Electronic Communications of the EASST
Pre-proceedings

Proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009)

Guest Editors: Reiko Heckel, Artur Boronat
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Preface

This volume contains the proceedings of the Eighth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2009), held in York (UK) on March 28-29, 2009, as a satellite event to the European Joint Conference on Theory and Practice of Software (ETAPS'09).

The GT-VMT workshop series serves as a forum for all researchers and practitioners interested in the use of graph-based notation, techniques and tools for the specification, modeling, validation, manipulation and verification of complex systems. Previous workshops have been organized in Geneva (2000), Crete (2001), Barcelona (2002 and 2004), Vienna (2006), Braga (2007) and Budapest (2008).

The aim of the workshop is to promote engineering approaches that provide effective sound tool support for visual modeling languages, enhancing formal reasoning at the semantic level (e.g., for model analysis, transformation, and consistency management) in different domains, such as UML, Petri nets, Graph Transformation or Business Process/Workflow Models.

This year's workshop will have a special focus on visualisation, simulation, and verification of domain-specific languages (DSLs) to improve the automation and quality in model-driven and/or service-oriented processes. This year we received 21 submissions, from which 10 were accepted and 4 are subject to a second review process after the workshop. Accepted papers balance theoretical and applied concepts, including tool support. The workshop program has been organized in five technical sessions, in two days:

Saturday, March 28, 2009	Sunday, March 29, 2008
Pattern Matching and Verification	Evolution
Simulation	Visual DSLs
Visual Transformations	

The final proceedings will be published in the Electronic Communications of the EASST after the workshop. ECEASST is a fully refereed online journal and provides a forum for practitioners, educators and researchers for disseminating innovative research in the area of software and system technology. The volumes in the ECEASST series are available online at http://www.easst.org/eceasst.

The organizers acknowledge the support by the European Association of Software Science and Technology (EASST) and the IST Integrated Project SENSORIA (Software Engineering for Service-Oriented Overlay Computers) funded by the European Union in the 6th framework program as part of the Global Computing Initiative.

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 2009

Reiko Heckel, Artur Boronat

PC Chairs of GT-VMT 2009

Programme Chairs

Artur Boronat (University of Leicester, UK)
Reiko Heckel (University of Leicester, UK)

Programme Committee

Paolo Baldan (University of Padova, Italy)
Paolo Bottoni (University of Rome, Italy)
Artur Boronat (University of Leicester, UK)
Andrea Corradini (University of Pisa, Italy)
Claudia Ermel (TU Berlin, Germany)
Gregor Engels (University of Paderborn, Germany)
Holger Giese (HPI Potsdam, Germany)
Reiko Heckel (University of Leicester, UK)
Gabor Karsai (Vanderbilt University, US)
Jochen Küster (IBM Zürich Research)
Juan de Lara (University of Madrid, Spain)
Tihamér Levendovszky (TU Budapest, Hungary)
Mark Minas (Universität der Bundeswehr München, Germany)
Francesco Parisi-Presicce (University of Rome, Italy)
Arend Rensink (University of Twente, Netherlands)
Gabriele Taentzer (Univ. of Marburg, Germany)
Dániel Varró (TU Budapest, Hungary)
Ehrard Weinell (RWTH Aachen University)
Martin Wirsing (Ludwig-Maximilians-Universität München, Germany)
Albert Zündorf (University of Kassel, Germany)

External Reviewers

Bapodra, Mayur
Becker, Basil
Biermann, Enrico
Brieler, Florian
Foss, Luciana
Gadducci, Fabio
Gerth, Christian
Guerra, Esther
Hildebrandt, Stephan
Horváth, Ákos
Jubeh, Ruben
Maier, Sonja
Mazanek, Steffen
Ráth, István
Seibel, Andreas
Soltenborn, Christian
Varró-Gyapay, Szilvia

Table of Contents

Session on Visual DSLs

Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions

Hartmut Ehrig[1] **, Frank Hermann**[1] **and Christoph Sartorius**[1]

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

Abstract: Model transformations are a key concept for modular and distributed model driven development. In this context, triple graph grammars have been investigated and applied to several case studies and they show a convenient combination of formal and intuitive specification abilities. Especially the automatic derivation of forward and backward transformations out of just one specified set of rules for the integrated model simplifies the specification and enhances usability as well as consistency.

Since negative application conditions (NACs) are key ingredient for many model transformations based on graph transformation we embed them in the concept of triple graph grammars. As a first main result we can extend the composition/decomposition result for triple graph grammars to the case with NACs. This allows us to show completeness and correctness of model transformations based on rules with NACs and furthermore, we can extend the characterization of information preserving model transformations to the case with NACs.

The presented results are applicable to several model transformations and in particular to the well known model transformation from class diagrams to relational data bases, which we present as running example with NACs.

Keywords: model transformation, triple graph grammars, completeness, correctness, negative application conditions

1 Introduction

Model transformations based on triple graph grammars have been introduced in [Sch94, KS06]. In order to define a general framework independent of the specific domain and target language the correspondences between source and target models are defined as relational mappings, where forward and backward transformation rules are derived automatically.

In [EEE+07] we showed how to analyze bi-directional model transformations based on triple graph grammars with respect to information preservation, which is based on a decomposition and composition result for triple graph grammar sequences. Moreover, completeness and correctness of model transformations have been studied on this basis in [EEH08b, EEH08c]. All formal results in these papers, however, do not consider negative application conditions (NACs), which are very important for several practical applications (see [SK08]). The main purpose of this paper is to extend TGGs with NACs on a formal basis.

As a main result we show completeness, correctness and information preservation of model transformations with NACs. Our new result can be used to check, whether a model transformation performed by an algorithm using triple graph transformations with NACs such as [SK08] is correct (see Section 7). The relationship between forward and backward model transformation sequences was analyzed already in [EEE+07] based on a canonical decomposition and composition result for triple transformations and this paper extends it to the case with NACs.

In Section 2 we review triple graphs and introduce the case study for a model transformation from class models to relational data base models. Section 3 reviews triple rules and triple graph transformations as introduced in [Sch94] and extends them to the case with NACs showing that the composition and decomposition result is also valid for this extension. The second main result of correctness and completeness of model transformations based on source consistent model transformations with NACs is presented in Section 4 and explained on a concrete model transformation sequence of the example. Section 5 shows how the characterization of information preserving bidirectional model transformations is extended to the case with NACs. Related and future work are discussed in sections 6 and 7, respectively.

2 Review of Triple Graphs

Triple graph grammars [Sch94] are a well known approach for bidirectional model transformations. Models are defined as pairs of source and target graphs which are connected via an intermediate correspondence graph together with its embeddings into these graphs. 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 triple graphs, while triple rules are defined in Sec. 3 together with the extension to negative application conditions (NACs).

Definition 1 (Triple Graph and Triple Graph Morphism) Three graphs G_S, G_C, and G_T, called source, connection, and target graphs, together with two graph morphisms $s_G : G_C \to G_S$ and $t_G : G_C \to G_T$ form a triple graph $G = (G_S \xleftarrow{s_G} G_C \xrightarrow{t_G} G_T)$. G is called *empty*, if G_S, G_C, and G_T are empty graphs.

A triple graph morphism $m = (s,c,t) : G \to H$ between two triple graphs $G = (G_S \xleftarrow{s_G} G_C \xrightarrow{t_G} G_T)$ and $H = (H_S \xleftarrow{s_H} H_C \xrightarrow{t_H} H_T)$ consists of three graph morphisms $s : G_S \to H_S$, $c : G_C \to H_C$ and $t : G_T \to H_T$ 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. A typed triple graph G is typed over a triple graph $TG = (TG_S \leftarrow TG_C \to TG_T)$ by a triple graph morphism $t_G : G \to TG$.

Example 1 *The type graph of the example is given in Fig. 1 showing the structure of class diagrams in the source component and relational databases in the target component. Classes correspond to tables and attributes to columns. Throughout the example, originating from [SK08] and [EEE+07], elements are arranged left, center, and right according to the component*

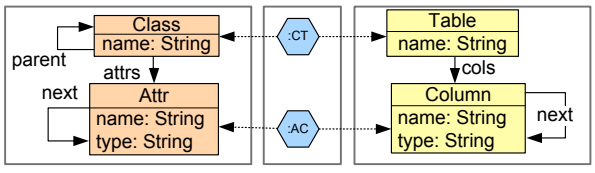


Figure 1: Triple type graph for CD2RDBM

types source, correspondence and target. Morphisms starting at a connection part are given by dotted arrow lines. Note that the case study is equipped with attribution, which is based on the concept of E-graphs [EEPT06].

The extension of the results of this paper to the case with attributes shall be straight forward, all results can be shown in the framework of weak adhesive HLR categories and hence, also for the category $\mathbf{AGraphs}_{ATG}$ of attributed graphs.

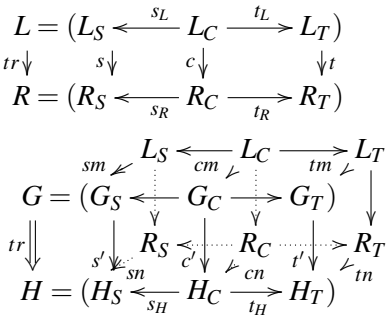
3 Triple Graph Grammars with NACs

Many model transformations use the concept of negative application conditions (NACs) introduced in [HHT96]. NACs can ensure termination and they can control the application of model transformation rules by defining forbidden structures as extensions of left hand sides of rules. If a forbidden structure is present around the selected match, the corresponding rule is not applicable and the match is invalid, i.e. NACs restrict the applicability of model transformation rules.

While triple graph grammars (TGGS) are an elegant way to descriptively define model transformations by defining triple rules that specify the synchronous creation of source and target model, formal results are mainly given for the case of TGGs without NACs. In this section we review triple rules, derivation of transformation rules and we define NACs for triple rules. The case study presents rules with NACs motivated by a similar model transformation in [SK08], where NACs are used to ensure well formed list structures.

A triple rule is used to build up source and target graphs as well as their connection graph, i.e. they are non-deleting. Structure filtering which deletes parts of triple graphs, is performed by projection operations only, i.e. structure deletion is not done by rule applications.

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 $tr = (s,c,t) : L \to R$. Given a triple rule $tr = (s,c,t) : L \to R$, a triple graph G and an injective 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 (H_S,s',sn), (H_C,c',cn) and (H_T,t',tn) in category **Graph** with induced morphisms s_H : $H_C \to H_S$ and $t_H : H_C \to H_T$. Morphism $n = (sn,cn,tn)$ is called comatch.



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.

Definition 3 (Triple, Source and Target Language) A set of triple rules TR defines the *triple language* $VL = \{G \mid \emptyset \Rightarrow^* G \text{ via } TR\}$ of triple graphs. *Source language* VL_S and *target language* are derived by projection to the triple components, i.e. $VL_S = proj_S(VL)$ and $VL_T = proj_T(VL)$,

where $proj_X$ is a projection defined by restriction to one of the triple components, i.e. $X \in \{S,C,T\}$.

Definition 4 (Derived Triple Rules) From each triple rule $tr = L \to R$ we have the following source, forward, target and backward rules:

$$\begin{array}{ccc} L_S & \leftarrow \emptyset \rightarrow & \emptyset \\ {\scriptstyle s}\downarrow & \downarrow & \downarrow \\ R_S & \leftarrow \emptyset \rightarrow & \emptyset \end{array} \qquad \begin{array}{ccc} \emptyset & \leftarrow \emptyset \rightarrow & L_T \\ \downarrow & \downarrow & {\scriptstyle t}\downarrow \\ \emptyset & \leftarrow \emptyset \rightarrow & R_T \end{array} \qquad \begin{array}{ccc} R_S & \xleftarrow{s\circ s_L} & L_C \xrightarrow{t_L} L_T \\ {\scriptstyle id}\downarrow & & {\scriptstyle c}\downarrow \quad \downarrow{\scriptstyle t} \\ R_S & \xleftarrow{s_R} & R_C \xrightarrow{t_R} R_T \end{array} \qquad \begin{array}{ccc} L_S & \xleftarrow{s_L} & L_C \xrightarrow{t\circ t_L} R_T \\ {\scriptstyle s}\downarrow & & {\scriptstyle c}\downarrow \quad \downarrow{\scriptstyle id} \\ R_S & \xleftarrow{s_R} & R_C \xrightarrow{t_R} R_T \end{array}$$

source rule tr_S | target rule tr_T | forward rule tr_F | backward rule tr_B

Source rules allow to create all elements of VL_S as restriction of VL, but they contain less restrictions for matches during transformation in comparison to their corresponding complete triple rules. Thus, they possibly allow to generate more elements than VL_S contains. This means that in general we have inclusion $VL_S \subseteq VL_{S0} = \{G_S \,|\, \emptyset \Rightarrow^* G_S \; via \; TR_S\}$ resp. $VL_T \subseteq VL_{T0} = \{G_T \,|\, \emptyset \Rightarrow^* G_T \; via \; TR_T\}$, where TR_S and TR_T are the sets of source resp. target rules derived from TR.

Definition 5 (General Negative Application Condition) Given a triple rule $tr = (L \xrightarrow{tr} R)$, a general negative application condition (NAC) (N,n) consists of a triple graph N and an injective triple graph morphism $n : L \to N$.
A match $m : L \to G$ is NAC consistent if there is no injective $q : N \to G$ such that $q \circ n = m$. A triple transformation $G \overset{*}{\Rightarrow} H$ is NAC consistent if all matches are NAC consistent.

Definition 6 (Source-Target Negative Application Condition) A source-target NAC (N,n) is a NAC with injective triple graph morphism $n : L \to N$ with $n = (n_S, id_{L_C}, id_{L_T})$ or $n = (id_{L_S}, id_{L_C}, n_T)$.
This means a source-target NAC is a NAC which only prohibits the existence of certain structures either in the source (*source NAC*) or in the target part (*target NAC*).

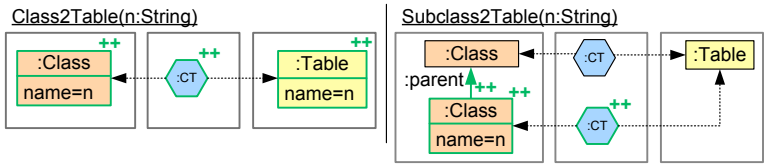


Figure 2: Rules for transforming classes to tables

In most usecases we encounter only source-target NACs, therefore we regard them as the standard case. In the following when speaking of NACs we always mean source-target NACs. If this is not the case we will explicitly refer to the term general NAC.

Definition 7 (Derived Triple Rules with NACs) Given a triple rule tr with NACs and let $\underline{tr}$ be its underlying triple rule without NACs. Let $\underline{tr_S}$, $\underline{tr_T}$, $\underline{tr_F}$ and $\underline{tr_B}$ be the derived rules from $\underline{tr}$ according to Def. 4. Then, source rule tr_S, target rule tr_T, forward rule tr_F and backward rule tr_B are given by the underlying rules $\underline{tr_S}$, $\underline{tr_T}$, $\underline{tr_F}$ and $\underline{tr_B}$, where additionally tr_S as well as tr_B contain all source NACs of tr and tr_T as well as tr_F contain all target NACs of tr.

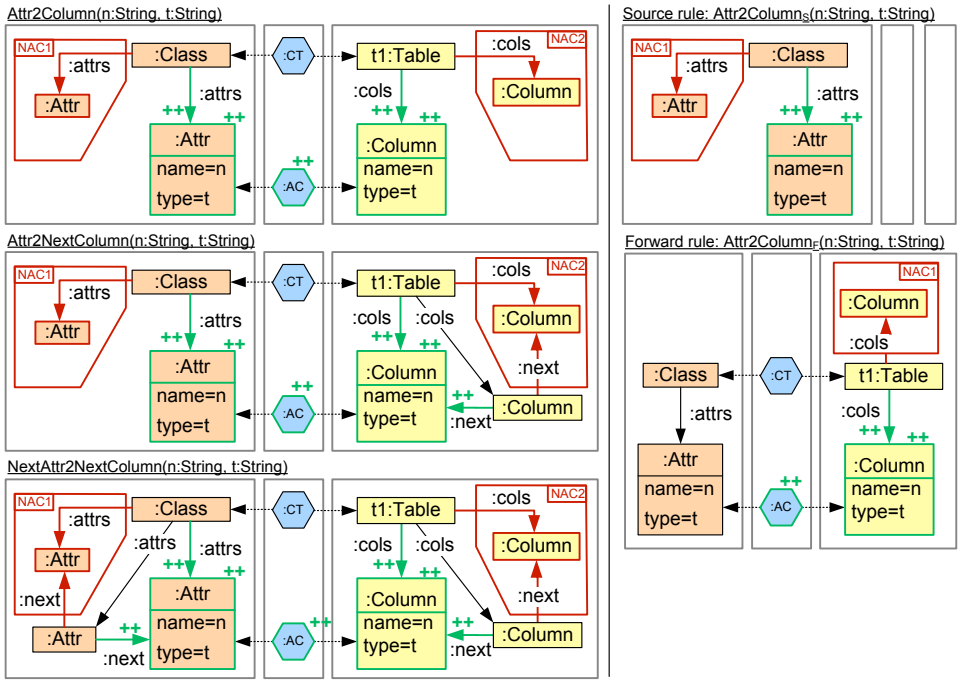


Figure 3: Rules for transforming attributes to columns and derived source and forward rule

Example 2 (Triple Rules) *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 green "++" and marked by green line coloring. 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. NACs are indicated by red frames with label "NAC" around the extension of the left hand side of a rule. Each forward rule is derived from a triple tr rule as follows: The source components which are created in tr are preserved by tr_F, i.e. they are in the left hand side. The source NAC is omitted and the rest of tr keeps the same. For example the forward rule of "Attr2Colum" is derived by omitting "NAC1" and adding to the left hand side the attribute node with its connecting edge to the class node shown on the right part of Fig. 3.*

Theorem 1 as a main technical result of the paper shows that TGT-sequences can be decomposed to source and forward sequences and composed out of them. All together this correspondence is bijective. The result uses the following notion of match consistency.

Definition 8 (Match and Source 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 $G_{00} \overset{tr_S^*}{\Longrightarrow} G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$ be a TGT-sequence with (mi_S, ni_S) being match and comatch of tri_S (respectively (mi_F, ni_F) for tri_F) then match consistency of $G_{00} \overset{tr_S^*}{\Longrightarrow} G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$ means that the S-component of the match mi is uniquely determined by the comatch ni_S $(i = 1,\ldots,n)$.

