the *iteration* of P as long as possible. Programs of the form $(P;(Q;R))$ and $((P;Q);R)$ have the same semantics and are considered as equal; by convention, both can be written as $P;Q;R$. We use $P\downarrow^+$ as a shortening of $P;P\downarrow$.

Notation. When the label of an element is either a or b we use the notation $a|b$. $\langle L \Rightarrow R \rangle$ is used a short form of $\langle L \leftarrow K \rightarrow R \rangle$, where K consists of the elements common to L and R. For an application condition with morphism $a\colon P \rightarrow C$, we omit P as it can be inferred from the left-hand side. We omit the application condition if it is satisfied by any match. To distinguish nodes with the same label, we sometimes add an identifier in the form of "label:id". We use source-code fragments as identifiers and therefore print them in a fixed-width font.

3 From Source Code to Source-Code Graphs

In this section, we introduce source-code graphs, the input for our type-checking program, and informally describe how source code is transformed into such graphs. A source-code graph is a graph representation of template definitions, declarations, and expression's specializations. The type signature of every declared method, function, and operator in a template definition is represented in the graph by an overload forest, see below. Expressions that involve parameterized types are represented in the graph by expression paths, explained shortly. For every declaration, the above mentioned graph representations are copied and the parameterized types are replaced by the actual types provided by the declaration.

The basic elements of our source-code graphs are nodes for template definitions, declarations, data types, names, type signatures, and expression trees. For quick reference, node and edge labels together with a short explanation are listed below. Note that a visualization of an edge as dashed or solid denotes a difference in labels.

Nodes				Edges	
D	declaration	p	actual parameter	$=$	comparison
E	error message	t	data type	c	cast without precision loss
ex	(sub)expression	T	template definition	d	deduced type of expression
ol	overloaded operator	W	warning message	p	parameter
op	operator name			pc	cast with precision loss
				r	return type
				R	recovered comparison

Template definitions are represented by T-nodes. Two declarations are equivalent if they are based on the same template and have equal lists of template parameters. Each class of equivalent declarations are represented by a D-node and denotes a future specialization. Each D-node has an incoming edge from the T-node representing the template definition the declaration means to instantiate. Possible template parameters are represented by t-nodes. Such parameters include classes, structures, fundamental types, and constants. Operator-, function- and method names are represented by op-nodes. In [AP07] we show a graph program that generates source-code graphs.

Example 1 *Consider the source code with two class-type templates, line 1 and 18, in Figure 3. The principal data type is the* `icon` *structure with a template parameter for its resolution type. The* `resolution` *structure has a constant template parameter for a (quadratic) resolution. For the two unique declarations in* `main`*, name lookup and type checking is needed for the expressions on lines 3, 20, 23, 24, 25, 40, and 41. In Section 4 we will show how the graph program* `TTC` *reports the type clash in the expression on line 41. Note that Figure 4 shows the source-code graph of the source code example from Figure 3 where the above mentioned lines are represented as expression paths. That source-code graph also shows the overload trees for the operators on line 3 and 22. For completeness, some overload paths representing used operations native to C++ are also included, e.g. comparison ($<$) of integers.*

We will now introduce some necessary graph-theoretic notions. In particular, we introduce and use expression paths and overload trees to define type safety for graphs. Expression paths rep-

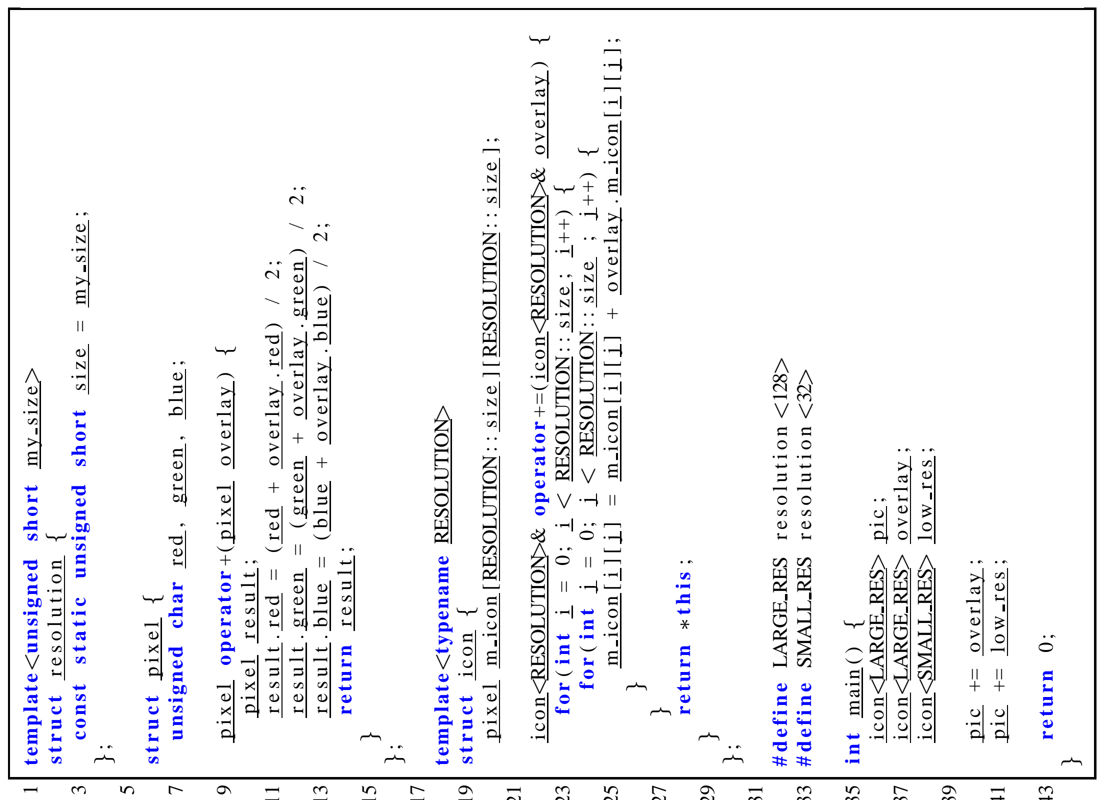


Figure 3: Two class-type templates.

resent the type information from an expression tree and are modeled by ex- and p-nodes. The root of the tree becomes an ex-node and has an incoming edge from the D-node that represents the specialization in which it will exist. Each ex-node has an edge to the op-node denoting the operation's name. We allow for operators with an arbitrary number of operands, so the children of the root in an expression tree are modeled by a path of p-nodes. If such a child is a subexpression, then the corresponding p-node has an edge to a new expression node. If it is not, then it denotes a type and its p-node has an edge to the t|D-node denoting that type.

Definition 4 (Expression paths) Given a graph G and a natural number i, an *i-expression path* in G is a path $ex\, p_0 \dots p_i$, where the head, ex, is an ex-node and $p_0, \dots, p_i$ are p-nodes such that, from every node p_k, $0 \leq k < i$, the only edge to another p-node is to p_{k+1}.

Example 2 *Figure 5 shows (to the left) an expression tree denoting line 41 in Example 1 together with its corresponding* 2*-expression path (to the right).*

We represent the type signatures of methods with overload forests, trees and paths. A method named `method` declared in class type `class` with n parameters is represented by a path of $n+2$ ol-nodes. The head of that path has an edge to the op-node representing the name `method`

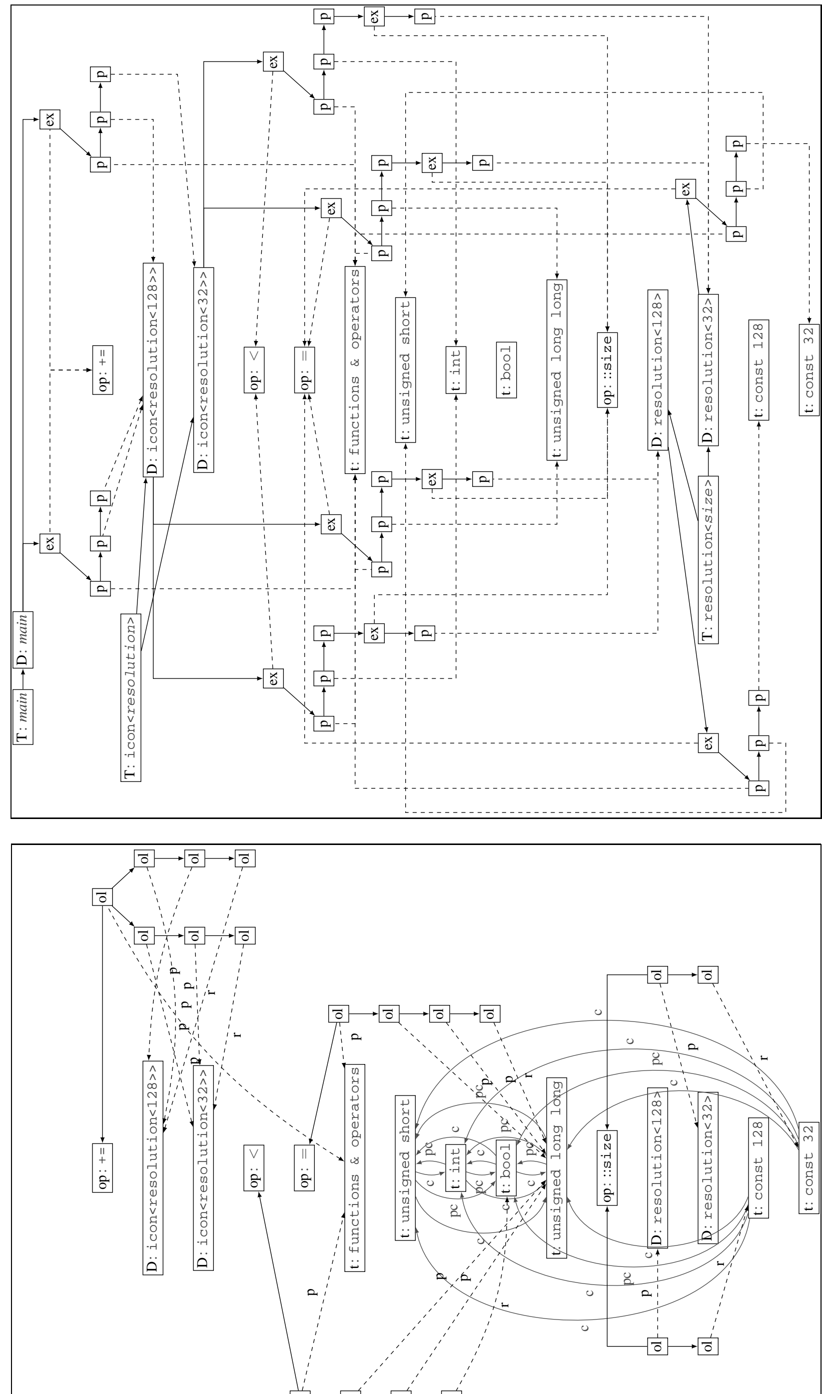


Figure 4: A source-code graph split in two for simpler representation, but note that the two subgraphs are not disjoint: the nodes with identical ids (see the center column) are identified.

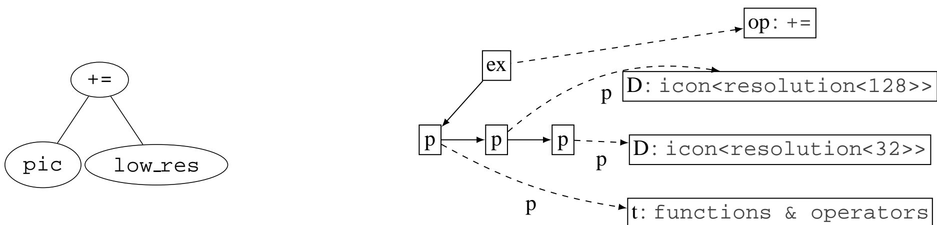


Figure 5: Expression paths represent expression trees.

and another edge to the t|D-node representing `class`. The ol-node at position k $(2 \leq k \leq n+1)$ in the path has an edge to the node denoting the type of the variable at parameter position $k-1$. The last ol-node in the path has an edge to the t|D-node denoting the return type of `method`. Functions are modeled as methods but as declared in a special class with a name not allowed in the source language, e.g. `functions & operators`. Operator overloading is modeled as functions. In the following operators, methods, and functions are collectively referred to as *operators*.

Definition 5 (Overload forest) A graph G contains an *overload forest* iff all ol-nodes in G are part of exactly one overload tree and there exist no pair of overload trees with equivalent roots, see below. An *overload tree* is a maximal connected subgraph T in G, consisting of only ol-nodes. T is maximal in the sense that, if an ol-node in T has an edge to an ol-node, then that node is also in T. Furthermore, T must have a tree structure, i.e. no cycles and every node has one parent, except for the root. For nodes in T the following holds for them in G: (1) Each internal (leaf) node has exactly one p-edge (r-edge) to a t|D-node, one edge from its parent, and no other incoming edges. (2) The root of T has an additional edge to an op-node. (3) No two siblings have a p-edge to the same t|D-node. (4) Every node has at most one child that is a leaf. Requirements 3 and 4 are necessary to prevent ambiguous type signatures. Two roots are *equivalent* iff there exists an op-node o and t|D-node t, such that both roots have edges to o and t. An *i-overload path* $o_0 \ldots o_{i+1}$ is a path in T from the root to a leaf. The t|D-node to which an r-edge exist from o_{i+1} is called the *return type* of the i-overload path.

Example 3 *The overload tree in Figure 6 has two* 2*-overload paths, representing the type signatures of two overloaded operators. The tree represents the operator template on line 22 and the two paths are generated for the two declarations on line 40 and 41 in Figure 3.*

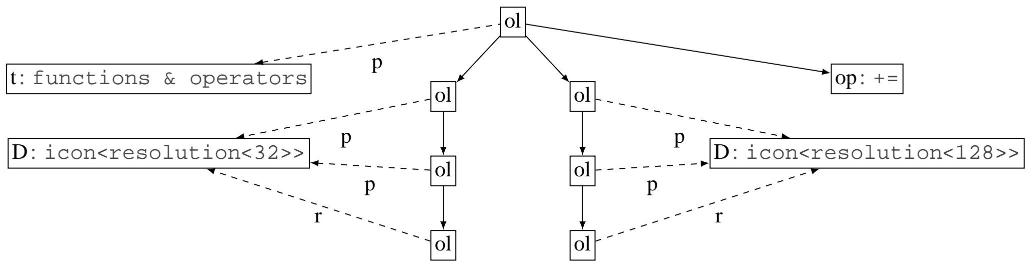


Figure 6: Two overload paths.

Remark 1. The size of a source-code graph grows linearly with the size of the source code that would be output by the template instantiation mechanism in a C++ compiler. An expression or declared operator that exists in such source code is represented only by a single expression path or overload path, respectively.

The main property in this paper is the one of type safety. A graph is type safe if for every expression path, there exists an overload path with the same type signature. This property corresponds to type safety for template instantiation in C++ programs, where every generated expression must be type checked against the existing and generated operators.

Definition 6 (Type-safe graphs) A graph G is *type safe* iff it contains an overload forest and is i-type safe for all natural numbers i. G is i*-type safe* iff every i-expression path in G is type safe. An i-overload path $o_0 \dots o_{i+1}$ makes the i-expression path $ex\, p_0 \dots p_i$ *type safe* iff:

1. There exists an op-node op and two edges: one from ex to op, the other from o_0 to op.
2. For all k, where $0 \le k \le i$, there exists a t|D-node t and two edges, one from o_k to t and the other is from p_k to either t or the head of a type safe j-expression path such that t is the deduced type of the that j-expression path.

The *deduced type* of the i-expression path is the t|D-node with an incoming r-edge from o_{i+1}.

It is easy to see that no overload path from Figure 6 makes the expression path in Figure 5 type safe.

4 The Type-Checking Program

This section describes the graph program `TTC` which performs the name lookup and type checks source-code graphs. The section also shows how message graphs are interpreted with graph conditions.

A schematic of how the subprograms of `TTC` interact is shown in Figure 7. Intuitively, `TTC` works as follows: The input is a source-code graph, each expression path is marked by `MarkExpression`, and `Compare` moves this marker through the path until it reaches the tail or a type clash. At a type clash, `Recover` either solves it or generates an error message. For markers at a tail, `Resolve` finds the deduced type of the expression path (i.e. it resolves the type of a subexpression). This chain is then iterated as long as new work is made available by `Recover` and `Resolve` for `Compare`. The yield of `TTC` is a message graph. Programs and rules are described in more detail below.

Definition 7 (`TTC`) Let the graph program `TTC` $=$ `MarkExpression`$\downarrow$;`TypeCheck`$\downarrow$ with the subprograms:

$$\begin{aligned}
\texttt{TypeCheck} &= \texttt{Compare}\downarrow^{+};\texttt{Recover}\downarrow;\texttt{AfterRecover}\downarrow;\texttt{Resolve}\downarrow\\
\texttt{Compare} &= \{\ \texttt{Lookup},\texttt{CompareNext},\texttt{FindType}\ \}\\
\texttt{Recover} &= \{\texttt{Cast},\texttt{Warning},\texttt{Error}\}\\
\texttt{Resolve} &= \left\{\begin{array}{c}\texttt{ResolveSubexpression},\\ \texttt{ResolveExpression1},\texttt{ResolveExpression2}\end{array}\right\}
\end{aligned}$$

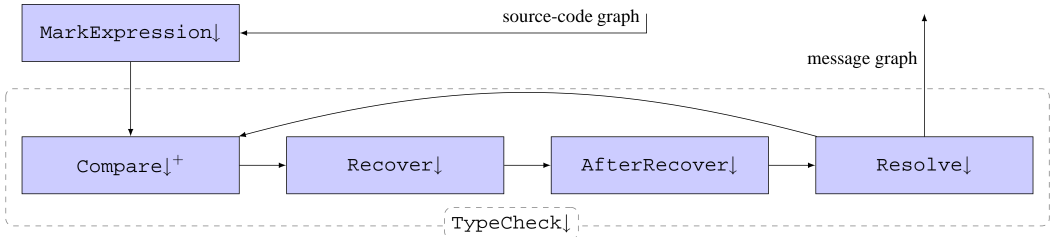


Figure 7: Structure of TTC.

MarkExpression↓ is executed before the actual type checking starts and markss each ex-node with an =-edge to show where the evaluation starts. To avoid duplicate evaluation, a loop is also placed at the ex-node and checked for in the application condition.

MarkExpression: ⟨⟨op ← ex ⇒ op $\overset{=}{\leftarrow}$ ex ↺⟩, ¬∃ (op ← ex ↺)⟩

Compare↓$^+$ consists of the rules Lookup, CompareNext, and FindType, see Figure 8. They move the =-edge generated by MarkExpression through the expression path as long as a matching overload path can be found. The program halts when it completes this matching or encounters a problem: that no overload path makes this expression path type safe or that this node in the path depends on the deduced type of a subexpression. The rule Lookup finds the overload tree for a marked expression's name. CompareNext matches the type signature of the expression path to an overload path parameter by parameter. The rule FindType is applied at the tail of the expression path and deduces the expression's type via the matched overload path's return type. The rule's application condition makes sure that this is actually the tail of the expression path.

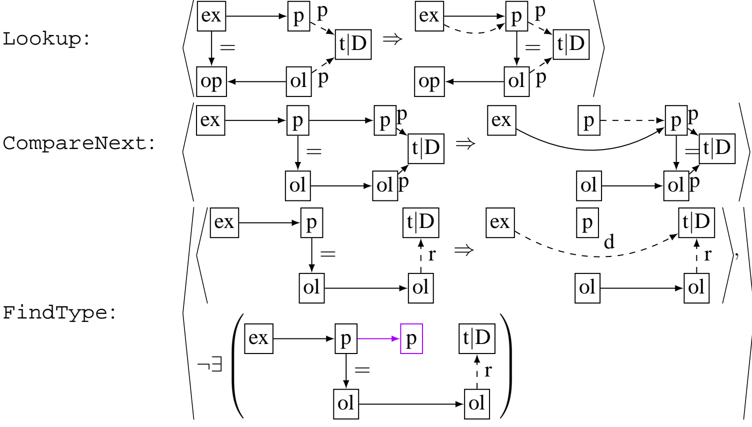


Figure 8: Rules in the program Compare.

`Recover↓` consists of the rules Figure 9 and tries to find alternative overload paths by implicit type casts. The rule `Cast` finds a type cast that causes no loss of precision. `Warning` works as `Cast` but uses a cast with a possible loss of precision. For this we generate a warning, a W-node with three outgoing edges: the location of the problem, the original type, and the cast type. The application condition make sure that `Cast` is preferred. The rule `Error` is applied when there is no solution by implicit type casts. An error node is therefore generated. It has three outgoing edges, to the p-node where it occurred, the faulting type, and a suggested type. The application condition limits `Error` from being applied where `Cast` or `Warning` could be applied instead.

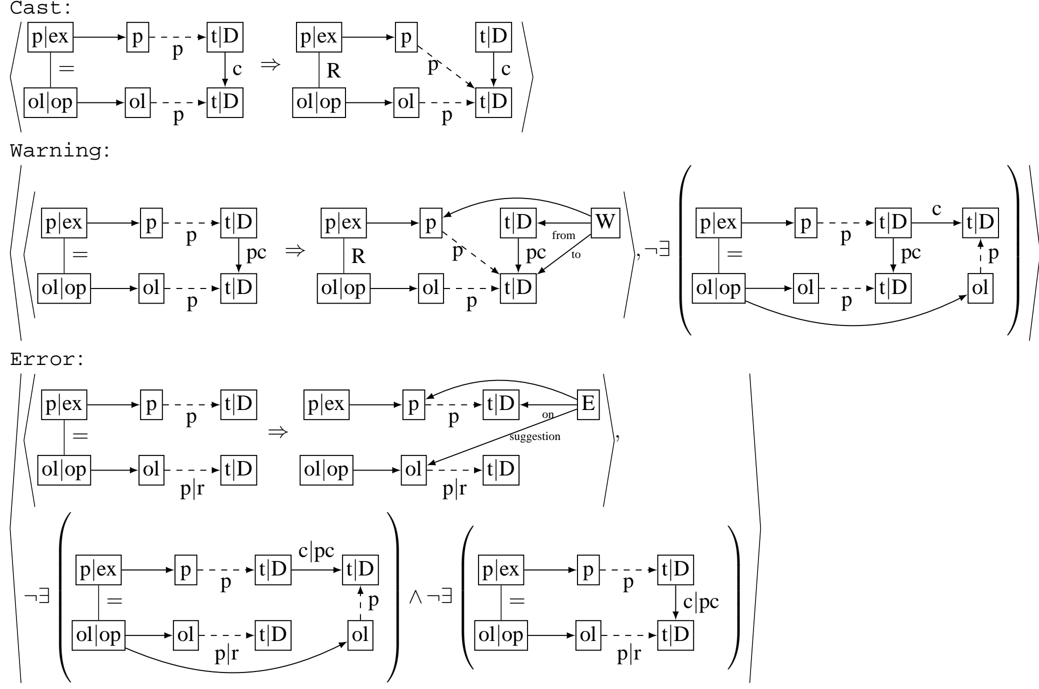


Figure 9: Rules for the program `Recover`.

`AfterRecover↓` performs some cleanup work for `Recover↓`, generated R-edges are reset to =-edges.

`AfterRecover:` ⟨ ex|p --R--> op|ol ⇒ ex|p --=--> op|ol ⟩

`Resolve↓` consists of three rules, as shown in Figure 10: `ResolveSubexpression` replaces a subexpression with its deduced return type. `ResolveExpression1` marks an expression as evaluated with a dashed edge. `ResolveExpression2` does the same in the special case when the return type is the same as the specialization in where the expression occurs.

Example 4 *The graph in Figure 11 is the yield of* `TTC` *applied to the overload tree from Figure 6 and the expression path from Figure 5. See [AP07] for more examples.*

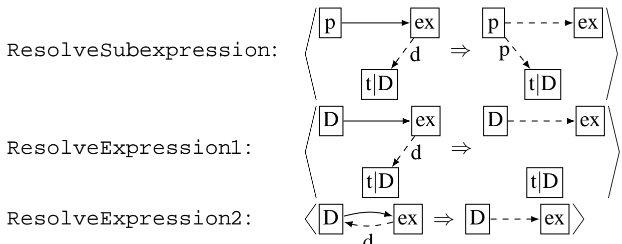


Figure 10: Rules in `Resolve`.

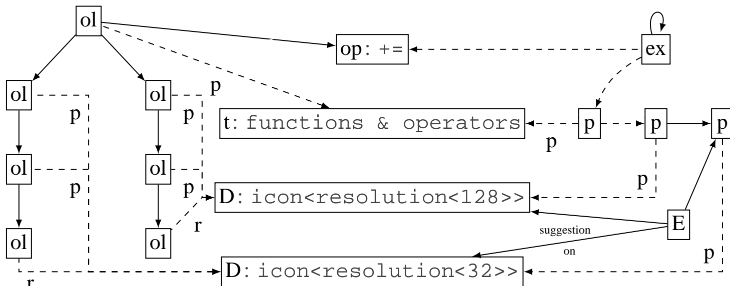


Figure 11: Portion of a message graph.

Remark 2. After the termination of TTC, graph conditions can help to interpret the message graph. A graph is an *error (warning)* graph iff it satisfies the condition $\exists(\emptyset \to \boxed{\text{E}})$ $(\exists(\emptyset \to \boxed{\text{W}}))$. A particular declaration can safely be instantiated if its corresponding D-node satisfies the condition $\neg\exists(\boxed{\text{D}} \to \boxed{\text{D}} \longrightarrow \boxed{\text{ex}})$. If that condition is not satisfied, then one of its expressions could not be resolved and the programmer must take appropriate actions. A message graph will contain all the detected errors for the corresponding source code. Graph conditions can therefore help programmers to locate the areas of the graph that contain such errors. An implementation of this approach should be able to highlight the areas containing the errors.

Remark 3. The size of the resulting message graph will not grow more than linearly with the size of the corresponding source-code graph. This is so since every expression tree can at most be marked by one error message. For size of source-code graphs, see Remark 1

5 Correctness and Termination

We now define correctness with respect to errors and termination for graph programs. We give the ideas for proving that TTC terminates and is correct w.r.t. errors.

Definition 8 (Correctness and Completeness) A graph program P is *correct with respect to errors* if for every pair $\langle G, H\rangle \in [\![P]\!]$, H is an error graph implies G is not a type-safe source-code graph. If the converse of the implication holds, we say that P is *complete w.r.t. errors*.

Theorem 1 (Correctness) *The graph program* `TTC` *is correct with respect to errors.*

Proof idea. Errors are only generated by `Recover` and consumes an =-edge that should have been consumed by `Compare`$\downarrow^+$, given that `TTC` were initially dealing with a type-safe source-code graph. A complete proof is given in [AP07].

Fact 1. The graph program `TTC` is not complete with respect to errors. E.g. `Recover` uses implicit type casts, and thereby avoids generating errors for some non-type-safe graphs. It has not yet been investigated whether or not other counterexamples exist.

Definition 9 (Termination) *Termination* of a graph program is defined inductively on the structure of programs: (1) Every rule p is terminating. (2) For a finite set $\mathscr{S}$ of terminating programs, $\mathscr{S}$ is a terminating program. (3) For terminating programs P and Q, $(P;Q)$ is terminating. Moreover, P^* and $P\downarrow$ is terminating if for every graph G, there is no infinite chain of derivations $G \Rightarrow_P G_1 \Rightarrow_P \ldots$ where $\Rightarrow_P$ denotes the binary relation $[\![P]\!]$ on graphs.

Theorem 2 (Termination) *The graph program* `TTC` *is terminating.*

Proof idea. `Compare` is applied at least once for every iteration of `TypeCheck` and consumes solid edges that are not generated by the other subprograms. A complete proof is given in [AP07].

6 Conclusions

We considered the template instantiation mechanism in C++ and showed how to write visual rules for type checking and error message generation. We informally described how source code was transformed into source-code graphs and defined type safety for graphs. We transformed source-code graphs into message graphs, a transformation given by the graph program `TTC` which type checked source-code graphs. The program automatically corrected some errors by implicit type casts. It emitted error messages for type clashes that it could not correct. Proof ideas were given for termination and correctness w.r.t. errors.
Further topics include:

1. Analysis of source-code graphs by generalized graph conditions. Graph properties like “There exists a warning or error node” can be expressed by graph conditions in the sense of [HP05]. It would be interesting to generalize graph conditions so more complex graph properties like “The graph is type safe” becomes expressible.

2. Debugging and a transformation from message graphs to source code. The error messages generated by `TTC` contained suggestions for remedies. In the double-pushout approach to graph transformation, a central property is the existence of an inverse rule, that when applied reverses the rewrite step of the rule [EEPT06]. In this way, the inverse rule allows for back tracking to a previous graph which can be manipulated to experiment with suggested remedies. The changes are logged in the output graph (message/change graph) and used by a source-code transformer to update the source code.

3. Implementation of the approach. This would include a formalization of the transformation from source code to source-code graphs and an extension of the set of considered template features.

Acknowledgements: This work is supported by the German Research Foundation (DFG) under grant no. HA 2936/2 (Development of Correct Graph Transformation Systems). We thank Annegret Habel for constructive suggestions that improved the paper.

Bibliography

[AG04] D. Abrahams, A. Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond*. Addison-Wesley Professional, 2004.

[AP07] K. Azab, K.-H. Pennemann. Type Checking C++ Template Instantiation (long version). Technical report 04/07, University of Oldenburg, 2007. Available at http://formale-sprachen.informatik.uni-oldenburg.de/%7Eskript/fs-pub/templates_long.pdf.

[CE00] K. Czarnecki, U. W. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, 2006.

[GPG04] T. Gschwind, M. Pinzger, H. Gall. TUAnalyzer–Analyzing Templates in C++ Code. In *Proc. of WCRE'04*. Pp. 48–57. IEEE Computer Society, 2004.

[HP01] A. Habel, D. Plump. Computational Completeness of Programming Languages Based on Graph Transformation. LNCS 2030, pp. 230–245. Springer, 2001.

[HP05] A. Habel, K.-H. Pennemann. Nested Constraints and Application Conditions for High-Level Structures. LNCS 3393, pp. 293–308. Springer, 2005.

[Jos99] N. M. Josuttis. *The C++ Standard Library: a tutorial and reference*. Addison-Wesley, 1999.

[PMS06] Z. Porkoláb, J. Mihalicza, Á. Sipos. Debugging C++ template metaprograms. In *Proc. of GPCE'06*. Pp. 255–264. ACM, 2006.

[PS04] D. Plump, S. Steinert. Towards Graph Programs for Graph Algorithms. LNCS 3256, pp. 128–143. Springer, 2004.

[Str00] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, 3rd edition, 2000.

[Str04] B. Stroustrup. Abstraction and the C++ Machine Model. LNCS 3605, pp. 1–13. Springer, 2004.

[Vel98] T. L. Veldhuizen. Arrays in Blitz++. LNCS, pp. 223–230. Springer, 1998.

[VJ02] D. Vandevoorde, N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, 2002.

A Graph-Based Type Representation for Objects

Cong-Cong Xing[1]

[1] cmps-cx@nicholls.edu
Department of Mathematics and Computer Science
Nicholls State University
Thibodaux, LA, USA

Abstract: Subtyping and inheritance are two major issues in the research and development of object-oriented languages, which have been traditionally studied along the lines of typed calculi where types are represented as a combination texts and symbols. Two aspects that are closely related to subtyping and inheritance – method interdependency, and self type and recursive object type – have either been overlooked or not received sufficient/satisfactory treatments. In this paper, we propose a graph-based notation for object types and investigate the subtyping and inheritance issues under this new framework. Specifically, we (1) identity the problems that have motivated this paper; (2) propose an extension to Abadi-Cardelli's ς-calculus towards fixing the problems; (3) present definitions of object type graphs followed by examples; (4) define subtyping and inheritance using object type graphs; (5) show how the problems can be easily resolved under object type graphs; and (6) summarize the contributions of this paper.

Keywords: Object type, graph transformation

1 Introduction

As pointed out by Markku Sakkinen in [Sak05], although in recent years the emphasis of the research and development in object-oriented programming (OOP) has shifted from programming languages (themselves) to larger entities such as components, environments, and manipulating tools, it does not mean that the existing object-oriented languages are perfect and no improvement is needed. In particular, *typing* is still a critical issue and a problem-prone area in the formal study of object-oriented languages, especially when type-related subjects, such as subtyping and inheritance, are considered.

One aspect related to subtyping is object method interdependencies: the invocation relationship among methods. The failure of keeping track of this invocation structure in object types can cause elusive programming errors which will inevitably occur, undermine the program reliability, and burden the program verification. One aspect related to inheritance is self type vs. recursive object type: which one is the *true* type of the self variable (in the context of inheritance). The failure of not distinguishing these two types sufficiently can lead to some well-known fundamental problems. While the former aspect has been overlooked in the literature, the latter has not received sufficient attentions and/or satisfactory treatments, in either theoretical studies (e.g., [AC96, FHM94, LC96, Liq98, BL95]) or main stream practice (e.g., Java and C++) of OOP. In the next section, we present concrete examples to illustrate this point.

2 Motivations

We present two problems that have motivated the writing of this paper.

2.1 Method Interdependency

We call a rectangle *free* if its two sides (height and width) are independent, *constrained* otherwise. In conventional type systems, the type of a free rectangle and the type of a constrained rectangle are not distinguished. We show, in this subsection, that this type confusion opens the door to let the different semantics of free rectangles and constrained rectangles be mixed, which is serious enough to be able to cause a program *not* to perform to its specification and thus weakens its reliability.