A TGT-sequence $G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$ is source consistent, if there is a match consistent sequence $\emptyset \overset{tr_S^*}{\Longrightarrow} G_{n0} \overset{tr_F^*}{\Longrightarrow} G_{nn}$. Note that by source consistency the application of the forward rules is controlled by the source sequence, which generates the given source model.

Theorem 1 (Decomposition and Composition of TGT-Sequences with NACs)

1. **Decomposition**: *For each TGT-sequence*
$$G_0 \overset{tr_1}{\Longrightarrow} G_1 \Rightarrow \ldots \overset{tr_n}{\Longrightarrow} G_n \tag{1}$$
with NACs there is a corresponding match consistent TGT-sequence
$$G_0 = G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \Rightarrow \ldots \overset{tr_{nS}}{\Longrightarrow} G_{n0} \overset{tr_{1F}}{\Longrightarrow} G_{n1} \Rightarrow \ldots \overset{tr_{nF}}{\Longrightarrow} G_{nn} = G_n \tag{2}$$
with NACs.

2. **Composition:** *For each match consistent transformation sequence* (2) *with NACs there is a canonical transformation sequence* (1) *with NACs.*

3. **Bijective Correspondence:** *Composition and decomposition are inverse to each other.*

Remark 1 (Injective matches) *Opposed to the version without NACs in [EEE+07] the matches of the triple rules are required to be injective. If we allow non-injective matches, then we must allow n and q in definition 5 to be non-injective as well.*

Proof of Theorem 1. This proof is based on the proof without NACs in [EEE+07] and Lemmas 1 and 2 below proven in [EHS09]. In a first step we want to decompose the match consistent NAC-consistent TGT-sequence (1) with injective matches into an intermediate version

$$G_0 = G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \overset{tr_{1F}}{\Longrightarrow} G_{11} \overset{tr_{2S}}{\Longrightarrow} \ldots \overset{tr_{nS}}{\Longrightarrow} G_{n(n-1)} \overset{tr_{nF}}{\Longrightarrow} G_{nn} = G_n \tag{3}$$

which is match consistent and NAC-consistent.

In [EEE+07] it has been shown that any tr is equal to the E-concurrent rule $tr_S \star_E tr_F$ without NACs with $E = L_F$ - the left hand side of the forward rule. Using this result following Lemma 1 multiple times we are able to split the triple rules with NACs until we obtain sequence (3).

Lemma 1 *The injective match of a triple rule tr is NAC-consistent if and only if the injective matches of the derived rules tr_S and tr_F are NAC-consistent.*

Now we want to reorder the rules until we have sequence (2). In [EEE+07] it has been shown that tr_{iS} and tr_{jF} are sequentially independent for $i > j$ without NACs. Following Lemma 2 multiple times finally leads to sequence (2) which is still match consistent and NAC-consistent.

Lemma 2 *Given sequentially independent rules tr_{2S} and tr_{1F} with NACs the following holds: The injective matches of $G_{10} \overset{(tr_{1F},m_1)}{\Longrightarrow} G_{11} \overset{(tr_{2S},m_2)}{\Longrightarrow} G_{21}$ are NAC consistent if and only if the injective matches of $G_{10} \overset{(tr_{2S},m_{2'})}{\Longrightarrow} G_{20} \overset{(tr_{1F},m_{1'})}{\Longrightarrow} G_{21}$ are NAC consistent too.*

Analogously we can transform sequence (2) back into sequence (1). The bijective correspondence follows from the bijective correspondence of the Concurrency Theorem and the Local

Church-Rosser Theorem in conjunction with the equivalence of the NAC-consistency according to Lemma 1 and 2.

□

4 Completeness and Correctness of Model Transformations with NACs

Model transformations with NACs from models of the source language VL_{S0} to models of the target language VL_{T0} can be defined on the basis of forward rules as shown in [EEE+07] without NACs. Vice versa, it is also possible to define backward transformations from target to source graphs using derived backward rules leading to bidirectional model transformations. In this section we analyze completeness and correctness of model transformations. Main results are based on the composition and decomposition result in Thm. 1 in Sec. 3.

Definition 9 (Model Transformation) $MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$ is a model transformation from G_S to H_T, if $G \overset{tr_F^*}{\Longrightarrow} H$ is source consistent with NACs, where G_S and H_T are the source and target graphs of G and H, respectively.

As pointed out already source consistency with NACs of $G \overset{tr_F^*}{\Longrightarrow} H$ means that the forward sequence is controlled by the corresponding source sequence $\emptyset \overset{tr_S^*}{\Longrightarrow} G$ which generates G. Model transformations are correct and complete with respect to the source and target language $VL_S = proj_S(VL)$ and $VL_T = proj_T(VL)$ (see Def. 3).

Theorem 2 (Correctness with NACs) *Each model transformation $MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$ is correct, i.e. $G_S \in VL_S$ and $H_T \in VL_T$.*

Proof. $(G \overset{tr^*}{\Longrightarrow} H)$ source consistent $\Rightarrow \exists\, (\emptyset \overset{tr_S^*}{\Longrightarrow} G \overset{tr_F^*}{\Longrightarrow} H)$ match consistent and $G_S = H_S$ $\Rightarrow \exists\, (\emptyset \overset{tr^*}{\Longrightarrow} H)$ by Thm. 1 $\Rightarrow H \in VL$ and $H_T \in VL_T$ and $G_S = H_S \in VL_S$. □

Theorem 3 (Completeness with NACs) *For each $H \in VL$: $\exists$ model transformation $MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$ with $G_S \in VL_S$, $H_T \in VL_T$. This means in particular:*

- *For each $H_T \in VL_T$: $\exists\, G_S \in VL_S$ and model transformation $MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$,*
- *For each $G_S \in VL_S$: $\exists\, H_T \in VL_T$ and model transformation $MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$.*

Proof. $H \in VL \Rightarrow \exists\, (\emptyset \overset{tr^*}{\Longrightarrow} H) \overset{Thm.1}{\Longrightarrow} \exists$ match consistent $(\emptyset \overset{tr_S^*}{\Longrightarrow} G \overset{tr_F^*}{\Longrightarrow} H)$ and $G_S = H_S \Rightarrow$ $G_S \in VL_S, H_T \in VL_T$ and $G \overset{tr_F^*}{\Longrightarrow} H$ is source consistent $\Rightarrow MT = (G_S, G \overset{tr_F^*}{\Longrightarrow} H, H_T)$ is model transformation. □

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. The resulting data base of the following model transformation is correctly typed and completely corresponds to the class diagram, which is the source model of the transformation.

Step	Forward Sequence Elements		Backward Sequence Elements	
	Matched	Created	Matched	Created
1	s1	c1,t1	t1	s1,c1
2	s1,c1,t1,s4,s9	c4	s1,c1,t1	s4,s9,c4
3	s1,s2,s7,c1,t1	c2,t2,t5	s1,c1,t1,t2,t5	s2,s7,c2
4	s1-s3,s6-s8,c1,t1,t2,t5	c3,t3,t6,t7	s1,c1,t1-t3,s2,c2,s7,t5-t7	c3,s3,s6,s8
5	s4,s5,s10,c4,t1,t3,t6	c5,t4,t8,t9	s4,c4,t1,t3,t4,t8,t9	c5,s5,s10

Table 1: Steps forward and backward model transformation

Example 3 *Fig. 4 shows triple graph G_5 of the model transformation $(G_S = G_{0,S}, G_0 \overset{tr_F^*}{\Longrightarrow} G_5, G_T = G_{5,T})$ with the following forward sequence: $G_0 \overset{Class2Table}{\Longrightarrow} G_1 \overset{Subclass2Table}{\Longrightarrow} G_2 \overset{Attr2Col}{\Longrightarrow} G_3 \overset{NextAttr2NextCol}{\Longrightarrow} G_4 \overset{Attr2NextCol}{\Longrightarrow} G_5$, where G_0 is generated by the corresponding source sequence $\emptyset \overset{tr_S^*}{\Longrightarrow} G_0$. All elements are labeled with numbers specifying the matches and the created objects for each transformation step according to the left part of Table 1. G_S is given by G_5 restricted to elements of the class diagram part. After creating the table and building up the correspondences to the class nodes in the first two derivation steps, rules for translating attributes are applied. All steps of the sequence respect the NACs and furthermore, they correspond to a suitable source sequence making the forward transformation source consistent. In the third step, rule "Attr2Column" is applied and translates attribute "s2" to column "t2". Attribute s_3 is generated after s_2 in the source sequence, which is required by the source NAC of "NextAttr2NextColumn". Thus, the corresponding forward transformation translates s_3 after s_2. The remaining two attributes are translated by "NextAttr2NextColumn" and "Attr2NextColumn", where the target NACs ensure that the created columns are inserted after the last existing one of table "t1". Thus, the ordering of the created columns is not completely determined by the source model itself, but depends on the chosen source sequence. The nodes and edges of correspondence and target component as well as the morphisms $(G_{5,S} \leftarrow G_{5,C} \rightarrow G_{5,T})$ are created during the forward transformation.*

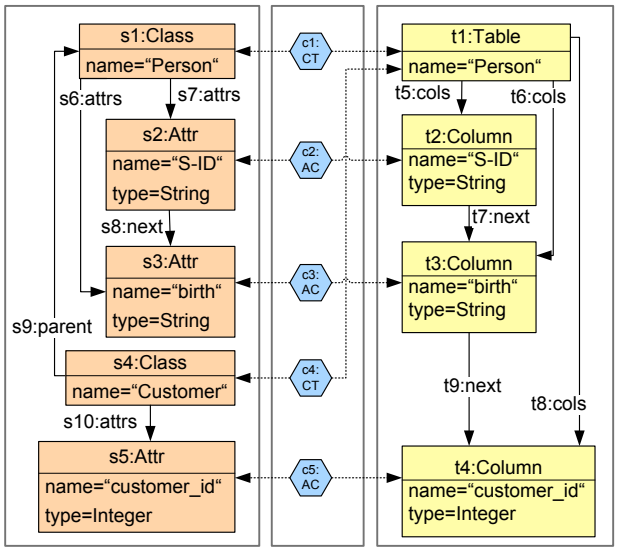


Figure 4: G_5 of Forward Sequence

5 Information Preserving 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 com-

position result, which is extended to the case with NACs in this paper (see Theorem 1).

Theorem 1 and its dual version lead to the following equivalence of forward and backward TGT-sequences with source-target NACs, which can be derived from the same general TGT-sequence.

Theorem 4 (Equivalence of Forward and Backward TGT-sequences with source-target NACs) *Each of the following TGT-sequences with source-target NACs implies the other ones where the matches are uniquely determined by each other.*

1. $G_0 \overset{tr_1}{\Longrightarrow} G_1 \overset{tr_2}{\Longrightarrow} G_2 \Longrightarrow ... \overset{tr_n}{\Longrightarrow} G_n$ (1)
2. $G_0 = G_{00} \overset{tr_{1S}}{\Longrightarrow} G_{10} \Longrightarrow ... \overset{tr_{nS}}{\Longrightarrow} G_{n0} \overset{tr_{1F}}{\Longrightarrow} G_{n1} \Longrightarrow ... \overset{tr_{nF}}{\Longrightarrow} G_{nn} = G_n$, (2)
   which is match consistent. In this case we have: $G_{00,T} = G_{n0,T}$, $G_{n0,S} = G_{nn,S}$.
3. $G_0 = G_{00} \overset{tr_{1T}}{\Longrightarrow} G_{01} \Longrightarrow ... \overset{tr_{nT}}{\Longrightarrow} G_{0n} \overset{tr_{1B}}{\Longrightarrow} G_{1n} \Longrightarrow ... \overset{tr_{nB}}{\Longrightarrow} G_{nn} = G_n$, (3)
   which is match consistent. In this case we have: $G_{00,S} = G_{0n,S}$, $G_{0n,T} = G_{nn,T}$.

Proof. Theorem 4 is a direct consequence of Theorem 1 concerning decomposition and composition of forward TGT-sequences with NACs and its dual version for target rules tri_T and backward rules tri_B where match consistency in Part 3 is defined by the T-components of the matches. □

Theorem 5 (Information Preserving Forward Transformation)
Each source consistent forward TGT-sequence $G \overset{tr_F^*}{\Longrightarrow} H$ *is backward information preserving, i.e. for* $K = (\emptyset \leftarrow \emptyset \rightarrow H_T)$, *there is a backward TGT-sequence* $K \overset{tr_B^*}{\Longrightarrow} H$, *which means that the source model* G_S *can be reconstructed from the target model* H_T*:*

$$G \overset{tr_F^*}{\Longrightarrow} H \overset{proj_T}{\longrightarrow} K \overset{tr_B^*}{\Longrightarrow} H \text{ with } G_S = H_S.$$

Proof. $G \overset{tr_F^*}{\Longrightarrow} H$ is source consistent which implies the existence of (2) $\emptyset \overset{tr_S^*}{\Longrightarrow} G \overset{tr_F^*}{\Longrightarrow} H$ being match consistent with $G_S = H_S$. By Theorem 4 with $G_0 = \emptyset$, $G_{n0} = G$, $G_{0n} = K$ and $G_n = H$ we obtain (3) $\emptyset \overset{tr_T^*}{\Longrightarrow} K \overset{tr_B^*}{\Longrightarrow} H$ being match consistent with $K_T = H_T$ and $H_S = G_S$ leading to $G \overset{tr_F^*}{\Longrightarrow} H \overset{proj_T}{\longrightarrow} K \overset{tr_B^*}{\Longrightarrow} H$. Hence, $G \overset{tr_F^*}{\Longrightarrow} H$ is backward information preserving. □

Example 4 *Example 3 Table 1 shows that for the given model transformation* $G_0 \overset{tr_F^*}{\Longrightarrow} G_5$ *according to Thm. 5 there is an inverse backward transformation* $G_5|_T \overset{tr_B^*}{\Longrightarrow} G_5$, *i.e. the source model can be reconstructed. However, there are also target consistent backward transformations* $G_5|_T \overset{tr_B^*}{\Longrightarrow} G_5'$ *with* $G_{5,S}' \neq G_{0,S}$, *because there are some class models with different inheritance relations corresponding to the given data base model.*

6 Related Work

Correctness of model transformations can be analyzed from different perspectives. Baleani et. al. motivate in [BFM+05] that correctness of model transformations for industrial tools should

be based on formal models in order to ensure correctness by construction. For this purpose they suggest to use a block diagram formalism, called synchronous reactive model of computation (SR MoC). However, correct interpretation of the model transformation rules does not imply a correct result, such that it is a model of the target language. Semantical correctness is discussed by Karsai et. al. in [KN06], where specific behavior properties of the source model shall be reflected in the target model. This property can be checked for a restricted class of models. In [EE08] semantical correctness is ensured by using the rules for the model transformation also for the transformation of the operational semantics, which is given by graph rules. This way the behaviour of the source model can be compared with the one of the target model by checking mixed confluence. However, this paper concentrates on syntactical correctness based on the integrated language generated by the triple rules.

Our example in this paper presents a model transformation with NACs from class diagrams to relational data bases and it is based on the grammars defined in [EEE+07] and especially on [SK08]. In contrast to the presented algorithm in [SK08] for controlling the model transformations we introduced NAC consistency based on source consistent forward sequences. In this way we could extend several important results to the case of TGGs with NACs. In particular, model transformations given by source consistent forward transformations are correct and complete with respect to VL by Theorems 2 and 3. While a formal proof of correctness for the above mentioned algorithm is not given in [SK08], completeness of the algorithm is effectively not ensured, because recursion calls may cause transformations that produce structures forbidden by other necessary rule applications.

But still the algorithm in [SK08] convinces to be an elegant approach for a restricted class of relations to efficiently detect correct rule orderings for a subset of model transformations. This opens the possibility to combine efficiency with the here presented results in the following way: Each model transformation with NACs given by an efficient algorithm can be checked to be correct by performing the test of source consistency presented as Fact 2 in [EEH08c], which is now also valid for model transformations with NACs according to Thm. 1.

Model transformations based on triple rules with NACs were also analyzed in [EP08] for a restricted class of triple rules with distinct kernel elements. Special NACs of forward rules ensure that kernels are not translated twice and kernel typing guarantees that each rule produces exactly one kernel. For this restricted class of triple graph grammars local confluence and termination can be analyzed and thus, model transformations can be checked for functional behaviour.

7 Conclusion

This paper focusses on syntactical correctness and completeness. In order to analyze these important properties we extended the composition and decomposition result for triple graph transformations in [EEE+07] to the case with NACs, i.e. TGT sequences with NACs can be decomposed into source and forward as well as target and backward transformations, respectively, and vice versa. Based on this fundamental property we have shown that source consistent model transformations are correct and complete with respect to the language given by the original triple rules. This extends the result in [EEH08a] to triple rules with NACs.

Source consistency of model transformations guarantees that each element of the source model

was matched by a model transformation rule and correspondences to target model elements were created. A suitable source sequence can be calculated by parsing the source model using the source rules and the corresponding forward transformation can be checked to be source consistent. Alternatively, forward transformations can be created by an arbitrary strategy and checked afterwards using the algorithm for checking source consistency presented in [EEH08c]. Source consistency is not restricted to cases, where all source nodes have to be connected via correspondence nodes. Therefore, correctness of many algorithms for model transformations based on triple rules with NACs can be checked using the source consistency check.

According to [EEH08a] model integration sequences can be characterized as special model transformation sequences, such that the results of this paper for model transformation can be transferred to model integrations based on triple rules in a next step.

In this paper we focused on NACs which specify conditions on separately source and target elements. They are sufficient to most model transformations, which were considered by case studies so far. However, future work will include the analysis of how to handle general NACs and their relevance for language specification. An interesting problem - which could be solved with general NACs - is termination, where a parsing of the source model is omitted. A possibility may be to introduce additional NACs for the forward rules, such that source elements, which are already in correspondence with target elements, cannot be matched again for translation. In this way termination for a restricted class of rules could be ensured automatically. But note that NACs, which are equal to the right hand side of a forward rule, are not sufficient, because in this case matches of the transformation are required to be essential.