Using the first-order ς-calculus notation [AC96], we can construct a free rectangle *fRect*, a constrained rectangle *cRect*, and their types *FR*, *CR* as follows:

$$FR \stackrel{\text{def}}{=} \mu(Self)\begin{bmatrix} h : int \\ w : int \\ mvh : int \to Self \\ mvw : int \to Self \\ geth : int \\ getw : int \end{bmatrix}, \quad fRect \stackrel{\text{def}}{=} \varsigma(s\!:\!FR)\begin{bmatrix} h = 1 \\ w = 2 \\ mvh = \lambda(i\!:\!int)(s.h \Leftarrow s.h + i) \\ mvw = \lambda(i\!:\!int)(s.w \Leftarrow s.w + i) \\ geth = s.h \\ getw = s.w \end{bmatrix},$$

$$CR \stackrel{\text{def}}{=} \mu(Self)\begin{bmatrix} h : int \\ w : int \\ mvh : int \to Self \\ mvw : int \to Self \\ geth : int \\ getw : int \end{bmatrix}, \quad cRect \stackrel{\text{def}}{=} \varsigma(s\!:\!CR)\begin{bmatrix} h = 1 \\ w = 2(s.h) \\ mvh = \lambda(i\!:\!int)(s.h \Leftarrow s.h + i) \\ mvw = \lambda(i\!:\!int)(s.w \Leftarrow s.w + i) \\ geth = s.h \\ getw = s.w \end{bmatrix},$$

where $\Leftarrow$ is the field update operation, and the intentions of methods in these two rectangles are obvious. Note that in *fRect*, the height (h) and the width (w) are independent, whereas in *cRect*, the width *depends on* the height ($w = 2(s.h)$). Also note that $FR = CR$, that is, the types of these two rectangles are confused (in conventional type systems).

Now, suppose we would like to have a function with the following specification (contract):

> *This function takes a rectangle and then doubles both its height and its width.*

With little effort, such a function can be written as:

$$ds \stackrel{\text{def}}{=} \lambda(r : FR)(r.mvh(r.geth)).mvw(r.getw).$$

It is easy to check that *ds* will double its argument's both sides when taking a free rectangle as argument. However, when *ds* takes a constrained rectangle as argument, for example $ds(cRect)$ (due to the fact $CR = FR$, *cRect* will type-check), it will fail to do so, as it is supposed to (by the specification). In detail,

$$ds(cRect) = (cRect.mvh(cRect.geth)).mvw(cRect.getw)$$
$$= \varsigma(s\!:\!CR)\begin{bmatrix} h = 2 \\ w = 6 \\ \ldots no\ change \ldots \end{bmatrix}.$$

Clearly, the height of *cRect* is doubled, but its width is *tripled* (not doubled)! The reason for this is the interdependency between the height and the width in *cRect*: when the height of *cRect* is changed to 2, its width is *implicitly* changed to 4 due to the width's dependence on the height.

Considering that the widely-agreed notion of program reliability refers to (e.g. [Seb07]) "program performs to its specification under all circumstances" and that the fact that ds does not live up to its specification when taking *cRect* as its argument, we argue that the reliability of ds, in the environment of conventional type systems, is substantially low. Furthermore, such elusive computation fault may be hard-to-detect when ds is embedded in large software systems. To resolve this problem effectively, ds should be written in such a way that it only takes free rectangles, that is, $ds(cRect)$ should be caught by the type checker. This observation calls for the separation of the type of free rectangles from that of constrained ones.

2.2 Self Type and Recursive Object Type

The notion of self type is coined to describe the type of the self variable in an object, especially when the object contains a self returning method. Then the question is: What is the (semantics of) self type? Is it just the (recursive) object type or something else? For example, using the notation of ς-calculus again, an object which consists of two methods, one returning the constant 1 and one returning the hosting object itself can be coded as $a \stackrel{\text{def}}{=} [l_1 = 1,\ l_2 = \varsigma(s\!:\!X)s]$, where s is the *self* variable and X is the type of s – the self type. How do we interpret X? One "natural" way is that X is just the object type itself (recursively defined), that is, $X = \mu(Y)[l_1 : int,\ l_2 : Y]$. As this interpretation works to some extent but runs into substantial problems (see, e.g., [AC96] for details), other explanations of the self type have been sought. For example, the second-order self quantifier [AC96] and the MyType [Bru94] are proposed. Nevertheless, no mater how self type is interpreted, an object type has always been managed to be a subtype of the associated self type. This setup, combined with inheritance and dynamic dispatch of methods, leads to the well-known "method-not-found" error as illustrated below (adapted from [Bru94]).

$$PT \stackrel{\text{def}}{=} ObjectType(MyType)\{x : int, eq : MyType \to Bool\}$$
$$CPT \stackrel{\text{def}}{=} ObjectType(MyType)\{x : int, c : color,\ eq : MyType \to Bool\}$$
$$pt_0 \stackrel{\text{def}}{=} object(self : MyType)\{x = 0, eq = fun(p : MyType)(p.x = self.x)\}$$
$$pt \stackrel{\text{def}}{=} object(self : MyType)\{x = 1, eq = fun(p : MyType)(p.x = self.x)\}$$
$$cpt \stackrel{\text{def}}{=} inherited\ from\ pt\ with\ \{c = red,\ eq = fun(p : MyType)[(p.x = self.x) \wedge (p.c = self.c)]\}$$
$$F \stackrel{\text{def}}{=} fun(p : PT)(p.eq(pt_0))$$

Given these definitions, it is easy to check that $pt_0 : PT$, $pt : PT$, and $cpt : CPT$. Note that in the definition of F, we actually have assumed (as [Bru94] does) that the type of pt_0, PT,

is a subtype of the self type associated with PT, $MyType$ in this case, so that $p.eq(pt_0)$ type-checks. Now, if inheritance implies subtyping (as we have been practicing in C++ and Java), then $cpt : CPT <: PT$ and $F(cpt)$ will type check. However, $F(cpt)$ will crash and produce a "method-not-found" error because $cpt.eq(pt_0)$ expects its argument, pt_0, to have a color field and uses that color field in the body of eq of cpt, but pt_0 does not have the color field.

Traditionally, it is this kind of problem that has prompted us to claim that "inheritance is not subtyping" [CHC90]. However, "inheritance implies subtyping" is a strongly desirable property in OOP. Without it, the software hierarchy build through inheritance will be much less useful since in this case a subobject (object from a subclass) cannot be regarded as having the same type with its superobject (object from the superclass), and cannot use any existing programs that have been written for superobjects. Program reusability will thus be greatly reduced. Towards keeping this hierarchy useful and resolving the method-not-found problem at the same time, we propose that an object type should not only be treated differently from its associated self type, but not be regarded as a subtype of its associated self type either.

3 Enhancing Object Types

In order to address the problems outlined in the previous section, we extend Abadi-Cardelli's ς-calculus by adding a mechanism called links that capture the method interdependencies in objects, and by distinguishing (recursive) object types from their associated self types. The terms (M) and types (σ) of this extended calculus are as follows.

$$
\begin{array}{lcl}
M & ::= & x \mid \lambda(x{:}\sigma).M \mid M_1M_2 \mid M.l \mid M.l \Leftarrow \varsigma(x{:}\mathscr{S}(A))M' \mid [l_i = \varsigma(x{:}\mathscr{S}(A))M_i]_{i=1}^{n} \\
\sigma & ::= & \kappa \mid t \mid \sigma_1 \rightarrow \sigma_2 \mid \mu(t)\sigma \mid A \mid \mathscr{S}(A) \\
A & ::= & \iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^{n} \quad L_i \subseteq \{l_1,\ldots,l_n\} \text{ for each } i
\end{array}
$$

x, $\lambda(x{:}\sigma).M$, and M_1M_2 are the standard λ-terms. $[l_i = \varsigma(x{:}\mathscr{S}(A))M_i]_{i=1}^{n}$ represents an object consisting of n methods, with names l_i and bodies M_i for each i. ς is the self-binder. $M.l$ means the invocation of method l in M. $M.l \Leftarrow \varsigma(x{:}\mathscr{S}(A))M'$ is the updating operation which evaluates to an object obtained by replacing method l in M by M'.

κ, t, $\sigma_1 \rightarrow \sigma_2$, and $\mu(t)\sigma$ are ground types, type variables, function types, and recursive types respectively. Object types are represented by $\iota(t)[l_i(L_i){:}\sigma_i]_{i=1}^{n}$ where each method l_i has type σ_i, and L_i is the set of *links* of l_i (defined below). ι is the self-type binder. An alternative way to represent self type is $\mathscr{S}(A)$ which denotes the self type associated with the object type A. The two notations are related by $A = \iota(t)[l_i(L_i){:}\sigma_i(t)]_{i=1}^{n} = [l_i(L_i){:}\sigma_i(\mathscr{S}(A))]_{i=1}^{n}$. Terms that can be of a self type are restricted to self (variable) or a modified self (for the sake of self variable specialization during inheritance).

Definition 1 Given an object $[l_i = \varsigma(s{:}\mathscr{S}(A))M_i]_{i=1}^{n}$. The only terms that are of type $\mathscr{S}(A)$ are s or $s.l_i \Leftarrow \varsigma(s{:}\mathscr{S}(A))M$ for some M.

The set of links, which is a part of the newly proposed object types, is defined as follows.

Definition 2 (*Links*) Given an object $a = [l_i = \varsigma(s:\mathscr{S}(A))M_i]_{i=1}^n$, (1) l_i is said to be ***dependent*** on $l_j(i \neq j)$ if there exists a M such that $a.l_i$ (or $a.l_i(\alpha)$ for some appropriate argument list α) and $(a.l_j \Leftarrow \varsigma(s:\mathscr{S}(A))M).l_i$ (or $(a.l_j \Leftarrow \varsigma(s:\mathscr{S}(A))M).l_i(\alpha)$) evaluate to different values; (2) l_i is said to be ***directly dependent*** on $l_j(i \neq j)$ if (a) l_i is dependent on l_j, and (b) if all such $l_k(i \neq k, j \neq k)$ where l_i is dependent on l_k and l_k is dependent on l_j, are removed from a, l_i is still dependent on l_j; (3)The set of ***links*** of l_i in object a (or equivalently, of M_i with respect to object a), denoted by $L_a(l_i)$ (or equivalently, by $L_a(M_i)$), contains exactly all such l_j on which l_i is directly dependent.

4 Object Type Graphs

The notion of links introduces new structures into object types. Object types are thus enriched but also become more complicated. To effectively analyze and reason about the structure of the new object types, we present a graph-based representation for object types – object type graphs (OTG).

4.1 Definitions

Definition 3 *(Directed Colored Graph)* A ***directed colored graph*** G is a 6-tuple (G_N, G_A, C, sr, tg, c) consisting of: (1) a set of ***nodes*** G_N, and a set of ***arcs*** G_A; (2) a ***color alphabet*** C; (3) a ***source map*** $sr : G_A \rightarrow G_N$, and a ***target map*** $tg : G_A \rightarrow G_N$, which return the source node and target node of an arc, respectively; and (4) a ***color map*** $c : G_N \cup G_A \rightarrow C$, which returns the color of a node or an arc.

Definition 4 *(Ground Type Graph)* A ***ground type graph*** is a single-node colored directed graph which is colored by a ground type.

Definition 5 *(Function Type Graph)* A ***function type graph*** $(s, G_1, G_2)_{(G_N, G_A, C, sr, tg, c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, and two type graphs G_1 and G_2, such that, (1) $c(s) = \rightarrow$; (2) there are two arcs associated with the starting node s, *left* arc $l \in G_A$ and *right* arc $r \in G_A$, such that $c(l) = in$, $c(r) = out$; l connects G_1 to s by $sr(l) = s_{G_1}$, $tg(l) = s$, and r connects s to G_2 by $sr(r) = s$, $tg(r) = s_{G_2}$, where s_{G_1} and s_{G_2} are the starting nodes of G_1 and G_2, respectively; (3) G_1 and G_2 are disjoint; (4) if there is an arc $a \in G_A$ with $c(a) = rec$, then $sr(a) = s_{G_i}$, $tg(a) = s$, $c(s_{G_i}) = \rightarrow$, $i = 1, 2$.

Definition 6 *(Object Type Graph)* An ***object type graph*** $(s, A, R, L, S)_{(G_N, G_A, C, sr, tg, c)}$ is a directed colored graph consisting exactly of a ***starting node*** $s \in G_N$, a set of ***method arcs*** $A \subseteq G_A$, a set of rec-colored arcs $R \subseteq G_A$, a set of ***link arcs*** $L \subseteq G_A$, and a set of type graphs S, such that (1) $c(s) =$ *self*. (2) $\forall a \in A$, $sr(a) = s$, $tg(a) = s_F$ for some type graph $F \in S$, and $c(a) = m$ for some method label m; $c(a) \neq c(b)$ for $a, b \in A$, $a \neq b$. (3) $\forall r \in R$, $c(r) = rec$, $tg(r) = s$, $sr(r) = s_F$ for some $F \in S$, and $c(s_F) =$ *self*. (4) $\forall l \in L$, $sr(l) = s_F$, $tg(l) = s_G$ for some $F, G \in S$, and $c(l) = bym$ for some method label m.

Remarks: Directed colored graph is the foundation of graph grammar theory [EPS73, Ehr78, Roz97]. Object type graphs are adapted from directed colored graphs. Ground type graphs are

trivial. Function type graphs are straightforward. They need to be defined because an object type graph may include them as subgraphs. An object type graph is formed by a starting node s and a set S of type graphs with each $F \in S$ being connected to s by a method arc that goes from s to F. The starting node s is colored by *self* and is used to denote the self type. The method interdependencies are specified by arcs in L. If $L(m)$ is the set of links of method m, then for each $l \in L(m)$ there is an arc (colored by *byl*) that goes from l to m. Recursive object types are specially indicated by rec-colored arcs in R.

For the sake of brevity, we drop the subscripts in $(s,G_1,G_2)_{(G_N,G_A,C,sr,tg,c)}$ and $(s,A,R,L,S)_{(G_N,G_A,C,sr,tg,c)}$ whenever possible throughout the paper.

4.2 Examples of OTG

We now provide some examples to illustrate the definitions introduced in the last section. Throughout this section, if the type of an object a is represented by a graph A, we will say the type of a is A, and vice versa.

Example 1 *In Figure 1, A, B, and C are the type graphs for the three ground types int, $real$, and $bool$ respectively. They are just a node colored by the appropriate ground types. D is the type graph for function type $int \to int$. E is the type graph for $(int \to B) \to int$, where B is the object type in Figure 2(a) which will be explained in the next example.*

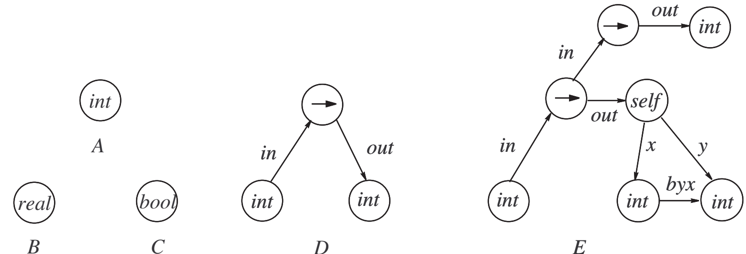


Figure 1: Examples of Ground Type Graphs and Function Type Graphs.

Example 2 *In Figure 2(a), graph A denotes the object type $[x:int, y:int]$, where methods x and y are independent of each other. Graph B denotes the type $[x:int, y(\{x\}):int]$ where y depends on x. Note that the direction of the link arc in B is from x to y (not from y to x), and that the link is colored by byx, signifying that changes made to method x will affect method y. For instance, an object of type A may be $[x=1,\ y=2]$ (which is actually a record), and an object of type B may be $[x=1,\ y=\varsigma(s:\mathscr{S}(B))(s.x+2)]$.*

Note also that although the presence of the link in B or the absence of the link in A serves as an extra condition (compared to conventional type systems) for selecting objects to be typed as A or B respectively, there are still infinitely many objects that are of type A or type B. For example, objects $[x=m, y=n]$ with $m,n \in \mathbf{N}$ are all of type A; objects $[x=n, y=\varsigma(s:B)(a(s.x)+b)]$ with $n,a,b \in \mathbf{N}$ are all of type B. In this sense, OTG is (still) an abstract specification of object behaviors.

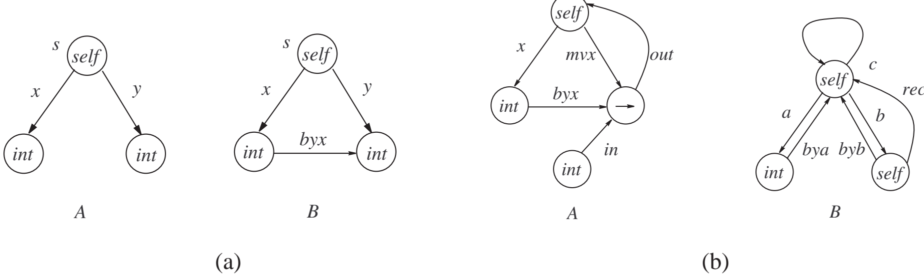


Figure 2: Example of Object Type Graphs

Example 3 *In Figure 2(b), A is the type graph for* $\iota(t)[x:int, mvx(\{x\}):int \rightarrow t]$ *which is the type of a simplified 1-d movable point* $[x=1, mvx=\varsigma(s:B)\lambda(i:int)(s.x \Leftarrow s.x+i)]$. *The facts that* mvx *depends on* x *and returns a modified self object are indicated by the* byx*-colored arc and the* out*-colored arc respectively. Note the direction of the* out*-colored arc goes to the starting node of the type graph directly, indicating that this is a self type (as opposed to recursive objet type). Graph B represents the type of the object*$[a=1,\ b=p,\ c(\{a,b\})=\varsigma(s:\mathscr{S}(B))s]$ *where* p *is some predefined object of type B. Here, note that the fact that* b *is of recursive object type is depicted by a* $self$*-colored node and a* rec*-colored arc going from this node to the starting node of the graph; and that the fact that* c *is of self type is depicted by its method arc going directly to the starting node of the graph. The difference between recursive object type and self type is clearly represented in object type graphs.*

5 Subtyping under OTG

Given the definition of OTG, we now investigate the issue of subtyping under OTG. Throughout the paper, we write $A_\sigma <: B_\tau$ iff $\sigma <: \tau$ where σ andτ are types and A_σ and B_τ are their type graphs. We first present the necessary definitions and then provide some subtyping examples.

5.1 Definitions

Definition 7 *(Type Graph Premorphism)* Let Φ be the set of ground types. Given two type graphs $G=(G_N, G_A, C, sr, tg, c)$ and $G'=(G'_N, G'_A, C', sr', tg', c')$, a ***type graph premorphism*** $f:G \rightarrow G'$ is a pair of maps $(f_N:G_N \rightarrow G'_N, f_A:G_A \rightarrow G'_A)$, such that (1) $\forall a \in G_A$, $f_N(sr(a)) = sr'(f_A(a))$, $f_N(tg(a)) = tg'(f_A(a))$, and $c(a)=c'(f_A(a))$; (2) $\forall v \in G_N$, if $c(v) \in \Phi$, then $c'(f_N(v)) \in \Phi$; otherwise $c(v)=c'(f_N(v))$.

Definition 8 *(Base, Subbase)* Given an object type graph $G=(s,A,R,L,S)$. The ***base*** of G, denoted by $Ba(G)$, is the graph $(s,A,t(A),L)$, where $t(A)=\{tg(a) \mid a \in A\}$. A ***subbase*** of G is a subgraph $(s,A',t(A'),L')$ of $Ba(G)$, where $A' \subseteq A$, $L' \subseteq L$, $t(A')=\{tg(a) \mid a \in A'\}$, and for each $l \in L'$ there exist $a_1, a_2 \in A'$ such that $sr(l)=tg(a_1)$ and $tg(l)=tg(a_2)$.

Definition 9 *(Closure, Closed)* The ***closure*** of a subbase $D = (s, A', t(A'), L')$ of an object type graph $G = (s, A, R, L, S)$, denoted by $Cl(D)$, is the union $D \cup E_1 \cup E_2$, where (1) $E_1 = \{l \in L \mid \exists a_1, a_2 \in A'$ with $tg(a_1) = sr(l),\ tg(a_2) = tg(l)\}$, and (2) $E_2 = \{l, h, a, t(l) \mid l, h \in L,\ a \in A,\ a \notin A',\ tg(l) = sr(h) = tg(a)$, and $\exists a_1, a_2 \in A'$ such that $tg(a_1) = sr(l),\ tg(a_2) = tg(h)\}$. A subbase D is said to be ***closed*** if $D = Cl(D)$.

Definition 10 *(Covariant, Invariant)* Given an object type graph (s, A, R, L, S). Let $t(A) = \{tg(a) \mid a \in A\}$. For each $v \in t(A)$, if v is not incident with any links, or if v is the target node of some links but not the source node of any links, then v is said to be ***covariant***; otherwise, v is said to be ***invariant***.

Definition 11 *(Object Subtyping)* Given two object type graphs $G = (s_G, A_G, \emptyset, L_G, S_G)$ and $F = (s_F, A_F, \emptyset, L_F, S_F)$. $F <: G$ if and only if the following conditions are satisfied: (1) There exists a premorphism f from $Ba(G)$ to $Ba(F)$ such that $f(Ba(G)) = Cl(f(Ba(G)))$. That is, $f(Ba(G))$ is closed. (2) For each node v in $f(Ba(G))$, let u be its preimage in $Ba(G)$ under f, $F_v \in S_F$ be the type graph with v as its starting node, and $G_u \in S_G$ be the type graph with u as its starting node. (i) If v is invariant, then F_v is isomorphic to G_u. (ii) If v is covariant, then $F_v <: G_u$.

Remarks: Type graph premorphism is adapted from graph morphism which is a fundamental concept in algebraic graph grammars [EPS73, Ehr78, Roz97]. It preserves the directions and colors of arcs and the colors of nodes up to ground types. The base of an object type graph singles the method interdependency information out of the entire object type graph so that the structure of the method interdependencies can be better studied. The closure of a subbase captures the complete behavior of the subbase by including, in addition to all methods and links in the subbase, a set E_2 of methods (and associated links) outside of the subbase in the following way: for any method l in E_2, (1) l depends on some methods inside the subbase, and (2) there exist some methods inside the subbase that depend on l.

5.2 Examples

We now present some simple subtyping examples.

Example 4 *Given the two type graphs in Figure 3(a), clearly we can find a premorphism f from base of A to base of B such that $f(Ba(A))$ is closed; note also that node v in B is covariant. Thus, $B <: A$. As an example, we can regard the object $[x = 1,\ y = \varsigma(s{:}\mathscr{S}(B))(s.x+1)]$ of type B as having type A.*

Example 5 *For the two type graphs in Figure 3(b), we can also find a premorphism from the base of A to the base of B, and node v in B is also covariant. But, node u in B is invariant which requires the corresponding node u' in A have the same color – pos (standing for positive integer) in order to have $B <: A$. But u' is colored by int, hence, $B \not<: A$.*

As an example to justify that $B \not<: A$, let $b = [x = 1,\ y = \varsigma(s{:}\mathscr{S}(B))(\log(s.x)+1)]$, it is easy to check $b : B$. If $B <: A$, then $b : A$, and in this case we can update the x field in b to a negative integer, say, -1, resulting an object like $b = [x = -1,\ y = \varsigma(s{:}\mathscr{S}(B))(\log(s.x)+1)]$. In this object, the invocation of method y will crash since log is not defined over negative integers.

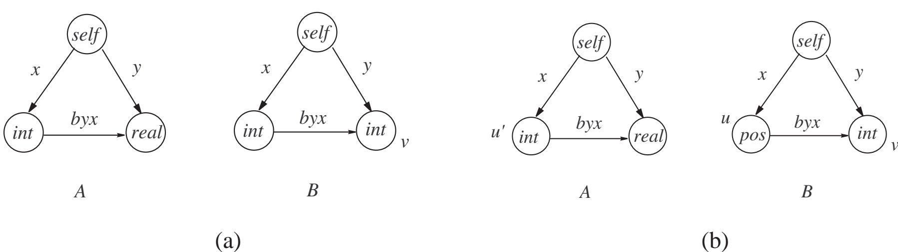


Figure 3: Examples of Object Subtyping

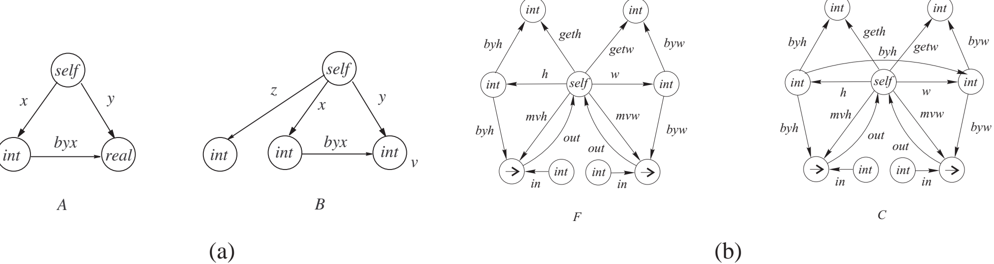


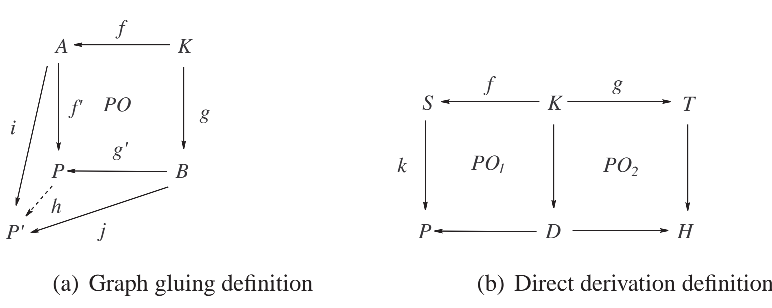
Figure 4: Examples of Object Subtyping

Example 6 *Considering the graphs A and B in Figure 4(a), it is easy to check (similar to the case of example 4) that $B <: A$. As an example for this subtyping, an object $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x+1),\ z = 1]$which is of type B, can clearly be regarded as having type A.*

Example 7 *Let us revisit the two object types in Figure 2(a). We have $B \not<: A$, since we cannot find a premorphism f from $Ba(A)$ to $Ba(B)$ such that $f(Ba(A))$ is closed. Similarly, there exists no such a premorphism g from $Ba(B)$ to $Ba(A)$ such that $g(Ba(B))$ is closed, so $A \not<: B$.*

One may wonder what kind of type (graph) can be of a subtype of A or B respectively. Any subtype of A must not have a link between methods x and y; and any subtype of B must have a link going from x to y. This is the structural requirement in Definition 11. As a result, object $[x = 1,\ y = 1]$ cannot be regarded as having the same type with the object $[x = 1,\ y = \varsigma(s : \mathscr{S}(B))(s.x+1)]$, and vice versa. One may contend that this subtyping is too restrictive so that some "good" subtyping instances are not allowed by it; we argue that this is the trade-off in the sense that the strictness of this subtyping can block and prevent potential programming errors, as shown in the next example.

Example 8 *As the last example, we show how the "free or constrained rectangles problem" described in section 2.1. The (new) types of the free rectangle fRect and the constrained rectangle cRect are depicted as F and C in Figure 4(b). Note that the independence between the height and the width in the free rectangle fRect and their dependency in the constrained rectangle cRect are faithfully shown by the absence and presence of a byh-colored link between methods h and w in F and C, respectively. It is easy to check that $C \not<: F$. So if we modify the function ds of section 2.1 by replacing its parameter type FR by the new type F in Figure 4(b), then the call $ds(cRect)$*



(a) Graph gluing definition

(b) Direct derivation definition

Figure 5: Definitions of Graph Gluing and Direct Derivation

will be rejected under OTG by the compiler since it does not type-check.

6 Inheritance under OTG

We now turn to the issue of inheritance. Some basic notions in graph grammar are needed before we can define inheritance formally.

6.1 Definitions

Definition 12 *(Type Graph Production)* A ***type graph production*** p is a pair of type graph premorphisms $f : K \to A_1$ and $g : K \to A_2$, where A_1 is called the *left side*, A_2 the *right side*, and K the *interface*. This is denoted as $p = (A_1 \stackrel{f}{\leftarrow} K \stackrel{g}{\rightarrow} A_2)$.

Definition 13 *(Type Graph Gluing)* Given two type graph premorphisms $f : K \to A$ and $g : K \to B$. The ***gluing*** of A and B along K is the ***pushout*** of $K \stackrel{f}{\rightarrow} A$ and $K \stackrel{g}{\rightarrow} B$ in the category formed by type graphs together with type graph premorphisms (Figure 5(a)).

Definition 14 *(Direct Derivation)* Given type graphs P, D, H, a type graph production $p = (S \stackrel{f}{\leftarrow} K \stackrel{g}{\rightarrow} T)$, and a premorphism $k : S \to P$ (called a context map). We say that H is ***directly derived*** from P via p by k, denoted by $P \stackrel{(p,k)}{\Longrightarrow} H$, if P is the result of gluing S and D along K and H is the result of gluing D and T along K (Figure 5(b)).

Definition 15 *(Unfolding Production and Operation)* A graph production $u = (S \stackrel{f}{\longleftarrow} K \stackrel{g}{\longrightarrow} G)$ is called an unfolding production if (1) K is a graph consisting of two nodes v_1 and v_2, and $c(v_1) = c(v_2) = self$; (2) S is a graph consisting of two nodes u_1 and u_2 and an arc t such that $c(u_1) = c(u_2) = self, c(t) = rec, sr(t) = u_2$, and $tg(t) = u_1$; (3) $f(v_i) = u_i, i = 1,2$; g is a partial morphism with $g(v_2) = s_G$ where s_G is the starting node of G. Given an unfolding production $u = (S \stackrel{f}{\longleftarrow} K \stackrel{g}{\longrightarrow} G)$, an object graph F, and a premorphism $i : S \to F$, we say F unfolds to P if $F \stackrel{(u,i)}{\Longrightarrow} P$.

Definition 16 *(Addition Production and Operation)* A graph production $a = (S \xleftarrow{f} K \xrightarrow{g} G)$ is called an addition production, if (1) K consists of only one node v, and $c(v) = self$; (2) S consists of only one node u, and $c(u) = self$; (3) $f(v) = u$ and $g(v) = s_G$ where s_G is the starting node of G. Given an addition production $a = (S \xleftarrow{f} K \xrightarrow{g} G)$, an object graph F, and a premorphism $j : S \to F$, we say that P is the result of adding G into F if $F \xRightarrow{(a,j)} P$.

Definition 17 *(Link Production and Operation)* A graph production $l = (S \xleftarrow{f} K \xrightarrow{g} G)$ is called a link production, if (1) G is a graph consisting of two nodes v_1, v_2 and an arc a connecting these two nodes. $c(v_i)$ $(i = 1,2)$ is either a ground type or a $\to$ or a $self$, and $c(a) = bym$, where m is the color of one of the methods in G; (2) K is a graph consisting of two nodes u_1 and u_2 with $c(u_i)$ is either a ground type or a $\to$ or a $self$, $i = 1,2$; (3) S is isomorphic to K, that is, f is an isomorphism from K to S; g is an injection with $g(u_i) = v_i$, $i = 1,2$; Given a link production $l = (S \xleftarrow{f} K \xrightarrow{g} G)$, an object graph F, and a premorphism $k : S \to F$, we say that P is the result of embedding G into F if $F \xRightarrow{(l,k)} P$.

Definition 18 *(Inheritance Construction of Object Type Graphs)* Given object type graphs F and G. F is said to be inherited from G if G can be transformed into F through a finite sequence of unfolding operations, addition operations, and link operations.

Definition 19 *(Inheritance)* Given object type graphs S and T, an object of type T can be constructed by inheritance from an object of type S if T is inherited from S.

The central idea here is that inheritance of objects should be guided and guarded by object types. We devise, through some basic graph transformation techniques, an "inheritance" notion on object types, and then use this notion to judge whether an object can be built through inheritance from another object.

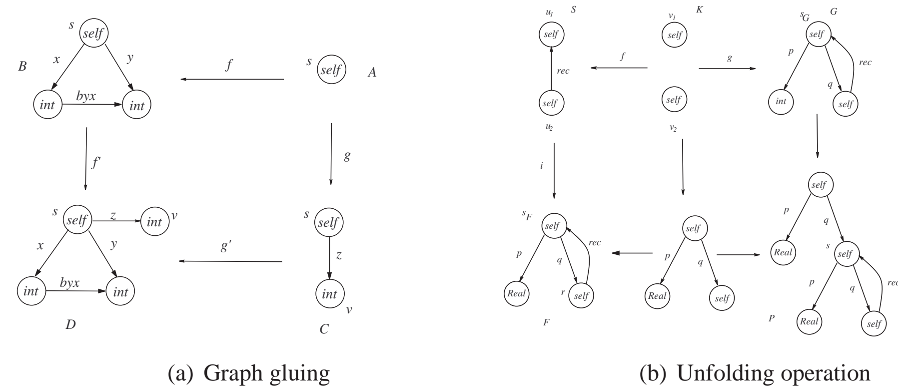
6.2 Examples

We now give some examples to demonstrate the graph operation definitions in the last section.