Bibliography

[BFM+05] M. Baleani, A. Ferrari, L. Mangeruca, A. L. Sangiovanni-Vincentelli, U. Freund, E. Schlenker, H.-J. Wolff. Correct-by-Construction Transformations across Design Environments for Model-Based Embedded Software Development. *Design, Automation and Test in Europe Conference and Exhibition* 2:1044–1049, 2005. doi:http://doi.ieeecomputersociety.org/10.1109/DATE.2005.105

[EE08] H. Ehrig, C. Ermel. Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation. In *Proc. International Conference on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 194–210. Springer Verlag, Heidelberg, 2008. http://tfs.cs.tu-berlin.de/publikationen/Papers08/EE08a.pdf

[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

[EEH08a] H. Ehrig, K. Ehrig, F. Hermann. From Model Transformation to Model Integration based on the Algebraic Approach to Triple Graph Grammars. In Ermel et al. (eds.), *Proc. Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT'08)*. Volume 10. EC-EASST, 2008. http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/19

[EEH08b] H. Ehrig, C. Ermel, F. Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars. In Karsai and Taentzer (eds.), *Proc. Third International Workshop on Graph and Model Transformation (GraMoT'08)*. ACM, New York, NY, USA, 2008. doi:http://doi.acm.org/10.1145/1370175.1370244 http://tfs.cs.tu-berlin.de/publikationen/Papers08/EEH08.pdf

[EEH08c] H. Ehrig, C. Ermel, F. Hermann. On the Relationship of Model Transformations Based on Triple and Plain Graph Grammars (Long Version). Technical report 2008/05, Technische Universität Berlin,Fakultät IV, 2008. http://iv.tu-berlin.de/TechnBerichte/2008/2008-05.pdf

[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

[EHS09] H. Ehrig, F. Hermann, C. Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions (Long Version). Technical report 2009/03, TU Berlin, 2009. http://iv.tu-berlin.de/TechnBerichte/2009/2009-03.pdf

[EP08] H. Ehrig, U. Prange. Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels. In Ehrig et al. (eds.), *Proc. International Conference on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 178–193. Springer Verlag, Heidelberg, 2008. http://tfs.cs.tu-berlin.de/publikationen/Papers08/EP08a.pdf

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

[KN06] G. Karsai, A. Narayanan. On the Correctness of Model Transformations in the Development of Embedded Systems. In Kordon and Sokolsky (eds.), *Monterey Workshop*. LNCS 4888, pp. 1–18. Springer, 2006.

[KS06] A. Königs, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. In *Proc. SegraVis School on Foundations of Visual Modelling Techniques*. Volume 148, pp. 113–150. Electronic Notes in Theoretical Computer Science, Elsevier Science, Amsterdam, 2006. http://www.sciencedirect.com/science/journal/15710661

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer (ed.), *WG94 20th Int. Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Springer Verlag, Heidelberg, 1994.

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In Ehrig et al. (eds.), *ICGT*. LNCS 5214, pp. 411–425. Springer, 2008. doi:10.1007/978-3-540-87405-8_28

Repotting the Geraniums: On Nested Graph Transformation Rules

Arend Rensink[1] **and Jan-Hendrik Kuperus**[2]

[1]rensink@cs.utwente.nl
Department of Computer Science
Universiteit Twente, Postbus 217, 7500 AE Enschede, The Netherlands

[2]jan-hendrik.kuperus@sogeti.nl
Sogeti Nederland BV, Postbus 76, 4130 EB Vianen, The Netherlands

Abstract: We propose a scheme for rule amalgamation based on nested graph predicates. Essentially, we extend all the graphs in such a predicate with right hand sides. Whenever such an enriched nested predicate matches (i.e., is satisfied by) a given host graph, this results in many individual match morphisms, and thus many "small" rule applications. The total effect is described by the amalgamated rule. This makes for a smooth, uniform and very powerful amalgamation scheme, which we demonstrate on a number of examples. Among the examples is the following, which we believe to be inexpressible in very few other parallel rule formalism proposed in the literature: *repot all flowering geraniums whose pots have cracked.*

Keywords: Geraniums, Graph Transformation, Rule Amalgamation, Quantified Rules, Nested Rules, Parallel Rules

1 Introduction

Standard graph transformation rules are existential. By this we mean that a rule applies wherever there exists a matching of its left hand side into the host graph, and the effect of its application is limited to the homomorphic image of its left hand side under the matching.[1]

The existentiality of rules certainly has advantages, such as their reversibility (at least in a DPO setting). However, there are certain types of transformation where this is clearly a limitation. For instance, if a certain change has to applied *universally*, that is, to *all* sub-graphs with a certain structure, then this can be quite cumbersome to model using existential rules.

This limitation has been recognised especially by tool builders, who after all are in the business of using graph transformations for practical cases; hence, virtually all graph transformation tools have some way to define *rule schemes* or *parallel rules*. (A thorough overview and comparison of related work follows later.) Not all of the solutions have a firm theoretical justification, but the work of Taentzer [28, 26] is ground-breaking in explaining the effect of parallel rule application in a general setting, namely as *rule amalgamation*.

In this paper we describe a way to specify rule amalgamation, based on the concept of *nested graph predicates* of [21, 11]. The basic idea is a very simple one: where a nested graph predicate

[1] An exception to this is the dangling edge deletion by SPO rules; indeed, the fact that this is not existential in the current sense is the reason why such rules cannot be mimicked by single DPO rules.

is essentially a diagram of graphs and graph morphisms, a *nested rule* is a similar diagram of "simple" rules and morphisms. The application of such a rule to a given host graph consists of first matching the graph predicate consisting of the left hand sides of the rule diagram (which will result in a set of match morphisms, each of which goes from a left hand side of one of the simple rules to the host graph), and then using the structure of the rule diagram as interaction scheme in terms of rule amalgamation. The interaction scheme *synchronises* the atomic rule applications according to the match morphisms.

We described a preliminary version of this idea in [22]. This has now been improved and implemented [13], so that we can report on some implementation and performance details. Furthermore, we give some more examples which show the expressiveness of the approach.

The geraniums in the title and abstract refer to the following challenge. We have a number of flower pots, each of which contains a number of geranium plants. These tend to fill all available space with their roots, and so some of the pots have cracked. For each of the cracked pots that contains a geranium that is currently in flower, we want to create a new one, and moreover, to move all flowering plants from the old to the new pot. *Create a single parallel rule that achieves this in a single application, without the use of control expressions.* The complexity of this example stems from the fact that it involves a nested universal quantification, which (as far as we are aware) cannot be expressed in other declarative rule formalisms proposed in the literature, with the possible exception of [9, 14].

The remainder of this paper is structured as follows. In Section 2 we recall the relevant concepts of rule amalgamation; in Section 3 we give a new presentation of nested graph predicates, and we show how these can be used to generate amalgamated rules, which we call nested rules in this paper. In Section 4 we discuss implementation issues, and demonstrate the use of nested rules, including the geraniums as well as some examples encountered in practice. Finally, in Section 5 we discuss related work, draw conclusions and discuss future work.

2 Rule amalgamation

We will first (briefly) recall the concepts of amalgamated graph transformation in the Single Pushout approach from [5], generalising from two rules along the lines of [27], resulting in a setup very similar to [12].

Definition 1 (Graph) A *graph* G is a tuple $\langle V, E, src, tgt \rangle$ consisting of a set of nodes V, a set of edges E, and source and target mappings $src, tgt \colon E \to V$. G is called *labelled* if there is also a function $lab \colon E \to L$ to a global set of labels L, and *simple* if $E \subseteq V \times L \times E$ such that src, lab and tgt are projections to the three components.

The examples in this paper are set in the category of simple labelled graphs, but for the purpose of the definitions one can imagine any pair of graph categories $\mathscr{G}_{\mathsf{tot}}, \mathscr{G}$ such that $\mathscr{G}$ has an initial object 0 and coproducts, and $\mathscr{G}_{\mathsf{tot}}$ is a full subcategory of $\mathscr{G}$ with initial object and coproducts which are preserved by the inclusion functor. We refer to the arrows in $\mathscr{G}$ as partial morphisms and to those in $\mathscr{G}_{\mathsf{tot}}$ as total morphisms.

Recall that a diagram D over a category $\mathscr{C}$ is a mapping from the nodes and edges of a graph

G_D to the objects and arrows of $\mathscr{C}$, such that $D(e)\colon D(src(e)) \to D(tgt(e))$ for all edges e. Diagram D *commutes* if for all parallel paths in G_D, the composition (in $\mathscr{C}$) of the edge images give rise to the same $\mathscr{C}$-arrow. As usual, we will often identify the elements of G_D with their images under D. In this paper we frequently use *tree-shaped diagrams*, in which G_D is a tree rooted in the initial object, i.e., has no cycles, no sharing (no distinct edges with the same target) and exactly one root rt_D (a node without incoming edge) such that $D(rt_D) = 0$. Note that a tree-shaped diagram trivially commutes. A tree-shaped diagram D is said to be en *instance* of another tree-shaped diagram D' if there exists a root-preserving graph morphism $i\colon G_D \to G_{D'}$ (called the *instantiation morphism*) such that $D' = D \circ i$. (So an instance may copy or ignore parts of D.)

Definition 2 (Rule and Sub-rule) A rule is a morphism $p\colon L \to R$ in $\mathscr{G}$. A rule p' is called a *sub-rule* of p if there exists a pair of total morphisms $e_L\colon L' \to L$ and $e_R\colon R' \to R$ such that $p' \circ e_L = e_R \circ p$, i.e., the following diagram commutes:

$$\begin{array}{ccc} L' & \xrightarrow{p'} & R' \\ {\scriptstyle e_L}\downarrow & & \downarrow{\scriptstyle e_R} \\ L & \xrightarrow{p} & R \end{array}$$

The pair $e = e(e_L, e_R)$ is called a *sub-rule embedding*.

Rules give rise to derivations in the usual way of the single-pushout approach.

Definition 3 (Match and Derivation) A *match* of a rule p in a graph G is a total morphism $m\colon L \to G$. Given a rule and a match, a *derivation* is a pushout in $\mathscr{G}$, depicted by the diagram

$$\begin{array}{ccc} L & \xrightarrow{p} & R \\ {\scriptstyle m}\downarrow & \mathrm{PO} & \downarrow{\scriptstyle m'} \\ G & \xrightarrow[d]{} & H \end{array}$$

m' is called the *comatch* and d the *derivation morphism*. Note that m' is in general not total.

We write $G \xrightarrow{p,m} H$ if a derivation as in the above diagram exists. Sub-rules and sub-rule embeddings form a category $\mathscr{R}$ (with the natural definition of identities and arrow composition) with an initial object and coproducts. Below we will sometimes call the objects of $\mathscr{R}$ *simple rules*, to contrast them with the notion of composite rule that we are about to define.

Definition 4 (Composite Rule) A composite rule schema S is a tree-shaped diagram over $\mathscr{R}$.

For instance, Figure 1 shows a composite rule schema that can be used to model the firing of a Petri Net transition. The rule morphisms are left implicit. In general, a composite rule schema corresponds to a *synchronisation rule*, and a composite rule instance (i.e., a tree-shaped diagram P that is an instance of S, in the sense discussed above) to a *component production set* in terms of [27], except that in that paper both kinds of diagrams are required to be bipartite graphs, and the component production set satisfies a certain completeness property.

Every diagram P over $\mathscr{R}$ induces several diagrams over $\mathscr{G}$, among which we will use:

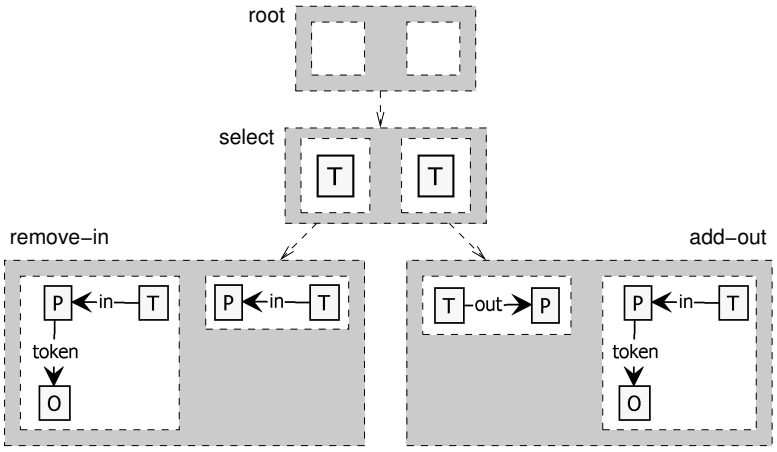


Figure 1: Composite rule schema for firing a Petri Net transition.

- The rule diagram D_P, consisting of the rule morphisms p for all objects p of P and the individual embedding morphisms e_L and e_R for all arrows e of D;
- The "left-hand-side" diagram L_P over $\mathscr{G}_{\mathsf{tot}}$ consisting of all the left hand sides L_p and the corresponding total morphisms e_L.

To define the derivations generated by a composite rule, first we extend the notion of a match.

Definition 5 (Composite Rule Match) Let S be a composite rule schema. A composite rule match of S in G consists of an instance P of S together with a set of matches $m_p\colon L_p \to G$ for all p in P, which, when added to L_P, make the resulting diagram commute.

A match of a composite rule schema S in a graph G is called a (partial) *covering* of G in [27]. Given such a match, as usual one can define the composite derivation (*star-parallel derivation* in [27]) either by taking the coproduct q of the rule diagram D_P and applying that as an ordinary rule (with respect to the unique match of q in G that is guaranteed by the coproduct construction), or by building the coproduct of the diagram consisting of the targets H_p of the individual derivations $G \xrightarrow{p,m_p} H_p$ together with the comatches m'_p. Due to the universal properties of coproducts, these two constructions are guaranteed to yield isomorphic results.

In order to get a useful notion of parallel transformation, the allowed rule schema matches have to be restricted. [27] identifies a number of possible criteria. The main contribution of this paper is to propose yet another criterion, which uses the theory of nested graph predicates introduced by us in [21] and later, independently, in [11].

3 Nested Graph Predicates

We give a new presentation of nested graph predicates, to make the connection with rule amalgamation clearer. A predicate will be a pair consisting of a tree-shaped graph diagram D over $\mathscr{G}_{\mathsf{tot}}$ and a formula generated by the following grammar, $\mathscr{L}$:

$$\phi \;::=\; \mathbf{tt} \mid \neg\phi \mid \phi \vee \phi \mid \exists x.\phi$$

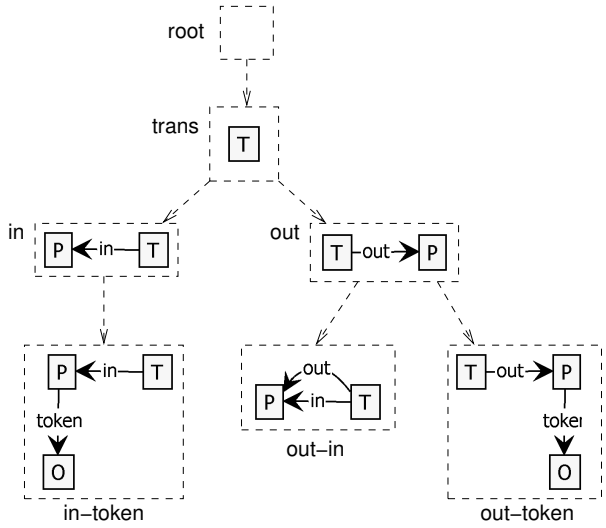


Figure 2: Graph diagram on which transition enabledness can be expressed

Here x denotes one of the non-root nodes of the graph G_D. Apart from the basic logic operators defined above, we also use $\wedge$, $\Rightarrow$, $\forall$ etc., defined in the standard way. Furthermore, we abbreviate $\exists x.\mathbf{tt}$ to x. Every such non-root node x has a unique incoming edge; we will denote this edge in_x.

For instance, some formulae over the diagram in Figure 2 are:

1. $\neg$out-in (which is an abbreviation of $\neg\exists$out-in.$\mathbf{tt}$), expressing that a given Petri Net does not have a loop;
2. $\forall$trans.$\forall$in.in-token, expressing that every transition of a Petri Net can fire;
3. $\exists$trans.$(\forall$in.in-token $\wedge\, \forall$out.$($out-token $\Rightarrow$ out-in$))$, expressing that there is an enabled transition according to the Condition/Event interpretation (in which all output places have to be empty, unless they are also input places).

Formulae are typed over the nodes of G_D. The type of a formula is a graph in D for which we need a matching into the subject graph before we can evaluate the formula; in other words, it represents the "free variables" of the formula. We write $\phi : t$ to denote that t is a type of ϕ. We only deal with formulae that are well-typed according to the following rules:

- $\mathbf{tt} : t$ for all nodes t of G_D;
- $\neg\phi : t$ if $\phi : t$;
- $\phi_1 \vee \phi_2 : t$ if $\phi_1 : t$ and $\phi_2 : t$;
- $\exists x.\phi : t$ if $t = src(in_x)$ and $\phi : x$.

ϕ is called *ground* if $\phi : 0$ (where 0 is the initial object of $\mathscr{G}$). For instance, we have $\neg\exists$out-in.$\mathbf{tt}$: out, whereas the other two example formulae above are ground.

In principle, formulae are evaluated over a given graph G; however, to define this properly we actually have to evaluate them over a given morphism $f : L \to G$, where L is one of the graphs in the diagram D. f in fact represents a matching of L in G that we have built up "so far" while establishing the validity of a larger formula ψ of which ϕ is a sub-formula. The meaning of $\exists x.\phi$

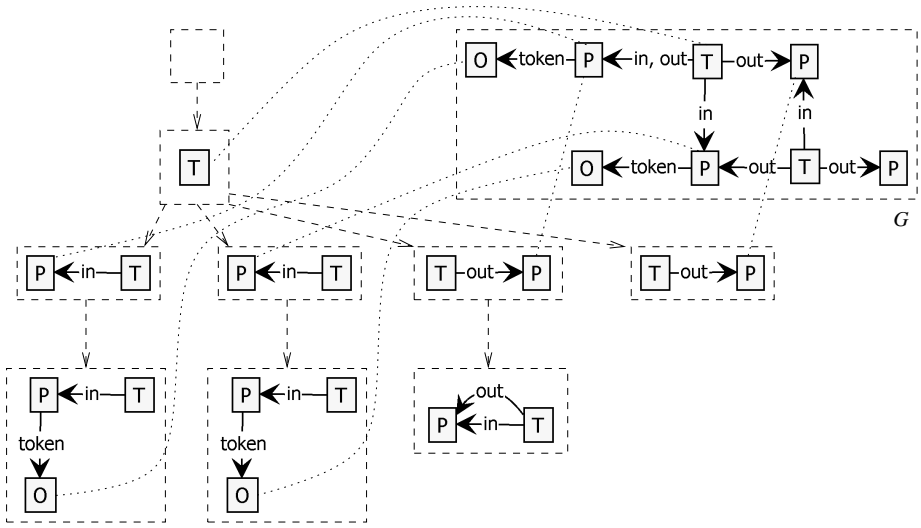


Figure 3: Proof of $\exists$trans.($\forall$in.in-token $\wedge$ $\forall$out.(out-token $\Rightarrow$ out-in)). *The dotted lines indicate some of the relevant node mappings.*

is that the matching f can be decomposed into $g \circ D(in_x)$ (where in_x is the unique edge in D with $tgt(in_x) = x$).