Example 9 *A graph gluing example is shown in Figure 6(a), where f and g map the only node in A to the $self$-colored node in B and C respectively, and D is the result of gluing B and C along A. Intuitively, this gluing operation entails the connection of B and C by identifying their starting nodes.*

Example 10 *Figure 6(b) shows an unfolding operation. $F \xRightarrow{(u,i)} P$, where $u = (S \xleftarrow{f} K \xrightarrow{g} G)$, $f(v_i) = u_i$, $i = 1,2$, $g(v_2) = s_G$, $i(u_1) = s_F$, $i(u_2) = r$. As we can see, P can be understood as constructed by deleting the rec-colored arc from F, and then glue the result with a copy of the original F by identifying the starting node of the former with the source node of the rec-arc of the latter.*

Example 11 *We finally show how the "colored-point problem" addressed in section 2.2 can be*



(a) Graph gluing

(b) Unfolding operation

Figure 6: Examples of Graph Gluing and Unfolding Operation

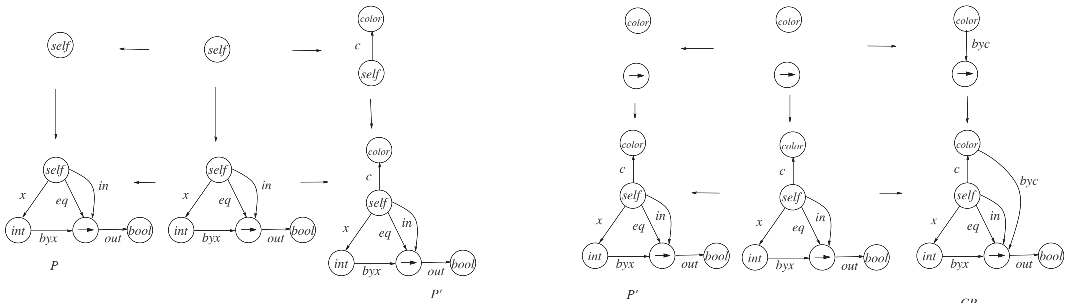


Figure 7: *CP* is Inherited from *P*.

resolved under OTG inheritance and subtyping. The type of points pt and pt_0 and the type of color point cpt are depicted as P and CP in Figure 7 respectively. We can see that CP is inherited from P (through one addition operation and one link operation), which means that cpt can be constructed by inheritance from pt (or pt_0). Moreover, it is easy to check that $CP <: P$ under OTG subtyping, which indicates that inheritance and subtyping are congruent in this case. (Note that this contrast with the "inheritance is not subtyping" slogan in the literature which is mainly motivated by this colored point example.) Finally, the crash of $F(cpt)$ is prevented since the function F, as defined in section 2.2 and in the literature, does not type-check under OTG typing. Note in its definition, $F = fun(p : PT)(p.eq(pt_0))$, with $PT = ObjectType(MyType)\{x : int, eq : MyType \rightarrow Bool\}$, $p.eq$ requires an argument of the self type associated with PT, $\mathscr{S}(PT)$, but pt_0 has type PT, and PT is neither the same as nor the subtype of $\mathscr{S}(PT)$ under OTG.

7 Resolution of the Problems

As examples of OTG subtyping and inheritance, we have demonstrated in the last section that the two problems outlined in section 2 can be successfully resolved under OTG subtyping and inheritance mechanisms. Here, we just summarize some major points.

- OTG subtyping takes into consideration the method interdependencies in objects. An object in which there is no dependence between two methods can never be regarded as having the same type with or a subtype of that of an object in which there is an interdependency between these two methods, and vice versa. The problem addressed in section 2.1 can be naturally resolved in OTG since there is an interdependency between height and width in constrained rectangles, and there is no such interdependency in free rectangles. Consequently, these two kinds of rectangles have different types.
- OTG inheritance replies on basic graph derivations. The fundamental idea in this respect is that inheritance on objects should be regulated using type information of the relevant objects. An "inheritance" relation over object types is first defined using graph derivations and then used to determine whether an object can be constructed by inheritance from another one.
- "Inheritance is not subtyping" has been advocated in the literature for quite a while. Despite that, the mainstream OOP still adheres to the practice that "inheritance indicates subtyping". One of the reasons for this is that without this practice, the software hierarchy built by inheritance would be almost useless. Thus this practice is highly desirable. The"colored point problem" described in section 2.2 is one of the motivating examples that has prompted "inheritance is not subtyping", because otherwise we will face some "method-not-found" error. Under OTG, we give this problem a new solution in the sense that "inheritance indicates subtyping" is retained and "method-not-found" error is avoided.

8 Related Work

Representing object types as directed colored graphs and subsequently addressing the subtyping and inheritance issues by graph transformations is our original idea. It uniquely connects the type theory of object-oriented languages to algebraic graph transformation theory. The foundations of type theory can be found in [Bar92, AC96, Pie02], and the recent results and directions in type theory research are reflected in, for example, [PRB07, Che07, DHC07]. The origin of algebraic graph grammar and graph transformation can be traced back to [EPS73, Ehr78], and [Roz97, EEPT06] present a comprehensive coverage of this research area. For current trends and developments in graph transformation, see for example, the proceedings of GT-VMT and ICGT [EG07, CEM+06].

Incidentally, it is interesting to note that the phrase "type graph" has been used inconsistently in the literature. For example, it is used to denote the disjunctive rational trees in Prolog type analysis and database query algebra [HCC93, Sch01], to facilitate the investigation of quantification in Type Logical Grammar [BS06], and to give types for (some other) graphs in graph transformation study [GL07, EEPT06]. Obviously, none of these relates to the object type graphs introduced in this paper in a clear manner.

9 Final Remarks

Subtyping and inheritance are two major issues in OOP. Although both issues have been studied extensively, problems still persist. Two particular problems, method interdependencies and "inheritance is not subtyping", are identified and subsequently addressed by a graph-computing

(OTG) approach in this paper. It is demonstrated that both problems can be resolved effectively under OTG subtyping and inheritance mechanisms.

Bibliography

[AC96] M. Abadi, L. Cardelli. *A Theory of Objects*. Springer-Verlag, New York, 1996.

[Bar92] H. Barendregt. Lambda Calculi with Types. In S. Abramsky (ed.), *Handbook of Logic in Computer Science*. Volume 2, pp. 117–309. Clarendon Press, Oxford, 1992.

[BL95] V. Bono, L. Liquori. A Subtyping for the Fisher-Honsell-Mitchell Lambda Calculus of Objects. In *Proc. of International Conference of Computer Science Logic*. LNCS 933, pp. 16–30. 1995.

[Bru94] K. Bruce. A paradigmatic Object-Oriented Programming Language: Design, Static Typing and Semantics. *Journal of Functional Programming* 4(2):127–206, 1994.

[BS06] C. Barker, C. chieh Shan. Types as Graphs: Continuations in Type Logical Grammar. *J. of Logic, Language and Information* 15(4), 2006.

[CEM+06] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, G. Rozenberg (eds.). *Proc. of ICGT'06*. LNCS 4178, Springer, 2006.

[CHC90] W. Cook, W. Hill, P. Canning. Inheritance is not Subtyping. In *Proc. of POPL*. Pp. 125–135. 1990.

[Che07] J. Chen. A typed intermediate language for compiling multiple inheritance. In *Proc. of POPL'07*. Pp. 25–30. 2007.

[DHC07] D. Dreyer, R. Harper, M. Chakravarty. Modular type classes. In *Proc. of POPL'07*. Pp. 63–70. 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Springer, 2006.

[EG07] K. Ehrig, H. Giese (eds.). *Proc. of GT-VMT'07*. http://eceasst.cs.tu-berlin.de/, 2007.

[Ehr78] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *Graph-Grammars and Their Applications to Computer Science and Biology*. LNCS 73, pp. 1–69. Springer-Verlag, 1978.

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph grammars: An algebraic approach. In *IEEE Conference of Automata and Switching Theory*. Pp. 167–180. 1973.

[FHM94] K. Fisher, F. Honsell, J. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nodic Journal of Computing* 1:3–37, 1994.

[GL07] E. Guerra, J. de Lara. Adding Recursion to Graph Transformation. In *Proc. of GT-VMT'07*. 2007.

[HCC93] P. V. Hentenrck, A. Cortesi, B. L. Charlier. Type Analysis of Prolog Using Type Graphs. Technical report, Brown University, Technical Report CS-93-52, 1993.

[LC96] L. Liquori, G. Castagna. A Typed Lambda Calculus of Objects. LNCS 1179, pp. 129–141. Sringer–Verlag, 1996.

[Liq98] L. Liquori. On Object Extension. In *ECOOP'98 Object-oriented Programming*. Lecture Notes in Computer Science 1445, pp. 498–522. Sringer–Verlag, 1998.

[Pie02] B. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[PRB07] P. Permandla, M. Roberson, C. Boyapati. A type system for preventing data races and deadlocks in the java virtual machine language. In *Proc. of LCTES'07*. Pp. 1–10. 2007.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1. World Scientific, 1997.

[Sak05] M. Sakkinen. Wishes for Object-oriented Languages. In *Proc. of Langages et Modeles a Objets (LMO 2005, invited talk)*. 2005.

[Sch01] K.-D. Schewe. On the Unification of Query Algebras and Their Extension to Rational Tree Structures. In *Proc. of 12th Australasian Database Conference*. Pp. 52–59. 2001.

[Seb07] R. Sebesta. *Concetps of Programming Languages*. Addison Wesley, 8th edition, 2007.

Using Graph Transformation Systems to Specify and Verify Data Abstractions

Luciano Baresi[1], Carlo Ghezzi[2], Andrea Mocci[3] and Mattia Monga[4]

{ baresi[1], ghezzi[2], mocci[3] } @elet.polimi.it
DeepSE Group
Dipartimento di Elettronica e Informazione - Politecnico di Milano
Piazza L. Da Vinci, 32 20133 Milano, Italy

[4] mattia.monga@unimi.it,
DICo - Università degli studi di Milano
Via Comelico, 39 20135 Milano, Italy

Abstract: This paper proposes an approach for the specification of the behavior of software components that implement data abstractions. By generalizing the approach of behavior models using graph transformation, we provide a concise specification for data abstractions that describes the relationship between the internal state, represented in a canonical form, and the observers of the component. Graph transformation also supports the generation of behavior models that are amenable to verification. To this end, we provide a translation approach into an LTL model on which we can express useful properties that can be model-checked with a SAT solver.

Keywords: Graph Transformation Systems, Specifications, Data Abstractions, Model Checking, SAT Solving

1 Introduction

Abstraction by specification [LG00] is the fundamental approach to abstract from implementation details by providing a high-level description of the behavior of a software component. Such an abstract description of the behavior is the *specification* of the software component.

In this paper, we focus on the specification of *data abstractions* [LG00], which can be viewed as a particular class of stateful components. In a data abstraction, the internal state is hidden; clients can interact with the component by invoking the operations exposed by its *interface*. Common examples of data abstractions are stacks, queues, and sets, which are usually part of the libraries of modern object oriented languages, such as JAVA.

Several formalisms have been proposed in the past for the specification of data abstractions. Among these, we mention the pioneering work on algebraic specifications [GH78, GTW78]. Recently, in the area of recovering specifications and program analysis, behavior models (see for example [DLWZ06, XMY06]) have been proposed as a simple formalism to relate the observable part of a data structure with the methods used to modify its internal state. In this paper, we propose a generative approach for the construction of behavior models based on graph transformation systems. Moreover, we found that the generative capabilities of graph transformation

tools are suitable for the generation of models that can be easily translated into logic models for verification. For this purpose, we propose a translation of graph-transformation generated models into a linear temporal logic, on which verification is possible via bounded model checking.

This paper is organized as follows. Section 2 recalls some background concepts on the specification of data abstractions. Section 3 presents the case of a traversable stack, a container whose specification has been critically analyzed in the past because of its subtle intricacies. Section 4 describes our approach to the specification of stateful components implementing data abstractions. Section 5 contains our bounded model checking approach to the verification. Finally, Section 6 concludes the paper.

2 Formalisms for Data Abstractions

Data abstractions hide their internal state and implementation and export a set of operations (methods), to allow clients to access their instances. Methods can be classified as:

- *constructors*, which produce a new instance of a class;
- *observers*, which return some view of the internal state;
- *modifiers*, which change the internal state.

A method might both modify the internal state and return some information about it. In this case, such an observer is called *impure*; otherwise, it is *pure*, that is, it has no side-effects and does not alter the internal state of the object.

Stateful components implementing data abstractions may be specified by providing pre- and post-conditions [Hoa69] for the methods exposed by the component's interface. Because a method may also change the internal state, this approach requires the introduction of a logical abstraction of the internal state, which complicates the specification of the component. For example, the JML language [LBR99] uses *models* for this purpose.

Algebraic specifications do not generally require an explicit abstraction of the hidden state, since it is implicitly taken into account through the use of axioms on sequences of operations. An algebraic specification is composed of two parts: the *signature* and the *set of axioms*. The algebraic signature defines the types (*sorts*) used in the set of axioms, and the signatures of the operations. The set of axioms is usually composed of a sequence of universally quantified formulae expressing equalities among terms in the algebra. For example, an algebraic specification for a class implementing a stack of strings may be characterized by the following axiom:

$$\forall s \in Stack, \forall e \in String : pop(push(s,e)) = s$$

which states that for every possible stack, the object obtained by the application of a push followed by a pop is equivalent to the original object. Algebraic specifications are supported nowadays by various languages and tools [GH93, Com04].

2.1 Behavior Models

A behavior model is a finite-state automaton that captures the relationship among the modifiers and return values of the observers. In a behavior model, each state is an abstraction of a set of

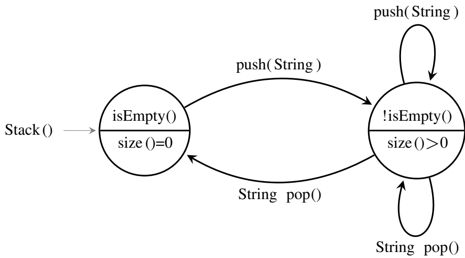


Figure 1: A simple abstracted behavior model of a Stack.

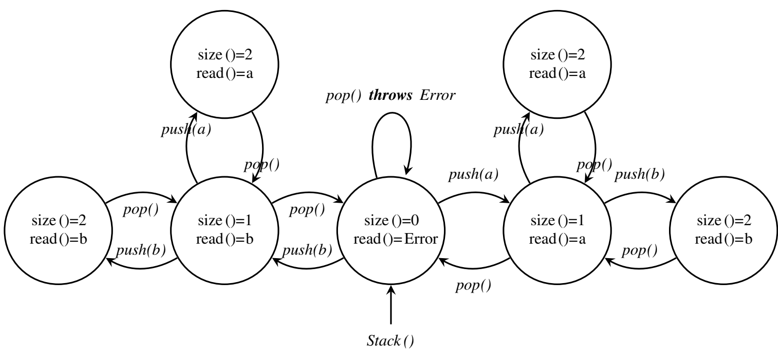


Figure 2: A deterministic behavior model for a Stack

different internal states of the actual component, identified by a simple predicate on the return values of a set of observers. Several different approaches to recover behavior models have been implemented for various purposes. For example, ADABU [DLWZ06] behavior models are characterized by states labeled with pure observers, while transitions are labeled with method names (without actual parameters). Other approaches, such as ABSTRA [XMY06], produce more precise finite state machine abstractions by using different techniques to represent the component's state. Figure 1 shows the simple behavior model of a stack produced by ADABU; in this model, the behaviors of all the stacks with size greater than zero are abstracted in a unique state.

Since infinitely many states are summarized in a finite-state machine, the resulting automaton is necessarily non-deterministic. However, a deterministic model is often more useful to reason about the properties of the component or to generate test cases; thus, we aimed at producing a deterministic specification, that is, one in which the application of each operation brings the automaton to a unique state, which represents a unique state of the object.

Figure 2 shows a partial deterministic model of the behavior of a stack. Deterministic models grow bigger and, for components with infinite possible internal states, they can only partially represent the behavior of the component. Deterministic models, such as the one of Figure 2, rely on the presence of a set of *instance pools* for the actual parameters and represent the behavior of the component for such parameters. In the example, the *push* method is tested with an instance pool for the input parameter composed of two different strings, “a” and “b”. In this paper, we present

```
public class MTStack {
  public MTStack() { .. }
  public void push(String element) { .. }
  public void pop() throws Error { .. }
  public void down() { .. }
  public void reset() { .. }
  public String read() throws Error { .. }
  public boolean isEmpty() { .. }
  public int size() { .. }
}
```

Figure 3: The public interface of Majster's Traversable Stack

an approach to the representation of the complete, unabstracted and thus deterministic behavior of stateful components defining data abstractions by using graph transformation systems.

3 Traversable Stack

In this section we introduce an example of data abstraction that is used as a running example to explain our approach. The example is inspired by a case study that generated a lively debate in the late 1970s in the community working on algebraic specifications. We will refer to the example as Majster's Traversable Stack (MTS) [Maj77], by the name of the author who first addressed the problem. The public interface of MTS is shown in Figure 3. For simplicity, we assume the contained object to be of type String. MTS defines the usual operations of a stack, such as *push* and *pop*, and allows for traversal by using a hidden pointer and by exposing the following operations:

- *down*, which traverses the stack circularly by moving the pointer stepwise towards the bottom of the stack;
- *read*, which yields the element returned by the pointer;
- *reset*, which moves the pointer on the top of the stack.

For example, let us consider a stack of three elements, obtained by applying the constructor and three *push* operations of three different elements, the strings "a", "b" and "c". In this case, the *read* observer returns "c". If a *down* operation is applied, the hidden pointer moves towards the bottom of the stack; thus, the *read* observer returns "b". If the hidden pointer reaches the bottom of the stack, a further application of the *down* operation brings the hidden pointer to the top of the stack.

MTS was introduced because the author argued that no finite set of axioms could specify the data abstraction in an algebraic way without using auxiliary functions, that is, purely in terms of the externally visible operations. For example, the specification of the *down* operation would require axioms like the following:

$$down^n\left(down^{n-1}\left(\ldots\left(down^1\left(push^n\left(\ldots push^2\left(push^1(MTStack(),o_1),o_2\right)\ldots\right),o_n\right)\right)\ldots\right)\right)=$$
$$push^n\left(push^{n-1}\left(\ldots push^2\left(push^1(MTStack(),o_1),o_2\right)\ldots,o_{n-1}\right),o_n\right)$$

Since n [1] is generic in the axiom formula, the specification would require an axiom for each $n>0$.

In the sequel, we will show how MTS can be rigorously specified in our approach, based purely on the operations exported by the data abstraction's interface.

4 A Graph Transformation approach

In this section we illustrate our approach to the specification of data abstractions based on graph transformations. Let us consider the behavior model for a stack depicted in Figure 2: it describes the behavior of all the stacks up to size 2. The specification of an abstract data type is given as a graph transformation system whose rules can be applied not only to generate such a partial (deterministic) model but also any other model that, for example, describes the behavior of stacks up to a generic size n.

In this paper we use the approach to graph transformation systems implemented in AGG [Tae04], which supports a rich set of features, such as negative application conditions, attributes defined as JAVA objects, and conditions on attributes. In the following, we define the *type graph* for MTS, which defines the kinds of graph nodes and edges that are needed in the specification, and then we define the graph transformation rules.

4.1 Type graph

The type graph for MTS is shown in Figure 4. A state of the behavior model is either a *null state* or an *object state* (a stack of strings). Conventionally, a null state represents the state of an instance object of the data abstraction before any application of a constructor. The *null state* is unique and represents the initial state of the behavior model. The type node representing an object state is labeled with a set of attributes, which represent the return values of the observers invoked when the object is in the corresponding state. Edges represent constructors and modifiers. Constructors link the null state node with a node representing the object state after applying a constructor, while modifiers are loops on the state node since they change the state of the instance. Each edge is labeled with attributes corresponding to the parameters, the return value, and the exception the method might rise when applied (exceptions are modeled as Boolean values).

As already said, behavior models rely on instance pools of parameters to be used to generate actual invocations of modifiers. For example, we can generate a behavior model for MTS that uses two strings, "a" and "b" as the possible contained objects. The type graph has a node for each type needed, labeled in the same way as the data abstraction node. For primitive types and

[1] In this and in the following formulae, the superscript j above each operation indicates the j-th subsequent operation of that kind.

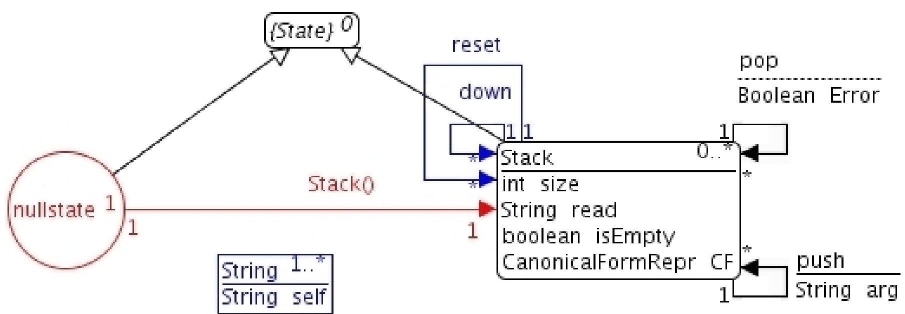


Figure 4: MTS Type Graph

Strings, we can *box* the value in a node that contains just an attribute representing the primitive type.

4.2 Rules and Canonical Form

Since data abstractions hide their internal state, the information gathered by just invoking observers might be insufficient to uniquely identify the object state. For example, let us consider two MTS of size 2, containing the same string “a” on the top and two different strings on the bottom, “a” for the first stack and “b” for the second. In both cases, let us consider the hidden pointer to be on the top of the stack. The invocation of the three observers (*read*, *size*, *isEmpty*) of the MTS is insufficient to reveal the different internal state, since they would return the same values (i.e., “a” for *read*, 2 for *size* and **false** for *isEmpty*).

Inspired by Veloso’s algebraic specification of MTS [Vel79], we use a *canonical form* for the data abstraction to identify each different object state. We define the canonical form as a language composed by method applications as tokens. The canonical form language must satisfy the following properties:

- each string of the language is composed of an initial constructor and a —possibly empty— sequence of modifiers;
- for each possible internal state, there is one and only one string of the canonical form language to represent it;
- for each string of the language, there is an internal state that is labeled by that string, such that the invocation of the corresponding sequence of methods produces an object of that state.

As a convenience, we identify a language of operations, which satisfies these properties, as a canonical form of the data abstraction. This explains why the object state node of the type graph on Figure 4 is enriched with an attribute (CF) describing its canonical form. According to this approach, any possible MTS instance can be represented with a string of the following canonical form language $\mathscr{L}_{MTS}$:

- ε, which conventionally labels the null state;

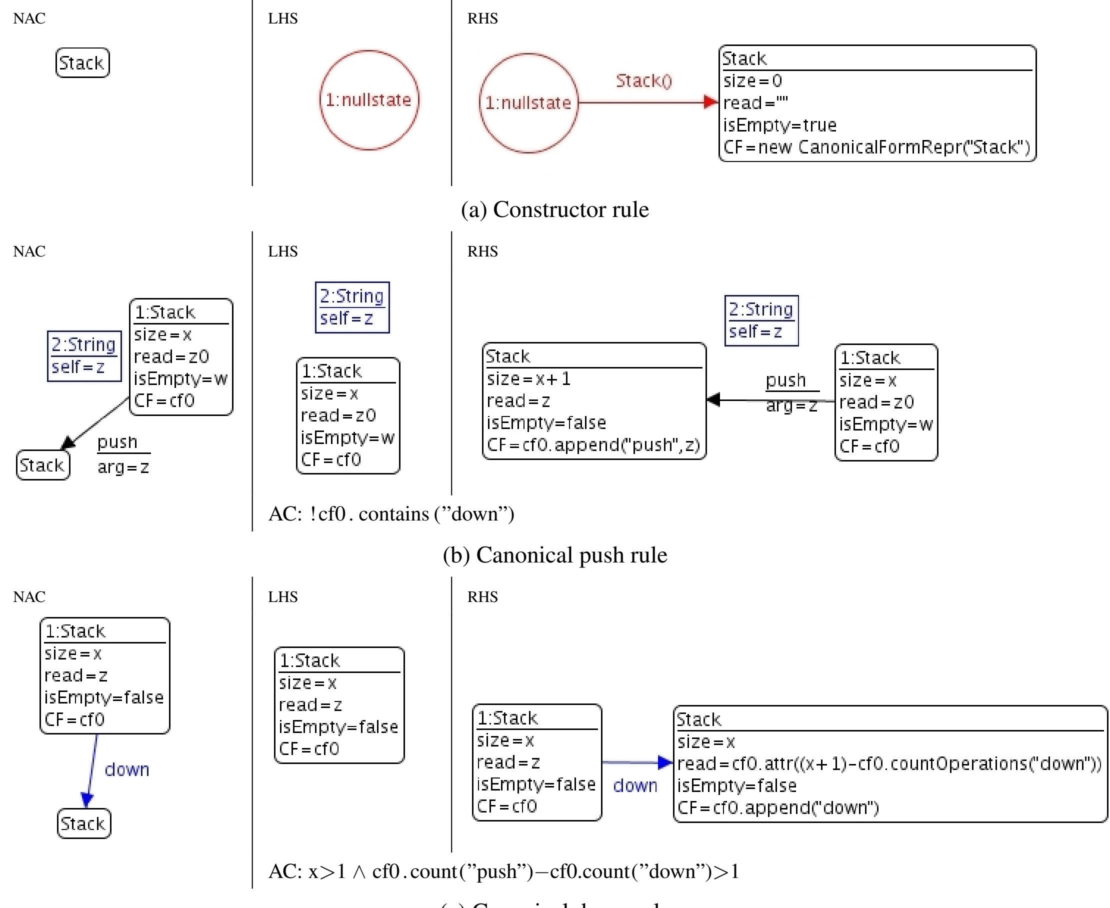


Figure 5: Partial GT specification of MTS (part I)

- $MTStack()$, that is, the empty stack after calling the empty constructor;
- $push^n\left(push^{n-1}\left(\ldots push^1\left(MTStack(), o_1\right)\ldots, o_{n-1}\right), o_n\right)$, that is, a stack after any non-empty sequence of *push*es;
- $down^m\left(down^{m-1}\left(\ldots\left(push^n\left(\ldots push^1\left(MTStack(), o_1\right)\ldots, o_n\right)\right)\ldots\right)\right), 1 \leq m < n$, that is, any non-empty and non-singleton stack with at least one down invocation but strictly less than the stack size.

For each internal state of the instances of the data abstraction, and thus for each string of the canonical form language, we can define which operations on that state are *canonical*, and those that are not. Given a state X labeled with a canonical form $\mathscr{X} \in \mathscr{L}_{MTS}$, an operation m is canonical in X if $\mathscr{X} \cdot m \in \mathscr{L}_{MTS}$.

For example, in the case of the canonical form of MTS described above, the language defines when a *push* operation has to be considered as canonical, that is, whenever no (canonical) *down* operations have been applied to the instance. Every other operation is *non-canonical*: such

operations bring the data abstraction in a state for which a corresponding different canonical string in the canonical language already exists. For example, the chosen canonical form language $\mathscr{L}_{MTS}$ implies that every *pop* operation is non-canonical.

Once the language has been identified, we can define GT rules dealing with canonical operations (explained in Section 4.3) and other rules dealing with non-canonical operations (explained in Section 4.4. For example, the language of canonical method applications defined in Section 3 for MTS can be used as a canonical form language for the specification of the data abstraction.

Since for each state of the instances there exists a corresponding string of the canonical form language, we can use it to label each state of the graph. We implemented the canonical form attribute type with an ad-hoc JAVA class that stores a list of strings, each representing an application of a canonical operation. Each string is stored together with all the actual parameters of the corresponding operation application, which can be accessed by invoking the observers on the instance of the canonical form.

4.3 Canonical Form Rules

A canonical form rule defines a state generation, i.e., it specifies how a new state in a canonical form can be generated from an existing one by applying a canonical operation. Canonical form rules share the following common template:

- The Left-Hand Side (LHS) of the rule is always composed of a single node for the data abstraction state, and a set of nodes representing the parameter's values needed for applying the canonical operation;
- The Right-Hand Side (RHS) preserves the nodes contained in the LHS and, furthermore, it adds the new canonical state and a new edge between the two data abstraction states, to represent the application of the canonical operation.

For example, the MTS requires a canonical rule for the default constructor, and other rules for the canonical applications of methods *push* and *down*.

Canonical constructor rules generate new transitions from the null state to new nodes representing the object state after the invocation of a constructor. The LHS of constructor rules is composed of the null state and a set of nodes representing the parameter values for the invocation of the constructor. The RHS adds the generated state and initializes the observer attributes. The canonical form representation of the generated node is initialized with a new value representing the application of the constructor. For example, MTS exposes only one constructor, which can be chosen as part of the canonical form. The canonical constructor rule is shown in Figure 5a.

Modifier rules differ from constructor rules just because the node in the LHS is a node representing an object state: it cannot be the null state. The node must be identified by a set of attribute conditions, which uniquely identify the correct state on which the rule must be applied.

Figure 5b shows the rule for operation *push*. Attribute conditions state that the canonical form cf0 must not contain any *down* operation. The rule introduces a new state, with *size* increased by one, the *read* observer returning the argument of the last *push* operation, and the canonical form modified by appending the *push* operation. Figure 5c shows the canonical *down* rule, which can be applied whenever the difference between the number of *push* operations and that of *down*

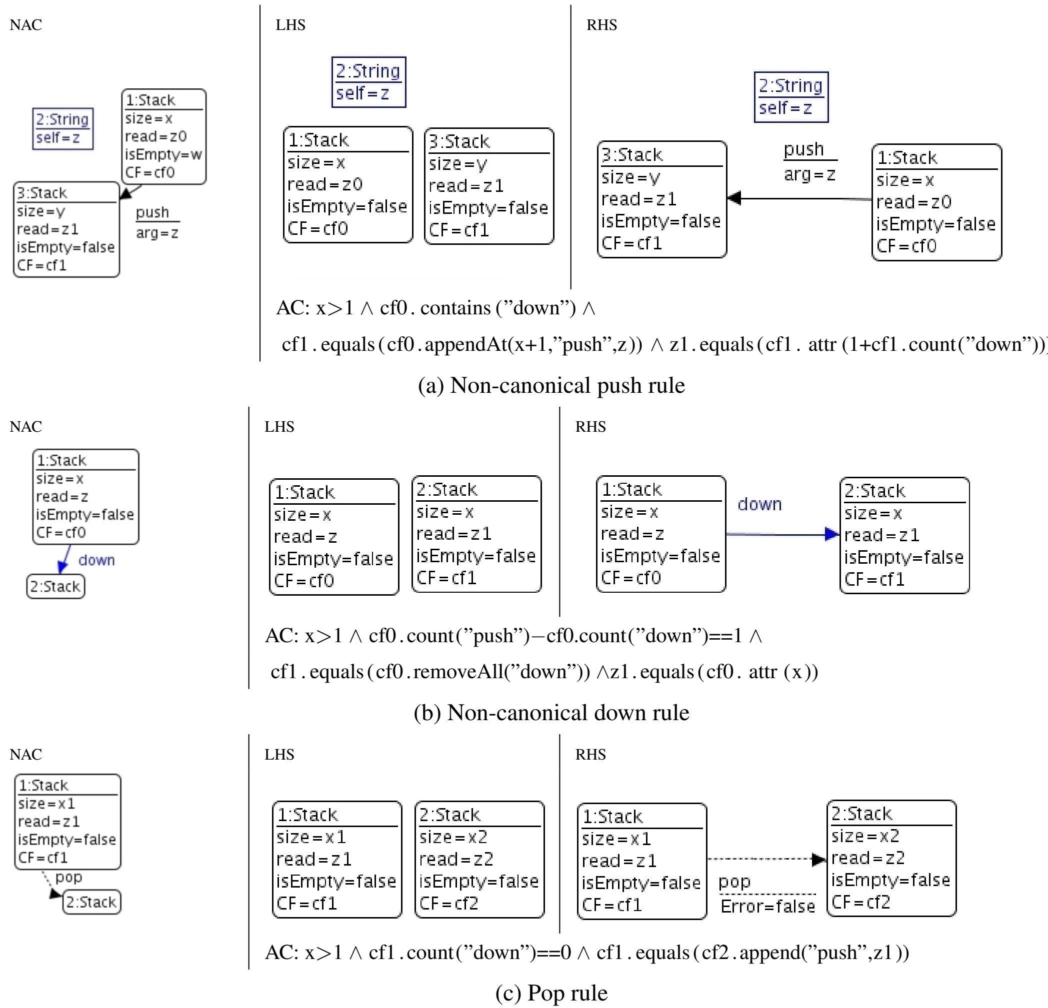


Figure 6: Partial GT specification of MTS (part II)

operations in the canonical form is greater than one. The introduced state has the same *size* as the previous and the *read* method returns the first attribute of the jth element of the normal form, where j = (x + 1) − cf0. countOperations (“down”). It is not hard to prove that with the defined canonical form language that element in the canonical form is always a *push* and that after each *down* operation the *read* method returns the corresponding element obtained by moving the hidden pointer towards the bottom of the stack. Negative Application Conditions (NACs) preserve the determinism of the generated behavior model.

4.4 Non-Canonical Method Rules

The generative approach of canonical constructors and modifiers rules defines the state space of the data abstraction. Thus, we represent non-canonical method applications as rules that add edges between existing nodes of the graph. With this approach, non-canonical method rules can

be applied iff the related states have been already generated by canonical form rules. The LHS and RHS of a non-canonical method rule must contain the two nodes, and the RHS of the rule creates the edge between them that represents the application of the method.

In the MTS case, method *pop* is not part of the canonical form. Thus, every transition corresponding to the application of a pop operation is added by non-canonical modifier rules (see Figure 6c). A more complex case regards operations *down* and *push*. In fact, these operations are not always part of a canonical form, depending on the context of the previous canonical operations applied to the object. For example, operation *down* is canonical only when applied after a sequence of $n > 1$ *push*es, starting with a constructor, and for a maximum of $n-1$ times. Otherwise, it is not canonical. Thus, a non-canonical rule (see Figure 6b) for operation *down* must be added to handle the non-canonical case. The non-canonical rule has a different context of application, represented by a different condition on attributes. In this case, the non-canonical *down* moves the hidden pointer to the top of the stack. In a similar way, the *push* operation is not canonical when applied after a *down* operation. In Veloso's specification, the operation is valid, and we can specify it with the rule of Figure 6a.

The MTS specification does not contain any example of non-canonical constructor rule. Such a rule would be similar to the case of non-canonical modifiers.

4.5 Canonical Form and Topology

In principle, attribute conditions on the canonical form could be expressed by topological patterns. For example, given the canonical push rule of Figure 5b, one can notice that the attribute condition that states that the canonical form does not contain any *down* operation could be expressed by the following informal topological constraint: there exists a path from the null state to the node on the LHS of the rule, starting with a constructor edge and composed only of push edges. Such topological conditions cannot be expressed with AGG, thus we used equivalent attribute conditions.

5 Verification

Our specification can be used to verify interesting properties of data abstraction specifications. Our verification approach has two steps: (1) AGG is used to generate a behavior model of the data abstraction; (2) the resulting model is translated into a Linear Temporal Logic (LTL) model, on which we apply SAT-solving [DP60] to model-check properties.

5.1 Model Generation

In general, model checking a specification expressed with graph transformations is not easy; several solutions have been proposed for the general case [BS06, RSV04]. However, the behavior of our rules, as expressed in the previous section, is limited: they can only add new nodes or edges to the graph. At each transformation step, the graph on which the rule is applied is left unmodified by the rule itself. For this reason, the model to be checked is an instance graph (i.e., a behavior model of the data abstraction) after a limited number of rule applications, and it can be generated by simply using AGG.

Graph transformation rules can generate infinitely many behavior models. For example, for any data abstraction with potentially infinite states, our specification approach can generate an unbounded number of graphs representing the behavior of the data abstraction. To model check the specification against certain properties we need to bound generated models by adding some attribute conditions on the canonical form rules. Non-canonical form rules are implicitly limited by the fact that they operate by only adding edges between existing nodes.

For example, suppose that we want to verify a given property on the MTS previously defined; we might choose to limit the scope of the verification to all the possible states of a MTS with $size < 4$. We can limit the application of canonical rules, and thus the size of the model, by adding this constraint as a constraint to the attributes of canonical form rules. In the case of MTS, the canonical rule to be constrained is the *push* canonical rule. If the size attribute of the node on the LHS is greater than or equal to 4, the *push* canonical rule is not applied, thus stopping the generation of new nodes.

5.2 LTL Translation

The model obtained by applying the rules of the graph grammar is translated into an LTL description which can be fed to ZOT [PMS07], together with the set of properties that should be satisfied by all the possible states of the data abstraction. ZOT produces a corresponding set of propositional formulae which can be automatically used as input for a SAT-solver.

The axioms for the logical model are built in the following way:

- We define a variable with finite domain for the state of the behavior model; the domain is a set of items defining the state in which an object of the data abstraction can be.
- Similarly, we define a set of variables for each observer attribute on each node representing a state of the object; for example, we might define a variable named *size* for the corresponding observer that can assume a set of different integer values, such as $\{0,1,2,3\}$ if we correspondingly limit the model.
- We define a variable called *do* that represents the name of the operation enabled in the transition; it can assume a set of values composed of the names of the constructors and modifiers, together with a string representation of their actual parameters.

Axioms A set of axioms is needed to characterize each state with the corresponding observer variables. Thus, the translation contains a set of implications of the following kind:

$$state = s_{1a} \Rightarrow (size = 1) \wedge (isEmpty = false) \wedge (read = a)$$

Enabled transitions can be expressed by defining axioms that denote which operations can be applied in a state. For example, the following axiom:

$$state = s_0 \quad \Rightarrow \quad (do = push_a \vee do = push_b \vee do = pop_E)$$

states that the only possible operations that can be done on the empty stack are *push* and *pop* operation, which is exceptional.

Finally, we need axioms to define the postcondition of the application of methods. For example, the axiom:

$$do = stack \Rightarrow X(state = s_0)$$

states that the next state after the invocation of the constructor is the state corresponding to the empty stack. Another example is given by the following implications:

$$(state = s_0 \land do = push_a \Rightarrow X(state = s_{1a})) \land (state = s_0 \land do = push_b \Rightarrow X(state = s_{1b}))$$

These implications define the different behavior obtained by pushing different elements on the empty stack. With all these axioms defined, we have an LTL specification on which each transition represents the application of an operation on instances of the data abstraction. For this reason, we can use the LTL operator X to predicate on which states are reached after the application of a sequence of methods.

5.3 Example Properties

Let us consider the following simple property, which states that the application of a *pop* after a *push* in any state brings the object to the same initial state:

$$\forall x \in state_vals \quad ((state = x) \land (do = push_a \lor do = push_b) \land X(do = pop) \\ \Rightarrow X^2(state = x))$$

Precisely, the property states that if we are in state x and we *push* a or b, and then we do a *pop*, the final state is again x. Another, more complex property, for the MTS is that for any state x where the size of the stack is greater than or equal to 2, the application of $n = size()$ *down* operations brings the object to state x:

$$\forall x \in state_vals, y \in \{2,3\} \quad ((state = x) \land (size = y) \land \big(\forall k (0 \leq k < y \Rightarrow X^k(do = down))\big) \\ \Rightarrow X^y(state = x))$$

We fed Zot with the LTL translation of a generated behavior model with states with $size < 4$, and checked it against these two properties. The LTL model, together with the negated properties, was found unsatisfiable in a few seconds; thus, the two properties are valid for the chosen bound.

6 Conclusions

This paper presents an approach to the specification and verification of data abstractions by using graph transformations. The generative nature of graph transformation provided a means to describe intensionally the (potentially infinite) behavior models of abstract data types, thus overcoming the limitations of plain finite-state automata. Graph transformations can be used to generate behavior models of the specified data abstraction which can be easily translated into

a finite set of axioms suitable for automatic verification, for example by using a SAT-solver. By using a generative approach, the complexity of the generated model can be tailored to the verification needs.

Our ongoing work is now on trying to infer behavior models defined through graph transformation systems by observing execution traces of existing module libraries.

Bibliography

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. (eds.), *ICGT 2006: Proceedings of 3rd International Conference on Graph Transformation*. Lecture Notes in Computer Science 4178, pp. 306–320. Springer, 2006.

[Com04] Common Framework Initiative. CASL *Ref. Manual*. LNCS 2960. Springer, 2004.

[DLWZ06] V. Dallmeier, C. Lindig, A. Wasylkowski, A. Zeller. Mining Object Behavior with ADABU. In *WODA 2006: Proceedings of 4th International Workshop on Dynamic Analysis*. May 2006.

[DP60] M. Davis, H. Putnam. A Computing Procedure for Quantification Theory. *J. ACM* 7(3):201–215, 1960.

[GH78] J. V. Guttag, J. J. Horning. The Algebraic Specification of Abstract Data Types. *Acta Informatica* 10(1), 1978.

[GH93] J. V. Guttag, J. J. Horning. *Larch: languages and tools for formal specification*. Springer-Verlag New York, Inc., New York, NY, USA, 1993.

[GTW78] J. A. Goguen, J. W. Thatcher, E. W. Wagner. An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types. In *Current Trends in Programming Methodology*. Volume 4, pp. 80–149. Prentice Hall, 1978.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *CACM* 12(10):576–580, 1969.

[LBR99] G. T. Leavens, A. L. Baker, C. Ruby. JML: A Notation for Detailed Design. In Kilov et al. (eds.), *Behavioral Specifications of Businesses and Systems*. Pp. 175–188. Kluwer Academic Publishers, 1999.

[LG00] B. Liskov, J. Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, Boston, MA, USA, 2000.

[Maj77] M. E. Majster. Limits of the "algebraic" specification of abstract data types. *ACM SIGPLAN Notices* 12(10):37–42, 1977.

[PMS07] M. Pradella, A. Morzenti, P. San Pietro. The symmetry of the past and of the future: bi-infinite time in the verification of temporal properties. In *ESEC-FSE '07: Proceedings*

of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. Pp. 312–320. ACM, New York, NY, USA, 2007.

[RSV04] A. Rensink, Á. Schmidt, D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In Ehrig et al. (eds.), *ICGT 2004: Proceedings of 2nd International Conference on Graph Transformation*. Lecture Notes in Computer Science 3256, pp. 226–241. Springer, 2004.

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *Application of Graph Transformations with Industrial Relevance*. LNCS 3062, pp. 446–456. Springer, 2004.

[Vel79] P. Veloso. Traversable stack with fewer errors. *ACM SIGPLAN Notices* 14(2):55–59, 1979.

[XMY06] T. Xie, E. Martin, H. Yuan. Automatic Extraction of Abstract-Object-State Machines from Unit-Test Executions. In *ICSE 2006: Proceedings of 28th International Conference on Software Engineering, Research Demonstrations*. Pp. 835–838. May 2006.

Parsing of Hyperedge Replacement Grammars with Graph Parser Combinators

Steffen Mazanek[1] **and Mark Minas**[2]

[1] steffen.mazanek@unibw.de
[2] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: Graph parsing is known to be computationally expensive. For this reason the construction of special-purpose parsers may be beneficial for particular graph languages. In the domain of string languages so-called parser combinators are very popular for writing efficient parsers. Inspired by this approach, we have proposed *graph parser combinators* in a recent paper, a framework for the rapid development of special-purpose graph parsers. Our basic idea has been to define primitive graph parsers for elementary graph components and a set of combinators for the flexible construction of more advanced graph parsers. Following this approach, a declarative, but also more operational description of a graph language can be given that is a parser at the same time.

In this paper we address the question how the process of writing correct parsers on top of our framework can be simplified by demonstrating the translation of *hyperedge replacement grammars* into graph parsers. The result are recursive descent parsers as known from string parsing with some additional nondeterminism.

Keywords: graph parsing, functional programming, parser combinators, hyperedge replacement grammars

1 Introduction

Graph languages are widely-used nowadays, e.g., for modeling and specification. For instance, we have specified visual languages using graph grammars [Min02]. In this context we are particularly interested in solving the membership problem, i.e., checking whether a given graph belongs to a particular graph language, and parsing, i.e., finding a corresponding derivation. However, while string parsing of context-free languages can be performed in $O(n^3)$, e.g., by using the well-known algorithm of Cocke, Younger and Kasami [Kas65], graph parsing is computationally expensive. There are even context-free graph languages the parsing of which is NP-complete [DHK97]. Thus a general-purpose graph parser cannot be expected to run in polynomial time for arbitrary grammars. The situation can be improved by imposing particular restrictions on the graph languages or grammars. Anyhow, even if a language can be parsed in polynomial time by a general-purpose parser, a special-purpose parser tailored to the language is likely to outperform it.

Unfortunately the development of a special-purpose graph parser is an error-prone and time-consuming task. The parser has to be optimized such that it is as efficient as possible, but still correct. Backtracking, for instance, has to be prevented wherever possible. Therefore, in a recent paper [MM07] we have proposed *graph parser combinators*, a new approach to graph parsing that allows the rapid construction of special-purpose graph parsers. Further we have introduced a Haskell [Pey03] library implementing this approach. It provides the generic parsing framework and a predefined set of frequently needed combinators.

In [MM07] we further have demonstrated the use of this combinator framework by providing an efficient special-purpose graph parser for VEX [CHZ95] as an example. VEX is a graph language for the representation of lambda terms. The performance gains mainly have resulted from the fact, that VEX is context-sensitive and ambiguous – properties many general-purpose graph parsers do not cope well with. The structure of VEX graphs is quite simple though, i.e., they basically are trees closely reflecting the structure of lambda terms. Only variable occurrences are not identified by names as usual; they rather have to refer to their binding explicitly by an edge. Nevertheless, the parser for VEX could be defined quite operationally as a tree traversal.

However, the operational description of languages like, e.g., structured Flowgraphs is much more difficult. Parsers get very complex and hard to read and verify. This brings up the question, whether graph parser combinators actually are powerful enough to express standard graph grammar formalisms. One such formalism are hyperedge replacement grammars [DHK97], which allow such languages to be described in a declarative and natural way.

Therefore, the main contribution of this paper is a method for the straightforward translation of hyperedge replacement grammars [DHK97] to parsers on top of our framework. The resulting parsers are readable and can be customized in a variety of ways. They are quite similar to top-down recursive descent parsers as known from string parsing where nonterminal symbols are mapped to functions. Unfortunately, in a graph setting we have to deal with additional nondeterminism: besides different productions for one and the same nonterminal, we also have to guess particular nodes occurring in the right-hand side of a production. We can use backtracking, but performance naturally suffers.

However, our approach can be used to build an initial, yet less efficient parser. Language-specific performance optimizations can then be used to improve the parser's efficiency step by step. Moreover, the presented approach offers the following benefits:

- Combination of declarative and operational description of the graph language.
- An application-specific result can be computed.[1]
- Context information can be used to describe a much broader range of languages.
- Robust against errors. The largest valid subgraph is identified.

This paper is structured as follows: We discuss the combinator approach to parsing in Sect. 2 and introduce our graph model in Sect. 3. We go on with the presentation of our framework in Sect. 4 and discuss the actual mapping of a hyperedge replacement grammar in Sect. 5. Finally, we discuss related work (Sect. 6) and conclude (Sect. 7).

[1] A general-purpose parser normally returns a derivation sequence or a parse tree, respectively. Several systems, however, provide support for attributed graph grammars.

2 Parser Combinators

Our approach has been inspired by the work of Hutton and Meijer [HM96] who have proposed monadic parser combinators for string parsing (although the idea of parser combinators actually is much older). The basic principle of such a parser combinator library is that primitive parsers are provided that can be combined into more advanced parsers using a set of powerful combinators. For example, there are the sequence and choice combinators that can be used to emulate a grammar. However, a wide range of other combinators are also possible. For instance, parser combinator libraries often include a combinator `many` that applies a given parser multiple times, while collecting the results.

Parser combinators are very popular, because they integrate seamlessly with the rest of the program and hence the full power of the host language can be used. Unlike Yacc [Joh75] no extra formalism is needed to specify the grammar. Functional languages are particularly well-suited for the implementation of combinator libraries. Here, a parser basically is a function (as we will see). A combinator like choice then is a higher-order function. Higher-order functions, i.e., functions whose parameters are functions again, support the convenient reuse of existing concepts [Hug89]. For instance, consider a function `symb` with type `Char->Parser` that constructs a parser accepting a particular symbol. Then we can easily construct a list `pl` of parsers, e.g., by defining `pl=map symb ['a'..'z']` (applies `symb` to each letter). A parser `lcl` that accepts an arbitrary lower-case letter then can be constructed by folding `pl` via the choice operator, i.e., `lcl=foldr choice fail pl`. Thereby, `fail` is the neutral element of `choice`.

At this point, we provide a toy example to give an impression of how a parser constructed with monadic combinators looks like. For that purpose we compare the implementation of a parser for the string language $\{a^k b^k c^k | k > 0\}$ and a graph parser for the corresponding language of string graphs as defined in [DHK97].

An important advantage of the combinator approach is that a more operational description of a language can be given. For instance, our exemplary language of strings $a^k b^k c^k$ is not context-free. Hence a general-purpose parser for context-free languages cannot be applied at all, although parsing this language actually is very easy: "Take as many a characters as possible, then accept the same number of b characters and finally accept the same number of c characters."

Using PolyParse [Wal07], a well-known and freely-available parser combinator library for strings, a parser for this string language can be defined as shown in Fig. 1a. The type of this parser determines that the tokens are characters and the result a number, i.e., k for a string $a^k b^k c^k$.

```
abc::Parser Char Int        abcG::Node->Grappa Int
abc =                       abcG n =
 do                          do
 as<-many1 (char 'a')        (n',as)<-chain1 (dirEdge "a") n
 let k=length as             let k=length as
 exactly k (char 'b')        (n'',_)<-exactChain k (dirEdge "b") n'
 exactly k (char 'c')        exactChain k (dirEdge "c") n''
 return k                    return k
```

Figure 1: Parsers for a) the string and b) the graph language $a^k b^k c^k$

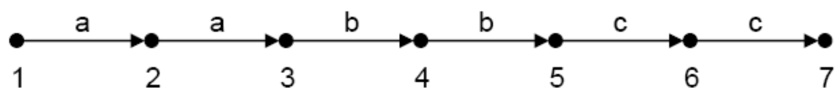


Figure 2: The string graph “aabbcc”

If the given word does not begin with a member of the language one of the calls of `exactly` fails.

The code is written in the functional programming language Haskell [Pey03]. The given parser uses the `do`-notation, syntactic sugar Haskell provides for dealing with monads. Monads in turn provide a means to simulate state in Haskell. In the context of parsers they are used to hide yet unconsumed input. Otherwise, all parsers in a sequence would have to pass this list as a parameter explicitly. Users of the library, however, do not have to know how this works in detail. They rather can use the library like a domain-specific language for parsing nicely embedded into a fully-fledged programming language.

In order to motivate our combinator approach to graph parsing, we provide the graph equivalent to the previously introduced string parser `abc`. Strings generally can be represented as directed, edge-labeled graphs straightforwardly. For instance, Fig. 2 provides the graph representation of the string “aabbcc”.[2]

A graph parser for this graph language can be defined using our combinators in a manner quite similar to the parser discussed above. It is shown in Fig. 1b. The main difference between the implementations of `abc` and `abcG` is, that we have to pass through the position, i.e., the node n we currently process.

3 Graphs

In this section we introduce hypergraphs and the basic Haskell types for their representation. Our graph model differs from standard definitions as found in, e.g., [DHK97], that do not introduce the notion of a *context*.

Let C be a set of labels and $type : C \to \mathbb{N}$ a typing function for C. In the following, a hypergraph H over C is a finite set of tuples (lab, e, ns), where e is a (hyper-)edge[3] identifier unique in H, $lab \in C$ is an edge label and ns is a sequence of node identifiers such that $type(lab) = |ns|$, the length of the sequence. The nodes represented by the node identifiers in ns are called *incident* to edge e. We call a tuple (lab, e, ns) a *context* in analogy to [Erw01].

The position of a particular node n in the sequence of nodes within the context of an edge e represents the so-called tentacle of e that n is attached to. Hence the order of nodes matters.

[2] In contrast to the string language there is a context-free hyperedge replacement grammar describing this language. However, it is quite complicated despite the simplicity of the language (cf. [DHK97]). An Earley-style parser for string generating hypergraph grammars like this is discussed in [SF04].

[3] We call hyperedges just edges and hypergraphs just graphs if it is clear from the context that we are talking about hypergraphs.

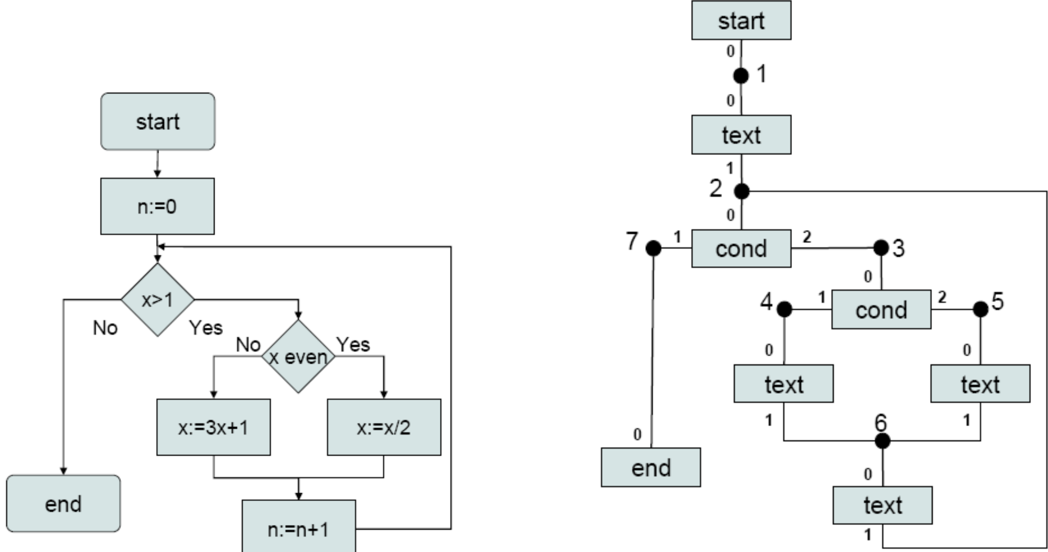


Figure 3: An exemplary Flowchart a) and its hypergraph representation b)

The same node identifier also may occur in more than one context indicating that the edges represented by those contexts are connected via this node.

Note, that our notion of hypergraphs is slightly more restrictive than the usual one, because we cannot represent isolated nodes. In particular the nodes of H are implicitly given as the union of all nodes incident to its edges. In fact, in many hypergraph application areas isolated nodes simply do not occur. For example, in the context of visual languages diagram components can be represented by hyperedges, and nodes just represent their connection points, i.e., each node is attached to at least one edge [Min02].

The following Haskell code introduces the basic data structures for representing nodes, edges and graphs altogether:

```
type Node = Int
type Edge = Int
type Tentacle = Int
type Context = (String, Edge, [Node])
type Graph = Set Context
```

For the sake of simplicity, we represent nodes and edges by integer numbers. We declare a graph as a set of contexts, where each context represents a labeled edge including its incident nodes.[4]

Throughout this paper we use Flowcharts as a running example. In Fig. 3a a structured Flowchart is given. Syntax analysis of structured Flowcharts means to identify the represented structured program (if any). Therefore, each Flowchart must have a unique entry and a unique exit point.

[4] In the actual implementation these types are parameterized and can be used more flexibly.

Flowcharts can be represented by hypergraphs that we call Flowgraphs in the following. In Fig. 3b the hypergraph representation of the exemplary Flowchart is given. Hyperedges are represented by a rectangular box marked with a particular label. For instance, the statement `n:=0` is mapped to a hyperedge labeled "text". The filled black circles represent nodes that we have additionally marked with numbers. A line between a hyperedge and a node indicates that the node is visited by that hyperedge.

The small numbers close to the hyperedges are the tentacle numbers. Without these numbers the image may be ambiguous. For instance, the tentacle with number 0 of "text" hyperedges always has to be attached to the node the previous statement ends at whereas the tentacle 1 links the statement to its successor. The Flowgraph given in Fig. 3b is represented as follows using the previous declarations:

```
fcg = {("start",0,[1]), ("text",1,[1,2]), ("cond",2,[2,7,3]),
       ("cond",3,[3,4,5]), ("text",4,[4,6]), ("text",5,[5,6]),
       ("text",6,[6,2]), ("end",7,[7])}
```

The language of Flowgraphs can be described using a hyperedge replacement grammar in a straightforward way as we see in the next section. We provide a special-purpose parser for Flowgraphs on top of our framework in Sect. 5.

4 Parsing Graphs with Combinators

In this section we introduce our graph parser combinators. However, first we clarify the notion of parsing in a graph setting.

4.1 Graph Grammars and Parsers

A widely known kind of graph grammar are hyperedge replacement grammars (HRG) as described in [DHK97]. Here, a nonterminal hyperedge of a given hypergraph is replaced by a new hypergraph that is glued to the remaining graph by fusing particular nodes. Formally, such a HRG G is a quadruple $G = (N,T,P,S)$ that consists of a set of nonterminals $N \subset C$, a set of terminals $T \subset C$ with $T \cap N = \emptyset$, a finite set of productions P and a start symbol $S \in N$.

The graph grammar for Flowgraphs can be defined as $G_{FC} = (N_{FC}, T_{FC}, P_{FC}, FC)$ where $N_{FC} = \{FC, Stmts, Stmt\}$, $T_{FC} = \{start, end, text, cond\}$ and P_{FC} contains the productions given in Fig. 4a. Left-hand side *lhs* and right-hand side *rhs* of each production are separated by the symbol $::=$ and several *rhs* of one and the same *lhs* are separated by vertical bars. Node numbers are used to identify corresponding nodes of *lhs* and *rhs*.

The derivation tree of our exemplary Flowgraph as introduced in Fig. 3b is given in Fig. 4b. Its leaves are the terminal edges occurring in the graph whereas its inner nodes are marked with nonterminal edges indicating the application of a production. The direct descendants of an inner node represent the edges occurring in the *rhs* of the applied production. The numbers in parentheses thereby identify the nodes visited by the particular edge.

A general-purpose graph parser for HRGs gets passed a particular HRG and a graph as parameters and constructs a derivation tree of this graph according to the grammar. This can be

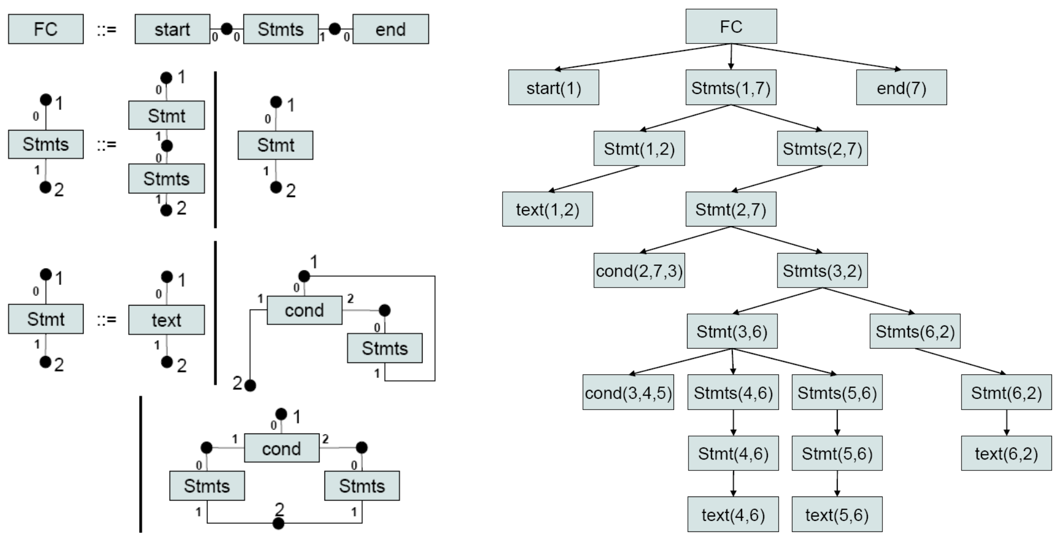


Figure 4: Flowgraphs, a) grammar and b) derivation tree of the example

done, for instance, in a way similar to the well-known algorithm of Cocke, Younger and Kasami [Kas65] known from string parsing (indeed, all HRGs can be transformed to the graph equivalent of the string notion Chomsky Normal Form). This approach has been elaborated theoretically by Lautemann [Lau89] and proven to be useful in practical applications, e.g., in [Min02] for the syntax analysis of diagrams.

Flowgraphs can be parsed with such a general-purpose graph parser in a straightforward way. However, as mentioned in the introduction there are graph languages that are not context-free (and thus cannot be described by a HRG) or that are highly ambiguous (thus causing most general-purpose parsers to perform poorly). Furthermore, here we are not interested in the derivation tree, but rather in the program represented by the graph, i.e., its semantics. For these reasons graph parser combinators are beneficial either way. We now briefly introduce the framework and describe how the HRG of Flowgraphs (and other HRGs similarly) can be translated into a graph parser on top of our framework.

4.2 The Combinator Library

Due to space restrictions in the following we focus on those types and functions that are needed to translate hyperedge replacement grammars schematically. Further information and a more general version of the framework can be found in [MM07]. First we provide the declaration of the type `Grappa` representing a graph parser:

```
newtype Grappa res = P (Graph -> (Either res Error, Graph))
```

This type is parameterized over the type `res` of the result. Graph parsers basically are functions from graphs to pairs consisting of the parsing result (or an error message, respectively) and the graph that remains after successful parser application.

Name	Type	Description
context	(Context->Bool)-> Grappa Context	A context satisfying a particular condition.
labContext	String->Grappa Context	A context with a particular label.
connLabContext	String->[(Tentacle,Node)]-> Grappa Context	A labeled context connected to the given nodes via the given tentacles.
edge	Tentacle->Tentacle->String-> Node->Grappa (Node,Context)	A labeled context connected to the given node via a particular tentacle also returning its successor (via the other, outgoing tentacle).
dirEdge	String->Node-> Grappa (Node,Context)	A directed edge, edge 0 1.

Table 1: Graph-specific primitive parsers

Name	Type	Description
oneOf	[Grappa res]->Grappa res	Returns the first successful parser of the input list, corresponds to	in grammars.
chain	(Node->Grappa (Node, res))-> Node->Grappa (Node, [res])	A chain of graphs, a node is passed through.
bestNode	(Node->Grappa res)-> Grappa res	Identifies the node from which the best continuation is possible, very expensive.
noDanglEdgeAt	Node->Grappa ()	Succeeds if the given node is not incident to an edge, handy for ensuring dangling edge condition.
allDifferent	[Node]->Grappa ()	Succeeds if the given nodes are distinct, handy for ensuring identification condition.
connComp	Grappa res->Grappa [res]	Applies the given parser once per connected component, while collecting the results.

Table 2: Some graph parser combinators

In general, the most primitive parsers are `return` and `fail`. Both do not consume any input. Rather `return` succeeds unconditionally with a particular result whereas `fail` always fails; thereby, backtracking is initiated.

In Table 1 we provide some important graph-specific primitive parsers. They are all nondeterministic, i.e., support backtracking on failure, and consume the context they return. Additionally we provide the primitive parser `aNode::Grappa Node` that returns a node of the remaining graph (with backtracking).

In Table 2 we briefly sketch some of the graph parser combinators provided by our library. Variations of `chain` have already been used in the introductory example, e.g., `chain1` that demands at least one occurrence.

```
fc::Grappa Program
fc = do
     (_,_,[n1])<-labContext "start"
     (_,_,[n2])<-labContext "end"
     stmts (n1,n2)

stmts::(Node, Node)->Grappa Program
stmts (n1, n2) = oneOf [stmts1, stmts2]
  where stmts1 = do
                 s<-stmt (n1, n2)
                 return [s]
        stmts2 = do
                 n'<-aNode
                 s<-stmt (n1, n')
                 p<-stmts (n', n2)
                 return (s:p)

stmt::(Node, Node)->Grappa Stmt
stmt (n1, n2) = oneOf [stmt1, stmt2, stmt3]
  where stmt1 = do
                connLabContext "text" [(0,n1),(1,n2)]
                return Text
        stmt2 = do
                (_,_,ns)<-connLabContext "cond" [(0,n1)]
                p1<-stmts ((ns!!1), n2)
                p2<-stmts ((ns!!2), n2)
                return (IfElse p1 p2)
        stmt3 = do
                (_,_,ns)<-connLabContext "cond" [(0,n1),(1,n2)]
                p<-stmts ((ns!!2), n1)
                return (While p)
```

Figure 5: A parser for Flowgraphs

5 Parsing Flowgraphs

In this section we directly translate the grammar given in Fig. 4a to a parser for the corresponding language using our framework. Our goal is to map a Flowgraph to its underlying program represented by the recursively defined type `Program`:

```
type Program = [Stmt]
data Stmt = Text | IfElse Program Program | While Program
```

In Fig. 5 the parser for Flowgraphs is presented. It is not optimized with respect to performance. Rather it is written in a way that makes the translation of the HRG explicit. For each nonterminal edge label l we have defined a parser function that takes a tuple of nodes $(n_1,\ldots,n_t)$ as a parameter such that $t = type(l)$. Several *rhs* of a production are handled using the `oneOf`

combinator. Terminal edges are matched and consumed using primitive parsers. Thereby their proper embedding has to be ensured. We use the standard list operator `(!!)` to extract the node visited via a particular tentacle from a node list `ns`.

For instance, `stmt3` represents the while-production. First, a "cond"-edge e visiting the nodes n_1 and n_2 via the tentacles 0 and 1, respectively, is matched and consumed. Thereafter the body of the loop is parsed, i.e., the `stmts` starting at the node visited by tentacle 2 of e, i.e., `ns!!2`, ending again at n_1. Finally the result is constructed and returned. If something goes wrong and backtracking becomes necessary, previously consumed input is released automatically.

The parser is quite robust. For instance, redundant components are just ignored and both the dangling and the identification condition are not enforced. These relaxations can be canceled easily – the first one by adding the primitive parser `eoi` (end of input) to the end of the definition of the top-level parser, the others by applying the combinators `noDanglEdgeAt` and `allDifferent`, respectively, to the nodes involved.

Note, that the implementation of `stmts` follows a common pattern, i.e., a chain of graphs between two given nodes. So using a combinator the parser declaration can be further simplified to `stmts=chain1Betw stmt`. Here, `chain1Betw` ensures at least one occurrence as required by the language. Its signature is

```
chain1Betw::((Node,Node)->Grappa a)->(Node,Node)->Grappa [a]
```

and it is defined exactly as `stmts` except from the fact that it abstracts from the actual parser for the partial graphs.

Performance

This parser is not very efficient. A major source of inefficiency is the use of `aNode` that binds a yet unknown node arbitrarily thus causing a lot of backtracking. This expensive operation has to be used only for the translation of those productions, where inner nodes within the *rhs* are not incident to terminal edges visiting an external node, i.e., a node also occurring in the *lhs*.[5] However, even so there are several possibilities for improvement. For instance, we currently try to make this search more targeted by the use of narrowing techniques as known from functional-logic programming languages [Han07]. Performance can be further improved if particular branches of the search space can be cut. For instance, we can prevent backtracking by committing to a (partial) result. In [MM07] we have demonstrated how this can be done in our framework. Finally, we can apply domain-specific techniques to further improve the performance. For instance, a basic improvement would be to first decompose the given graph into connected components and apply the parser to each of them successively. We provide the combinator `connComp` for this task. However, this step can only be applied to certain languages and at the expense of readability.