Formally, the semantics of the logic is expressed by a relation $f \models \phi$ where $\phi : t$ and $f : D(t) \to G$ is a total morphism in $\mathscr{G}$:

- $f \models \mathbf{tt}$ always holds;
- $f \models \neg\phi$ if $f \not\models \phi$;
- $f \models \phi_q \vee \phi_2$ if $f \models \phi_1$ or $f \models \phi_2$;
- $f \models \exists x : \phi$ if $g \models \phi$ for some g such that $f = g \circ D(in_x)$.

If ϕ is ground, we also write $tgt(f) \models \phi$ instead of $f \models \phi$. For instance, if G is the Petri Net depicted on the right of Figure 3, then the figure shows that there is an enabled Condition/Event transition, as expressed by the example formula 3 above.

A formula ϕ is in positive form if it does not contain negations (but may contain **ff**, $\wedge$ and $\forall$). Every formula is equivalent to a positive form formula, which can be obtained easily by "pushing" negations inward. For instance, formula 3 above is equivalent to $\exists$trans.($\forall$in.$\exists$in-token.**tt** $\wedge$ $\forall$out.($\exists$out-in.**tt** $\vee$ $\forall$out-token.**ff**)).

If ϕ is a ground positive form formula, then a *proof diagram* of $G \models \phi$ is defined to be a commuting diagram P over $\mathscr{G}_{\mathsf{tot}}$, consisting of an instance Q of D with instantiation morphism $i : G_Q \to G_D$, augmented with a graph G and for all nodes v of Q a morphism $f_v : Q(v) \to G$. Furthermore, for every node v of Q there is a set Ψ_v of sub-formulae of ϕ such that $\phi \in \Psi_{rt_Q}$, and for all $\psi \in \Psi_v$, $\psi : i(v)$ and the following conditions are satisfied:

- $\psi \neq \mathbf{ff}$
- If $\psi = \psi_1 \vee \psi_2$, then either $\psi_1 \in \Psi_v$ or $\psi_2 \in \Psi_v$;
- If $\psi = \psi_1 \wedge \psi_2$, then $\psi_1 \in \Psi_v$ and $\psi_2 \in \Psi_v$;

- If $\psi = \exists x.\psi'$, then v has an outgoing edge e with $i(e) = in_x$ and $\psi' \in \Psi_{tgt(e)}$.
- If $\psi = \forall x.\psi'$, then for all $g\colon D(x) \to G$ such that $f_v = g \circ D(in_x)$, v has an outgoing edge e with $i(e) = in_x$, $f_{tgt(e)} = g$ and $\psi' \in \Psi_{tgt(e)}$.

A proof diagram is called *minimal* if it does not have spurious edges; i.e., the only edges are those necessitated by the last two bullets above. For instance, Figure 3 is a minimal proof diagram, if v and w are the two occurrences of out in the diagram then $\Psi_v = \{\exists\text{out-in}.\mathbf{tt}\}$ and $\Psi_w = \{\forall\text{out-token}.\mathbf{ff}\}$.

Predicate-driven amalgamation. The step from nested graph predicates to amalgamated rules is very small: rather than interpreting formulae over diagrams over $\mathscr{G}_{\mathsf{tot}}$, we use tree-shaped diagrams over $\mathscr{R}$, i.e., composite rule schemas. The interpretation of ϕ over S is defined to be its interpretation over the left-hand-side diagram L_S. The following is a key insight:

Proposition 1 *Given a composite rule schema S, a closed formula ϕ interpreted over S, and a graph G, a minimal proof diagram of $G \models \phi$ is a composite match of S in G.*

For instance, we can turn the diagram in Figure 2 into a diagram over $\mathscr{R}$ by replacing the graph in-token by the rule remove-in of Figure 1, replacing out by add-out, and turning all other graphs into identity rules (i.e., based on identity production morphisms). The resulting diagram "refines" Figure 1. The formula $\exists\mathsf{trans}.(\forall\mathsf{in}.\text{in-token} \wedge \forall\mathsf{out}.(\text{out-token} \Rightarrow \text{out-in}))$, which previously just expressed the existence of an enabled transition in a Condition/Event net, now encodes the firing of such a transition under the condition that it is enabled.

The developments in this section culminate in the following definition, which we will use in the remainder of the paper:

Definition 6 (Nested Rule) A nested graph transformation rule is a tree-shaped diagram S over $\mathscr{R}$ with a formula $\phi \in \mathscr{L}$ over S. A match of such a rule is a minimal proof diagram of ϕ over L_S, and a rule derivation is the composite derivation with respect to such a minimal proof diagram.

4 Implementation and examples

The theory of nested rules has been implemented in GROOVE [20], with some restrictions. Nested rules in GROOVE have been used and shown their value in several applications. In this section we discuss some of the implementation choices and show some applications.

4.1 GROOVE implementation

The main functionality of GROOVE is to explore the complete state space of a graph transformation system. Every derivation gives rise to a transition, and independent derivations interleave, giving rise to a size blow-up that is at worst exponential in the number of independent derivations.

A composite rule derivation can combine a large number of simple rule derivations. Apart from the ease of specification, this has the advantage that the number of transitions as well as the number of interleaving points between transitions decreases, in some cases quite dramatically.

For the purposes of practical use, we have made the following choices.

Modified positive form formulae. Rather than the full logic defined above, GROOVE only supports restricted positive form formulae, as defined by the following syntax:

$$\begin{aligned} \phi &::= \exists x.(\textstyle\bigwedge_{k\in K} \neg x_k) \wedge (\bigwedge_{i\in I} \psi_i) \\ \psi &::= \forall x.(\textstyle\bigwedge_{k\in K} \neg x_k) \wedge (\bigvee_{j\in J} \phi_j) \end{aligned}$$

where $\neg x_k$ abbreviates $\forall x_k.\mathbf{ff}$, and I, J, K are arbitrary index sets. Thus, disjunction is restricted to existentially quantified sub-formulae and conjunction to universally quantified sub-formulae. It can be proved (in fact, it indirectly follows from [21]) that this is no real restriction, in the sense that every formula is equivalent to a "normal form" formula in this restricted syntax, but we will not elaborate on this point here.

Single-graph representation. One of the disadvantages of nested rules as formulated in Definition 6 is that they consist of two parts, a rule diagram and a formula. In GROOVE, we have chosen to include all of these into a single graph representation. For this purpose, we introduce special *quantifier nodes* that stand for the $\forall$- and $\exists$-quantifiers of the formula and are arranged (using special in-labelled edges) in a tree of alternating quantifiers. The root of this tree is an $\exists$-node which is left implicit, so that a simple rule is just a special case of a composite rule.

The "fresh" nodes of the quantified graphs, i.e., those nodes that are not in the codomain of the incoming morphisms, are attached to the corresponding quantifier nodes using special `at`-labelled edges. For fresh edges of the quantified graph, this solution does not work since GROOVE does not support edges on edges; instead, if such a fresh edge does not have fresh end nodes, we include the name of the quantifier as a prefix of the edge label.

As an example, Figure 4 shows the firing rule of Condition/Event nets in this one-graph representation. As usual in the GROOVE notation, non-RHS elements (which are to be deleted) are dashed thin blue (or dark grey), non-LHS elements (which are to be created) are wider solid green (or light grey), and NAC elements are wide, closely dashed red (or dark grey). The dotted nodes and edges form the tree of quantifiers (where the root is omitted); to make the connection with the diagram in Figure 2 explicit, we have named all quantifier nodes. For the existential `out-in-quantifier` this name is in fact necessary as it occurs as a prefix in one of the `in`-edges, to associate this edge with the quantifier.

Non-vacuous universal quantification. If no match of x exists in a given host graph, the formula $\psi = \forall x.\phi$ is true irregardless of ϕ. In this case, ψ is said to be *vacuously true*. Consequently, a universally quantified nested (sub-)rule may be vacuously applicable, in which case the rule has no effect. Sometimes this may be just what one wants, as in the firing rule of Figure 4: for a transition with no input places, the sub-rule `in` is always enabled and has no effect. However, quite often vacuous derivations are not intended. Though non-vacuity can always be enforced through an application condition, we have included a special quantifier node, denoted $\forall^{>0}$, which guarantees that the sub-rule is matched at least once. Thus, $\forall^{>0}x.\phi$ is equivalent to $\exists x.\phi \wedge \forall x.\phi$.[2]

[2] Thus, the difference between $\forall$ and $\forall^{>0}$ is very similar to that between optional and obligatory set nodes in PROGRES.

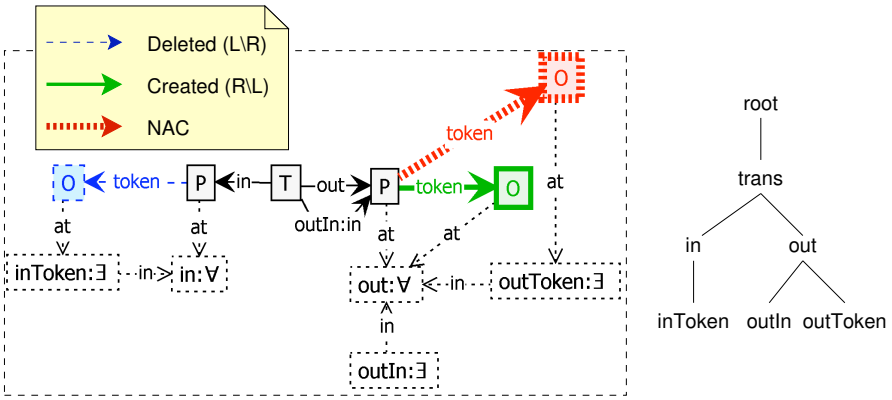


Figure 4: Nested C/E firing rule in GROOVE *syntax, with the explicit tree structure shown to the right*

4.2 Examples

We now show some other applications of nested rules.

Geraniums. The title challenge of this paper is to create a new pot for every cracked flower pot with at least one flowering geranium, and to transfer all flowering geraniums in the cracked pot to the new one. This is an example of a rule that needs two nested universal quantifiers: an outer quantifier for the pots, and an inner quantifier for the plants in the pots. This puts the rule beyond what can be formulated in other approaches to parallel graph transformation rules, such as the cloning rules of [18] or the set nodes and star rules in PROGRES [24], except in the extension recently proposed in [9] — see Section 5 for a more extensive discussion.

In GROOVE, a first attempt is given on the left side of Figure 5. An example derivation is shown in Figure 6. However, this rule is incorrect as it also creates new pots for cracked pots that do not contain any flowering geraniums. To rule this out, we need the non-vacuous universal quantifier discussed above. However, we cannot simply replace the plants-quantifier by $\forall^{>0}$, since then the rule requires that *all* cracked pots have at least one flowering geranium, hence it would become inapplicable for a graph like the one in Figure 6. To resolve this, we have to add

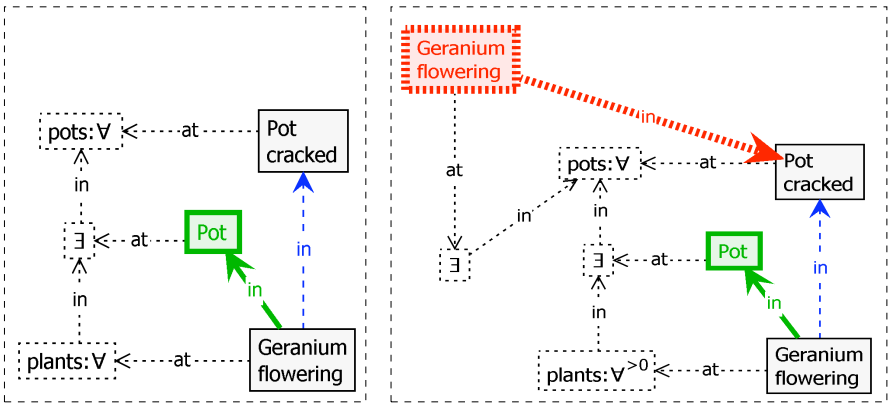


Figure 5: Incorrect and corrected versions of the geranium rule.

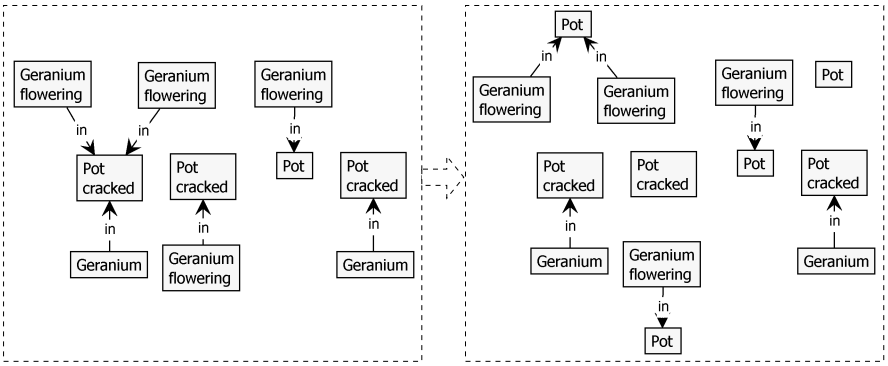


Figure 6: Example derivation of the left hand rule of Figure 5

a disjunct to the pots-quantifier, resulting in the rule on the right hand side of Figure 5.

Sierpinski Triangles Another example that is very suited to specification in nested rules is the Sierpinski case described in [29]. This involves a challenge to give a graph grammar that generates all Sierpinski triangles (a certain fractal shape) up to an arbitrary depth. One step of the generation process involves replacing all up-pointing sub-triangles by a more involved graph (which contains three new up-pointing triangles). A nested GROOVE rule that specifies this given in Figure 7.

In [29], we have described a sequential GROOVE solution to the Sierpinski case, and we have remarked that the above parallel rule has (only) slightly better performance. This may be surprising in the light of the fact that the sequential solution generates many more intermediate states. However, in this particular case no real state space exploration is needed: instead, a "linear" exploration strategy is used that selects a single rule application and never backtracks. In this type of exploration, generating the intermediate states causes only little overhead.

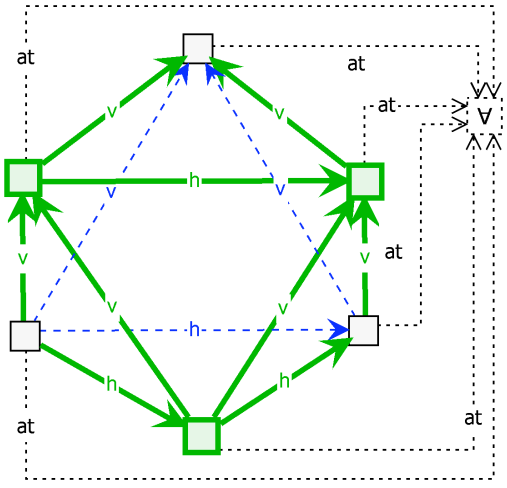


Figure 7: Nested rule specifying one Sierpinski triangle generation step.

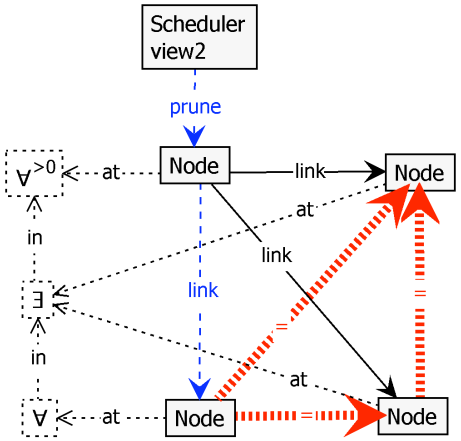


Figure 8: One of the rules of the ad-hoc network connectivity protocol in [3]. The =-labelled NAC-edges are injectivity constraints.

Network gossipping protocol. In [3] we describe a GROOVE model of an ad-hoc network connectivity protocol. The paper shows that this model gives rise to a large symmetry reduction, so that larger network instances can be modelled than with other specification methods (although the size of the state space is still exponential in the size of the network). Nested rules have been used here in several places, to reduce the number of derivation steps and especially the number of interleavings of steps. In contrast to the previous example, in this case it is important to explore the state space in full, and indeed without the use of nested rules the advantage with respect to other methods to some degree disappears. An example rule where nesting has been exploited is shown in Figure 8: this specifies that all but two outgoing link-edges have to be removed from every network node.

5 Evaluation

We have shown how to integrate the concepts of nested graph predicates and rule amalgamation. It turns out that these concepts mesh together quite well, and give rise to a usable specification formalism for parallel rules. This formalism has been implemented in GROOVE; we have given several practical examples where this type of rule has been very useful.

On the downside, it turns out that nested rules can be complicated to write. This is mainly due to the chosen single-graph representation: especially when the rules become larger, the fact that all nesting levels are combined in a single graph makes the resulting figure hard to read. An alternative is to use a hierarchical graph syntax, where the quantifier nodes are containers for the graph elements associated with them.

Related work. We briefly review alternative approaches to parallel rule specification.

First of all, node replacement systems [7] have a natural notion of parallelism due to the fact that, when a node is replaced, all incident edges, no matter how many, are modified as well. For the *star grammars* [4] this is generalised so that not only incident edges but their opposite nodes can be duplicated as often as necessary. This roughly corresponds a single universal quantifica-

tion in terms of our nested rules; in fact, every adaptive star rule can easily be formulated as a nested rule with a single universal quantifier.

PROGRES [24] and also FuJaBA [19] feature so-called *set nodes*, which are essentially single universally quantified nodes. Furthermore, PROGRES has *star rules*, which are essentially rules that are entirely universally quantified. An interesting extension to set nodes can be found in [9], which allows to specify *set regions* rather than just set nodes. Since these set regions can be nested, this comes close to our notion of nested quantifiers, and we conjecture that this formalism can in fact specify the geranium rule. Unfortunately, the paper does not provide enough information to be sure.

Some approaches are based on rule schemas with sub-graphs that can be cloned or copied before applying the rule: for instance, [18, 1, 17]. The latter two have the interesting option of specifying connections *between* the clones, which may for instance be ordered in a linear list. This is outside the capabilities of our nested rules. On the other hand, each of these approaches deals with a single level of (universal) quantification only, and so we believe that they cannot solve the geranium challenge.

As we have made clear, our nested rules are built on the principle of rule amalgamation. Other papers that have shown the power of amalgamation for specifying parallel rules (in particular, the Petri net firing rule) are [15, 8].