So we can start with an easy to build and read parser for a broad range of languages. It may be less efficient, however, it can be improved step by step if necessary. Further it can be integrated and reused very flexibly, since it is a first-class object.

[5] The function `aNode` can also be used to identify the start node in our introductory example `abcG`.

6 Related Work

Our parser combinator framework basically is an adaptation of the PolyParse library [Wal07]. The main distinguishing characteristics of PolyParse are that backtracking is the default behavior except where explicitly disallowed and that parsers can be written using monads. There is an abundance of other parser combinator libraries besides PolyParse that we cannot discuss here. However, a particularly interesting one is the UU parser combinator library of Utrecht University [SA99]. It is highly sophisticated and powerful, but harder to learn for a user. Its key benefit is its support for error correction. Hence a parser does not fail, but a sequence of correction steps is constructed instead.

Approaches to parsing of particular, restricted kinds of graph grammar formalisms are also related. For instance, in [SF04] an Earley parser for string generating graph languages has been proposed. The diagram editor generator DiaGen [Min02] incorporates an HRG parser that is an adaptation of the algorithm of Cocke, Younger and Kasami. And the Visual Language Compiler-Compiler VLCC [CLOT97] is based on the methodology of positional grammars that allows to parse restricted kinds of flex grammars (which are essentially HRGs) even in linear time. These approaches have in common that a restricted graph grammar formalism can be parsed efficiently. However, they cannot be generalized straightforwardly to a broader range of languages like our combinators.

We have demonstrated that semantics can be added very flexibly in our framework. The graph transformation system AGG also provides a flexible attribution concept. Here, graphs can be attributed by arbitrary Java objects [Tae03]. Rules can be attributed with Java expressions allowing complex computations during the transformation process. AGG does not deal with hypergraphs. However, it can deal with a broad range of graph grammars. These are given as so-called parse grammars directly deconstructing the input graph. Critical pair analysis is used to organize reverse rule application.

In [RS95] a parsing algorithm for context-sensitive graph grammars with a top-down and a bottom-up phase is discussed. Thereby first a set of eventually useful production applications is constructed bottom-up. Thereafter viable derivations from this set are computed top-down. Parser combinators generally follow a top-down approach, although in a graph setting bottom-up elements are beneficial from a performance point of view.

Finally there are other approaches that aim at the combination of functional programming and graph transformation. Schneider, for instance, currently prepares a textbook that provides an implementation of the categorical approach to graph transformation with Haskell [Sch07]. Since graphs are a category, a higher level of abstraction is used to implement graph transformation algorithms. An even more general framework is provided in [KS00]. The benefit of their approach is its generality since it just depends on categories with certain properties. However, up to now parsing is not considered.

7 Concluding Remarks

In this paper we have discussed graph parser combinators, an extensible framework supporting the flexible construction of special-purpose graph parsers even for context-sensitive graph grammars. It already provides combinators for the parsing of several frequently occurring graph patterns. We even may end with a comprehensive collection of reusable parser components.

Parser combinators are best used to describe a language in an operational way. For instance, we have provided a parser for the graph language $a^k b^k c^k$ as a toy example. Similar situations, however, also appear in practical applications as, e.g., discussed in [Kör07]. We further have provided a schema for the straightforward translation of hyperedge replacement grammars into a parser on top of our framework. The resulting parser is not efficient. It is rather a proof of concept. Languages like our exemplary Flowgraphs can be parsed very efficiently using a standard bottom-up parser. However, the main benefit of our framework is that language-specific optimizations can be incorporated easily in existing parsers, e.g., by providing additional information, using special-purpose combinators, heuristics or even a bottom-up pass simplifying the graph.

Parsing generally is known to be an area functional languages excel in. In the context of string parsing a broad range of different approaches have been discussed. However, in particular the popular combinator approach has not been applied to graph parsing yet. With the implementation of our library we have demonstrated that graph parser combinators are possible and beneficial for the rapid development of special-purpose graph parsers.

Future work

Our approach is not restricted to functional languages though. For instance, in [AD01] the translation of string parser combinators to the object-oriented programming language Java is described. We plan to adapt this approach in the future to, e.g., integrate graph parser combinators into the diagram editor generator DiaGen [Min02]. This hopefully will allow the convenient description of even more visual languages.

The parsers presented in this paper suffer from the fact that purely functional languages are not particularly dedicated to deal with incomplete information. For instance, we have discussed why inner nodes occurring in the right-hand sides of productions have to be guessed. Multi-paradigm declarative languages [Han07] like Curry [Han] are well-suited for such kinds of problems. We currently reimplement our library in a functional-logic style to overcome these limitations. This work will also clarify the relation to proof search in linear logic [Gir87]. Here, the edges of a hypergraph can be mapped to facts that can be connected to a parser via so-called linear implication ($\multimap$). During the proof the parser consumes these facts and at the end none of them must be left.

We further plan to investigate error correction-strategies in a graph setting. For instance, in the context of visual language editors based on graph grammars this would allow for powerful content assist. Whereas in a string setting error-correcting parser combinators are well-understood already [SA99], not much has been done with respect to graphs yet. Admittedly, we do not expect to find an efficient solution to this problem.

Bibliography

[AD01] D. S. S. Atze Dijkstra. Lazy Functional Parser Combinators in Java. Technical report UU-CS-2001-18, Department of Information and Computing Sciences, Utrecht University, 2001.

[CHZ95] W. Citrin, R. Hall, B. Zorn. Programming with Visual Expressions. In Haarslev (ed.), *Proc. 11th IEEE Symp. Vis. Lang.* Pp. 294–301. IEEE Computer Soc. Press, 5–9 1995.

[CLOT97] G. Costagliola, A. D. Lucia, S. Orefice, G. Tortora. A Parsing Methodology for the Implementation of Visual Systems. *IEEE Trans. Softw. Eng.* 23(12):777–799, 1997.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*. Chapter 2, pp. 95–162. World Scientific, 1997.

[Erw01] M. Erwig. Inductive graphs and functional graph algorithms. *J. Funct. Program.* 11(5):467–492, 2001.

[Gir87] J.-Y. Girard. Linear Logic. *Theoretical Computer Science* 50:1–102, 1987.

[Han] Hanus, M. (ed.). Curry: An Integrated Functional Logic Language. Available at `http://www.informatik.uni-kiel.de/~curry/`.

[Han07] M. Hanus. Multi-paradigm Declarative Languages. In *Proceedings of the International Conference on Logic Programming (ICLP 2007)*. Pp. 45–75. Springer LNCS 4670, 2007.

[HM96] G. Hutton, E. Meijer. Monadic Parser Combinators. Technical report NOTTCS-TR-96-4, Department of Computer Science, University of Nottingham, 1996.

[Hug89] J. Hughes. Why functional programming matters. *Comput. J.* 32(2):98–107, 1989.

[Joh75] S. C. Johnson. Yacc: Yet Another Compiler Compiler. Technical report 32, Bell Laboratories, Murray Hill, New Jersey, 1975.

[Kas65] T. Kasami. An efficient recognition and syntax analysis algorithm for context free languages. Scientific report AF CRL-65-758, Air Force Cambridge Research Laboratory, Bedford, Massachussetts, 1965.

[Kör07] A. Körtgen. Modeling Successively Connected Repetitive Subgraphs. In *Applications of Graph Transformations with Industrial Relevance: International Workshop, AGTIVE'07, Kassel, Germany, October 2007. Proceedings*. LNCS, 2007. to appear.

[KS00] W. Kahl, G. Schmidt. Exploring (finite) Relation Algebras using Tools written in Haskell. Technical report 2000-02, Fakultät für Informatik, Universität der Bundeswehr, München, 2000.

[Lau89] C. Lautemann. The Complexity of Graph Languages Generated by Hyperedge Replacement. *Acta Inf.* 27(5):399–421, 1989.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[MM07] S. Mazanek, M. Minas. Graph Parser Combinators. 2007. To appear in Proc. of 19th Internat. Symp. on the Impl. and Appl. of Functional Languages.
http://www.unibw.de/steffen.mazanek/dateien/ifl2007

[Pey03] S. Peyton Jones. *Haskell 98 Language and Libraries. The Revised Report.* Cambridge University Press, 2003.

[RS95] J. Rekers, A. Schürr. A parsing algorithm for context sensitive graph grammars. Technical report 95-05, Leiden University, 1995.

[SA99] S. D. Swierstra, P. R. Azero Alcocer. Fast, Error Correcting Parser Combinators: a Short Tutorial. In Pavelka et al. (eds.), *26th Seminar on Current Trends in Theory and Practice of Inform.* LNCS 1725, pp. 111–129. 1999.

[Sch07] H. J. Schneider. Graph Transformations - An Introduction to the Categorical Approach. 2007. `http://www2.cs.fau.de/~schneide/gtbook/`.

[SF04] S. Seifert, I. Fischer. Parsing String Generating Hypergraph Grammars. In Ehrig et al. (eds.), *Graph Transformations*. Lecture Notes In Computer Science 3256, pp. 352–267. Springer, 2004.

[Tae03] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *AGTIVE*. Lecture Notes in Computer Science 3062, pp. 446–453. Springer, 2003.

[Wal07] M. Wallace. PolyParse. 2007. `http://www.cs.york.ac.uk/fp/polyparse/`.

Visual Design and Reasoning with the Use of Hypergraph Transformations

Ewa Grabska[1], **Grażyna Ślusarczyk**[2] **and Truong Lan Le**[3]

[1] uigrabsk@cyf-kr.edu.pl
[2] gslusarc@uj.edu.pl
The Faculty of Physics, Astronomy and Applied Computer Science
Jagiellonian University, Kraków, Poland

[3] lan@ippt.gov.pl
Polish-Japanese Institute of Information Technology, Warszawa, Poland

Abstract: This paper deals with visual design and reasoning. A visual language with its internal representation in the form of attributed hierarchical hypergraphs is discussed. Hypergraph attributes allow for defining and analysing constraints imposed by design knowledge. Operations on hypergraphs which reflect modifications of design diagrams are also presented. The approach is illustrated by examples of designing floor-layouts.

Keywords: visual design, graph transformations, hypergraphs, knowledge-based reasoning

1 Introduction

This paper describes a knowledge-based decision support design system where designs are configurations of visual elements. Both partial and complete design solutions are represented in the form of diagrams forming a specific visual language, called the layout language. The syntactic knowledge of this language is defined by means of attributed hierarchical hypergraphs.

The proposed approach constitutes an attempt to solve the problem of transforming the visual design knowledge into a computer internal representation for a system which is to support the designer in the early stage of the design process [GGLŁŚ, GŚG]. The compatibility between the proposed visual language and the internal representation that serves as a base for reasoning about designs is discussed.

In our approach each diagram drawn by the designer is represented in the form of an attributed hierarchical hypergraph. Hyperedges of hypergraphs represent both diagram components and the multi-argument relations among them. Hierarchical hyperedges correspond to groups of diagram components. Hypergraphs nested in these hyperedges correspond to subcomponents of the diagram parts. Hierarchical hypergraphs not only reflect the top-down way of designing but also enable the designer to consider the project on the specified level of detail and allow one to express relations between components on different hierarchy levels. Attributes assigned to hyperedges encode the semantic design knowledge.

Diagram modifications made by the user are reflected in the hypergraph representation, which is correspondingly changed using operations defined on hypergraphs. When the designer decides

to divide a chosen area of the layout into smaller parts the operation called hyperedge development [GŚG], is used. A reverse operation, called hyperedge suppression, which allows one to redesign the chosen area, is defined in this paper.

The paper strongly recommends the designer to interact with the computer system on the level of a visual language. The knowledge stored in the attributed hypergraph representation of diagrams allows the system to reason about layouts and to support the designer by suggesting further steps and prevent him/her from creating designs not compatible with the specified constraints or criteria. The approach is illustrated by examples of designing floor-layouts.

2 Related work

This paper deals with supporting conceptual design phase by a knowledge-based system. Nowadays detailed design and design documentation phases are usually well supported in CAD tools such as ArchiCAD, Architectural Desktop and AllPlan, etc. [Szu]. Although there are many computational tools for describing, editing, analyzing, and evaluating design projects [Min], there still exists lack of consistency between knowledge visualization of the given domain (architecture, construction, machine building) and its internal representation in a computer program. This is one of the reasons that the initial design phase, called conceptual design, is the least supported one.

Our approach proposes a visual representation of designs together with their internal representation. Such a method allows one to equip the design editor with intelligent assistants supporting creative design by reasoning about the project features and suggesting design modifications. When developing a visual language the role of sketches in the conceptual design phase was taken into consideration [Gol]. Therefore elements of our language contain general ideas about design objects and are not treated as completed design components used in design visualizations in all CAD tools.

Graphs and hierarchical structures are used quite frequently in knowledge-based design tools [SWZ]. Our approach is based on a formal model of hierarchical hypergraphs introduced in [DHP] and extended in [Ślu]. This type of hypergraphs enables us to express multi-argument relations between elements on different hierarchy levels, which is essential in design. Our hypergraphs can be treated also as an extension of hypergraphs used by [Min], which are too restrictive in expressing relations.

3 Design Diagrams and Attributed Hierarchical Hypergraphs

In this section we present an example of a visual language, called a layout language, which enables the designer to create and edit floor-layouts. A vocabulary of this language is composed of shapes corresponding to components like rooms, walls, doors, windows, while the rules specifying possible arrangements of these components constitute its syntactic knowledge. Elements of the layout language are diagrams seen as simplified architectural drawings.

Let us consider a floor-layout presented in Figure 1(a) and its design diagram shown in Figure 1(b). A design diagram is composed of polygons which are placed in an orthogonal grid. These polygons represent components of a floor-layout, like functional areas or rooms. Mutual

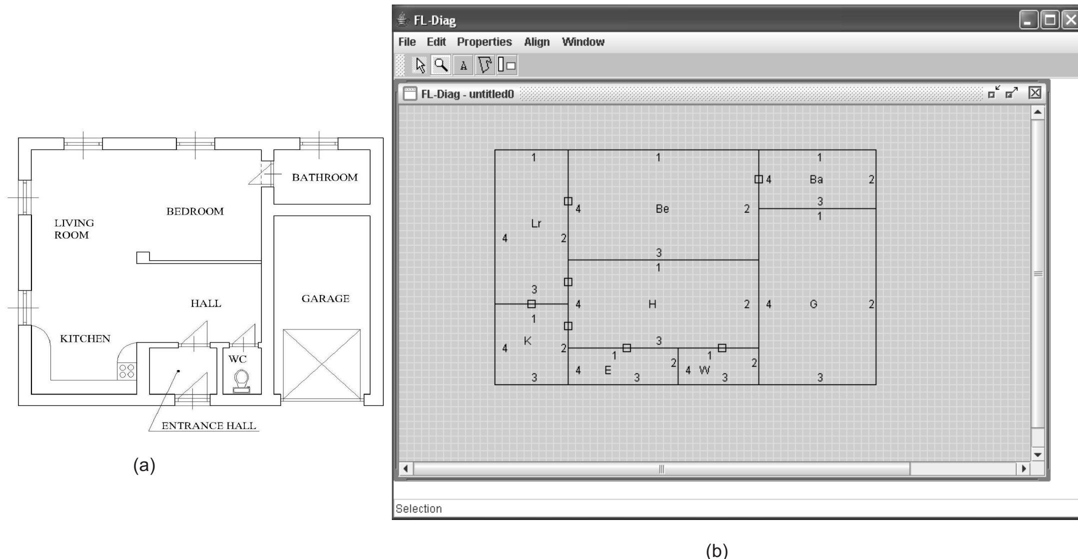


Figure 1: a) An architectural drawing, b) a design diagram

location of polygons is determined by design criteria. Lines with small squares on them represent the accessibility relation among components, while continuous lines shared by polygons denote the adjacency relations between them. The accessibility relation between two areas is specified when the existence of a wall with doors, a fragment of a wall or lack of a wall are planned in a floor-layout design. When a wall dividing two areas is not planned, the location of a line representing the accessibility relation is determined by the specified sizes of the areas. The sides of each polygon are ordered clock-wise starting from the top left-most one. In a design diagram only qualitative coordinates are used i.e., only relations among graphical elements (walls) are essential.

Each design diagram has its internal representation in the form of an attributed hierarchical layout hypergraph. Such a hypergraph contains two types of hyperedges, which can represent components and spatial relations on different levels of details. Hyperedges of the first type are non-directed and correspond to layout components. Hyperedges of the second type represent relations among components and can be either directed or non-directed in the case of symmetrical relations. Considering floor-layout design only spatial relations which are by nature symmetrical (accessibility and adjacency) are taken into account.

An example of the internal representation of the diagram presented in Figure 1(b) is shown in Figure 2. This hypergraph is composed of eleven component hyperedges, three of which are hierarchical ones, and fourteen relational hyperedges, where half of them represent the accessibility relation and the other half the adjacency relation. The relational hyperedge connecting node 2.2 of the hyperedge labelled *Lr* and node 4.6 of the hyperedge labelled *Be* is one of the hyperedges expressing relations between components nested in different parent hyperedges.

To represent features of layout components and relations between them attributing of nodes and hyperedges is used. Attributes represent properties (like shape, size, position, number of windows or doors) of elements corresponding to hyperedges and nodes.

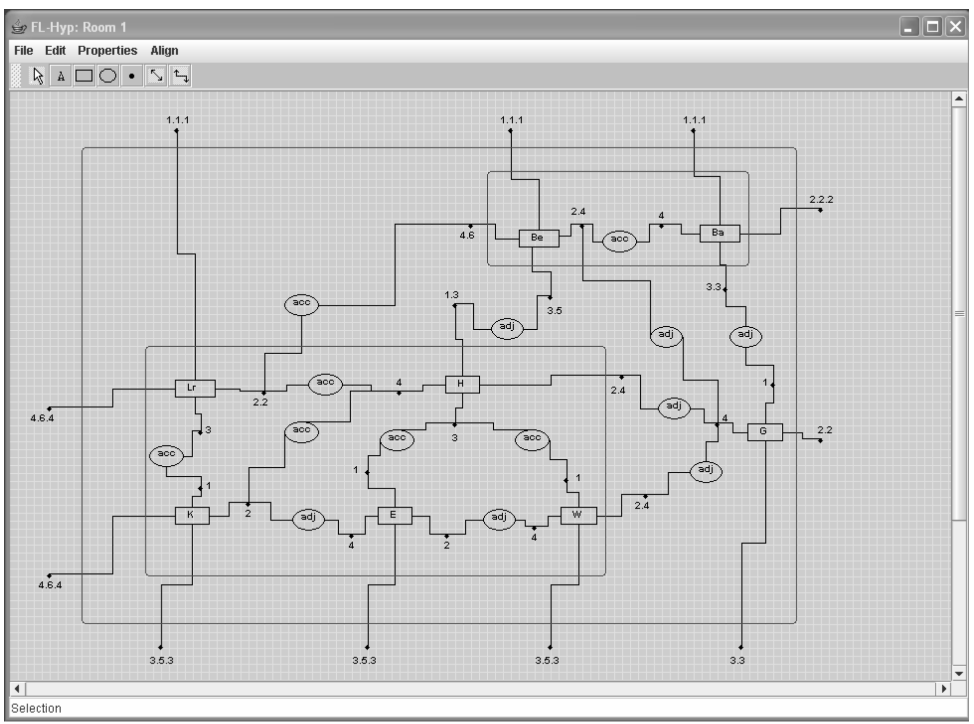


Figure 2: A hierarchical layout hypergraph corresponding a design diagram from Figure 1(b)

The proposed hierarchical layout hypergraph constitutes a modification of hypergraphs presented in [Min, Ślu] and is defined as follows.

Let $[i]$ denote the interval $[1,i]$ of natural numbers (with $[0] = \emptyset$) and let $\Sigma = \Sigma_E \cup \Sigma_V$, where $\Sigma_E \cap \Sigma_V = \emptyset$, be a fixed alphabet of hyperedge and node labels, respectively. Let At be a set of hyperedge and node attributes.

Definition 1 An attributed hierarchical layout hypergraph over $\Sigma = \Sigma_E \cup \Sigma_V$ and At is a system $G = (E_G, V_G, s_G, t_G, lb_G, att_G, ext_G, ch_G), where:$

1. $E_G = E_G^C \cup E_G^R$ is a nonempty finite set of hyperedges, where elements of E_G^C represent object components, while elements of E_G^R represent relations and $E_G^C \cap E_G^R = \emptyset$,
2. V_G is a nonempty finite set of nodes,
3. $s_G : E_G \longrightarrow (V_G)^*$ and $t_G : E_G \longrightarrow (V_G)^*$ are two mappings assigning to hyperedges sequences of source and target nodes respectively, in such a way that $\forall e \in E_G^C \; s_G(e) = t_G(e)$,
4. $lb_G = lb_{E_G} \cup lb_{V_G}$, where:
   - $lb_{E_G} : E_G \longrightarrow \Sigma_E$ is a hyperedge labelling function, such that $\Sigma_E = \Sigma_E^C \cup \Sigma_E^R \wedge \Sigma_E^C \cap \Sigma_E^R = \emptyset \wedge \forall e \in E_G^C \; lb_{E_G}(e) \in \Sigma_E^C \wedge \forall e \in E_G^R \; lb_{E_G}(e) \in \Sigma_E^R$,
   - $lb_{V_G} : V_G \longrightarrow \Sigma_V$ is a node labelling function,

5. $att_G = att_{E_G} \cup att_{V_G}$, where:
   - $att_{E_G} : E_G \to P(At)$ is a hyperedge attributing function,
   - $att_{V_G} : V_G \to P(At)$ is a node attributing function,
6. $ext_G : [n] \to V_G$ is a mapping specifying a sequence of hypergraph external nodes,
7. $ch_G : E_G^C \to P(A)$ is a child nesting function, where $A = E_G \cup V_G$ is called a set of hypergraph atoms, and such that one atom cannot be nested in two different hyperedges, a hyperedge cannot be its own child, source and target nodes of a nested hyperedge e are nested in the same hyperedge as e.

Hyperedges of the layout hypergraph are labelled by names of components or relations. A sequence of source and target nodes is assigned to each hyperedge and express potential connections to other hyperedges. Moreover, for each hierarchical hypergraph a sequence of external nodes is determined. A child nesting function ensures that one atom cannot be embedded in two different hyperedges. It also guarantees that there are no ancestor-descendant cycles in a hypergraph, which means that a hyperedge cannot be its own child.

4 Operations on Hierarchical Layout Hypergraphs

It is known that a design process cannot be a priori defined in an algorithmic way. During a design process the designer often modifies a design diagram and/or changes design goals before he gets a plausible solution. To reflect these changes in our internal representation of a visual language we equip the proposed system with operations acting on hierarchical layout hypergraphs. They allow to create and modify hypergraphs representing structures of objects being designed.

The *hyperedge development* operation, which enables to represent a model structure on a more detailed level, was defined in [GŚG]. It takes two hierarchical hypergraphs as arguments and nests in a hyperedge corresponding to an object element in the first hypergraph the second hypergraph representing the components of this element and relations among them. The nodes connected to a developed hyperedge are substituted by the corresponding external nodes of the child hypergraph.

Now we define in a formal way an operation called a *hyperedge suppression*, which is a reverse to a development operation. It is useful when the designer wants to redesign the chosen area. Then he/she has to remove the existing area division before dividing it in a different way. A suppression operation takes as an argument a hierarchical layout hypergraph and removes the nested contents of one of its earlier developed component hyperedges.

Let us consider Figure 2 and Figure 3. Figure 3 presents the result of applying a suppression operation consisting in removing a hypergraph H nested in the hyperedge $\widetilde{e}$ labelled L corresponding to the living area (Figure 2).

Let $S_G(e)$ and $T_G(e)$ denote sets of all nodes specified by sequences of source and target nodes of a hyperedge e in a given hierarchical layout hypegraph G.

Definition 2 Let G be an attributed hierarchical layout hypergraph over $\Sigma = \Sigma_E \cup \Sigma_V$ and At, and $\widetilde{e} \in E_G^C$ be a component hyperedge of G such that there exists a hierarchical layout hypergraph

H over Σ and At, and $ch^+_G(\widetilde{e}) = E_H \cup V_H$, where ch^+_G denotes the transitive closure of ch_G.
Let EXT_H denote a set of external nodes of H, V denote a finite set of nodes such that $V \cap V_G = \emptyset$, E denote a set of all relational hyperedges of $E^R_G \setminus E^R_H$ and connected with nodes of EXT_H, and E' denote a finite set of relational hyperedges such that $E' \cap E_G = \emptyset$.
Let a function $substr(string_1, string_2)$ return a substring of $string_1$ without the prefix $string_2$.

The hyperedge suppression operation is defined by three functions:

1. a suppression function $sup : EXT_H \to V$ defined in such a way that
$\forall v, w \in EXT_H \; substr(lb_G(v), |lb_H(v)|) = substr(lb_G(w), |lb_H(w)|) \Rightarrow sup(v) = sup(w)$,
is a surjection which determines the correspondence between a set of new nodes and external nodes of H assigning one node to all nodes with labels which differ only by a prefix number.

2. a suppression labelling function $lb_{sup} : V \to \Sigma_V$ defined in such a way that
$\forall v \in V \; lb_{sup}(v) = substr(lb_G(w), |lb_H(w)|)$, where $w \in sup^{-1}(v)$,
is a mapping assigning to new nodes labels obtained by removing prefixes from labels of the corresponding external nodes of H.

3. a suppression embedding function $emb_{sup} : E \to E'$ is a surjection which determines the correspondence between a set of new relational hyperedges and relational hyperedges connected with external nodes of H.

The result of the hyperedge suppression operation is an attributed hierarchical layout hypergraph $\widetilde{G} = (E_{\widetilde{G}}, V_{\widetilde{G}}, s_{\widetilde{G}}, t_{\widetilde{G}}, lb_{\widetilde{G}}, att_{\widetilde{G}}, ext_{\widetilde{G}}, ch_{\widetilde{G}})$ over Σ and At, where:

1. $E_{\widetilde{G}} = (E_G \setminus E_H \setminus E) \cup E'$,
2. $V_{\widetilde{G}} = (V_G \setminus V_H) \cup V$,
3. $s_{\widetilde{G}} : E_{\widetilde{G}} \to (V_{\widetilde{G}})^*$ and $t_{\widetilde{G}} : E_{\widetilde{G}} \to (V_{\widetilde{G}})^*$ are defined in such a way that:
   - $s_{\widetilde{G}}(\widetilde{e}) = t_{\widetilde{G}}(\widetilde{e}) \subseteq \widetilde{V}^*$, where $\widetilde{V} = S_G(\widetilde{e}) \cup V$,
   - $\forall e \in E_G \setminus E_H \setminus E \setminus \{\widetilde{e}\} \; s_{\widetilde{G}}(e) = s_G(e) \wedge t_{\widetilde{G}}(e) = t_G(e)$,
   - $\forall e \in E' \; s_{\widetilde{G}}(e) = f(e) \wedge t_{\widetilde{G}}(e) = f(e)$ where $f : E' \to (V_{\widetilde{G}})^*$,
4. $lb_{\widetilde{G}} = lb_{E_{\widetilde{G}}} \cup lb_{V_{\widetilde{G}}}$, where:
   - $\forall e \in E_G \setminus E_H \setminus E \; lb_{\widetilde{G}}(e) = lb_G(e)$,
   - $\forall e_2 \in E' \; lb_{\widetilde{G}}(e_2) = lb_G(e_1)$, where $e_1 \in emb^{-1}_{sup}(e_2)$,
   - $\forall v \in V_G \setminus V_H \; lb_{\widetilde{G}}(v) = lb_G(v)$,
   - $\forall v \in V \; lb_{\widetilde{G}}(v) = lb_{sup}(v)$,
5. $att_{\widetilde{G}} = att_{E_{\widetilde{G}}} \cup att_{V_{\widetilde{G}}}$, where:
   - $\forall e \in E_G \setminus E_H \setminus E \; att_{\widetilde{G}}(e) = att_G(e)$,
   - $\forall e_2 \in E' \; att_{\widetilde{G}}(e_2) = att_G(e_1)$, where $e_1 \in emb^{-1}_{sup}(e_2)$,

- $\forall v \in V_G \setminus V_H \; att_{\widetilde{G}}(v) = att_G(v)$,
- $\forall v \in V \; att_{\widetilde{G}}(v) = h(v)$, where $h : V \rightarrow P(At)$,

6. $ext_{\widetilde{G}} : [\widetilde{n}] \rightarrow V_{\widetilde{G}} \times \Sigma_V$.
7. $\forall e \in E_G^C \setminus \{\widetilde{e}\} \; ch_{\widetilde{G}}(e) = ch_G(e) \wedge ch^+_{\widetilde{G}}(\widetilde{e}) = \emptyset$.

As the result of the hyperedge suppression operation the hypergraph H is removed from the component hyperedge $\widetilde{e}$ of G. The external nodes of H are substituted by the corresponding nodes of V which become new source nodes of the hyperedge $\widetilde{e}$. The correspondence among nodes is established by a suppression function *sup* on the basis of the node labels. The source nodes of $\widetilde{e}$ have the same labels as before a development operation on $\widetilde{e}$.

The suppression embedding function replaces the relational hyperedges connected to the external nodes of H by new relational hyperedges connecting nodes of G to source nodes of $\widetilde{e}$. The way of replacing each hyperedge by the corresponding set of new hyperedges is specified on the basis of design constraints.

Let us come back to the example of the suppression operation removing a hypergraph H nested in the hyperedge $\widetilde{e}$ labelled L corresponding to the living area (Figure 2 and Figure 3). Node replacing specified by the function *sup* determines six new nodes (a set V) corresponding to ten external nodes of the nested hypergraph. The labels of new nodes are obtained by removing prefixes coming from the corresponding external nodes of the removed hypergraph. For example two nodes with labels 2.4 representing east walls of the hall and toilet, respectively, are merged into one node with label 4 representing the north side of the area L, while three nodes labelled 3.5.3 are merged into one node labelled 5.3 and representing the south wall of the living area. The four relational hyperedges which were connected with nodes 2.2, 1.3, 2.4 and 2.4 ($E = 2.2, 1.3, 2.4, 2.4$) are replaced by three new relational hyperedges of E' connected with nodes 2, 3 and 4 of the hyperedge labelled L according to the suppression embedding function. As the result of this suppression operation a hierarchical layout hypergraph shown in Figure 3 is obtained.

5 Visual reasoning

Visual languages play a similar role in design as design sketches as they enable to follow changes made in design diagrams. Nowadays computer systems with visual languages should be equipped with intelligent tools capable of assisting the user in a computational process. Such tools ought to be able to reason on the basis of the internal representations of visual language elements.

The presented system is able to reason about the diagram being designed and suggest the designer modifications which are needed and warn him/her against creating solutions not compatible with the specified constraints. To this end the system is equipped with a set Sr of predicates of the form $r : P(\mathscr{H}) \times \mathscr{K} \rightarrow \{TRUE, FALSE\}$, where $\mathscr{H}$ denotes a family of attributed hierarchical layout hypergraphs over Σ and At, and $\mathscr{K}$ denotes design knowledge. Each predicate tests whether the predefined criteria are satisfied for the presently generated hierarchical layout hypergraph.

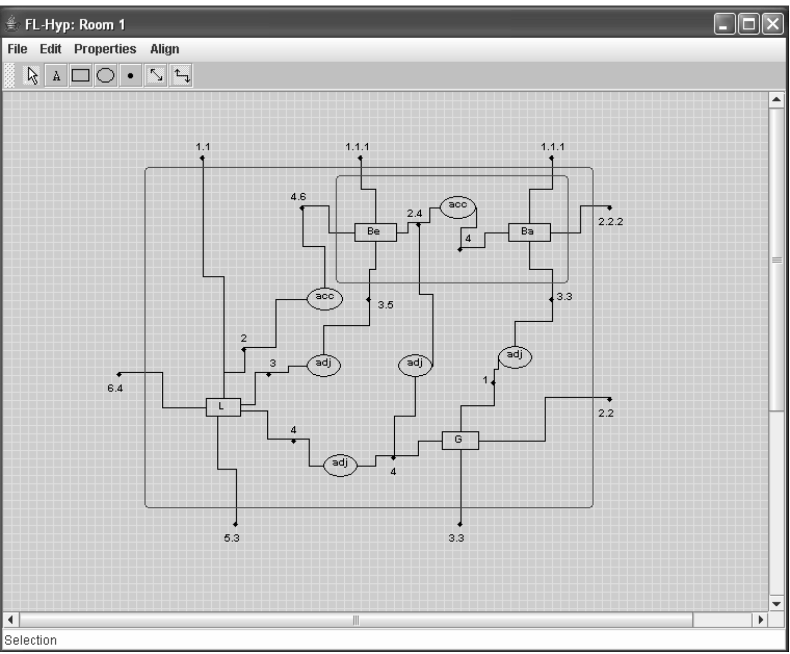


Figure 3: The hierarchical layout hypergraph obtained by removing a hypergraph from the hyperedge representing the sleeping area

Design knowledge is divided into syntactic knowledge K_s and interpretation knowledge K_i [CRRBG]. The predicates of Sr are divided into syntactic predicates (St) based on syntactic knowledge and semantic ones (Sm) based on interpretation knowledge. Each predicate of St has a specified subhypergraph which is searched for in a layout hypergraph created so far. After each hypergraph development operation, which nests a hypergraph in a hyperedge, for each component hyperedge of a nested hypergraph a predicate $r1 \in St$, which tests the accessibility of the corresponding room or area, is activated. This predicate searches for a subhypergraph shown in Figure 4(a), where *Lab* denotes a label of a component hyperedge. For example in a hypergraph nested in a hyperedge representing the whole apartment (Figure 2) and presented in Figure 6(b) for both component hyperedges labelled S and L one such subhypergraph is found, while for a hyperedge labelled G representing a garage a searched subhypergraph is not found. The designer is notified that there is no inside connection of a garage and it is accessible only from the outside of the house.