Another approach that needs mentioning in this context is that of *synchonised hyperedge replacement*; see, e.g., [14]. A central concept in this formalism is to combine "local" rules into larger ones, using *synchonisation algebras* to determine how rules are to be combined. It is claimed in [30] that this is powerful enough to repot the geraniums.

Finally, another method altogether for repotting the geraniums is by using control expressions rather than a single parallel rule or rule schema. There have been many proposals for powerful control languages; we would like to mention PROGRES, FuJaBA's storyboards, but also the recent notions of *recursive rules* [10, 32], which in fact have no extraneous control conditions but rather integrate them with the rules themselves. We would also categorise the use of the (very powerful) pattern definitions in model transformation tools such as TEFKAT [16] and VIATRA2 [31] as control expressions, though admitedly the dividing line grows thin in these cases.

Future work. We see nested rules as a first step towards the ability to specify an arbitrary transformation as a single transaction with atomic execution. The planned next step is to enhance the GROOVE control language with an atomicity statement that turns an arbitrary control statement into such a transaction.

Another potentially useful extension is the introduction of *counting quantifiers*, being existential quantifiers that assert the existence of a given, fixed number of distinct instances of a sub-graph (other than 1, which is the default meaning of existential quantification). For instance, the rule in Fig. 8 could be simplified using such a feature.

Bibliography

[1] D. Balasubramanian, A. Narayanan, S. Neema, F. Shi, R. Thibodeaux, and G. Karsai. A subgraph operator for graph transformation languages. In [6].

[2] A. Corradini, H. Ehrig, U. Montanari, L. Ribeiro, and G. Rozenberg, eds. *Third International Conference on Graph Transformations (ICGT)*, vol. 4178 of *LNCS*. Springer, 2006.

[3] P. Crouzen, J. van de Pol, and A. Rensink. Applying formal methods to gossiping networks with mCRL and Groove. *SIGMETRICS Perform. Eval. Rev.*, 36(3):7–16, 2008.

[4] F. Drewes, B. Hoffmann, D. Janssens, M. Minas, and N. V. Eetvelde. Adaptive star grammars. In [2], pp. 77–91.

[5] H. Ehrig and M. Löwe. Parallel and distributed derivations in the single-pushout approach. *TCS*, 109(1&2):123–143, 1993.

[6] K. Ehrig and H. Giese, eds. *Graph Transformation and Visual Modeling Techniques (GT-VMT)*, vol. 6 of *Electronic Communications of the EASST*. EASST, 2007.

[7] J. Engelfriet and G. Rozenberg. Node replacement graph grammars. In [23], pp. 1–94.

[8] C. Ermel, G. Taentzer, and R. Bardohl. Simulating algebraic high-level nets by parallel attributed graph transformation. In Kreowski, Montanari, Orejas, Rozenberg, and Taentzer, eds., *Formal Methods in Software and Systems Modeling*, vol. 3393 of *LNCS*, pp. 64–83. Springer, 2005.

[9] C. Fuss and V. E. Tuttlies. Simulating set-valued transformations with algorithmic graph transformation languages. In [25], pp. 442–455.

[10] E. Guerra and J. de Lara. Adding recursion to graph transformation. In [6].

[11] A. Habel and K.-H. Pennemann. Nested constraints and application conditions for high-level structures. In Kreowski, Montanari, Orejas, Rozenberg, and Taentzer, eds., *Formal Methods in Software and Systems Modeling*, vol. 3393 of *LNCS*, pp. 293–308. Springer, 2005.

[12] R. Heckel, J. Müller, G. Taentzer, and A. Wagner. Attributed graph transformations with controlled application of rules. In Valiente and Rossello Llompart, eds., *Colloquium on Graph Transformation and its Application in Computer Science*, Technical Report B–19. Universitat de les Illes Balears, 1995.

[13] J.-H. Kuperus. Nested quantification in graph transformation rules. Master's thesis, Department of Computer Science, University of Twente, 2006.

[14] I. Lanese and E. Tuosto. Synchronized hyperedge replacement for heterogeneous systems. In Jacquet and Picco, eds., *Coordination Models and Languages (COORDINATION)*, vol. 3454 of *LNCS*, pp. 220–235. Springer, 2005.

[15] J. de Lara, C. Ermel, G. Taentzer, and K. Ehrig. Parallel graph transformation for model simulation applied to timed transition Petri Nets. In Heckel, ed., *Graph Transformations and Visual Modelling Techniques (GT-VMT)*, vol. 109 of *ENTCS*, pp. 17–29, 2004.

[16] M. Lawley and J. Steel. Practical declarative model transformation with tefkat. In Bruel, ed., *MoDELS Satellite Events*, vol. 3844 of *LNCS*, pp. 139–150. Springer, 2006.

[17] J. Lindqvist, T. Lundkvist, and I. Porres. A query language with the star operator. In [6].

[18] M. Minas and B. Hoffmann. An example of cloning graph transformation rules for programming. In Bruni and Varró, eds., *Graph Transformation and Visual Modeling Techniques (GT-VMT)*, vol. 211 of *ENTCS*, pp. 241–250, 2006.

[19] J. Niere and A. Zündorf. Using FUJABA for the development of production control systems. In Nagl, Schürr, and Münch, eds., *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*, vol. 1779 of *LNCS*, pp. 181–191. Springer, 2000.

[20] A. Rensink. The groove simulator: A tool for state space generation. In Pfaltz, Nagl, and Böhlen, eds., *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*, vol. 3062 of *LNCS*, pp. 479–485. Springer, 2004.

[21] A. Rensink. Representing first-order logic using graphs. In Ehrig, Engels, Parisi-Presicce, and Rozenberg, eds., *Second International Conference on Graph Transformations (ICGT)*, vol. 3256 of *LNCS*, pp. 319–335. Springer, 2004.

[22] A. Rensink. Nested quantification in graph transformation rules. In [2], pp. 1–13.

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

[24] A. Schürr. Programmed graph replacement systems. In [23], pp. 479–546.

[25] A. Schürr, M. Nagl, and A. Zündorf, eds. *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, vol. 5088 of *LNCS*. Springer, 2008.

[26] G. Taentzer. *Parallel and Distributed Graph Transformation: Formal Description and Application to Communication-Based Systems*. PhD thesis, TU Berlin, 1996.

[27] G. Taentzer. Parallel high-level replacement systems. *TCS*, 186(1-2):43–81, 1997.

[28] G. Taentzer and M. Beyer. Amalgamated graph transformations and their use for specifying AGG — an algebraic graph grammar system. In Schneider and Ehrig, eds., *Graph Transformations in Computer Science*, vol. 776 of *LNCS*, pp. 380–394. Springer, 1994.

[29] 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, and T. Vajk. Generation of Sierpinski triangles: A case study for graph transformation tools. In [25], pp. 514–539.

[30] E. Tuosto. Private communication, 2009.

[31] D. Varró and A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.*, 68(3):214–234, 2007.

[32] G. Varró, Á. Horváth, and D. Varró. Recursive graph pattern matching with magic sets and global search plans. In [25], pp. 456–470.

Parallelization of Graph Transformation Based on Incremental Pattern Matching

Gábor Bergmann[1], **István Ráth**[2], **Dániel Varró**[3]

[1 2 3] (bergmann,rath,varro)@mit.bme.hu, http://www.mit.bme.hu/eng/
Méréstechnika és Információs Rendszerek Tanszék (MIT),
Budapesti Műszaki és Gazdaságtudományi Egyetem (BME), Budapest, Hungary

Abstract: Graph transformation based on incremental pattern matching explicitly stores all occurrences of patterns (left-hand side of rules) and updates this result cache upon model changes. This allows instantaneous pattern queries at the expense of costlier model manipulation and higher memory consumption.

Up to now, this incremental approach has considered only sequential execution despite the inherently distributed structure of the underlying match caching mechanism. The paper explores various possibilities of parallelizing graph transformation to harness the power of modern multi-core, multi-processor computing environments: (i) incremental pattern matching enables the concurrent execution of model manipulation and pattern matching; moreover, (ii) pattern matching itself can be parallelized along caches.

Keywords: graph transformation, incremental pattern matching, parallelization

1 Introduction

Nowadays, a main challenge of software engineering is the adaptation to parallel computing architectures. In order to increase execution speed, algorithm designers need to think of new ways to exploit the computing power of multi core processors instead of purely relying on more efficient processor designs. Experience has shown that this is a complicated task, and no general solution exists; whether parallel execution can actually be effectively applied depends largely on the problem itself.

Model transformation is an application domain where speed optimization based on parallel execution has a lot of potential, especially in case of large, industrial models. In fact, model transformations seem to be an ideal target for parallel execution as in practical transformations, many similar, or almost identical model structures need to be traversed and transformed. Frequently, these model manipulation sequences are non-conflicting, which naturally calls for an execution model where these sequences are executed on the available processors in parallel.

Using a graph transformation (GT) [EEKR99] based approach for model transformations, there are even more possibilities for the exploitation of parallelism. Besides model manipulation sequences, graph transformations involve a graph searching phase, which is targeted at finding the matches of a graph pattern. However, despite the recent optimization activities in the graph transformation community, which have been reported at tool contests [SNZ08, Gra08], GT tools rarely exploit parallel execution for further improvement both in terms of execution speed and

scalability with model sizes.

Incremental pattern matching [BÖR+08] offers an entirely different execution model compared to local search-based implementations. The match sets for all patterns involved in the graph transformation are computed in an initialization phase prior to execution (e.g. when the model itself is loaded into memory), and as the transformation progresses, this match set cache is incrementally updated as the model graph changes (update phases). Thus, model search phases are reduced to fast read-from-cache operations, in exchange for the overhead imposed by cache update phases which occur synchronously with model manipulation operations. Benchmarking [BHRV08] has shown that in certain scenarios, this approach leads to several orders-of-magnitude increases in speed.

In the current paper, we introduce novel extensions to the incremental pattern matcher of the VIATRA2 framework, which is based on RETE networks [For82], to exploit parallelism based on asynchronous model updates and multi-threaded match set caching.

First, *update phases may be executed concurrently* to the model transformation's main execution thread. In this case, the cache validation thread of the match set may execute concurrently with model manipulation sequences or textual output emission, e.g. in the case of code generation transformations. This approach aims to reduce the overhead imposed by update phases, in the case when parallel computing power is available.

Then, if further scaling up is required, the implementation of the *match set cache updating can be multi-threaded*. It is important to point out that both of these approaches are significantly different from parallelized pattern search. Finally, as incremental pattern matching provides fast cache-reading operations, it supports *parallel transformation execution* by allowing simultaneous access to caches from multiple threads. By improving this scenario with concurrent update phases, model manipulation protected by locks will no longer force other transformation threads to wait for the termination of the time-consuming update. As a consequence, read-intensive transformations are expected to scale well with parallel computational capacity.

The rest of the paper is structured as follows. Section 2 gives a brief introduction on graph transformations. Section 3 describes RETE, and its implementation in the VIATRA2 model transformation framework. The main contributions of the paper are presented in Section 4, where we present ways of parallelizing both pattern matching and model manipulation. Finally, we discuss related work in Section 5 and conclude the paper in Section 6.

2 Foundations of model transformation

This section gives an overview on the foundations of the specification and simulation of modeling languages. In order to specify the abstract syntax of the modeling language, the concept of metamodeling is used. For transforming models to other models or generated code, and simulating the behaviour of models, the paradigm of graph transformation [Roz97] is applied.

2.1 Model transformation example: Petri nets

In this paper, we will use the transformation of Petri nets as a demonstration for parallelization concepts. These demonstrating Petri net transformations include Petri net firing as a model

simulation example.

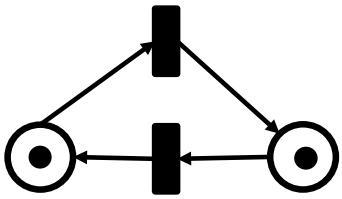


Figure 1: A sample Petri net.

Petri nets (Figure 1) are widely used to formally capture the dynamic semantics of concurrent systems due to their easy-to-understand visual notation and the wide range of available analysis tools. Petri nets are bipartite graphs, with two disjoint sets of nodes: *Places* and *Transitions*. Places may contain an arbitrary number of Tokens. A token distribution (marking) defines the state of the modelled system. The state of the net can be changed by firing enabled transitions. A transition is enabled if each of its input places contains at least one token and no place connected with an inhibitor arc contains a token (if no arc weights are considered). When firing a transition, we remove a token from all input places (connected to the transition by Input Arcs) and add a token to all output places (as defined by Output Arcs).

2.2 Foundations of metamodeling

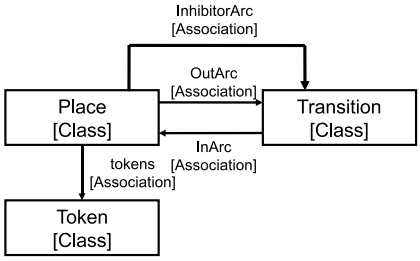


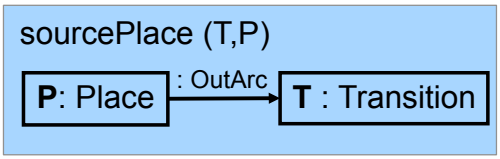
Figure 2: Petri net metamodel.

A *metamodel* describes the abstract syntax of a modeling language. Formally, it can be represented by a type graph. Nodes of the type graph are called *classes*. A class may have attributes that define some kind of properties of the specific class. *Inheritance* may be defined between classes, which means that the inherited class has all the properties its parent has, but it may further contain some extra *attributes*. *Associations* define connections between classes. Figure 2 shows a simple Petri net metamodel.

2.3 Graph patterns and graph transformation

Graph patterns are frequently considered as the atomic units of model transformations [VB07]. They represent conditions (or constraints) that have to be fulfilled by a part of the instance model in order to execute some manipulation steps on the model. A basic graph pattern consists of graph elements corresponding to the metamodel. A *negative application condition* (NAC) prescribes contextual conditions for the original pattern which are forbidden in order to find a successful match. Figure 3 presents a simple graph pattern consisting of a Place P, a Transition T and an OutArc A to enumerate the source places connected to a given transition.

Graph transformation (GT) [EEKR99] provides a high-level rule and pattern-based manipulation language for graph models. Graph transformation rules can be specified by using a left-hand side – LHS (or precondition) graph (pattern) determining the applicability of the rule, and a right-hand side – RHS (postcondition) graph (pattern) which declaratively specifies the result model after rule application. Elements that are present only in (the image of) the LHS are




```
pattern sourcePlace(T, P) = {
  transition(T);
  place(P);
  outArc(A, P, T);
}
```

Figure 3: Matcher for the *sourcePlace* pattern

deleted, elements that are present only in the RHS are created, and other model elements remain unchanged (in accordance with the single-pushout approach in VIATRA2). For instance, a GT rule may specify how to remove (or add) a token from a place, as shown in Figure 4.

```
// Removes a token from the place 'Place'.
gtrule removeToken(in Transition, in Place) = {
 precondition find sourcePlaceWithToken
    (Transition, Place, Token);
 postcondition find sourcePlaceWithoutToken
    (Transition, Place, Token);
}
// Adds a token from the place 'Place'.
gtrule addToken(in Transition, in Place) = {
 precondition find targetPlaceWithoutToken
    (Transition, Place);
 postcondition find targetPlaceWithToken
    (Transition, Place, Token);
}
```

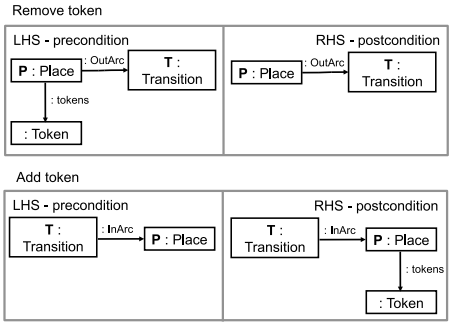


Figure 4: Graph transformation rules for firing a transition

Complex model transformation can be assembled from elementary graph patterns and graph transformation rules using some kind of control language. In our examples, we use abstract state machine (ASM) [BS03] for this purpose as available in the VIATRA2 framework. The following transformation simulates the firing of a transition, i.e. the removal of tokens from input places and the addition of tokens to output places (see Figure 5).

```
rule fireTransition(in T) = seq {
 if (find isTransitionFireable(T)) // confirm that the transition is fireable
 seq {
  forall Place with find sourcePlace(T, Place) // remove tokens from all source places
   do apply removeToken(T, Place); // GT rule invocation
  forall Place with find targetPlace(T, Place) // add tokens to all target places
   do apply addToken(T, Place);
 }
}
```

Figure 5: Transformation program for firing a transition

3 RETE-based incremental graph pattern matching

The incremental graph pattern matcher of the VIATRA2 framework adapts [BÖR+08] the RETE algorithm, which is a well-known technique in the field of rule-based systems.

RETE network for graph pattern matching RETE-based pattern matching relies on a network of nodes storing *partial matches* of a graph pattern. A partial match enumerates those model elements which satisfy a subset of the constraints described by the graph pattern. In a relational database analogy, each node stores a *view*. Matches of a pattern are readily available at any time, and they will be incrementally updated whenever model changes occur.

Input nodes serve as the underlying knowledge base representing a model. There is a separate input node for each entity type (class), containing a view representing all the instances that conform to the type. Similarly, there is an input node for each relation type, containing a view consisting of tuples with source and target in addition to the identifier of the edge instance.

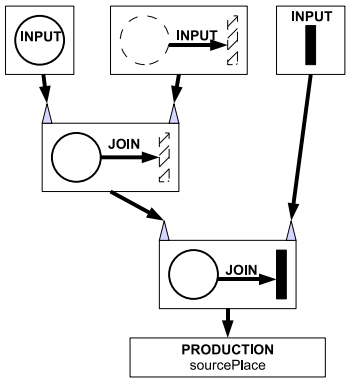


Figure 6: Simple RETE matcher

At each *intermediate node*, *set operations* (e.g. filtering, projection, join, etc.) can be executed on the match sets stored at input nodes to compute the match set which is stored at the intermediate node. The match set for the entire pattern can be retrieved from the output *production node*. One kind of intermediate node is the *join node*, which performs a natural join on its parent nodes in terms of relational algebra; whereas a *negative node* contains the set of tuples stored at the primary input which do *not* match any tuple from the secondary input (which corresponds to anti-joins in relational databases).

As an illustration, Figure 6 shows a RETE network matcher built for the *sourcePlace* (see Figure 3) pattern illustrating the use of join nodes. By joining three input nodes (the top-most nodes on Figure 6), this sample RETE net enforces two entity type constraints ('Place' and 'Transition' entity types on the left and right input nodes) and an edge (connectivity) constraint (corresponding to the relation connecting the 'Place' and 'Transition' entity types), to find pairs of Places and Transitions connected by an out-arc.

Updates after model changes. Input nodes receive notifications about each elementary model change (i.e. when a new model element is created or deleted) and release an update token on each of their outgoing edges. Such an update token represents changes in the partial matches stored by the RETE node. Positive update tokens reflect newly added tuples, and negative updates refer to tuples being removed from the set. Upon receiving an update token, a RETE node determines how the set of stored tuples will change, and release update tokens of its own to signal these changes to its child nodes. This way, the effects of an update will propagate through the network, eventually influencing the result sets stored in production nodes.

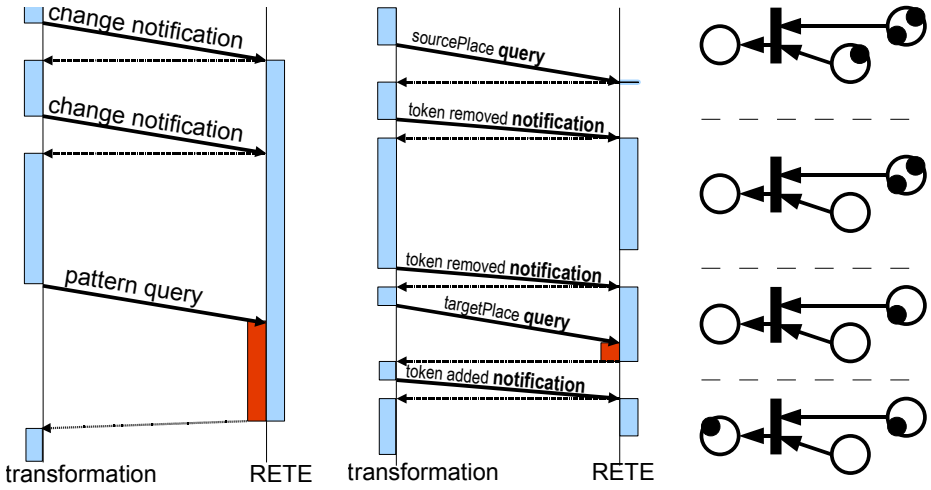
The match set can be retrieved from the network instantly without re-computation, which makes pattern matching very efficient. As a trade-off, there is increased memory consumption, and update operations become more complex.

4 Parallel transformation with incremental pattern matching

This section presents our conceptual contributions to the parallel execution of model transformations. First, Subsection 4.1 will discuss in detail how the asynchronous RETE approach allows the update phases to be executed in the background, while the transformation continues uninterrupted. In Subsection 4.2, we generalise this approach to multiple RETE threads for systems with more than two CPU cores, based on the multi-threaded RETE maching set cache. The proposed pattern matcher is applied to a multi-threaded model manipulation context in Subsection 4.3 to let the model manipulation phase take advantage of the number of CPU cores. Finally, the resulting system is evaluated in Subsection 4.4.

4.1 Concurrent pattern matching and model manipulation

Contrary to our previous work, the RETE net implementation used throughout this paper relies on *asynchronous* message passing. This involves a *message queue* attached to the network, containing update messages manifested as objects. Each message object specifies a recipient node, the tuple representing the update, and the sign (insertion or deletion). The message consumption cycle fetches the first message from the queue and delivers it to the appropriate node; the node will place any propagated output messages to the end of the queue, thereby achieving asynchronous messaging. Change notifications issued by model manipulation are simply put into the queue; then the update propagation phase consists of looping the message consumption cycle until the queue becomes empty.



(a) General concept (b) *fireTransition* concurrently (c) Petri net states

Figure 7: Concurrent pattern matching

Using asynchronous messaging, the load on the main thread of the transformation can be reduced by executing the incremental pattern matcher (which consumes change messages from the queue) in a separate thread. When the transformation manipulates the model (see Figure 7(a)),

it only has to send the new update message to the message queue, and continue its operation. The thread of the pattern matcher will execute the update propagation in the background, ideally, without imposing a performance penalty on the transformation thread. When the message queue becomes empty, the RETE network has reached a *fixpoint*; the pattern matcher thread then goes to sleep and will not resume its operation until a new update message is posted.

When the transformation initiates pattern matching, it has to assure that background update propagations have terminated and the matches stored at the production nodes are up-to-date; so if necessary, it will have to sleep until RETE reaches its fixpoint.

Figure 7(b) shows how the Petri net firing rule *fireTransition* (defined in Figure 5) may behave in such a concurrent system. (i) First, the set of source places is fetched instantaneously from the pattern matcher. (ii) Then, one token is deleted in every source place, each of them issueing a notification to the pattern matcher thread that results in some update propagation in the RETE net. (iii) Next, the list of target places is retrieved after update propagation is finished. (iv) Finally, a new token is created at each target place, resulting in subsequent notifications. Figure 7(c) displays the corresponding states of the Petri net model.

Performance expectations and initial results. While the local search based pattern matchers operate with cheap model changes and costly pattern queries, the sequential RETE-based matcher [BÖR+08] has a moderate overhead on model change balanced by instant pattern queries. This novel concurrent incremental pattern matching approach combines the advantages of the former two: it has cheap model manipulation costs, and potentially instant pattern queries. Although the transformation might have to wait for the termination of the background pattern matcher thread, the worst case of this time loss is still comparable to the update overhead of the original RETE approach.

This concurrent approach is expected to improve performance over a non-concurrent implementation (as described in Section 3) if there are comparatively infrequent pattern matcher queries and complex model changes between them. This would correspond to a *forall* style control flow when all matches of a pattern are obtained first, and then each of them is processed (potentially) simultaneously, which is common in model-to-model transformation scenarios. This complements the traditional advantage of incremental pattern matching, which manifested especially on *as long as possible* style control flows: when single matches are selected and processed one by one until there are no matches of the pattern. Initial experiments[1] have shown that the concurrent approach improves performance by up to 20% on the Sierpinsky benchmark of [SNZ08]. For building a Sierpinsky-triangle of 8, 9 and 10 generations, our original RETE ran for 2.6s, 8.3s, and 26.2s, while the concurrent solution took 2.2s, 6.9s, 22.8s to terminate, respectively.

4.2 Multi-threaded pattern matching with RETE

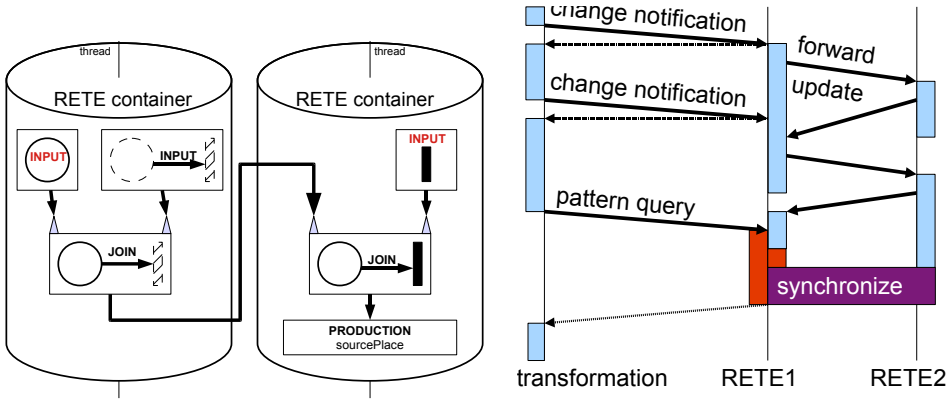
The concurrent patten matching approach can be improved further given that the hardware architecture is capable of running multiple threads efficiently. There are various approaches of parallelizing the RETE algorithm, see Section 5 for details. Here we present our simple solution, mostly for illustration purposes.

[1] Environment: 2.2GHz Intel Core 2 Duo processor, Windows Vista, Sun Java 1.6.0_11, 1GB heap memory

The basic idea is to employ multiple pattern matcher threads to consume update messages. However, if these threads share the same message queue and RETE nodes, and multiple threads could access the same node simultaneously, this could easily lead to complex inconsistency problems, which could not be easily avoided by locks.

Our proposal splits the network into separate RETE *container*s, each of which is responsible for matching a set of subpatterns. A container has its own distinct set of nodes, and assigns each RETE container to a dedicated pattern matcher thread consuming update messages of a dedicated queue. Each container is responsible for forwarding messages to its nodes using the dedicated message queue. This way, two threads are not allowed to operate on the same RETE node, thus maintaining mutual exclusions is not necessary.

Forwarding messages between two containers is accomplished by enqueueing the message in the target container. Figure 8(a) depicts a parallel version version of Figure 6 illustrating how a RETE net can be split into several containers for parallel execution.



(a) RETE containers (b) Message sequence

Figure 8: Multi-threaded pattern matching

If a container runs out of update messages to process, it reaches a *local fixpoint*, otherwise it remains *active*. The *global fixpoint* is reached when all containers are in a local fixpoint. In order to retrieve up-to-date and consistent match sets, the transformation thread has to wait for a global fixpoint; see Figure 8(b) for illustration. This thread synchronisation goal, however, is not trivial to accomplish, since a container can leave its local fixpoint and become active again before a global fixpoint is reached due to incoming messages from other, still active containers. To address this issue, we have developed a termination algorithm based on logical clocks that is able to determine global fixpoints [Ber08], which is not presented here for space considerations.

Performance expectations. It is important to point out that the performance of such a system may depend highly on the amount of synchronization and replication that is necessary when messages are passed between the containers. In theory, it would seem beneficial if the subpatterns (deployed to separate RETE containers) had a low number of interconnections, but further research is necessitated to achieve this in practice. An ideal application scenario would be sev-

eral transformations or parts of the same transformation that are known to use different patterns; allowing easy, straightforward splitting and parallelisation of the RETE net, with a low amount of inter-connectedness. By partitioning the patterns into relatively independent containers, a multi-threaded RETE pattern matcher may achieve high performance.

4.3 Multi-threaded model manipulation

In case of incremental pattern matching, the usefulness of optimizing the pattern matcher has its limitations, as significant CPU time is spent on the rule application itself. Further gains in performance can only be achieved by accelerating the execution of rule application and model manipulation. For this purpose, we exploit multi-core architectures to provide multi-threading for the model manipulation component as well.

Several approaches aim to achieve serializable (i.e. thread-safe) parallelisation of graph transformation rule applications, either for rule instances within one transformation sequence, or for separate transformation runs. Most advanced solutions exploit the declarative nature and concurrency theory of graph transformation to execute non-conflicting rules in parallel [Mez07].

Unfortunately, in many practical cases, model transformations are complemented intertwined with hard-to-analyze imperative actions, or simply, there are too many conflicts. In this case, low-level solutions are required for parallel execution by using a locking system on the model (in analogy with how locks are used for scheduling imperative transactions). The lock system can have, for instance, a model-level, an element-level, or hierarchy-based lock granularity; also, a read lock / write lock model is preferable.

Multi-threading for model manipulation can be achieved easier if pattern matching is performed in a separate thread, as described in Subsection 4.1 (or multiple threads, as in Subsection 4.2). When model manipulation threads change the model, they send update notifications atomically, which involves inserting an update message addressed to the appropriate input node into the message queue of the node's container. When a transformation thread requires the set of matches for a certain pattern, the pattern matcher call returns them immediately if the network is in a global fixpoint, or suspends the thread (but not others) until that fixpoint state is reached.

Performance expectations. A high amount of synchronization can diminsh performance both through waiting and overhead. Parallel execution of read-intensive transformations is relatively conflict-free, therefore it does not heavily suffer from waiting, and can scale up to multiple CPU cores efficiently. A suggested application scenario would be parallel code generation, with each thread producing a separate output file from a corresponding aspect of the source model. Code generation is usually a read-only operation; we also believe that using different aspects of the model aids in the partitioning of the RETE net.

4.4 Initial evaluation of parallel transformation

For evaluation purposes, we have extended VIATRA2 with multiple transformation threads, concurrent pattern matching, and model-level R/W locking. We used this system to measure the performance of parallel code generation, namely generating PNML [JKW02] descriptions of several Petri nets within the model space. This application scenario has the advantage that trans-

formation jobs are entirely read-only. However, since all generation jobs basically follow the same algorithm and use similar Petri-nets, it does not easily lead to partitioning the RETE net; therefore, we used concurrent pattern matching, but with a single RETE container. The initial expriments[2] show that this system has a quasi-linear scalability. A Petri-net generated by the procedure in [BHRV08] as "Size 50000" was used as a sample input. Two PNML code generators in sequence ran for 2.9s each, 5.8s altogether. Two code generators in parallel ran for 3.7s each, but they took only 3.9s altogether. Due to the complexity (and sometimes strange characteristics) of parallel algorithms, further measurements are required to compare its performance with non-incremental approaches.

5 Related work

Incremental pattern matching. Incremental updating techniques have been widely used in different fields of computer science (including view updates in relational databases [GMS93]). In graph/model transformation tools, PROGRES [SWZ99] supports incremental attribute update performing immediate invalidation of partial matchings, while the validation of partial matchings are only computed on request (i.e., when a matching for the LHS is requested). The transformation engine of TefKat [LS05] performs an SLD resolution based interpretation to construct and incrementally maintain a search space tree representing the trace of transformation execution [HLR06]. The uniform, incremental handling of model elements and patterns can be considered a unique, advanced feature of the approach. [VVS06] proposes to store a tree for the partial matches of a pattern, and incrementally updates it upon model changes.

RETE networks. RETE networks [For82], which stem from rule-based expert systems, have already been used as an incremental graph pattern matching technique in several application scenarios including the recognition of structures in images [BGT90], and the co-operative guidance of multiple uninhabited aerial vehicles in assistant systems as suggested by [MMS08]. Our contribution extends this approach by supporting a more expressive and complex pattern language.

Parallel RETE. There is also some work in literature in the context of parallel or distributed RETE implementations. For instance, [AT98] focuses on parallelizing rule applications, [MK90] parallelizes pattern matching. Unfortunately, certain approaches focusing on expert systems are hard to be accessed, e.g. due to vague patent descriptions [Lin05], and certain industrial solutions might not be published at all. Anyhow, these approaches rarely provide proofs to guarantee the global termination of local updates as mentioned in Subsection 4.2, which is specific to our model transformation context.

Parallel graph transformations. In addition to large amount of theoretical work on concurrent and parallel aspects of graph transformation, relatively little practical work has been carried out. Some advanced solutions were proposed by G. Mezei [Mez07] who analyses pattern conflicts and groups executable rules into *independence blocks* to execute them in parallel. Further contributions also introduced parallel pattern search for first occurence and all occurrences. Our current work is complementary to his work, as it offers parallelization with a different pattern matching paradigm. Future research shall be conducted to identify how to combine the strength of the two approaches.

[2] Environment: 2.2GHz Intel Core 2 Duo processor, Ubuntu 8.10, x64 OpenJDK 1.6.0_0, 1.5GB heap memory

6 Conclusion

The paper introduced various techniques for parallelizing graph transformation systems using incremental pattern matching. More specifically, we discussed how to exploit the power of modern computers with multiple processor cores, tailored to the specialities of incremental pattern matching. Our approach decouples model manipulation and pattern matching, and then parallelizes each of these phases.

We also sketched conditions when the proposed solutions are expected to perform best: (i) transformations with longer model manipulation sequences, (ii) transformation runs accessing different patterns, and (iii) transformation runs that are read-intensive.

Finally, an initial implementation of all the three presented ideas has been integrated to the VIATRA2 model transformation framework, and an initial performance evaluation of these parallelization techniques was carried out.

Acknowledgements: This work was partially supported by EU projects SENSORIA (IST-3-016004) and SecureChange (ICT-FET-231101), and the László Schnell Foundation.

Bibliography

[AT98] M. M. Aref, M. A. Tayyib. Lana—Match algorithm: a parallel version of the Rete—Match algorithm. *Parallel Comput.* 24(5-6):763–775, 1998. doi:http://dx.doi.org/10.1016/S0167-8191(98)00003-9

[Ber08] G. Bergmann. Incremental graph pattern matching and applications. Master's thesis, Budapest University of Technology and Economics, May 2008. http://home.mit.bme.hu/~bergmann/publications/bergmann_diploma_2008.pdf

[BGT90] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In Ehrig et al. (eds.), *Graph-Grammars and Their Application to Computer Science*. Lecture Notes in Computer Science 532, pp. 174–189. Springer, 1990.

[BHRV08] G. Bergmann, A. Horváth, I. Ráth, D. Varró. A Benchmark Evaluation of Incremental Pattern Matching in Graph Transformation. In *ICGT*. 2008.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental Pattern Matching in the VIATRA Model Transformation System. In Karsai and Taentzer (eds.), *Graph and Model Transformation (GraMoT 2008)*. ACM, 2008.

[BS03] E. Börger, R. Särk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook on Graph Grammars and Computing by Graph Transformation.* Volume 2: Applications, Languages and Tools. World Scientific, 1999.

[For82] C. L. Forgy. Rete: A fast algorithm for the many pattern/many object pattern match problem. *Artificial Intelligence* 19(1):17–37, September 1982.

[GMS93] A. Gupta, I. S. Mumick, V. S. Subrahmanian. Maintaining views incrementally (Extended abstract. In *In: Proc. of the International Conf. on Management of Data, ACM*. Pp. 157–166. 1993.

[Gra08] GraBaTs - Graph-Based Tools: The Contest. 2008. http://www.fots.ua.ac.be/events/grabats2008/.

[HLR06] D. Hearnden, M. Lawley, K. Raymond. Incremental Model Transformation for the Evolution of Model-Driven Systems. In Nierstrasz et al. (eds.), *MoDELS*. Lecture Notes in Computer Science 4199, pp. 321–335. Springer, 2006.

[JKW02] M. Jungel, E. Kindler, M. Weber. The Petri Net Markup Language. In *In S. Philipi, editor, Algorithmen und Werkzeuge fur Petrinetze (AWPN), Koblenz*. June 2002.

[Lin05] P. Lin. System and method to distribute reasoning and pattern matching in forward and backward chaining rule engines. US Patent application 20050246301, 02 2005.

[LS05] M. Lawley, J. Steel. Practical Declarative Model Transformation With Tefkat. In Bézivin et al. (eds.), *Proc. of the International Workshop on Model Transformation in Practice (MTiP 2005)*. October 2005. http://sosym.dcs.kcl.ac.uk/events/mtip05/.

[Mez07] G. Mezei. Supporting Transformation-Level Parallelism in Model Transformations. In *Automation and Applied Computer Science Workshop*. Budapest, Hungary, 2007.