The example syntactic predicate $r2$ searches in a nested hypergraph for a subhypergraph shown in Figure 4(b). The labels Dr and K denote a dining-room and a kitchen, respectively. The predicate checks if a dining-room is located near a kitchen and is accessible from it. If these conditions are not satisfied for the existing dining-room the designer obtains a suitable piece of information.

Predicates belonging to a set Sm enable the system to perform semantic analysis of the generated hypergraph. The semantic reasoning about a design diagram is performed on the basis of identifiers and present values of attributes assigned to hypergraph atoms. To each component hy-

Figure 4: a) A subhypergraph of a predicate $r1$, b) a subhypergraph of a predicate $r2$

peredge the attribute *area*, which value specifies the area of the corresponding space, is assigned. To relational hyperedges labelled *acc* the attribute *type* is assigned. Its value specifies the way in which the adjacent spaces are accessible (by the door, by lack of a fragment or a whole wall between them). To hypergraph nodes the attributes *position* and *window_number* are assigned. They specify the location of the walls and the numbers of windows which the walls contain.

The example semantic predicate $r3$, which is activated for each room located in a design diagram, sums up values of the *window_number* attribute of all nodes assigned to a corresponding component hyperedge representing this room. Comparison of the obtained number with the specified architectural norms allows the system to reason about the lightning of the room and the loss of heat, and warn the designer about the possible problems.

Another example of semantic reasoning concerns deducing shape of rooms on the basis of identifiers of nodes assigned to hyperedges representing diagram components. First, a syntactic predicate, which searches for composition hyperedges connected with six or eight nodes, is activated. For the found hyperedges of the first type, a semantic predicate $r4$ tests if the corresponding areas have the shape of a letter L, while for the hyperedges of the second type, a semantic predicate $r5$ tests if the corresponding areas have the shape of a letter T. Predicate $r4$ searches for two pairs of nodes, each of them representing two non-collinear and parallel walls, and such that walls represented by these pairs are not perpendicular (hyperedges S and L in Figure 6(b)). Predicate $r5$ searches for three nodes representing three parallel walls with the same orientation and such that at most two of them are collinear, one node representing a wall which is parallel to the mentioned three walls but has the opposite orientation. Two remaining pairs of nodes should represent parallel and non-collinear walls with opposite orientations.

6 Implementation

In this section a structure of the system *HGSDR* (Hypergraph Generator Supporting Design and Reasoning) is presented. The system allows the designer to edit diagrams and automatically applies operations on hierarchical layout hypergraphs being internal representations of diagrams. It also gives the possibility to define constraints and reason about design diagrams.

The system is written in Java and contains four modules (Figure 5): a graphical interface for editing objects and constraints, a constraint module for reasoning about diagrams, a hierarchical layout hypergraph generator and a control module for hypergraph visualization. The graphical interface enables the designer to construct diagrams of visual elements directly accessible in the editor or taken from a library of objects defined by the user or an external library of domain-oriented objects. A set of visual primitives, which are directly accessible contains points, line segments, polygons and eliptic arcs. Texts can be used to describe primitives. The graphical interface allows the designer to interactively create and edit both objects, their attributes and constraints concerning these attributes.

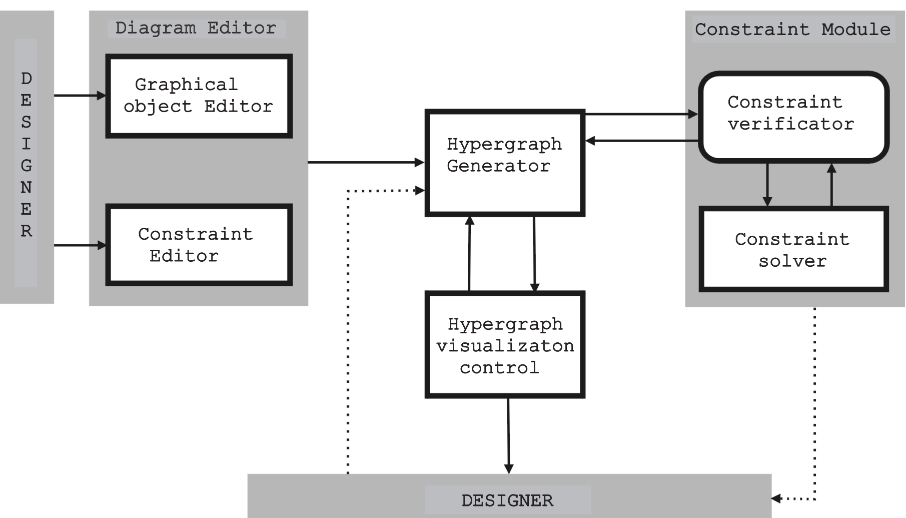


Figure 5: A system architecture

The constraint module contains syntactic and semantic reasoning rules. It activates the rules, solves constraint equations defined on hypergraph attributes and sends messages to the designer. It prompts the designer which diagram transformations are needed. When the designer changes values of attributes, which are connected by specified constraints (like distance or location), the system can force the appropriate constraints to be satisfied by changing sizes of some objets and drawing a new diagram again.

The hierarchical layout hypergraph generator automatically creates hypergraphs, where hyperedges correspond to diagram components and relations between them. The control module for hierarchical hypergraph visualization verifies and updates an existing hierarchy and enables to group components of the same hierarchy level and show relations between these groups and other components being on different levels of the hypergraph hierarchy.

7 Case Study

Let us consider an example of creating a design diagram presented in Figure 1(b), which corresponds to the layout of a one storey house with a garage shown in Figure 1(a). The first diagram drawn by the designer represents the area of the whole apartment. The initial hypergraph representing this diagram, that is automatically generated (Figure 6(a)), is composed of one hyperedge connected with four external nodes representing sides of the area and placed in the diagram according to the geographical location of the sides they correspond to.

In the next step, the designer divides the whole apartment area into three parts representing a living area, sleeping area and a garage, respectively (Figure 7(a)). As a consequence, the hyperedge development operation on the layout hypergraph is invoked automatically. As the result of this operation the layout hypergraph representing the three areas and adjacency relations between them is nested in the hyperedge representing the whole apartment. The obtained

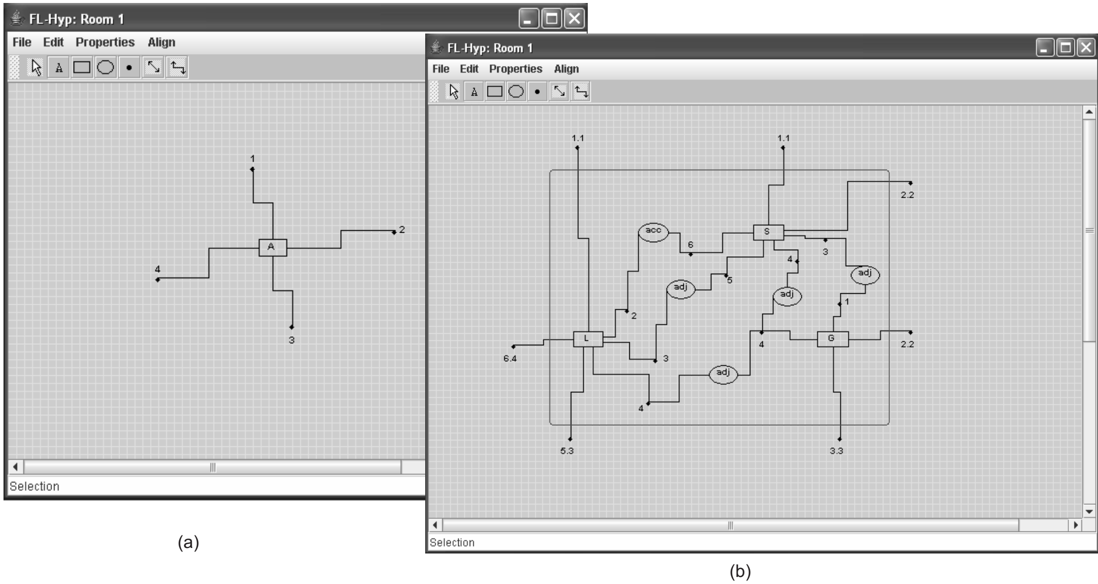


Figure 6: a) A component hyperedge representing the whole area of the apartment, b)a hierarchical layout hypergraph obtained as a result of a hyperedge development operation

hierarchical layout hypergraph is shown in Figure 6(b).

The four external nodes of the hyperedge shown in Figure 6(a) are replaced by seven external nodes of the nested layout hypergraph in respect to their geographical orientations. The labels of the external nodes of the layout hypergraph nested in the hyperedge representing the apartment are concatenated with labels denoting the number of the parent hyperedge node they replaced. The node number 1 of the apartment is replaced by nodes representing north sides of the living area (*L*) and the sleeping area (*S*). Both new nodes are labelled 1.1, where the first part of this label denotes that they correspond to first sides of areas *L* and *S*, while the second part of the label is inherited from the node which they replaced. The node number 2 is replaced by node 2 of the sleeping area and node number 2 of the garage, where both of them represent east sides of the diagram and are labelled 2.2. The node number 3 is replaced by node number 3 of the garage (labelled 3.3) and node number 5 of the living area (labelled 5.3). The node number 4 of the apartment is replaced by the node number 6 of the living area (labelled 6.4).

Then, the sleeping area is divided by the designer into a bedroom and a bathroom, which are adjacent to each other (Figure 7(b)). This modification of the design diagram results in nesting the layout hypergraph representing these rooms and adjacency between them in the hyperedge labelled *S* representing the sleeping area. As the result of applying the hyperedge development operation the hierarchical layout hypergraph presented in Figure 3 is obtained.

Six external nodes of the hyperedge labelled *S* are replaced by seven corresponding external nodes of the nested hypergraph. The labels of these seven nodes are concatenated with labels denoting numbers of the parent hyperedge nodes they replaced. For example, the node labelled 1.1 of the sleeping area is replaced by nodes representing north walls of the bedroom (*Be*) and bathroom (*Ba*) (nodes 1.1.1 and 1.1.1, respectively).The relational hyperedges which were connected

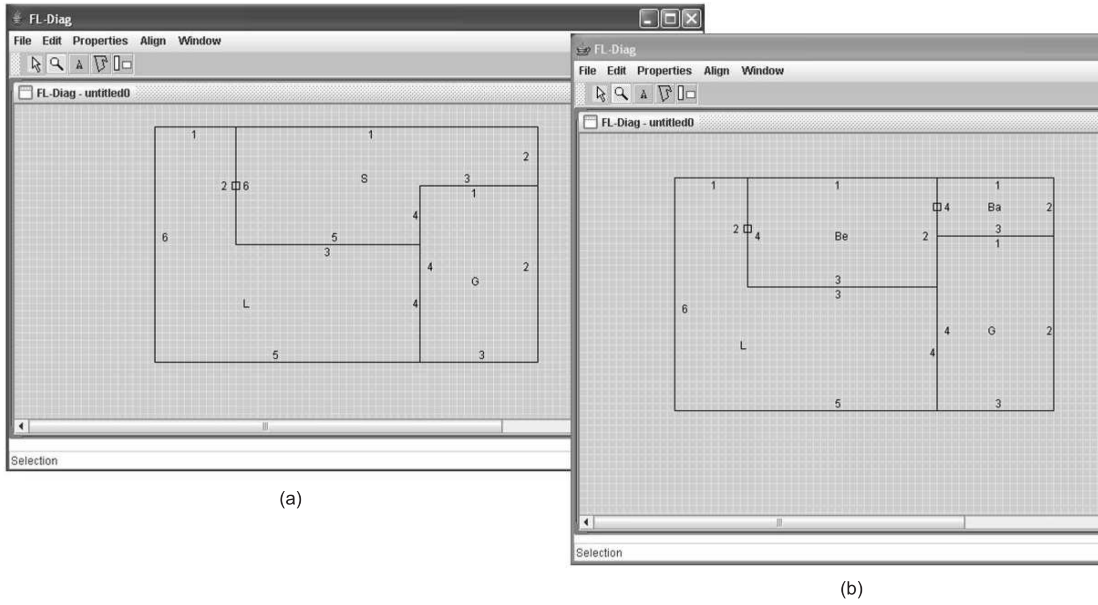


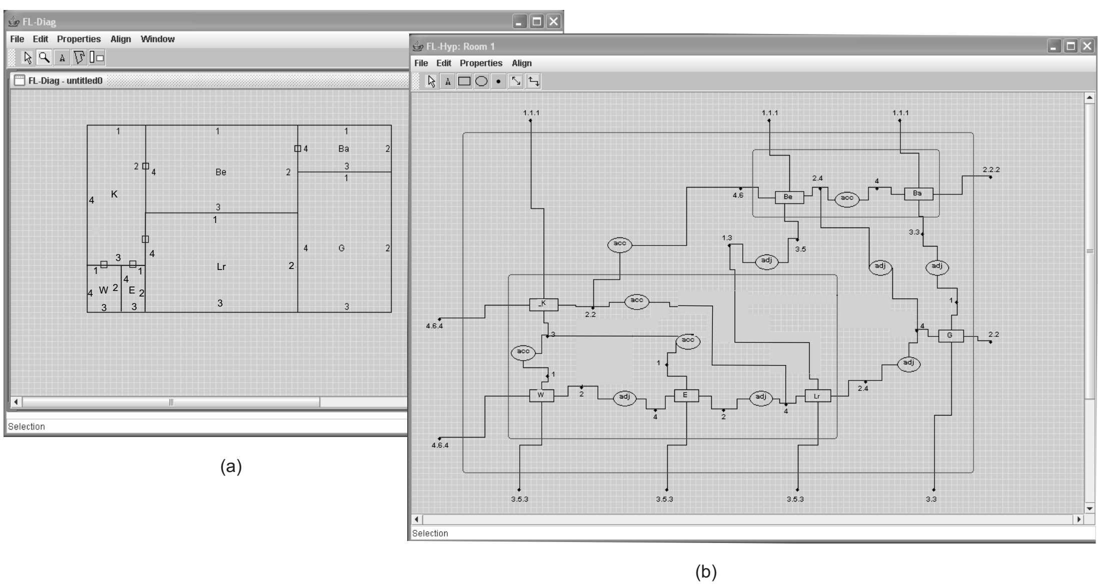
Figure 7: a)Three areas of the apartment, b) the apartment with the divided sleeping area

with nodes 3, 4, 5 and 6 of the hyperedge labelled *S* are replaced by new relational hyperedges connected with nodes 3.3, 2.4, 3.5, and 4.6 of the nested hypergraph, respectively.

Then, the designer divides the living area into five rooms representing a living room, kitchen, hall, entrance and a toilet, respectively (Figure 1(b)). As the result of the next hyperedge development operation, which nests the layout hypergraph representing these five rooms and adjacency relations between them in the hyperedge representing the living area the hierarchical layout hypergraph shown in Figure 2 is obtained. Six external nodes of the hyperedge labelled *L* are replaced by the corresponding external nodes of the nested layout hypergraph. For example, the node labelled 4 is replaced by two nodes 2.4 representing east walls of the hall and of the toilet (*W*), respectively, while the node 5.3 by three nodes labelled 3.5.3 which correspond to south walls of the kitchen (*K*), entrance (*E*) and toilet, respectively. Three relational hyperedges which were connected with nodes 2, 3 and 4 of the hyperedge labelled *L* are replaced by four new relational hyperedges. For example, the hyperedge which was connected with node 4 of the living area is replaced by two relational hyperedges, one connected with node 2.4 of the hall and another one with node 2.4 of the hyperedge labelled *W* (Figure 2).

If the designer is not satisfied with the layout of the living area and decides to change it, then she/he removes the division of this area and goes back to the diagram shown in Figure 7(b). As a consequence, the hyperedge suppression operation on the layout hypergraph is invoked automatically. As the result of this operation the hierarchical layout hypergraph shown in Figure 3 is obtained. The nodes which were divided by the development operation are merged again. Node replacing is specified in such a way that the merged nodes are given the same labels which they had before the development operation was used.

In the next step the living area is divided into four spaces corresponding to a living-room,



(a)

(b)

Figure 8: a) A redesigned diagram, b)a hierarchical layout hypergraph corresponding to it

kitchen, entrance and a toilet. The design diagram obtained as a result of a new division of *L* and the corresponding hierarchical layout hypergraph are presented in Figure 8(a) and Figure 8(b).

8 Conclusions

This paper is the next step in developing a visual language to support innovative design. In our approach the designer's modifications of diagrams are reflected by operations performed on their hypergraph representations. The way of reasoning about design diagrams on the syntactic and semantic level, based on knowledge stored in attributed hierarchical layout hypergraphs, that enables the system to suggest the designer modifications of created solutions, is also described.

The presented system is tested on designing floor-layouts. Other applications will concern visual languages for designing gardens in different styles and designing three-dimensional forms of buildings. The present implementation is written in such a way that each new application requires only some changes in the editor module.

Bibliography

[CRRBG] R. D. Coyne, M. A. Rosenman, A. D. Radford, M. Balachandran, J. S. Gero. *Knowledge-based Design System*. Addison-Wesley, Sydney, 1990.

[DHP] F. Drewes, B. Hoffmann, D. Plump. Hierarchical Graph Transformation. In J. Turyn (ed.), *Proc. of FOSSACS 2000*, LNCS 1784, pp. 98–113, Springer, 2000.

[Gol] G. Goldschmidt. The Dialetic of Sketching. *Creativity Research Journal*, 4, 1991.

[GGLŁŚ] E. Grabska, K. Grzesiak-Kopeć, J. Lembas, A. Łachwa, G. Ślusarczyk. Hypergraphs in Diagrammatic Design. In K. Wojciechowski et al. (eds.), *Proc. of the International Conference ICCVG 2004*. Computer Vision and Graphics, pp. 111–117, Springer, 2006.

[GŚG] E. Grabska, G. Ślusarczyk, M. Glogaza. *Design Description Hypergraph Language.* In M. Kurzyński et al. (eds.), Computer Recognition Systems 2, Advances in Soft Computing 45, pp. 763–770, Springer, 2007.

[GLŁŚG] E. Grabska, J. Lembas, A. Łachwa, G. Ślusarczyk, K. Grzesiak-Kopeć. Hierarchical Layout Hypergraph Operations and Diagrammatic Reasoning. *Machine Graphic & Vision*, in print, 2007.

[Min] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44, pp. 157–180, 2002.

[Pal] W. Palacz. Algebraic Hierarchical Graph Transformation. *Journal of Computer and System Sciences* 68, pp. 497–520, 2004.

[SWZ] A. Schürr, A. Winter, A. Zündorf. Graph grammar engineering with PROGRES. In W. Schäfer, P. Botella (eds.), *Proc. of the 5th European Software Engineering Conference (ESEC95)*, LNCS 989, pp. 219–234, Springer-Verlag, Berlin, 1995.

[Szu] J. Szuba. *Graphs and Graph Transformations in Design in Engineering*. PhD thesis, PAS, Warszawa, 2005.

[Ślu] G. Ślusarczyk. Hierarchical Hypergraph Transformations in Engineering Design, *Journal of Applied Computer Science*, 11(2), pp. 67–82, 2003.

Graph Transformation Model of a Triangulated Network of Mobile Units

Stefan Gruner

Dept. of Comp. Sc. | Univ. of Pretoria | 0002 Pretoria | South-Africa | sg@cs.up.ac.za

Abstract: A triangulated network of mobile units is modelled by means of a graph transformation system in which graph nodes are labelled with geometric coordinates and edges are labelled with distances. Nodes represent mobile units and edges represent wireless radio communication links between them. Under concurrency the model can describe interesting practical scenarios, for example swarms of taxis in an urban environment. The contribution features the enhancement of a graph transformation system by trigonometric calculations. By the way it is also shown that the classical "negative edge condition" has only limited applicability if a strict locality principle is assumed, and –vice versa– that there are reasonable modeling cases in which this locality principle itself fails to suffice.

Keywords: Attributed Graph Transformation, Mobile Network, Concurrency, Locality.

1 Scenario

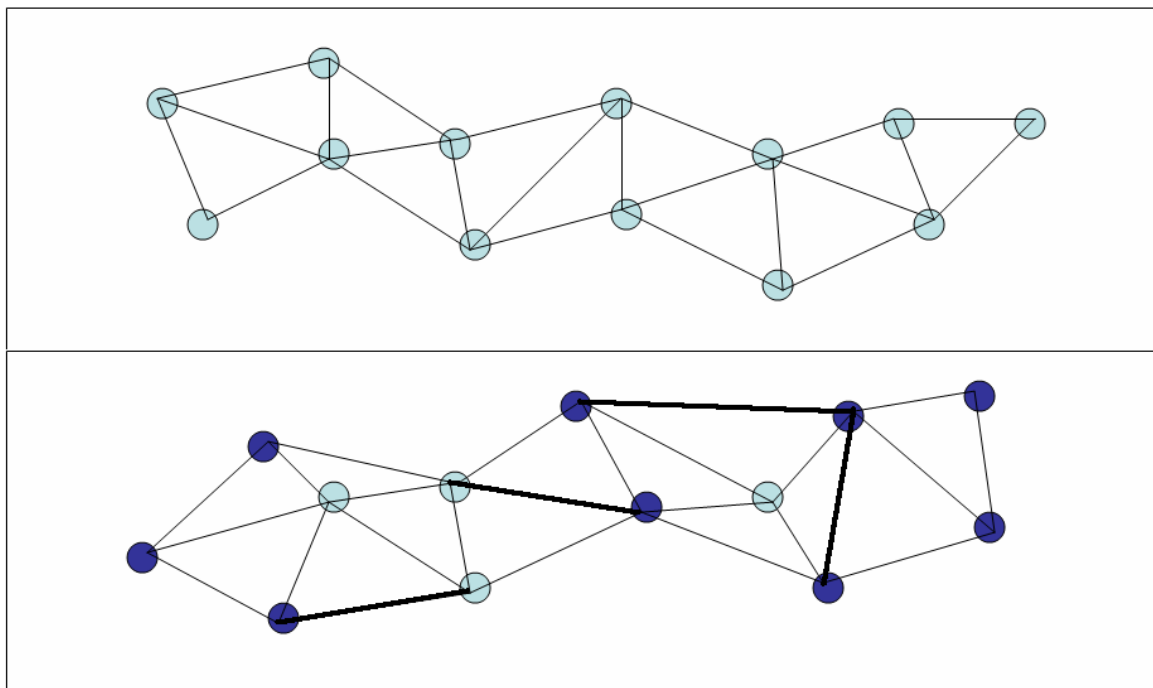


Figure 1: *Triangulated network before* (top) *and after* (bottom) *the movement of some nodes. Dark shaded nodes have changed their positions, and some communication links have also been reconfigured in their topological positions, which is depicted by thicker lines.*

Imagine an inner-city scenario in which a swarm of independent yet cooperating taxis keep each other informed about sources of customers to be picked up, traffic jams in the streets, etc. Every taxi is equipped with a simplistic communication device to keep in contact with a small number of other units in a not-too-far distance. To avoid confusion between the units, two *locality constraints* are imposed on every unit:

- The number *N* of communication partners to each unit is limited.
- The communication distance *D* between each unit is limited, too.

The network is self-organizing like a swarm of fish, thus not controlled by any central agency. Furthermore, the network structure shall be *triangulated* (as further defined below), such that:

- It has a well-manageable regular internal structure, as depicted in Figure 1, whereby:
- Configurations can be simply (re)-calculated with the usual formulae of trigonometry.

In the following sections of this paper a simple yet effective graph transformation model to such a self-organizing mobile network is developed. The underlying graph transformation techniques themselves being rather "classical" (except of a new interpretation of the "negative edge" condition) the value of this study is to be found in its general and uniform representation of a practically relevant scenario – enabling simulative experiments to study the behaviour of such systems, locally and globally, under various settings of its key parameters.

2 Technical Preliminaries

The *graph transformation paradigm* used in this paper combines the PROGRES system's syntax [SWZ99] and denotational semantics [Sch96] of program-embedded attributed graph transformation with the VISIDIA system's *message-passing* operational semantics [BGM01] [LMS95] as follows:

- Edges in the left-hand-side of a graph transformation rule represent communication link between two units, whereby the liveness of such links must be acknowledged by means of message-passing between the connected units.
- The *non*-existence of an edge between two units *u* and *u'* (negative edge condition) can thus only be recognized *indirectly* via a third unit *u"* to which both *u* and *u'* are linked. Any rule with a left-hand-side combining only *u* and *u'* with a "negative edge" (which would be perfectly legal in PROGRES with its "omnivident" global viewpoint) is thus *meaningless* under this strictly local operational perspective.
- An edge decorated with an explicit edge-attribute *e* indicates that some information *e* is (or can be) shared between the two adjacent nodes by means of message-passing as described above. Edge-attributes are only "syntactic sugar" for the sake of legibility of the model specification – in the terminology of PROGRES: "derived" (not "intrinsic").
- The operational semantics (implementation) of an edge between an ordinary node *u* and a multi-node *u** (which represents a finite set of nodes in the neighbourhood of *u*) requires *star-synchronisation* under *mutual exclusion,* as explained in [BGM01].
- Where mutual exclusion is not locally required, arbitrarily many transformation rules may be applied across the network graph at any time, in any paradigm of concurrency; see for example [LMS99].
- *Isomorphisms* are generally used to map the transformation rules' left-hand-sides into the model graph for application – with special treatment of the multi-nodes [SWZ99].

The *triangulation property* of the network graphs in this paper is defined as follows: A *ring* of size r is a graph with r nodes $\{V_0, \ldots V_{r-1}\}$ and r edges $\{E_0, \ldots E_{r-1}\}$ such that for all $0 \leq i < r$: $E_i = (V_i, V_{i+1 \bmod r})$ and no further edges exist between those nodes than just these ones. Then a *planar* graph is called *triangulated* if all its ring-shaped sub-graphs are of size $r = 3$. Thereby it is not required that all possibilities of triangle-building are fully exhausted: As it can be seen in Figure 1, there might be "incomplete" triangle subgraphs $\{V_x, V_y, V_z\}$ with edges (V_x, V_y) and (V_y, V_z) but no ring-closing edge (V_z, V_x).

3 Model Specification

In the following, the terms "attribute" (from the PROGRES terminology) and "label" (from the VISIDIA terminology) are used synonymously. In the model developed in this paper for the mobile units scenario, nodes and edges of the network graph are labelled as follows:

- An edge can carry a variable label *D*, representing a geometric distance between two nodes (in whatever vector-space, naïvely two-dimensional Euclidean). Also remember what has generally been said about edge labels in the previous section.
- Nodes carry several labels of the following kind:
  - *C* is a vector-type label representing the coordinates (position) of the node in the chosen vector-space (environment).
  - *A* or *P* are auxiliary labels (modelling artefacts without a corresponding property in the modelled world), used for the treatment of concurrency and mutual exclusion as further explained below.
  - A label *OFF* is used to model defective nodes (units) which cannot communicate. In the absence of this label the unit is assumed to be operative; nodes with edges cannot be *OFF*.
  - A label *NEW* can be used to signify previously non-existing units which have just arrived in the operational terrain covered by the network, or also to signify the recovery of a previously defective node. New nodes cannot be connected immediately – in other words: nodes with edges cannot be of type *NEW*.

Consequently the graph transformation rules of the model are labelled, too. However, there are also *generic* rules in which some of the labels (node labels or edge labels) are omitted. This means that such a rule can be applied without taking the instance-value of the omitted label into account. For example, if a graph transformation rule depicts an edge without edge label *D* then this edge could be mapped to any edge in the model graph. Thus, the model is based on a simple hierarchic type system, with *ANY* being the only super-type to all the other concrete types and no intermediate types in between. Omission of a type is regarded as equivalent to the explicit labelling of the according entity with the *ANY* symbol – see [Sch97] for comparison.

3.1 Locally Mutual Exclusion of Activities

The first rule of the model checks if a node, which intends to change its geographic position, is free to do so. If this is the case, it takes a token which prevents any immediate neighbour from becoming active as well. As usual it is assumed that the choice of an anchor-place for rule application is made non-deterministically (at random). In Figure 2 this rule is depicted. It tells us that a node can pick an activity token (for mutual exclusion in the local neighbourhood) if

this node itself as well as all its adjacent nodes are currently passive. As mentioned above, the model rules shall be *denotationally* interpreted like in PROGRES, whereby an additional star symbol (*) is used here as “lexical sugar” to emphasize the mapping of a multi-node to the entire neighbourhood (star) of a central node (here: node 1). Moreover it must be kept in mind how two overlapping rule-applications in the same local neighbourhood are prevented by the *operational* semantics of the VISIDIA system [BGM01][LMS99], due to which it can never happen that two adjacent units find themselves both in an active state *A* at the very same time. This means that simultaneous activities of two immediately adjacent units in the scenario are modelled via pseudo-simultaneous interleaving of mutually exclusive actions in rapid pace.

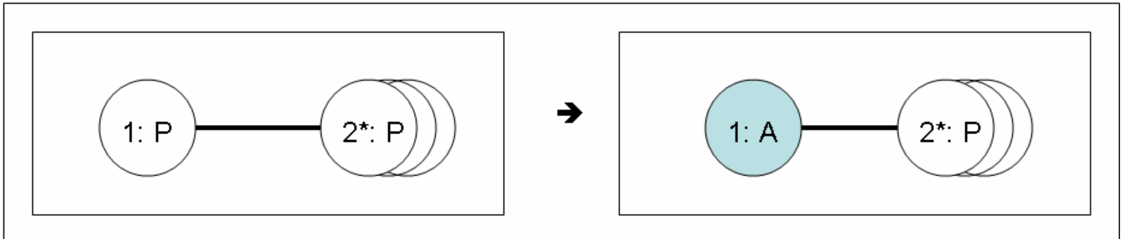


Figure 2: *Rule which explicitly models mutual exclusion of activities in a neighbourhood. The passive centre node of a star may pick an activity-token if all its neighbours are passive too. An operational semantics like the one implemented in the VISIDIA system guarantees that this rule itself is not multiply applied in the same local neighbourhood at the same time, which prevents two adjacent nodes from taking the activity token A simultaneously.*

Thus, the issue of local mutual exclusion is treated in this paper at two different levels: *explicitly* in the model specification by means of the *A* and *P* labels, as well as *implicitly* by VISIDIA operational rule-application semantics (message exchange along the communication channels). In non-overlapping (remote) regions of the model graph, however, a rule may well be multiply applied, at the same time, in genuine non-interleaving concurrency. Technically speaking this explicit modelling of local mutual exclusion would not be necessary; the implicit star-synchronisation of a VISIDIA kind of operational semantics would suffice. Nevertheless the explicit exclusion model with labels *A* and *P* was chosen for the sake of conceptual clarity.

3.2 Movement of Units

After a node has acquired activity-status (see above) it can change its location according to the mobile unit scenario. This is modelled with node labels representing coordinates in the chosen vector space. “Motion” is thus nothing but node relabelling, interpreted in terms of the chosen domain. When a node has been “moved” (by relabelling), two further actions must take place:

- The adjacent edges, labelled with the distances between the active node and its passive neighbour nodes, must also get relabelled to correctly represent the new geographic configuration.
- Then, the active node must release its exclusion label *A* and return to a passive state *P*.

As shown below, all these actions can be expressed in one single graph transformation rule. For the recalculation of positions and distances, the graph grammar system is augmented with

a simple trigonometric calculus which must be executed while the graph grammar rules are applied. The authors of PROGRES system have demonstrated how this can be done [SWZ99].

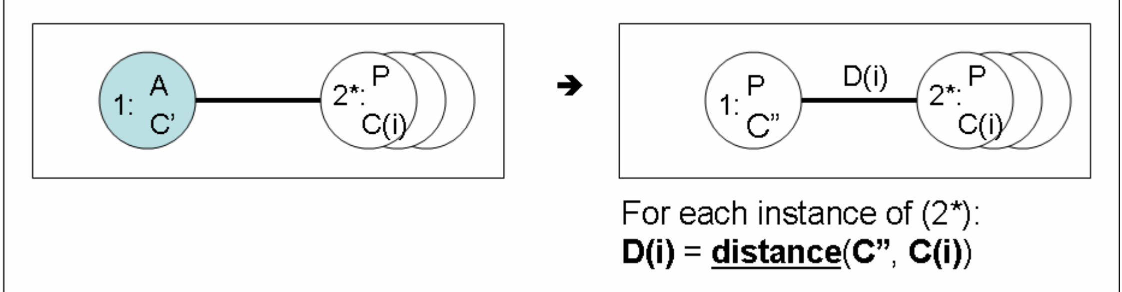


Figure 3: *Movement rule. The distance is calculated by an imported function, according to the PROGRES system. Thereafter the active node gives up its mutual exclusion token and returns into a passive state.*

Note that the movement of a node can temporarily destroy the desired triangulation property of the underlying network. For reasons of model simplicity (which means: a small number of small rules) it has been decided to temporarily concede the violation of this global topologic network property and to fix any violation with an equally simple set of repair-rules, rather than trying to design a complicated graph transformation system with large (non-local) rules for the sake of avoiding the violation of the triangulation property in the first place. (The repair-rules will be shown in the subsequent section.) Also note that the usage of large (non-local) rules would undermine the concept of a self-organizing network which is not controlled by a central agency. Small (local) rules, on the other hand, can be easily applied by the network nodes themselves as described in the literature to the concurrent graph relabeling paradigm [LMS99].

3.3 Communication Breakdown

In the mobile unit scenario it should be realistically assumed that communication can break down from time to time (which can also result in a temporary violation of the triangulation property and will also be treated by repair-rules as described in the subsequent section). In this paper any instance of a communication breakdown has one of the following three reasons; (see Future Work section below for further considerations):

- Communication breakdown due to a unit-internal technical defect;
- Communication breakdown due to too far distance between the communicating units;
- Communication breakdown because a unit cannot cope with too large numbers of communication partners any more; (the connectivity degrees of network nodes are assumed to be limited).

For all these cases, model rules are provided in the following. The activity status (*A* or *P*) of the nodes involved is irrelevant here, because the activity status is only a model artefact by means of which simultaneous movements of adjacent nodes are simulated (through small-step interleaving under mutual exclusion). Where motion is not involved at all, explicit distinction between *A*-nodes and *P*-nodes is obsolete as well.

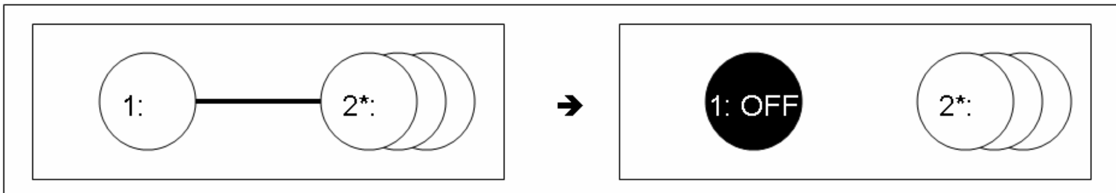


Figure 4: *Transformation rule describing a communication breakdown due to internal defect.*

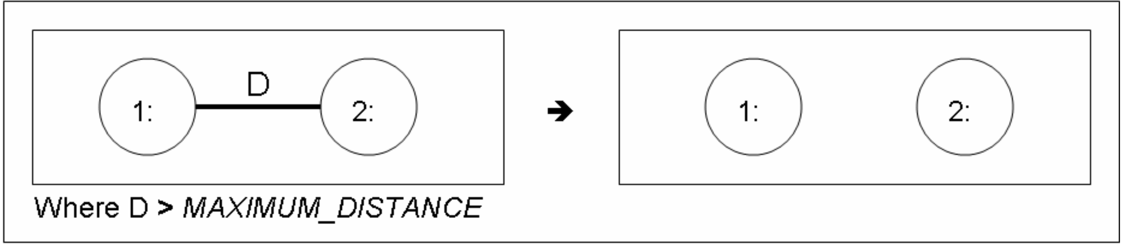


Figure 5: *Rule describing a communication breakdown due to long distance (weak signal).*

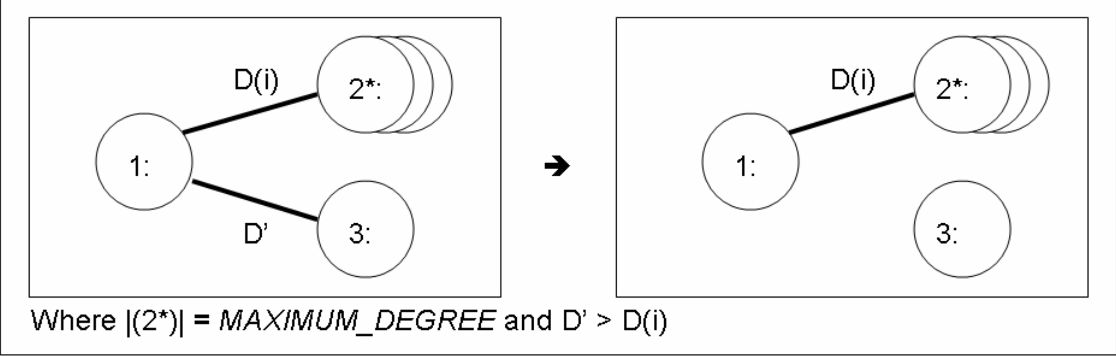


Figure 6: *Too many communication partners: the most remote one is abandoned. Classical graph transformation systems, without counting features, cannot process such kind of rules.*

As far as Figure 6 is concerned one might ask the question how a unit (here: 1) can have more than the maximal number of communication partners in the first place. The answer is found in the dynamic and self-optimizing nature of the system to be modelled: Temporary violations of global "invariant" properties (here: the maximal degree of connectivity) are admitted for a short period of time, such that (for example) a better communication link can be established before a worse one is given up. (This issue will become clearer in the subsequent paragraphs where the reparation of violated triangulations and the establishment of new communication links are modeled.) Also note that the rule depicted in Figure 6 requires a graph transformation paradigm which allows for *counting* the magnitude |N| of the neighbourhood of a centre node, which means that *neither* the classical PROGRES system [SWZ99] *nor* the classical VISIDIA system [BGM01] can be used to implement this crucial rule.

In this sub-section in only remains to be said that a defective (thus: isolated) node must be able to get back into operational mode, as a precondition to the re-establishment of its participation in the communicative network. This is modelled by the rather trivial rule depicted in Figure 7.

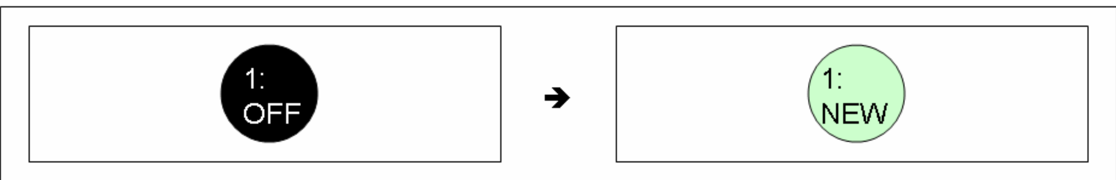


Figure 7: *A defective unit gets repaired. (It is as if a new unit would appear in the landscape.)*

3.4 Reparation of Locally-Temporarily Violated Network Invariants

The scenario model is generally assumed to possess a number of global invariants, such as the triangulated structure of the network graph, the maximal communication distance between two units, or the maximal degree of connectivity (number of communication partners) throughout the network. On the other hand it has already been mentioned that the dynamic character of the scenario will locally lead to violations of those invariants for short periods of time,[‡] until some repair-mechanisms restore the desired homogeneity of the model graph. These mechanisms are modelled explicitly (as part of the scenario specification) in the following paragraphs.

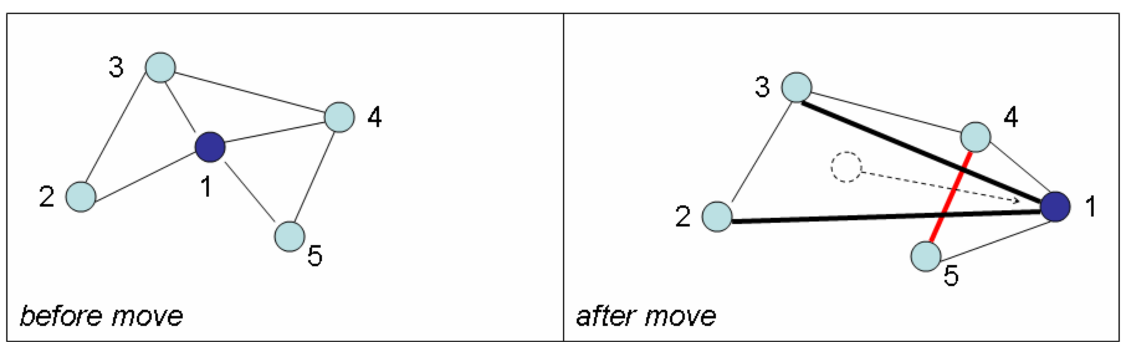


Figure 8: *Scenario in which the triangulation property is violated after a far-reaching move of unit 1 (see red line). The edges between units 1 and 2, respectively units 1 and 3, are now crossing the edge between units 4 and 5, such that this sub-graph is not planar any more.*

Consider the following situation depicted in Figure 8, in which some unit makes a move over some longer distance (by means of the motion rule of Figure 3) which leads to a local violation of the desired triangulation property. The strategy dealing with such situations consists of two subsequent steps:

- One of two edges which are "crossing" each other –which can only be described in terms of the vector-space geometry functions with which the graph transformation system is augmented– will be deleted; for the purpose of network optimization this is typically the longer link (with a greater distance label *D*).
- If such a deletion leads to a subsequent destruction of the network's triangulation property elsewhere, new communication links must be established at those locations, as it is sketched in Figure 9.

[‡] An analogy in the physical nature can be found in the realm of quantum physics, whereby spontaneous appearances of short-lived virtual particles can temporarily violate the macro-law of energy preservation for a short period of time in a small area of space.

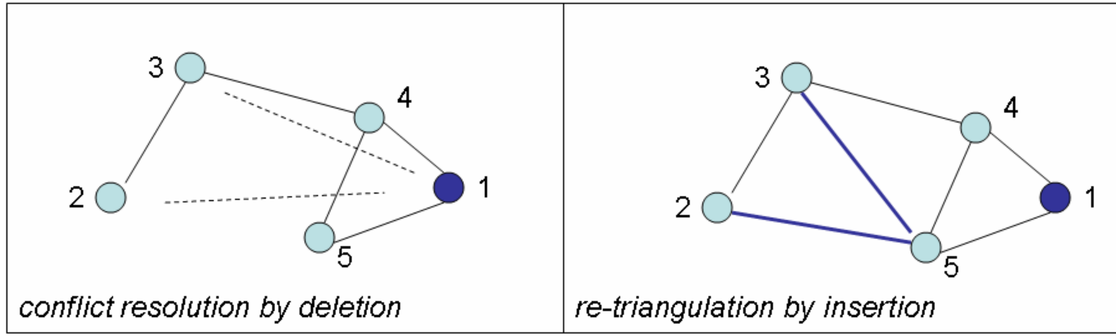


Figure 9: *Continuation of the scenario of Figure 8. In a first step* (left) *the conflicting links are deleted (see dotted lines). Thereafter* (right), *new links must be established elsewhere to re-establish the desired triangulation property (see thick blue lines).*

In the following, the graph transformation rules are shown by means of which scenarios like the one depicted in Figure 8 and Figure 9 can be effectively modelled. Note that this is the point of the specification at which the most complicated calculations in terms of the underlying geometry must be imported (in PROGRES style) into a graph transformation rule in order to confirm the existence of a cut-point between two finite lines in the underlying vector-space.

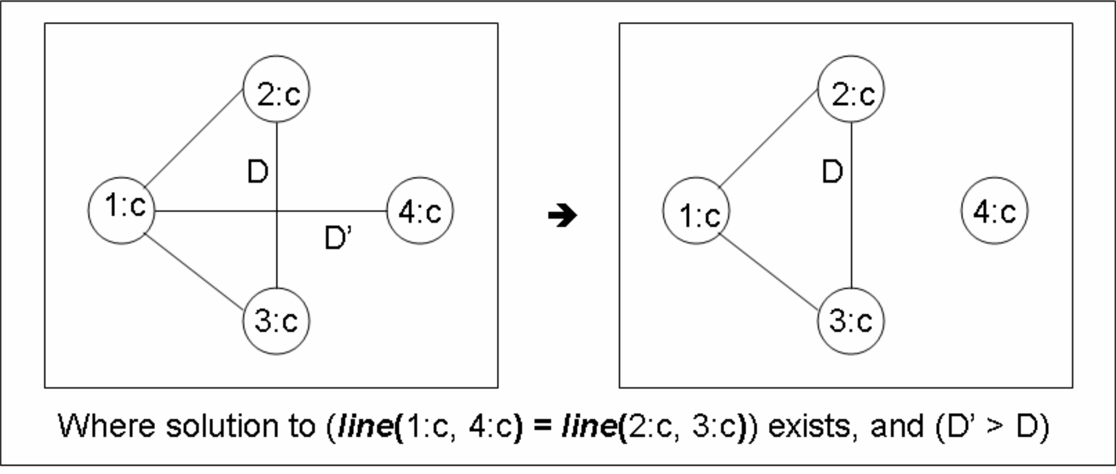


Figure 10: *Graph rule describing the detection and deletion of a conflicting network link.*

In Figure 10 as well as in Figure 11 the "violated" triangle is formed by the nodes 1, 2 and 3. Note that the edges between nodes 1 and 2, respectively between 1 and 3, are *not* redundant for the detection of the cross-point, because these edges represent *communication* links: Node 1 is the central unit in this scenario, which detects the cross-point between line *D* and line *D'* by communication with its neighbours 2 and 3. A rule designed like the ones in Figure 10 and Figure 11, but *without* the edges between nodes 1 and 2, respectively 1 and 3, would imply the *global* (omnivident, communication-less) perspective of a classical graph transformation system such as PROGRES [SWZ99] – in contradiction to VISIDIA's operational semantics of locality and message-passing [BGM01] [LMS99], to which the model of this paper adheres. In this paradigm the left-hand-side of a rule with such a purpose can only be a connected graph.

Also note that node 4 of Figure 10 could *possibly* remain *isolated* after the according rule has been applied – ditto for the rules depicted in Figure 5 and Figure 6. In such a case the isolated node would assume a *NEW* state, like the one which is depicted in Figure 7. The according graph transformation rule is not depicted in this paper for reason of triviality: every unit knows intuitively (by itself) when it does not have any communication partners, thus when to relabel itself to *NEW*.

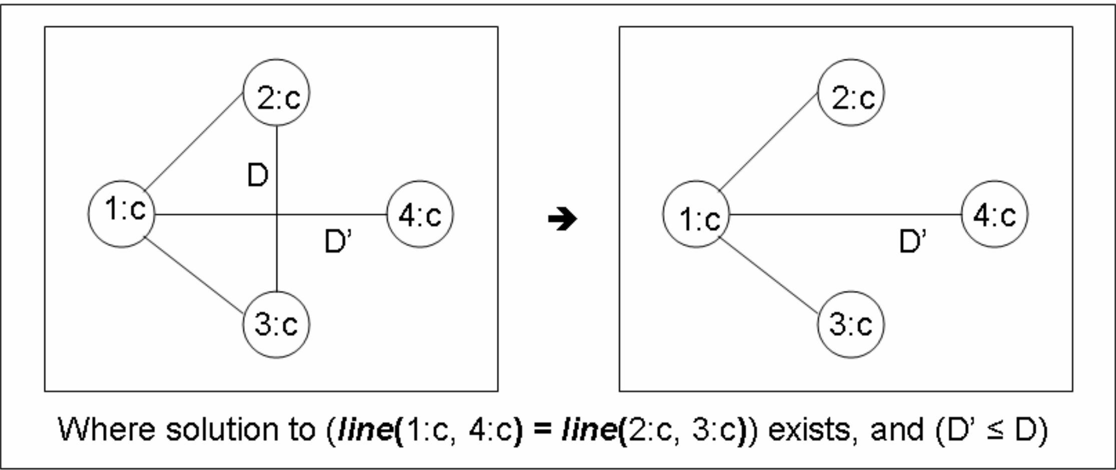


Figure 11: *Ditto, this time with the other one of the two conflicting links being deleted.*

Further note that the rules of Figure 10 and Figure 11 together ensure *local optimization* by determining the *longer link to be deleted* and the shorter link to remain. It is obvious that two rules are needed for this optimization purpose: one for the case that $D > D'$, and one for the case that $D \leq D'$. These rules are designed as ordinary rules (not as a star-rules) such that only one edge in a conflict situation is deleted per rule application. This design option has been chosen in order to keep the amount of destruction within a region of the model graph as small as possible (thus also the amount of required restaurations) for conservative reasons. In case that more than one conflicting edge must be destroyed, these rules can be applied repeatedly until all conflicts are resolved.

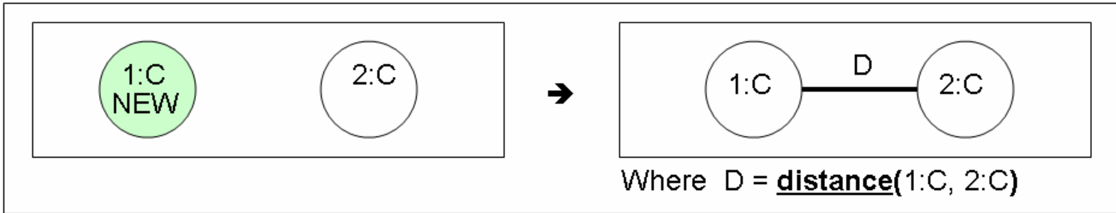


Figure 12: *Rule modeling a new unit receiving a radio signal and establishing a network link with its hitherto unknown newly found communication partner. No VISIDIA-semantics here!*

After an edge has been deleted as shown above, or after a defective unit has come back into operation mode, there are opportunities for the (re)-establishment of communication links for the sake of the already mentioned global triangulation property. In the case of a new node,

which has per der definitionem no communication links (and therefore no knowledge about the existence of any other units) the new node cannot do anything but wait and listen until it receives a signal from another, hitherto unknown nearby unit. A communication link between these two units is then formally established, and the now connected node looses its *NEW* status, as depicted in Figure 12. Per default the value of *D* must be smaller than maximum, otherwise no radio signal could have been received at all. Should the newly established link violate any of the already mentioned global network invariants then the repair-rules (Figure 6, Figure 10, Figure 11) would be in place again to rectify the temporary topological flaw in an optimizing manner.

At this point it is important to note that the graph transformation rule of Figure 12 is the only rule of the scenario model which can *not* be explained in terms of VISIDIA's message-passing semantics. Here, and only here, it is necessary to assume the "omnivident" perspective of the PROGRES paradigm. The reason is that, from a strictly local perspective, nodes 1 and 2 of Figure 12 cannot know anything about each other unless a communication link exists – which is however not the case in the left-hand-side of that rule. From a local perspective, the event of receiving a signal from a hitherto unknown unit comes as an unpredictable surprise to the receiving unit. It is not possible to verify the existence of the left-hand-side situation of Figure 12 through message exchange, because any message exchange already implies the situation of the right-hand-side of Figure 12. When the software prototype to this scenario model is being implemented (see section Future Work), this problem must be solved *ad-hoc* by means of data structures which do *not* correspond to the "pure" VISIDIA theory; (possibly: representation of a unit's surrounding "landscape", storing information about the presence of radio signals).

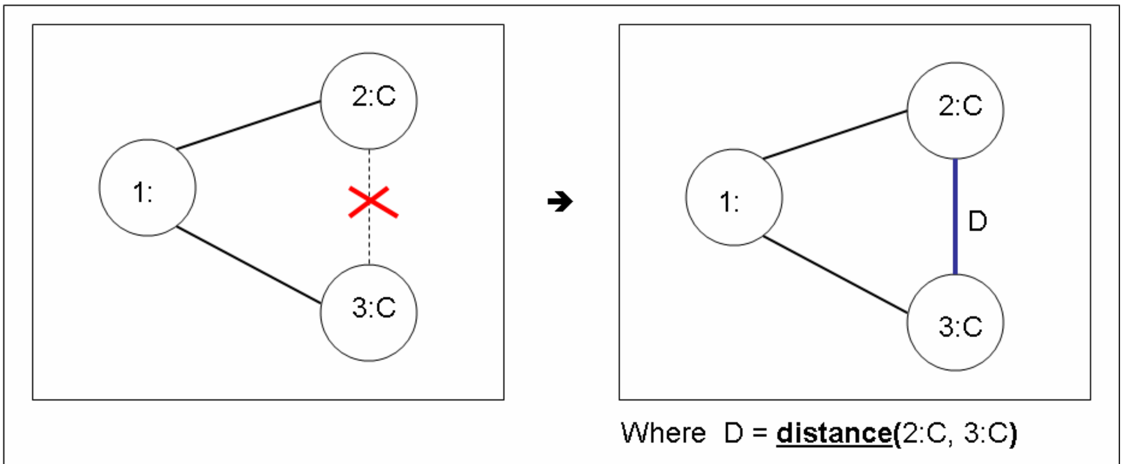


Figure 13: *Transformation rule describing the establishment of a new triangulation-link.*

Finally it must be explained how new "triangle" configurations in the model graph can come into existence, as it is shown in Figure 13. Per default, the distance *D* (between the newly connected nodes 2 and 3) cannot exceed the maximum. The dotted and crossed-out line in the left-hand side of the rule depicted in Figure 13, which models the establishment of a new triangulation, represents a *negative application condition* known from graph transformation systems such as PROGRES [SWZ99]. In the model of this paper (with its strict principle of

locality in which the classical "omnividence" is no longer given), it must be explained how such an easily-drawn dotted line can be effectively implemented. This is done again by means of message passing (as explained in the Technical Preliminaries section above). Last but not least it should be mentioned that also the application of this rule (Figure 13) can lead to temporal violations of network invariants, but also in this case the already mentioned repair rules can be applied to fix the flaw.

3.5 Global Dynamics through Local Processes

All graph transformation rules in the model –except the one of Figure 12– are designed in such a way that they can be applied and executed by the network nodes themselves (without any global super-instance) with the techniques of the concurrent relabeling paradigm [BGM01]. Obviously no neighbourhood (star) radius is greater than 1 in these rules, which makes their application (in terms of message exchange within a neighbourhood) especially easy. Indirect communication (of radius greater than 1 via router nodes) does not occur.

The dynamics of the mobile network of communicating units as a whole is thus a result of the concurrent activities of the individual units within the network. The activity of each individual unit is a simple process, following a simple protocol, which comprises at least the following algorithmic steps (described in pseudo-code as follows):

```
WHILE (operative) // process cycle for every unit
      {
       WHILE (repair_rule_applicable) {apply repair_rule};  // as often as possible
            IF (move_is_desired)         {apply move_rule}; // once per cycle
            IF (link_rule_applicable)    {apply link_rule}; // once per cycle
      };
```

In this rule-application protocol, priority is given to the application of repair-rules to maintain the desired global topological network invariants as mentioned above. Alternative designs of the protocol could obviously lead to different global network behaviour, which would make up an interesting question for experimental studies; (see Future Work section below).

4 Related Work

There are many formalisms and approaches to the modelling of distributed systems as a whole or particular aspects thereof. As far as the domain of graph transformation is concerned (which is the theme of GT-VMT-2008), a similar mobile network scenario and a similar modelling approach to it was recently published by *Casteigts* and *Chaumette* [CCh05]. It differers from the model presented in this paper as far as the embedding of geometric calculations for positions and distances into the transformation rules is concerned. *Heckel* and *Guo* described a layered graph transformation model of roaming cellphones being transferred from one base station to another one [HGu04]. Their scenario is thus similar to the scenario presented above (and also intended to be subject to simulative experiments), but geometric considerations such as network triangulation or distances between units do not play a role in their paper. Moreover, their model is developed in the classical "omnivident" paradigm, not in the paradigm of local

communications. The short-paper [GHe04], published by the same authors, is little more than an abstract summary of [Hgu04] but it provides a nice description of the typical characteristics and difficulties of distributed mobile systems, as well as a nice discussion of broader related work. *Knirsch* and *Kreowski* where amongst the first ones to model agent systems on a high level of abstraction [KKr00]. Their formalism and notation appears quite similar to the one of [LMS99] but remains rather vague as far as the issue of mobility is concerned. *Chalopin*, *Godard*, *Métivier* and *Ossamy*, on the other hand, have successfully modelled agent mobility on a given network in terms of graph relabelling with message-passing semantics [CGM06], however their graph model is static and does not allow for restructurings of the network itself.

5 Future Work

To date, none of the existing graph transformation software packages offers experimental simulation environments to such an extent that hundreds of nodes of pixel-size could be displayed in motion on a computer screen. So far, existing graph transformation software packages have put all emphazise on the visualisation of structural or *topological* properties of their model graphs, not on the visualisation of their *geometric* properties and node motion, as it would be required for the communicating mobile unit scenario described in this paper. The implementation of such a software system is planned, whereby the graph transformation rules described in this paper shall be hard-coded into the prototype for the sake of runtime speed.

Once such a prototype would be available, the necessary empirical validation of the introduced concepts would be possible. Then it could be investigated through experimental observation and measurement, for example:

- How frequently does the global network invariant (triangulation) get locally violated?
- How quickly are those local violations being repaired?
- How does the size of a neighbourhood (e.g.: maximally 5 communication partners or maximally 8 per unit) as well as the length of the maximal communication-distance influence the behaviour of the entire network as a swarm-like super-unit?
- How smoothly does the network re-organize its structure when many nodes are on the move into different directions?
- How would a different rule application protocol, individually executed by each unit, affect the behaviour of the network system as a whole?
- (Etc.)

The model developed in this paper is still quite simplistic as far as the physical properties of communication links are concerned. It has been simply assumed that the maximal distance of communication is a constant D for all units in the network. In reality one would have to deal with locally variable parameters Ď, depending (for example) on weather-conditions, objects obstructing the transmission of radio signals, etc. The graph transformation model of the scenario would have to be amended accordingly – not so much in terms of the graphical rewrite-rules, but rather in terms of the functional calculations into which those rules have been embedded.

References

[BGM01] M. Bauderon, S. Gruner, Y. Métivier, M. Mosbah, A. Sellami: *Visualisation of Distributed Algorithms based on Graph Relabeling Systems*. ENTCS Vol.50, No.3, Pages 227-237, Elsevier, 2001.

[CCh05] A. Casteigts, S. Chaumette: *Dynamicity Aware Graph Relabeling Systems (DA-GRS): A Local Computation based Model to describe MANet Algorithms*. Proc. 17th IASTED Internat. Conf. on Parallel and Distr. Computing and Systems (PDCS05), pages 231-236, Pheonix, November 2005.

[CGM06] J. Chalopin, E. Godard, Y. Métivier, R. Ossamy: *Mobile Agent Algorithms vs. Message Passing Algorithms*. LNCS Vol.4305, pages 187-201, Springer, 2006.

[GHe04] P. Guo, R. Heckel: *Modeling and Simulation of Context-Aware Mobile Systems*. Proc. 19th Internat. Conf. on Automated Software Eng. (ASE'04), IEEE Computer Society, 2004.

[HGu04] R. Heckel, P. Guo: *Conceptual Modeling of Styles for Mobile Systems: a Layered Approach based on Graph Transformation*. Proc. IFIP Conf. on Mobile Information Systems (MOBIS), Oslo, September 2004.

[KKr00] P. Knirsch, H.-J. Kreowski, *A Note on Modeling Agent Systems by Graph Transformation*. LNCS Vol.1779, pages 79-86, Springer, 2000.

[LMS95] I. Litovsky, Y. Métivier, E. Sopena: *Different Local Controls for Graph Relabelling Systems*. Math. Syst. Theory Vol.28, pages 41-65, 1995.

[LMS99] I. Litovsky, Y. Métivier, E. Sopena: *Graph Relabeling Systems and Distributed Algorithms*. H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol.3, pages 1-56, World Scientific, 1999.

[Sch96] A.. Schürr: *Logic Based Programmed Structure Rewriting Systems*. Fundamenta Informaticae Vol. 26, No. 3/4: Special Issue on Graph Transformations, pages 363-385, IOS Press, 1996.

[Sch97] A.. Schürr: *Programmed Graph Replacement Systems*. G. Rozenberg (ed.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol.1, pages 479-546, World Scientific, 1997.

[SWZ99] A.. Schürr, A. Winter, A. Zündorf: *The PROGRES Approach: Language and Environment*. H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.): Handbook of Graph Grammars and Computing by Graph Transformation, Vol.2, pages 487-550, World Scientific, 1999.

Acknowledgments

During a visit to the University of Bordeaux in March 2006 I discussed several ideas related to this paper with my colleagues, particularly *Mohamed Mosbah* and *Serge Chaumette*. Thanks also to *Andy Schürr* and *Reiko Heckel* as well as three anonymous reviewers of GT-VMT-2008 for their valuable hints and suggestions.

Some Applications of Graph Transformations in Modeling of Mechanical Systems

Stanisław Zawiślak, Łukasz Szypuła, Mirosław Myśliwiec and Adam Jagosz

Department of Fundamentals of Machine Building
University of Bielsko-Biała, Poland

Abstract: In the paper, some graphs representing mechanical systems are analyzed. The transformations of graphs representing a gear and a truss are considered. These transformations of the graphs allow for derivation of simplified calculation courses in some cases e.g. gear ratios or forces. Furthermore, these simplifications are mutually connected with adequate changes of the graph or sub-graphs in step-wise manner taking into account the functional schemes of the artifacts. The derived calculation methods utilize other algebraic objects associated with a graph e.g. cut matrices and sets of fundamental cycles.

Keywords: gear functional scheme, Hsu's graph, f-cycle, cut matrix, kinematical analysis

1 Introduction

Graphs are used as models of versatile technical systems e.g. electrical and electronic systems, railways and road networks, phone networks and mechanical systems. The last mentioned area of application is relatively new and it is relatively narrowly known. Nevertheless the achievements within this area of investigation are crucial for AI applications in mechanical design or AI-aided design [9]. The most essential and simultaneously wide introduction to the graph representation of mechanical systems can be found in books Tsai [11], Rudolph [6] and Kaveh [4] describing some different aspects, respectively. The valuable contributions to the discussed field have been done by Hsu [3], Shai [9], the author [15-19] and many others.
Graph transformations are used in engineering applications mainly in civil engineering [1,2,10] but recently this tool was also used in mechanical engineering [5,7,8,13]. The graphs' application in civil engineering focus mainly on a layout of civil engineering structures e.g. trusses, buildings or floor arrangements in buildings. In paper [5], the reference review connected with an application of graph-grammars in mechanical design has been made. The described methodology was used for generation of new designs of gears i.e. their functional schemes. In paper [7], the synthesis of mechanisms based upon an application of graph grammars is presented. The task of synthesis and enumeration of all possible designs of a particular mechanical artifact can also be performed by means of the graph-based approach [11]. Firstly, graphs are used e.g. for encoding of a functional scheme of a planetary gear or a geometrical structure of a truss. Secondly, base on some algebraic objects related to the graphs (e.g. matrices, polynomials or matroids [15]) the calculation methods are derived and used. The transformations of the graphs assigned to planetary gears are shown in the paper. Some adequate transformations are connected with the changes of drives and the related changes of passage of a rotational speed and power throughout a gear. Simultaneously these transformations cause simplification of adequate equation systems in an automatic way. The whole process makes possible to analyze simplified functional schemes and it allows for

derivation of relevant simplified kinematic equation systems for consecutive considered work modes in case of the automatic gear boxes. The goal of this paper is to show which graph transformation are used in modeling of gears and trusses as well as what is the mechanical interpretation of these transformations which – moreover - are different from other approaches shown in the cited references. It is worth to underline that in recent years, it was very rare to analyze versatile mechanical artifacts in the light of common graph transformations approach. On the contrary, usually just single objects were considered or other aspects of graph models were highlighted. Moreover, it has to be added that O. Shai [9] compared several graph-models of artifacts from the AI knowledge transformation perspective.