[MK90] M. Mahajan, V. K. P. Kumar. Efficient parallel implementation of RETE pattern matching. *Comput. Syst. Sci. Eng.* 5(3):187–192, 1990.

[MMS08] A. Matzner, M. Minas, A. Schulte. Efficient Graph Matching with Application to Cognitive Automation. In Schürr et al. (eds.), *AGTIVE 2007*. Springer Verlag, 2008.

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

[SNZ08] A. Schürr, M. Nagl, A. Zündorf (eds.). *Applications of Graph Transformations with Industrial Relevance, Third International Symposium, AGTIVE 2007, Kassel, Germany, October 10-12, 2007, Revised Selected and Invited Papers*. Lecture Notes in Computer Science 5088. Springer, 2008.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES approach: language and environment. Pp. 487–550, 1999.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Sci. Comput. Program.* 68(3):214–234, 2007.

[VVS06] G. Varró, D. Varró, A. Schürr. Incremental Graph Pattern Matching: Data Structures and Initial Experiments. In Karsai and Taentzer (eds.), *Graph and Model Transformation (GraMoT 2006)*. Electronic Communications of the EASST 4. EASST, 2006.

Visual compilation of behavioral modeling languages

Erhard Weinell

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

Abstract: Implementing behavioral modeling languages, i.e. those describing the processing of data rather than data structures, is a challenging task. Domain specific languages especially suffer from this fact, as their initial realization as well as later evolution should be enabled in a rapid way. Existing solutions frequently found in the DSL community often apply code generation mechanisms for this purpose. These solutions are usually not restricted to specific kinds of DSLs, but their support is often rather rudimentary.

In our project, we explicitly target behavioral languages operating on graph-based, hence discrete data structures. Consequently, we can offer improved realization support for a restricted set of DSLs. To this end, the present paper introduces a simple transformation language tailored to build DSL compilers. Starting from a DSL's abstract syntax representation, the created compilers generate graph queries or transformation rules for further processing by the provided machinery.

Keywords: execution framework, behavioral modeling languages, graph transformations

1 Introduction

Employing *specialized* or *domain-specific languages* (DSLs) in software development processes promises higher developer productivity by easing specific tasks during development. During the past years, numerous projects have consequently proposed such languages and surrounding toolchains for various application areas. Representatives tackle very different aspects of programming such as numerical algorithms, software architectures, and web applications[1].

DSLs are often designed as a company's auxiliary means rather than a leading product, therefore their development should only require restricted amounts of resources. Therefore, there is need to aid the design and implementation of DSLs, which is addressed by specific frameworks and meta-languages. *Behavioral* DSLs describing inspection or manipulation of data, rather than data itself, are especially challenging with this regard: The modeled specification documents need to be evaluated according to the language's semantics. A common approach for this task is to *derive code* for general purpose languages (GPLs) using source-to-source transformations. Tools to model such transformations are readily available, e.g. Stratego/XT [Vis03] for DSLs with textual concrete syntax. However, depending on the conceptual *gap* between source

[1] The *Annotated Bibliography* hosts numerous examples, see http://homepages.cwi.nl/~arie/papers/dslbib/

DSL and target GPL, this can result in complex translations being hard to develop and to maintain. Furthermore, means to technically access data storages have to be incorporated into the generated code.

In our project, we support the development of specialized behavioral languages related to the *processing of discrete models*. Such languages can describe querying, editing, and translation of models using specially tailored language constructs and pragmatics. In the following, these are referred to as *graph languages*. In contrast, we do not explicitly support *static* modeling languages, e.g. to describe data structures. Furthermore, we provide a graph-oriented view on data e.g. stored in a relational database or in a model repository. A basic yet extensible language and execution engine for graph transformations, which can be considered a very-high-level virtual machine for declarative "machine code", supports the evaluation of behavioral specification documents. Behavioral DSLs can thus be implemented by compilation on a high level of abstraction, therefore leveraging the aforementioned conceptual gap.

Compilation in our case requires the translation from abstract syntax models of a DSL's specification documents into graph transformation rules. This can be naturally achieved through model transformations, which allow to capture the functionality required for this purpose. More specifically, a compiler needs to implement uni-directional translations, while keeping trace informations for later debugging. Though our initial aim was to re-use existing model transformation languages for this purpose, numerous ones showed to yield inconveniently verbose specifications in our scenario. Therefore, a *specialized translation mechanism* has been developed to enable more concise solutions. This mechanism being introduced in the present paper offers constructs from a *fixed translation target domain*, and eases its use by certain (overridable) *heuristics*.

Structure of the remaining paper: Section 2 gives a concise architectural overview on the proposed approach to show the projects "big picture". Afterwards, Section 3 introduces a running example before Section 4 presents the transformation tooling. Section 5 discusses related fields in the literature, and Section 6 concludes.

2 Overview

Figure 1 shows a coarse-grained overview on the presented approach as introduced above. From the figure's left side, one can note that DSLs are integrated with our framework based on their abstract syntax model. However, the construction of editing environments or processing tools like parsers is not considered in this project, which therefore need to be supplied externally. The abstract syntax models of specifications need to be exported from such tools to the graph-oriented database system DRAGOS, which we apply as model repository. Consequently, specifications are stored as graph structure instantiating a DSL-specific graph schema (i.e. metamodel).

The middle column depicts the specification after translation into the framework's core language, called DRAGULA. Although details on this language are out of the scope of the present paper, more insights are given in [Wei08a, Wei08b]. Based on this second representation, the DRAGULA Execution Engine is able to evaluate a given specification through a sequence of graph transformations. For this purpose, the execution engine operates on a host graph constitut-

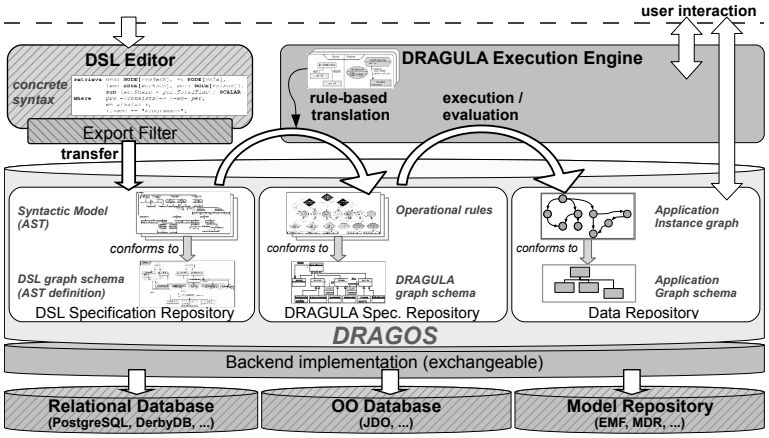


Figure 1: Architectural overview

ing the system's runtime data structures. This third graph structure, which might be visualized by graph browsers or suitable UI frameworks, is shown as right column in the figure.

The transition from the specification's DSL-specific model to its DRAGULA counterpart is related to the compilation of source code to executable byte code, as followed e.g. by Java or the ATL. Unlike these two languages' backends, DRAGULA does not settle at a low-level of abstraction comparable to stack machines. Instead, DRAGULA constitutes a high-level declarative language itself based on graph transformation concepts. Implementing compilers from behavioral DSLs to this core language therefore is a *comparatively* easy task, which needs to be aided by proper tool support, nevertheless. A transformation language dedicated to this specific task has been developed in the context of the proposed framework, which will be introduced in the following.

For its main contribution, this paper uses an example-oriented approach instead of a rigorous formal definition. This is motivated by space restrictions and the main goal to present an understandable, yet realistic scenario. An elaborated language definition will be subject of a future publication.

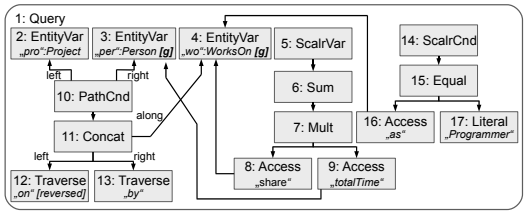
3 Example Graph Language

To illustrate the use of the framework introduced above, this section presents a concrete example language for *querying* graph structures. In the following, the compilation of this language to DRAGULA is explained. Figure 2 illustrates a query conceptually related to GReQL [KW99], a query language originally developed for a reengineering environment. As host graph schema, we assume a data model related to project management applications. Specifically, the query should report all projects and the total labor time spent on the respective one by all of its programmers.

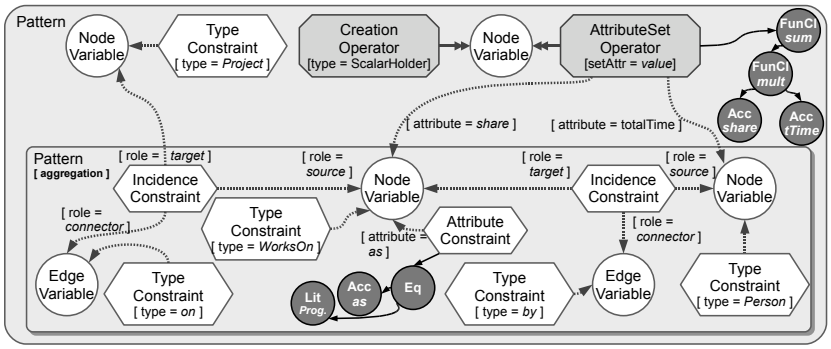
The textual concrete syntax depicted in Figure 2a side comprises two parts, namely entities and values being queried (upper part), and predicates that need to be fulfilled by these entities (lower part). The **`grouped`** keyword indicates that entities found for `per` and `wo` do not yield distinct

```
query
  pro: Project,
  grouped per: Person,
  grouped wo: WorksOn,
  sum( wo.share
     * per.totalTime )
where
  pro <-on- wo -by-> per,
  wo.as = "Programmer"
```

(a) Concrete DSL syntax



(b) Abstract DSL syntax



(c) DRAGULA compilation derived from Figure 2b

Figure 2: Sample graph query in DSL and DRAGULA syntax

results, but all of their valid assignments should be grouped together for further processing[2]. These grouped variables are used in the scalar expression below to sum up the multiplication's results. The query's lower part checks connectivity of entities, e.g. `wo` having a directed link towards `pro` of type `on`. Also, the attribute `as` of `wo` is compared to the depicted value.

Figure 2b shows the query's abstract syntax, which will be used for compilation to DRAGULA. Considering the query language's static semantics, applied identifiers have been resolved to cross-references, e.g. for the left and right legs of path conditions. Therefore, this structure constitutes a (abstract semantics) *graph* instead of an (abstract syntax) *tree*.

The third part Figure 2c finally shows the same query expressed in DRAGULA core language. The basic building blocks of DRAGULA rules are called Patterns, depicted as rounded boxes in the figure. The outer pattern contains a NodeVariable, for which a value is retrieved from the underlying data storage. By attaching a TypeConstraint, this value is restricted to a specific type, `Project` in this case. Another NodeVariable is used to store the scalar value calculated by the aggregation expression. Due to the fundamentally graph-oriented paradigm of DRAGULA, no dedicated scalar variables are available. Therefore, a new node is *created* after pattern matching, using the depicted CreationOperator. Afterwards, its value attribute is set to the resulting total labor time spent on the matched project entity using an AttributeSetOperator. This value is calculated by evaluating a tree of expression elements (gray circles) representing predefined functions (FunCl) and accesses to entity attributes (Acc). For the latter purpose, the referred entity variables must

[2] In terms of SQL, this construct is the inverse notation of `GROUP BY`, which enumerates all non-grouped elements.

be attached to the respective AttributeSetOperator.

A second, nested pattern contains all variables and related constraints representing **`grouped`** elements in the concrete syntax. In this example, it therefore comprises NodeVariables for `wo` and `per`, as well as EdgeVariables to match their respective connectivity. Incidence between nodes and edges is expressed using IncidenceConstraints, whilst additional TypeConstraints ensure that traversed edges are of the desired type (`on` resp. `by`). An AttributeConstraint compares the as attribute of the attached variable's value to the literal `Programmer`. The nested pattern's aggregation annotation[3]. causes the execution engine to find *all valid assignments* for its variables. To the surrounding pattern, these results are returned as one *vector of entities* for each variable.

Conclusion: This example shows a basic usage scenario for our framework, the implementation of a language specialized on graph queries[4]. The DRAGULA language embraces separation of concerns, which is achieved by distinguishing between *variables* to be assigned, and *constraints* that have to be fulfilled for these assignments. Pattern matching can therefore handle both typed and untyped models, check incidence between nodes and edges considering or neglecting direction, etc. Though not required in this example, DRAGULA supports nested graph structures and hypergraphs, too. Manipulations of graph structures are expressed by semi-imperative (since order-independent) *operators*. The relatively verbose modeling style, though impeding readability, eases automated processing, as different language aspects are expressed using distinct entities. Furthermore, DRAGULA can easily be extended by additional language constructs in form of new entity types.

4 Simple Visual Transformations

Up to now, we have seen a single example for expressing a query in the DRAGULA graph language. Consequently, the question how to handle such translations in general arises. Model transformations are the natural choice to compile specialized languages or DSLs to DRAGULA, as their according documents are represented by graph-based models. Although the literature hosts numerous languages suitable for this purpose such as triple-graph-grammars [Sch94], these have shown to be too generic for convenient and concise modeling of the required translations – as will be discussed in Section 5. Therefore, we propose a novel language especially focused on using the DRAGULA language as target meta-model. The achieved results, however, can be transfered to other target graph languages as well, albeit presumably not to low-level machine code.

The *Simple Visual Transformation Language* (SViTL) is a rule-based declarative language to relate source- and target-side specification fragments to each other. For disambiguation, SViTL rules are therefore referred to as *SViTL pairs* or simply *pairs* in the following. In addition to a DSL's abstract syntax elements, the source-side may comprise more advanced constructs found in many graph transformation languages, namely *NACs* and *path expressions*. PROGRES-like optional or set-valued elements are not supported, neither are nested or recursive patterns as

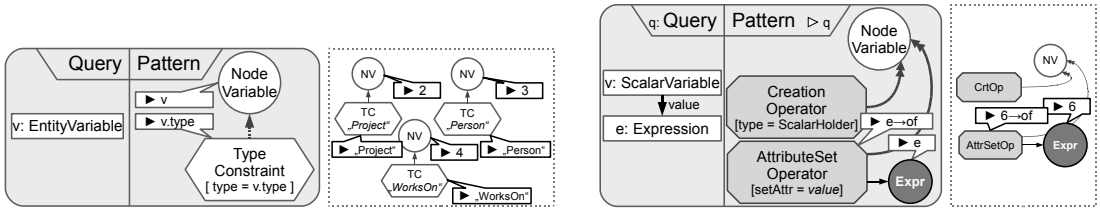
[3] In addition, nested patterns can also model boolean composition of queries, e.g. using `or`, `not`

[4] In a wider definition of *domain*, it could be considered a DSL as well.

e.g. offered by VIATRA2. The target-side only supports elements of the DRAGULA language, without any "advanced" constructs.

4.1 Translation of basic language constructs

As initial SViTL pairs, we consider the *variable* concept in both languages of Figure 2. First, Figure 3a relates EntityVariables to pairs of DRAGULA variables and constraints. Evaluating this pair creates the depicted DRAGULA elements *for each match* of the pair's source-side, as suggested by the adjoining host graph fragments (surrounded by the dotted box). Furthermore, the pair's target-side comprises so-called *anchor definitions* (the balloon-shaped elements in the figure) attached to the DRAGULA elements. These constructs *anchor* the created DRAGULA elements to source-side elements or scalar values. As can be seen in the figure, NodeVariables are anchored to the respective objects' identifiers from Figure 2b, whereas TypeConstraints refer to the source-side element's type attribute. Anchor definitions must be unique for each pair's target-side, only one definition referring to a given source-side element is allowed.



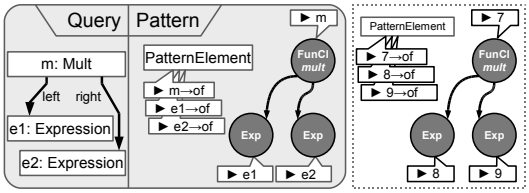
(a) Entity Variables (b) Scalar Variables

Figure 3: SViTL pairs handling variable declarations

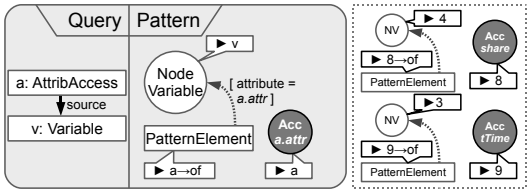
Second, as discussed in Section 3, a dedicated ScalarHolder node must be created at runtime to return a scalar value from a query. The SViTL pair shown in Figure 3b provides the required elements, including an Expression object on both sides. The of-anchor's purpose in this pair is to associate all elements of the DRAGULA expression tree being constructed with the AttributeSetOperator by means of common anchors. Note, that the present pair does not consider the actual type of the source-side's expression node (Sum, c.f. Figure 2b), nor does it specify a corresponding type at the DRAGULA-side. Instead, concrete type informations are provided by other pairs later on. Figure 3b additionally presents the use of *anchor labels* to distinguish anchor definitions for the same source-side object. Formally, the uniqueness constraint of anchor definitions stated above applies to equally labelled ones only. Assigning no label at all represents a "default" label, but is not treated specially in any way.

The further processing of expression trees is depicted in Figure 4, starting with arithmetic operations in Figure 4a. Note, that this pair does not refer to an AttributeSetOperator directly, but to the general type PatternElement instead. This way, expression trees used by set-operators and constraints can be handled uniformly. Figure 4b finally handles access to attributes of entity variables. For this purpose, the container needs to be connected to the respective entity variable. The created edge as well as the Access expressions receive the requested attribute's name. This is a simplified depiction: In reality, a unique combination of the entity variable's identifier and

the requested attribute are used as *alias* for the collected value, to avoid ambiguities.



(a) Binary operators, specifically multiplication



(b) Access to entity attributes

Figure 4: SViTL pairs handling expression trees (created fragments restricted to the outer pattern)

Up to now, we have examined SViTL pairs to represent variables and scalar expressions based on DRAGULA elements. When *executing* these pairs to compile the query from Figure 2, *conceptually* all target-sides are created *independently*, and in *arbitrary order* for all source-side matches. Afterwards, all anchored elements referring to the same anchor are *glued together*. For this purpose, all incident edges and attribute values are diverted to a single element. Furthermore, if the elements being glued are of different types, then one must be a direct or indirect subtype of the other. In this case, the more specific type "wins" in that it is taken for the glueing result.