2 Graphs as models of mechanical systems

The possibility of representation of a mechanical system *M* by means of a graph *G* consists in simplification and representation of the system *M* by means of relations between its elements. These relations can be then turned into graphs where elements of relations are presented as edges with adequate weights. There are versatile graph representations of a planetary gear [15]. The review of some more frequently used approaches is given in [18]. Other graph-based methods of modeling of mechanical systems are described in works [9, 11]. A graph $G\ (V, E\ W)$ is a weighted graph, where: $|V| = n$, $|E| = m$ and a function $W: E \rightarrow \{set\ of\ weights\}$. The function *W* as well as the set of weights depend on a considered artifact and a considered problem. A single weight is usually assigned to a single edge e.g. see explanation given to Fig. 1. However in some cases - when advanced mechanical analysis is performed upon a graph model of an artifact (i.e. a truss) - several weights are assigned to a single edge. For example, in case of a truss (Fig. 2) when stresses are analyzed every edge represents a rod made of metal - therefore some weights are as follows: cross-section area of the bar [m^2], force acting along the bar [N], physical and mechanical properties of the bar material e.g. density [kg/m^3] etc. [19]. Moreover, sometimes instead of weights the different line types are used for edges pictograms aiming for easiness in interpretation of a weighted graph. In Fig.1 the functional scheme of a gear and the assigned weighted graph are presented. Two parallel short lines perpendicular to the main axis indicate that scheme is symmetrical but the symmetrical image under the axis is omitted. Planets *2* and *7* in Fig. 1a can look like geared wheels presented in Fig. 1c, arm *1* is omitted. Meshing of elements *6* (planet) and *3* (wheel with internal toothing) can look like it is shown in Fig. 1d. A relation between a pair of two gear elements i.e. the fact that two geared wheels are in mesh - is shown in a graph as stripped-line edge. For example: geared wheels *2* and *7* (so called planets) in Fig 1a are turned into the vertices *2* and *7* in the graph in Fig. 1b. Because these planets cooperate in a mechanical sense (their tooth are especially in contact; generally speaking in mesh) therefore the edge *{2,7}* is drawn as a stripped one. In case of a pair of gear elements: arm and planet – their adequate vertices are connected via continuous line starting from polygon e.g. element *1* and *2* (arm and planet) are represented as vertices *1* and *2* connected via the edge *{1,2}* drawn as continuous. The rotational axis of the planet is fixed to the arm. The arm assures a constant distance between the main gear axis and the axis of the planet allowing for their mutual rotational movement. We can considered different elements as input and output of the gear e.g.: 5-1, 1-3 or 5-3, respectively. The Hsu's method of assignment of a graph to a gear scheme has been utilized. The method can be summarized as follows: vertices of graphs and adequate gear elements are notated by means of the same labels i.e. natural numbers. The relations between elements of the gear are considered as follows:

- all pairs which rotate around the same main axis (rotational pairs) are represented by vertices of a (shaded) polygon e.g. *1,3,4* and *5*. Therefore we have pairs: *{1,3},{1,4}, {1,5}, {3,4, {3,5}* and *{4,5}*. The whole full graph being a subgraph of the considered graph of the planetary gear represents the whole set of these rotational pairs (6 in case of rectangle, 10 in case of pentagon etc.). The rule was introduced that this full graph is replaced in figures by a shaded polygon. This approach allow for achieving a more readable form of the graph. This approach is dedicated only to the illustrative form and in further detailed considerations connected e.g. with distinguishing of cycles (called here f-cycles [11]) all edges of full graph are theoretically considered despite that they are hidden in the drawn polygon;

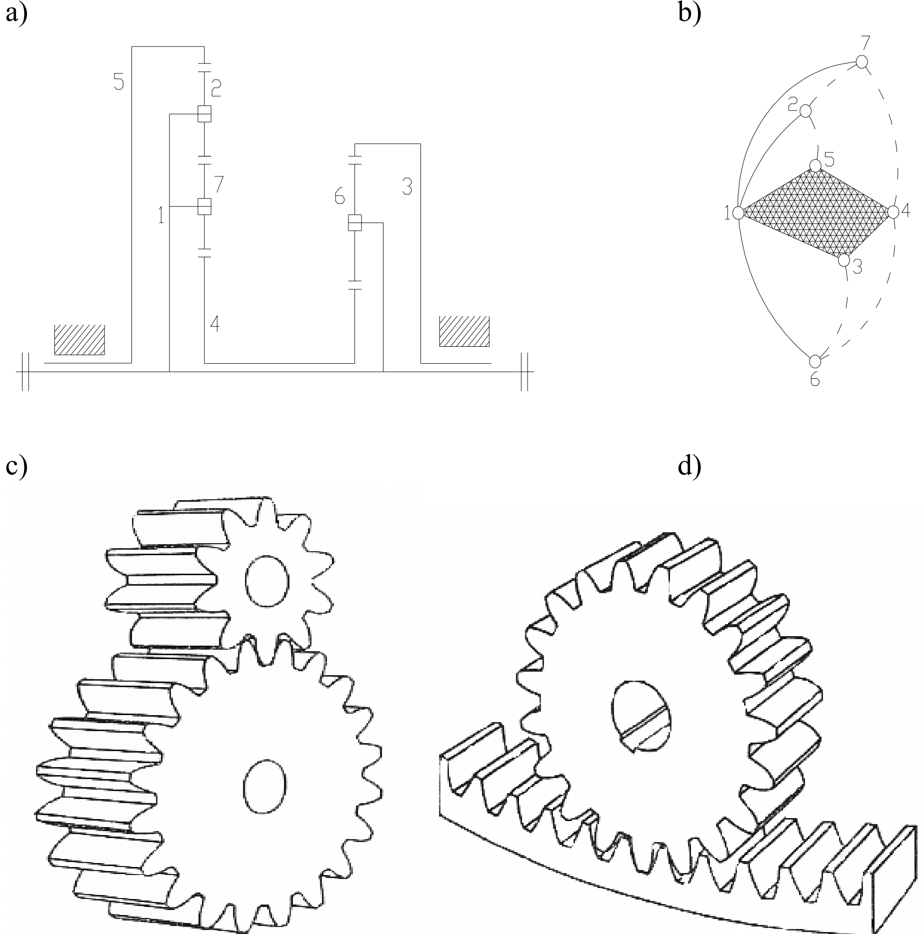


Fig. 1. Planetary gear (a), its Hsu's graph (b), external (c) and internal (d) meshings

- pairs of elements being in mesh (co-operation of two toothed wheels, geared wheels) are represented via stripped lines (edges) e.g. *{4,6}, {3,6}, {4,7}, {5,2}* and *{2,7}*. The meshing between the planets *2* and *7* is external (Fig.1c) whereas the meshing between wheels *2* and *5* is internal (Fig.1d) what will have respectable meaning in writing the equations describing the kinematics of a gear wheel but it is not encoded in a graph explicitly. Internal meshing means that the wheel *5* has its tooth inside a toothing ring of the element 5;

- rotational pairs “planet wheel – arm (carrier)” are represented via continuous edges i.e.: *{1,7}, {1,2}* and *{1,6}*.

The powerfulness of the graph representation of mechanical systems consist in usage of versatile algebraic structures connected with the graph and derivation of some calculation methods utilizing these algebraic objects (chapters 3.2 and 3.3, below). The tasks which can be performed by means of graph-based models are as follows:

- analysis of gear ratios [11],
- solution of a reverse problem i.e. assignment of a functional scheme to a gear graph [5, 17],
- enumeration of all gear schemes fulfilling a particular constraint i.e. having less then e.g. *7* rotational elements [11],
- calculation of forces in a truss loaded by external discrete forces (acting in nodes) [14],
- checking a stiffness of a truss [4]

and many others [11].

In the underneath consideration e.g. a cut matrix for trusses and f-cycles for gears graph models will be applied in adequate calculations. A truss is a structure made of rods. Neglecting the type and dimensions of joints the simplified model is obtained. Such a model is frequently used in an introductory engineering calculations. At the beginning an assignment of a graph to a truss is done in a natural way: nodes of truss are converted into vertices and bars into edges, respectively. The model and the transformation steps which simplify some calculations are described underneath. Transformations of graphs which are assigned to mechanical systems have several goals:

- derivation of calculation methods in simple manner taking into engineering knowledge transformed into a field of graphs (obtained methods are equivalent to the traditional ones),
- automation of calculation courses and
- constructing a design form in an algorithmic way via step by step performed routine.

3 Transformations of graphs being models of mechanical systems

Graph representations of automated gear boxes can be used for an automation of a ratio calculation as well as some other tasks. The essence of transformation relays in every case in different task:

- creation of a functional scheme step by step – several mutually related facts about graph are taken into account,
- creation of a system of equations (describing e.g. a truss) in algorithmic way – assuring the proper arrangement of these equations,
- simplification of a gear functional scheme and simultaneously simplification of an adequate graph. It also helps in an analysis of the direct path of passing a rotational movement from the input to the output of a gear. Moreover it allows for a ratio calculation for every drive upon a simplified subgraph. Underneath, firstly, the problem of creation of a functional scheme of an exemplary gear is analyzed.

3.1. Building of a gear functional scheme based upon its graph

The problem of conversion of a graph being a model of a mechanical system into its functional scheme has not been fully solved until now [7,11] especially in a fully automatic or algorithmic way. Underneath, the proposal is formulated how to interpret the consecutive

phases of creation of a functional scheme of a gear as a process of expanding a subgraph which evolves from an initial chosen vertex up to the whole graph. The task of drawing (in the graph) of particular edges and vertices (and drawing paralelly adequate mechanical elements) is performed in algorithmic way analyzing a graph and a scheme in a cross reference manner – one choice implies the next one in accordance with clear, unequivocal rules.
Assignment of the graph presented in Fig. 1b to the scheme shown in Fig. 1a is illustrated in Table 1 (in what follows denoted by T1). The procedure of assignment of a functional scheme to a graph can be considered in following steps:

(i) choice of a vertex in a polygon – the vertex *1* was chosen. It implies that all the edges starting from *1* are considered simultaneously i.e. *{1,2}, {1,6}, {1,7}*. They are drawn as bold lines in box T.1(1,2). The interpretation of the scheme in box T.1(1,1) is as follows: *1* – is the main axis of a gear. Moreover, *1* – is a carrier for the planets *2,6* and *7* because the graph edges are drawn by means of continuous lines,

(ii) edge *{2,7}* is a stripped line therefore planets *2* and *7* are a pair of geared wheels in mesh /see: first row of the table, box T1(1,2)/,

(iii) choice of the vertex *3*. It is the second consecutive vertex of the polygon therefore its representative i.e. gear element *3* (geared wheel with an internal toothing) rotates around the main axis. The edge *{3,6}* is drawn as a stripped line – see box T.1(2,2) – therefore elements *3* and *6* are in mesh, element *3* is added in the created functional scheme - box T.1(2,1),

(iv) choice of the vertex *4*. It is the polygon node so the element *4* rotates around the main axis, furthermore the edges *{4,6}* and *{4,7}* are drawn as stripped lines - box T.1(3.2). Therefore the element *4* is in mesh with two planets simultaneously – in mechanical sense it means that it is so called sun. This fact is illustrated in box T.1(3.1),

(v) choice of the vertex *5*. It is the last vertex of the polygon - box T.1(4.2). So, element *5* rotates also around the main axis. The edge *{5,2}* indicates that the elements *2* and *5* are in mesh - box T.1(4.1). The functional scheme has been fully done.

The transformation of the graph is here shown symbolically as turning more and more parts of the graph into bold lines. So transformation of a subgraph is really simple: adding graph elements one after another but simultaneously the functional scheme has to be built in accordance with the graph and the mechanical point of view. Finally the bold subgraph is turning into a final complete graph fully drawn as bold - so the procedure is finished. This procedure is needed for a task of creating of a family of design solutions of gears. The properties of graphs representing gears have been formulated [11]. Upon generating of the family of the graphs which fulfil these conditions [11] – the atlases of designs are built i.e. the sets of functional schemes of particular gears. However in many papers the process is finished on the phase of graphs generation believing that the final step could be done by a reader.
Therefore performance a conversion: “graph-functional scheme” described above – causes that the family of designs is understandable for engineers. This phase was frequently omitted in papers dealing with graph models of gears but sometimes it had caused that the existence of some mistakes [18] was not revealed. Only analysis of the gear scheme by an experienced engineer allows for full analysis of the machines from mechanical point of view. So, it seems that a fully algorithmic procedure has to be prepared applying an expert system. It exceeds the range and aim of the present paper.

Table 1. Building of the functional scheme upon the gear graph

Graph Representation of planetary gear		
	Functional scheme	Graph representation
	1	2
1		
2		
3		
4		

Here only the rough idea of transformation of the gear graph is given. Till now, it has not been turned into the computer program. More automatic and computerized version is described in [5].

3.2. Truss analysis

Second analyzed exemplary mechanical system is a truss – considered in a simplified version formulated in a previous chapter for statically determined trusses. We focus our attention on a problem of forces calculation but many other tasks can be also performed [4, 9]. The applied procedure of assignment of a graph to a truss is as follows:

(a1) truss is a linear graph itself, let's number/label its vertices by natural numbers from *1* to *n* or n capital letters *A, B, C*, etc.,

(a2) turning a graph into a directed graph is performed via introducing arcs according to the rule that direction of an arc is established from a vertex of a lower number to a vertex of greater number or taking into account a sequence of characters of the Latin alphabet,

(a3) additional tree is introduced which allows for systematic assignment of the cut matrix.

The discussed calculation method does not depend on labeling (numeration) of the graph vertices and the tree assignment where the root of the tree can be placed outside the existed graph or one of its vertices. The assignment of direction can be done in arbitrary way (it arises directly from numeration). Every arc symbolizes the direction of acting force. If the resulting forces are obtained as negative then it means that the direction of a force is reverse. If all forces are positive it means that all directions of acting forces are the same as direction of graph edges (arcs). The exemplary truss and its graph is presented in Fig. 2 a-f.

The derived method of the forces calculation [14] can be described in the following steps:

(s1) 2D truss is a graph itself considering nodes as vertices and bars(rods) as edges,

(s2) choice the special reference vertex, frequently the left hand, bottom node of the truss is chosen, enter the vertices numbering in an arbitrary way, it represents an external loading,

(s3) entering of the orientation of edges: from vertices of lower to upper number label, the method does not depend on numbering of vertices,

(s4) creation of the special tree - which branches connect the chosen vertex with all other ones creating the original truss, it is an additional (theoretical) tree allowing for further steps. The orientation of branches is from the chosen vertex to the truss nodes. Additional remark: the introduced tree allows for performance of all other tasks in an algorithmic manner as well as it has mechanical meaning – allows for inserting external forces to the calculation course,

(s5) creation of the cut matrix ($_2B$) of this new graph (Fig. 2f) – placing the columns associated with the branches of the tree at the beginning. It is a well-known fact from graph theory: elements of the cut matrix are as follows: $\{0, 1, -1\}$ what is relevant to the fact that a considered edge does not belong, it goes inside or it goes outside of a particular cut, respectively. Moreover – the cuts are generated by the tree branches, therefore the submatrix corresponding to them is a E matrix. The cut matrix for the considered truss (Fig. 2a) is presented by the formula (1):

$$
{}_2B = \begin{array}{c} \\ I \\ II \\ III \\ IV \\ V \\ VI \end{array}
\begin{array}{cccccc|ccccccccc}
I & II & III & IV & V & VI & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline
1 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -1 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1
\end{array} \qquad (1)
$$

where: *I, II, ..., VI* in the upper row denoted additional tree arcs and in the initial column – cuts connected with the end points of these arcs, respectively,

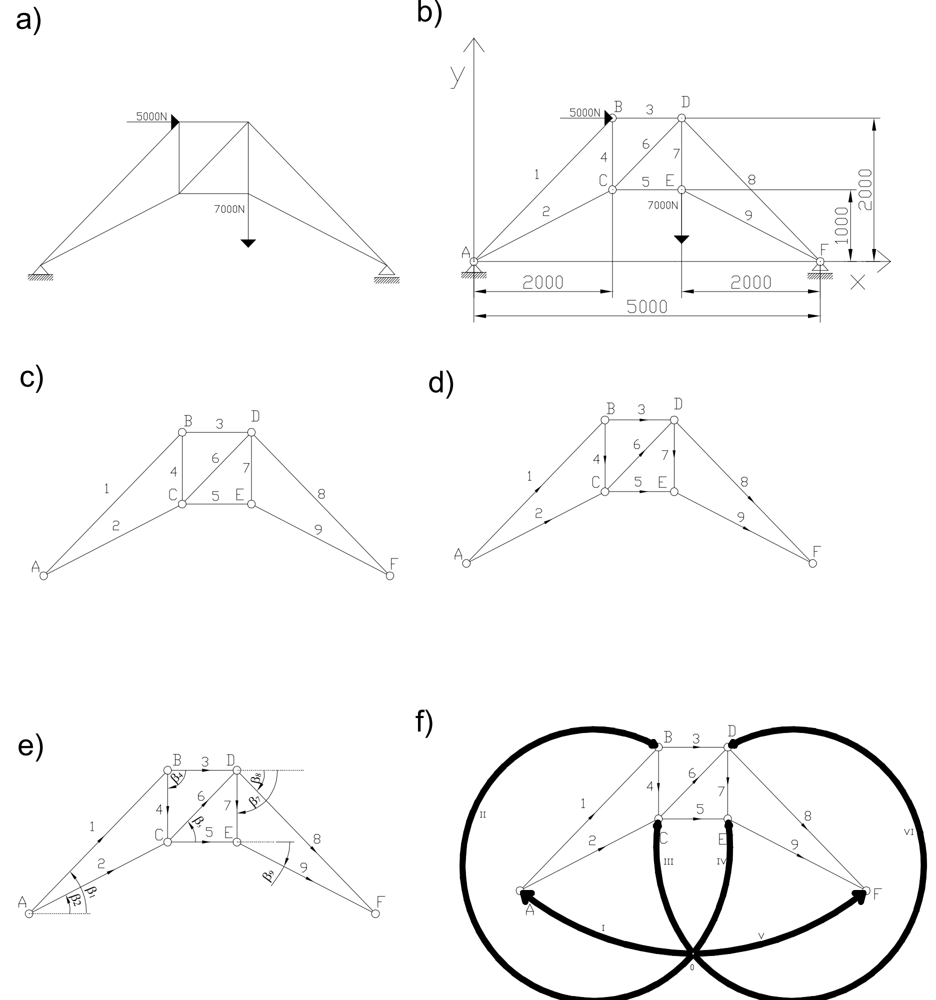


Fig.2 Truss and its graphs, (a) truss, (b) dimensions, (c) linear graph, (d) directed graph, (e) angles of truss rods ($\beta_3 = \beta_5 = 0$), (f) graph with an additional tree

(s6) creation of so called generalized (or rearranged) cut matrix where elements *0,1* and *–1* are replaced by matrices of rank *2 x 2* where these elements are doubled on the main diagonal and the remaining elements are zero (it is equivalent to consider *x* and *y* components separately). After performance of these actions, the first submatrix corresponding to branches of the tree is still a unit matrix. Therefore, the cut-matrix can be divided into two sub-matrices, first of them is the matrix representing the branches of the tree i.e. *E* matrix (with *1* on the main diagonal). Finally, the remaining submatrix (i.e. ${}_{20}\boldsymbol{B}^{R}$) is used for creation of the system of equations,

(s7) creation of trigonometry functions (or transformation) matrix – collecting the angles of inclinations of forces in the nodes and the rods according to the orthogonal coordinate system connected with some set of vertices, the original of the introduced co-ordinate system is placed in the point chosen in step (s2). It allows for considerations of *X* and *Y* components of forces. The matrix is denoted by $\boldsymbol{C}_{\beta}$,

(s8) creation of the vector matrix of external forces $\boldsymbol{F}_{Z}$*'*,

(s9) creation of the system of equations describing a static analysis of the statically-determined truss, additionally the rows connected with fixed supports has to be crossed out. In mechanical sense it means that reaction of the ground are not calculated via this approach,

(s10) solution of the system by finding the inverse matrix for the matrix generated in step (s7),

(s11) finding the wanted forces in the rods – collected in matrix ***S*** – using the formula (2):

$$S = [{}_{20}\boldsymbol{B}^{R}\ \boldsymbol{C}_{\beta}]^{-1}\ \boldsymbol{F}_{Z}\text{'} \tag{2}$$

where:

S – forces in bars,

${}_{20}\boldsymbol{B}^{R}$ – special matrix connected with the transformed graph representation of a truss,

$\boldsymbol{C}_{\beta}$ – matrix of cosinus and sinus functions, upon the geometrical dimensions of the truss elements,

$\boldsymbol{F}_{Z}$*'* – external forces.

The preciseness of the method depends on the procedure of creating a reverse matrix. In many cases the reaction of the ground are needed for mechanical analyses, so they can be additionally calculated upon classical attitude. It is a slight drawback of the graph-based approach.

External forces are described by formula (7), underneath.

The analysis of the truss is performed according to the above presented steps (s1)-(s11). Step (s6) consists in inserting the formulas:

$$1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix};\quad -1 = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix};\quad 0 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \tag{3}$$

The matrix ${}_{20}B$ has the following form:

$$
{}_{20}B = \begin{bmatrix}
-1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1
\end{bmatrix} \tag{4}
$$

Matrix of trigonometrical functions is as follows:

$$
C_\beta = \begin{bmatrix}
\frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\frac{\sqrt{2}}{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{2\sqrt{5}}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & \frac{\sqrt{5}}{5} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{\sqrt{2}}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{2}}{2} & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{2\sqrt{5}}{5} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -\frac{\sqrt{5}}{5}
\end{bmatrix} \tag{5}
\qquad
F'_z = \begin{bmatrix} 5000 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ -7000 \\ 0 \end{bmatrix} \tag{6}
$$

For example: the rod *1* is inclined by 45°. Therefore sin 45° = cos 45° = *sqrt(2)/2*; these are elements of the matrix C_β i.e.: C_β *(1,1) and* C_β *(2,1)* . The results obtained upon the formula 2 are as follows - forces in [N] in rods are: *S = [9337, -12989, 11 600, -6600, - 12400, 1131, -*

13200, 17536, - 13864], respectively. In the considered method the data representing the truss structure and the dimensions are separated in two distinct matrices. It is very important feature which is not characteristic for some other traditional methods. This fact allows also for e.g. evolutionary approach to truss design and optimization [19] what was done in the master thesis of the co-author (A. Jagosz). It was also shown that the graph method gives the same results as e.g. traditional one as well as achieved by means of FEM. Underneath (a) is used instead of Fig 2a. The transformations in Fig. 2 are as follows: (a)&(b)→(c) modeling in natural way – considering a structure as a graph; (c) →(d) according to the rule "a1", (d) →(e) calculation the angles [with a direction] in algorithmic way based upon vector and scalar products of adequate vectors; (d) →(e) adding tree according to the rule "s4". The final graph is suitable for performing of the force calculations.

3.3. Analysis of gear ratios

Third considered problem is calculation of ratios of an automatic gear box. The scheme of the gear can be found in document [12]. The gear consists of 6 elements: geared wheels, planets and carriers, furthermore it is equipped in the system of two brakes (Br-1,Br-2) and two clutches (Cl-1, Cl-2). Activation of some sequences of these control elements allows for changing the drives. In Table 2, the sequences of the control elements and the graphs describing the obtained mechanical systems are listed. Here, Hsu's graph is utilized but the authors of [12] Freudenstein's graphs had used. Due to the fact that here we focus our attention on graph transformations only the graph-theoretical aspects are discussed. The applied method of graph modeling – i.e. Hsu's graph as well as the applied notation are different than in [12], however the same results were obtained (after some recalculations). In [12] some different rules for assignment of negative elementary ratios had been utilized. The consecutive sequences of the control elements i.e. clutches and brakes causes allow that different drives are obtained (different output rotational speeds). The exemplary transformed graphs assigned to the available drives are shown in Table 2. The graph-based method of gear ratio calculation consists of the following steps: (st1) assignment of a graph to a gear; (st2) distinguishing of so called f-cycles – where every f-cycle contains a stripped line edge [i.e. two elements of gear are in mesh, their codes are assigned to the stripped line-ends]; (st3) derivation of the system of equation describing the kinematics of gear elements; (st4) solution of the system finding a searched ratio. In step "(st3)" every equation in the system is created according the algorithmic rule for dealing with indicators.

Therefore, for every f-cycle $(i,j)k$ the equation of f-cycle can be written automatically in the following form:

$$\omega_i - \omega_k = \pm N_{j,k} (\omega_j - \omega_k) \qquad (7)$$

where:

ω_i – rotational speed of the element i,

$N_{j,i}$ – ratio; $N_{j,i} = D_j / D_i = z_j / z_i$, $z_j > 0$, $z_i > 0$ in all cases; D and z with adequate indicators describes diameters and numbers of tooth on the particular geared wheels i and j, respectively,

Sign + for internal gearing, sign – otherwise.

Elements i and j are in mesh, element k is a carrier (an arm). The graph transformations are here connected with changes of the adequate functional schemes where the so called redundant

elements are omitted. Moreover transformed graphs have different sets of f-cycles which fully describe the kinematics for the consecutive stages (writing equations based on the formula (7)). The adequate sequences of the control elements (clutches, brakes) are listed in column 1 in Table 2.

Table 2. Transformed Hsu's graphs representing the available drives of the automatic transmission [12] and allowable adequate kinamatic analyses

	Activated control elements	Hsu's graph representing the adequate gear version - performing its movements via other rotating elements	Derived equation systems – describing kineamtics of the gear mode
	1	2	3
1	Cl-1, Br-1 1-st drive	F-cycles: (2,4)1; (3,5)4; (2,3)1 and (5,6)4.	$\begin{cases}\omega_2-\omega_1=+N_{42}(\omega_4-\omega_1)\\ \omega_3-\omega_4=-N_{53}(\omega_5-\omega_4)\\ \omega_2-\omega_1=-N_{32}(\omega_3-\omega_1)\\ \omega_5-\omega_4=+N_{65}(\omega_6-\omega_4)\end{cases}$ $\omega_1=0$ Solution: $\frac{\omega_4}{\omega_6}=\frac{N_{32}N_{65}}{N_{32}N_{65}+(N_{42}+N_{32})N_{35}}$
2	Cl-1, Br-2 2-nd drive	F-cycles: (3,5)4 and (5,6)4.	$\begin{cases}\omega_3-\omega_4=-N_{53}(\omega_5-\omega_4)\\ \omega_5-\omega_4=+N_{65}(\omega_6-\omega_4)\end{cases}$ $\omega_3=0$ Solution: $\frac{\omega_4}{\omega_6}=\frac{N_{65}}{N_{65}+N_{53}}$
3	Cl-1, Cl-2 3-rd drive	Direct passage of rotational movement	$\omega_4/\omega_6=1$
4	Cl-2, Br-1 reverse drive	F-cycles: (3,5)4 and (5,6)4.	$\begin{cases}\omega_2-\omega_1=+N_{42}(\omega_4-\omega_1)\\ \omega_2-\omega_1=-N_{32}(\omega_3-\omega_1)\end{cases}$ $\omega_1=0$ Solution: $\frac{\omega_4}{\omega_3}=-\frac{N_{32}}{N_{42}}$

Elements are considered as redundant when their movements do not have any influence on the output rotational speed. Transformation of the gear graph consists especially in: neglecting of these vertices (and adjacent edges) which represent the temporary redundant gear elements.

4 Final remarks

In the paper, the transformations of graphs representing mechanical systems were considered. It was shown that in case of some exemplary mechanical artifacts e.g. a planetary gear, an automatic gear and a truss – the applied transformations of their graphs allow for an effective and an algorithmic performance of some different engineer tasks: (t1) assignment of gear functional scheme to a graph, (t2) ratio analysis and (t3) calculation of truss forces. The relatively wide reference review was done, focusing attention only on transformation aspects of graph-based models of mechanical systems. General review on graphs in mechanics can be found in [11,15]. The summary of the considered problems of modeling mechanical systems by means of graphs can be shown via a scheme presented in Fig. 4. Graphs are here considered as Artificial Intelligence tools because after knowledge transformation from mechanics to graph theory the problem can be solved automatically. After retransformation of the obtained solution from graph theory field the solution understandable for mechanical engineers is obtained.

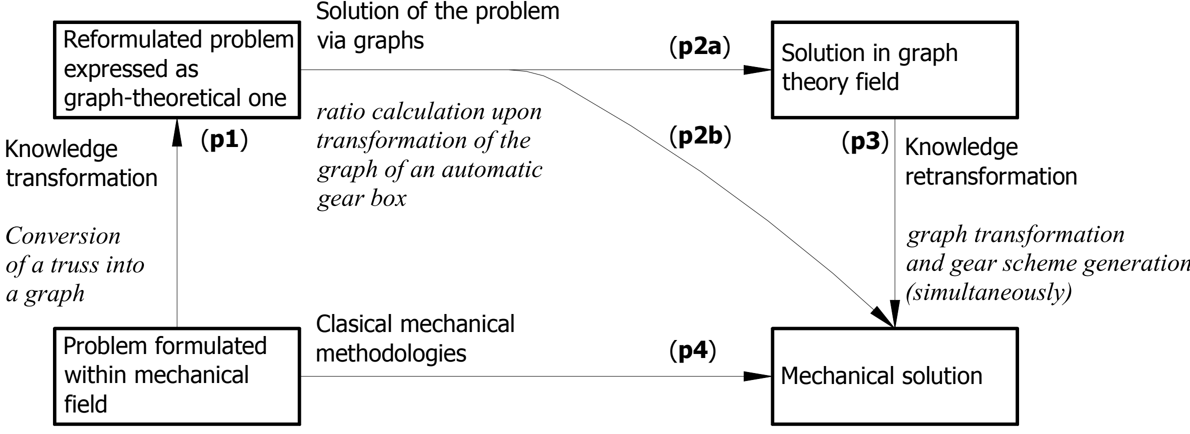


Fig.4. Scheme of usage of graph-based models of mechanical systems from AI point of view

The path (p1) in the scheme is illustrated in the paper via conversion truss-graph upon some graph transformations. The result (forces in the truss rods) is explicit so path (p2b) is used. In the case of ratio calculation we also move along the path (p2b). The path (p2a) is actively used is some tasks described in references e.g. for enumeration of designs [11]. Then retransformation is needed what was illustrated in the paper via conversion graph-functional scheme of a gear. Even wider range of mechanical tasks can be done using the graph-based models what is described in the quoted references, especially in the unique books [4,11]. Further authors' investigations are focused on usage of graph transformations in finding of the degenerate structures of gears and search for the redundant geared wheels or other redundant elements in the considered gear structures.

References

1. Borkowski A., Grabska E., Nikodem P., Strug B.: On genetic search of optimum layout skeletal structures, Progress Advances in Intelligent Computing in Engineering, VDI Verlag, No 180 , pp. 149-157, 2002.
2. Hliniak G., Strug B.: Graph grammars and evolutionary methods in graphic design, Machine Graphics & Vision, No 9, 1/2, pp. 5-13, 2000.
3. Hsu C.-H.: Graph notation for the kinematic analysis of differential gear trains, Journal of the Franklin Institute, Vol. 329 (5), pp. 859–867, 1992.
4. Kaveh A.: Structural Mechanics: Graph and Matrix Methods, Research Studies Press Ltd, Hertfordshire, 1995.
5. Li X., Schmidt L.C., He W., Li L., Qian Y.: Transformation of an EGT grammar: New grammar, new designs. ASME J. of Mech. Des., Vol. 126, No 4, pp. 753-756, 2004.
6. Rudolph S.: Semantic validation scheme for graph-based engineering design grammars; Chapter in the book - Ed. J. S. Gero: Design computing and cognition'06 (Part 7), pp. 541-560, Springer Netherlands, 2006.
7. Schmidt L. C., Shetty H., Chase S.: A graph grammar approach for structure synthesis of mechanisms, ASME Journal of Mechanical Design, Vol. 122, pp. 371-376, 2000.
8. Schmidt L. C., Cagan J.: GGREADA: A graph grammar-based machine design algorithm, Research in Engineering Design, Vol. 9, pp. 195-213, 1997.
9. Shai O.: Transforming engineering problems through graph representations, Advanced Engineering Informatics, Vol. 17, No. 2, pp. 77 - 93, April, 2003.
10. Szuba J., Schürr A., Borkowski A.: GraCAD – graph-based tool for conceptual design. Graph Transformation: Ist International Conference; Proceedings ICGT, pp.363-377, 2002.
11. Tsai L.-W.: Enumeration of kinematic structures according to function, CRC Press, Boca Raton, FL 33487, USA, 2001.
12. Tsai L-W., Magrab E.B., Mogalapalli S.N.: A CAD system for the optimization of gear ratios for automotive automatic transmissions, Department of Mechanical Engineering University of Maryland College Park, Maryland 20742. September 27, 1992.
13. Westfechel B.: Graph-based product and process management in mechanical design, RWTH Aachen, 1999.
14. Wojnarowski J., Bogucki Z.: Mechanics with exemplary calculation exercises (in Polish), Silesian University of Technology, Gliwice, 1991.
15. Wojnarowski J., Kopeć J., Zawiślak S.: Gears and graphs, Journal of Applied and Theoretical Mechanics, Vol. 44, No. 1, pp.139-162, 2006.
16. Wojnarowski J., Zawiślak S.: Modeling of mechanical system by means of matroid, Mechanism and Machine Theory, Vol. 36, No 6, pp. 717-724, 2001.
17. Zawiślak S.: Graph-based methodology as a support of conceptual design of planetary gears, Teoria maszyn i mechanizmów, Ed.: Józef Wojnarowski, Mirosław Galicki, Part 1, Oficyna Wydawnicza Uniwersytetu Zielonogórskiego, Zielona Góra, pp. 405-411, 2006.
18. Zawiślak S.: Artificial intelligence aided design of gears based on graph-theoretical models, IFToMM 2007: Twelfth World Congress in Mechanism and Machine Science, Besancon, France, June 17-21, 2007, (on line: www.iftomm2007.com).
19. Zawiślak S., Jagosz A.: An application of evolutionary algorithms and graph-based method of stress calculation for truss optimization, (in preparation) Polish Conference on Evolutionary Algorithms and Global Optimization, Warsaw TU, June 2008.

Author Index

Arbab, Farhad . . . 195
Azab, Karl . . . 249

Baresi, Luciano . . . 277
Biermann, Enrico . . . 222
Bottoni, Paolo . . . 65
Braatz, Benjamin . . . 95
Brandt, Christoph . . . 95
Brieler, Florian . . . 151
Bucchiarone, Antonio . . . 181

Costa, David . . . 195

Ehrig, Hartmut . . . 9, 137
Ehrig, Karsten . . . 9

Galeotti, Juan . . . 181
Ghezzi, Carlo . . . 277
Grabska, Ewa . . . 305
Grohmann, Davide . . . 109
Gruner, Stefan . . . 319

Hassan, Abubaker . . . 123
Hermann, Frank . . . 9
Hoffmann, Kathrin . . . 208

Jagosz, Adam . . . 332

Karsai, Gabor . . . 23
Kleppe, Anneke . . . 79
Köhler, Christian . . . 195
Kumar, Rahul . . . 51

Lambers, Leen . . . 137, 208
Le, Truong . . . 305

Mackie, Ian . . . 123
Maier, Sonja . . . 165
Manning, Greg . . . 235
Mazanek, Steffen . . . 291
Mercer, Eric . . . 51
Miculan, Marino . . . 109
Minas, Mark . . . 151, 165, 291
Mirenkov, Nikolay . . . 65
Mocci, Andrea . . . 277
Modica, Tony . . . 222
Monga, Mattia . . . 277
Myśliwiec, Miroslaw . . . 332

Narayanan, Anantha . . . 23

Padberg, Julia . . . 208
Pennemann, Karl-Heinz . . . 249
Plump, Detlef . . . 235
Prange, Ulrike . . . 208
Proenca, Jose . . . 195

Rein, Alexander . . . 208
Rensink, Arend . . . 79

Sato, Shinya . . . 123
Ślusarczyk, Grażyna . . . 305
Szypula, Lukasz . . . 332

Taentzer, Gabriele . . . 137
Tolvanen, Juha-Pekka . . . 7

Vangheluwe, Hans . . . 179

Watanobe, Yutaka . . . 65
Weinell, Erhard . . . 37

Xing, Cong-Cong . . . 263

Yoshioka, Rentaro . . . 65

Zawislak, Stan . . . 332