Considering the host graph fragments of the two previous figures, the following Figure 5 shows the result after glueing. For instance, both node variables anchored to the source-side object 3 are glued together. The resulting variable is therefore connected to a TypeConstraint and to the AttributeSetOperator. Likewise, the Expression construct anchored to 8 in Figure 4a is refined to an Access by Figure 4b. As one can see, processing SViTL pairs is not limited to tree-like structures on the source-side, as no explicit traversal strategy needs to be defined. Therefore, graph-based specifications like the ASG of our example query can be processed without additional means like e.g. explicit rule ordering.

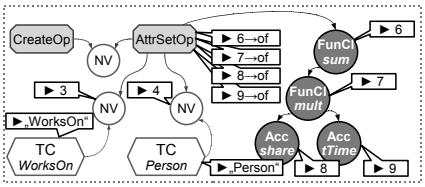


Figure 5: Glued DRAGULA elements

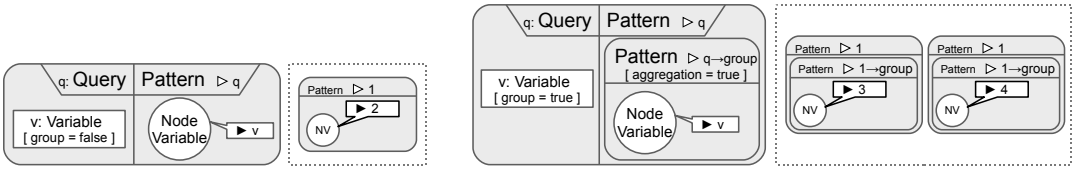
4.2 Translation of Scope-effecting constructs

Up to now, all DRAGULA fragments on a pair's right side are placed into a single common Pattern. However, as can be seen in Figure 2, it is crucial to support the creation of nested structures and to select the proper container for each created element, as this effects the pattern's semantics. At the same time, translations should only be concerned with this issue if required, so that complexity of basic translation rules does not increase. Furthermore, we cannot assume a tree-like structure of the source model, which would allow to pass containers of target-side elements along this hierarchy. SViTL handles containment informations implicitly whereever possible, giving developers means to override this behavior if required.

The basic idea to assign the desired container pattern to DRAGULA elements is to record their respective *neighborhoods*. Unless stated otherwise, a neighborhood comprises all elements created simultaneously when applying a SViTL pair. When glueing equally anchored elements together, their respective neighborhood informations are combined as well, such that the glued element is possibly contained in multiple neighborhoods. If a neighborhood contains a pattern, then it is used as container for all remaining elements. Otherwise, container informations are propagated between neighborhoods, as will be discussed below.

Modeling container-related informations. Concerning the pattern structure of Figure 2c, we need to specify that *non-grouped* variables belong to an outer pattern, whilst all *grouped* ones are assembled in a nested pattern. These requirements need to be modeled *explicitly* using additional SViTL pairs. The remaining elements, e.g. constraints and attribute expressions, should however be assigned to the most suitable container automatically. Otherwise, SViTL specifications would need to verbosely specify containers for each created element. Neighborhoods, together with heuristics-based propagation of container informations, solve this problem.

The two additional SViTL pairs depicted in Figure 6 provide the required informations regarding the variables' containers. In contrast to previous pairs, the target-side container is *explicitly* anchored to the source-side, query q in this case. This notation includes the created pattern into the target-side neighborhood, which thus comprises the pattern and the variable in Figure 6a. Likewise, Figure 3b applied these means to relate all elements concerning scalar variables to the same container, anchored to the query object 1. All other patterns presented before do not relate target-side elements to any container, as no anchor definition is given for the respectively depicted pattern.



(a) Non-grouped variables

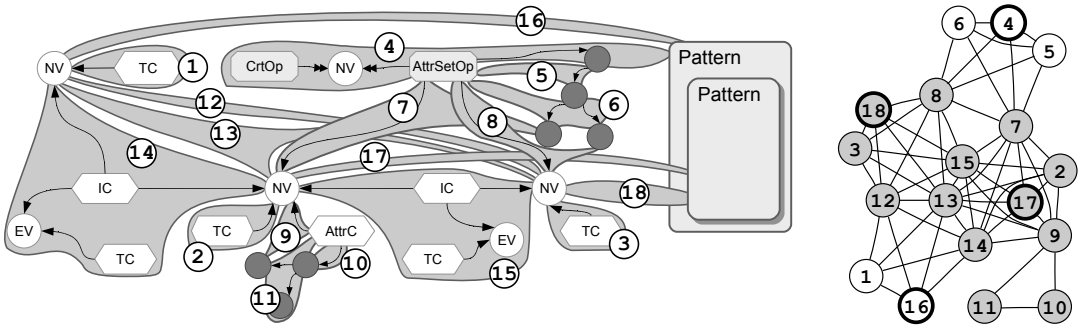
(b) Grouped variables

Figure 6: Grouping specification for variables

Finally, Figure 6b explicitly creates a nested pattern, and a variable contained therein. Neighborhoods do not span pattern boundaries. Instead, separate neighborhoods are created for each target-side pattern, including all of its contained elements. In the present case, the target-side node variable is therefore neighbored to the inner pattern, but not to the outer one. In addition, the hierarchy relation between the two patterns is recorded.

Relating neighborhoods to containers. Having discussed means to specify container hierarchies, we now examine how these additional informations effect the processing of the example query. Figure 7a depicts the created data structures after applying all SViTL pairs for all possible matches, and after glueing their respective results. In principle, it comprises the same informations as the resulting pattern in Figure 2c, except for containment properties. In addition,

all neighborhoods are labeled and visualized by filled curves in the figure. Each neighborhood stems from exactly one match of a SViTL pair, e.g. `1` to `3` from applying the pair in Figure 3a, `4` from Figure 3b, etc. Pairs modeling connectivity constraints (being omitted in this paper due to the lack of space) yield the neighborhoods `12` to `15`. The two pairs from Figure 6 create the neighborhoods `16` to `18`.



(a) Neighborhoods in the resulting pattern

(b) Neighborhood relations graph

Figure 7: Resulting neighborhoods and their interrelations

Due to subsequent glueing of target-side elements, each one can participate in multiple neighborhoods. By examining which neighborhoods share common elements, we can directly derive the *neighborhood relations graph* from Figure 7b as follows: Neighborhoods are represented by nodes, and undirected edges indicate those neighborhoods sharing at least one created element. Nodes with thick borders especially denote neighborhoods comprising a pattern, as these are treated specially in the following.

Using the neighborhood relations graph, we can relate each neighborhood to a suitable pattern. From a given neighborhood, we determine the *shortest path* to any neighborhood containing a pattern, i.e. `4`, `16`, `17`, or `18`. The considered neighborhood is then associated with the pattern of the neighborhood reachable by the shortest available path. If this pattern is not uniquely determined, we require the patterns in question to be in hierarchical order, i.e. one of them is transitively contained within the other. In case this condition is fulfilled, we use the *innermost* pattern as heuristic solution. In the present example, only two patterns with direct hierarchical order occur. The resulting mapping of neighborhoods to patterns is indicated in Figure 7b: White nodes belong to the outer pattern in Figure 7a, whereas gray ones belong to the inner pattern.

Assigning created elements to containers. In the previous subsection, we identified proper containers for each *neighborhood*. To finally create and fill the desired target pattern, the acquired informations still have to be propagated to the respective DRAGULA elements. Again, we follow a heuristic approach here: Each DRAGULA element is placed into the *outermost* pattern any of its neighborhoods is associated with. Again, we assume that a hierarchical order between possible containers exists.

Using this algorithm, it is possible to derive the final pattern structure for the example query

as shown in Figure 2c. For example, the upper variable and its attached constraint are placed into the outer pattern due to neighborhood `1`. In contrast, the connectivity check in `14` is moved to the inner pattern, as the inner one dominates in the abovementioned algorithm. Furthermore, the expression tree for the scalar variable is placed into the outer pattern. This is caused by neighborhood `6`, although the Access expressions are also related to the inner pattern's variables.

Design-rational of the SViTL heuristics. Having introduced SViTL and its heuristics to treat containment properties, we shortly discuss the rational behind these design decisions.

- Neighborhoods are associated with the *innermost* related pattern. This way, entities not concerned with pattern hierarchies are placed into more deeply nested patterns whenever possible. For example, the connectivity check attached to the outer variable comprises several elements not being contained in any other neighborhood. This pattern fragment is therefore "pushed" into the hierarchy to the level of its deepest related neighborhood, i.e. the pattern holding grouped variables as defined by the explicitly anchored variable. In case this behavior is not intended, a SViTL pair can define an explicit anchoring to place this check into the outer pattern instead.
- Having determined all neighborhoods' possible containers, each DRAGULA element is placed into the *outermost* candidate. This decision is derived from the observation that an outer pattern's elements are more likely referenced from an inner one than in reverse. Therefore, the outer pattern is considered as *declaring scope* for all of its neighborhoods and their respective elements.

Besides these two heuristics, explicit anchoring can always be applied to denote an element's container manually. For example, if the container anchoring would have been omitted in Figure 3b, neighborhoods `4` to `6` would have been assigned to the inner pattern instead of the outer one.

4.3 Formal properties

Having introduced SViTL as formalism for language translation, two of its formal properties should be discussed in the remainder of this section, namely termination and confluence. Although we cannot give a formal proof for these aspects, intuitive arguments are provided.

Termination. Termination guarantees that a translation result is obtained in finite time, assuming a finite input model. To ensure this property, the calculation of matches, the control of individual translation steps, and the post processing need to be considered.

SViTL pairs are always executed for all of the source side's matches. These can be determined a priori, as the translation process does not alter the input model. Using only basic language constructs on the LHS, as applied in the present case study, always yields a finite number of matches. For path expressions, the language definition has to ensure that cycles in the source model are detected (and broken) when evaluating transitive closures. Translation control obviously terminates, as each pair is processed only once for all of its matches. Post processing, comprising the glueing of elements and assigning containers, only considers a finite number of elements. For

each of these, a finite number of properties, comprising edges and attribute values are modified. To conclude, the termination of SViTL translations can be derived from these arguments.

Confluence. Besides termination, all alternative translations of a given input model should yield semantically equivalent results. This property is sufficiently guaranteed if the translation can be shown to be confluent, such that all obtainable target structures are (syntactically) equivalent up to isomorphism. To show this property, we examine possibly arising non-determinisms in the translation process, as these state the decision points where evaluations may diverge.

In SViTL, non-determinism is caused by the matches of a given pair's LHS, which may be processed in random order by the execution engine. This ordering, however, does not effect the outcome of the translation, as all results are created independently of each other. Due to the same reason, application order of individual SViTL pairs does not effect the translation's outcome. Hence, all non-deterministic choices up to this point yield equal result states.

Furthermore, the effect of the post processing steps need to be taken into account. Here, rule application order again does influence the result, as all priority decisions are based on inter-element dependencies. These dependencies comprise type hierarchy for the glueing of elements, and mutual containment properties for the selection of container patterns, both of which are not influenced by non-deterministic decisions.

Conclusion. To conclude, this paper gives an informal and, from the viewpoint of theory, admittedly unsatisfying argumentation of termination and confluence. At least for the considered query language and the example query, existence and uniqueness of the resulting DRAGULA rule has been validated using the GROOVE system [Ren03]. Of course, this neither confirms these properties for other queries or even other language translations. Due to SViTL's close relation to triple-graph-grammars, however, formal foundations such as presented in [EP08] might be a suitable starting point for further analysis.

5 Related Work

Besides SViTL, there are numerous established transformation languages readily available. The probably most prominent approach among the graph transformation community are triple graph grammars (TGGs) as introduced by Schürr [Sch94] in the nineties. TGGs are a general framework for translation purposes, and hence cannot take specifics of the source or target model into account. SViTL in contrast has been explicitly tailored for using DRAGULA as target language, supporting nested structures and an automated, heuristics-based container assignment strategy. Propagation of container informations would have to be specified manually in each TGG rule, whereas they can be omitted in most SViTL pairs in the provided case study.

As second difference, the rule application strategy of SViTL is not driven by context patterns like in TGGs. Instead, developers can safely neglect rule dependencies, as each SViTL pair is evaluated independently of the rest. For example, concatenation of path expressions can therefore be specified independently of whether a variable being traversed is explicitly given (as in Figure 2), or an anonymous variable should be created instead. For the explicit case, an additional pair relating the concatenation element to the selected variable needs to be given. Using

TGGs, a third rule creating the anonymous variable is needed plus either rule priorities or negative application conditions to control its applicability – this rule and especially the required control elements are not needed in SViTL.

A benefit of TGGs vital for many application scenarios [Ste08] is their bidirectional applicability. In the application scenario followed by SViTL, bidirectionality is not required, as this would resemble a (seldomly needed) decompiler directly integrated into the compiler. Therefore, SViTL excludes this feature.

Being conceptually very close to TGGs, the abovementioned arguments also hold for the OMG's standard on model transformations QVT[5].

BOTL [BM03] follows a slightly different approach by treating all objects created during rule application separate from their context. Similar to SViTL, objects with equal identifiers are merged afterwards. However, SViTL is not tied to a singular and globally known identifier, but uses common anchors to combine related model fragments. Therefore, SViTL pairs can be modeled by focussing on cross-cutting aspects of the processed ASG, rather than its identified elements. As remarked in [KS06], both approaches cannot be applied in cases where user-interactive selection of competing rules is needed. In the case of compiler construction, user-interactiveness can be neglected, though.

[LG08] uses patterns instead of rules for declarative model transformations. In contrast to BOTL and SViTL, it performs static analysis of the transformation specification, and infers dependencies between patterns. From these, operational rules are created. Compared to our work, the purely static information derivation is advantageous concerning pattern analyzability. However, we can provide more complex constructs like path expressions, which presumable are hard to provide in the cited work.

6 Conclusion

This paper proposes an approach to construct DSL compilers from declarative specifications. Although there are numerous approaches with similar mechanics, we offer tools for easy adaptation to specific domains. To this end, the present paper discusses SViTL as language for declaratively mapping DSLs to the provided core language. On the technical side, the DRAGULA language has been implemented based on the DRAGOS graph database system, as introduced in Section 2. An editing environment for SViTL specifications has been implemented based on the Eclipse framework. Currently, we are finalizing a transformation-based implementation using the DRAGULA language.

The major advantage of SViTL over existing solutions is its strong integration with the presented framework, which is achieved through backend-specific language constructs and an according execution mechanism. Explicit support for hierarchical structures is usually not found in existing transformation approaches. In SViTL, this is achieved through an automated, heuristics-based solution specific for the targetted DRAGULA language. Furthermore, the quasi-parallel application strategy allows to abstract from rule orderings, which are otherwise modeled through priorities or NACs.

[5] *Query/View/Transformation*, see http://www.omg.org/docs/ptc/05-11-01.pdf

Future Work. In the future, we would like to determine completeness of model transformation rules, i.e. that a DSL's constructs are appropriately covered by SViTL pairs. As result, developers could be better guided in constructing DSL compilers if they identify concepts not covered by the constructed compiler yet.

Bibliography

[BM03] P. Braun, F. Marschall. Transforming Object Oriented Models with BOTL. *Elec. Notes in Theoretical Comp. Sci.* 72(3):103–117, 2003.

[EHRT08] H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (eds.). *Graph Transformations, 4^{th} International Conference (ICGT)*. Lect. Notes in Comp. Sci. 5214. Springer, 2008.

[EP08] H. Ehrig, U. Prange. Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels. Pp. 178–193 in [EHRT08].

[KS06] A. Königs, A. Schürr. MDI: A Rule-based Multi-document and Tool Integration Approach. *Software and Systems Modeling* 5(4):349–368, 2006.

[KW99] B. Kullbach, A. Winter. Querying as an enabling technology in software reengineering. In *Proc. of the 3^{rd} Europ. Conf. on Software Maintenance and Reengineering*. Pp. 42–50. IEEE Computer Society Press, 1999.

[LG08] J. de Lara, E. Guerra. Pattern-Based Model-to-Model Transformation. Pp. 426–441 in [EHRT08].

[Ren03] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *AGTIVE*. Lect. Notes in Comp. Sci. 3062, pp. 479–485. Springer, 2003.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Mayr et al. (eds.), *WG*. Lect. Notes in Comp. Sci. 903, pp. 151–163. Springer, 1994.

[Ste08] P. Stevens. Towards an Algebraic Theory of Bidirectional Transformations. Pp. 1–17 in [EHRT08].

[Vis03] E. Visser. Program Transformation with Stratego/XT: Rules, Strategies, Tools, and Systems in Stratego/XT 0.9. In Lengauer et al. (eds.), *Domain-Specific Program Generation*. Lect. Notes in Comp. Sci. 3016, pp. 216–238. Springer, 2003.

[Wei08a] E. Weinell. Adaptable Support for Queries and Transformations for the DRAGOS Graph-Database. In Schürr et al. (eds.), *Proc. of the 3^{rd} Intl. Workshop on Applications of Graph Transformation with Industrial Relevance (AGTIVE)*. Lect. Notes in Comp. Sci. 5088, pp. 396–411. Springer, 2008.

[Wei08b] E. Weinell. Extending Graph Query Languages by Reduction. In Ermel et al. (eds.), *Graph Transformation and Visual Modeling Techniques, 7^{th} Intl. Workshop*. Elec. Comm. of the EASST 10. 2008.

Improved Flexibility and Scalability by Interpreting Story Diagrams

Holger Giese[1], **Stephan Hildebrandt**[1] **and Andreas Seibel**[1]

[1] [holger.giese|stephan.hildebrandt|andreas.seibel]@hpi.uni-potsdam.de
System Analysis and Modeling Group,
Hasso Plattner Institute for Software Systems Engineering,
University of Potsdam, Germany

Abstract: In this paper, we present an interpreter for Story Diagrams working on Eclipse Modeling Framework (EMF) models. The interpreter provides a more flexible and, under certain circumstances, a more scalable solution than the compiled Java code generated from Story Diagrams by Fujaba. Story Diagrams can now be modeled and executed within Eclipse. They can be modified and re-executed by the Story Diagram interpreter immediately without recompiling the source code and restarting the application. Our implementation also supports higher-order transformations by using Story Diagrams to modify other Story Diagrams. While interpretation obviously results in performance drawbacks, we demonstrate that the Story Diagram interpreter is able to improve the performance in certain worst-case situations compared to the average generated code. This is achieved by a dynamic ordering of the matching process, which considers the actual number of elements in an association at runtime. Such a dynamic ordering can minimize the matching effort considerably. In contrast, Fujaba generated code uses a static matching strategy. Whereas the Fujaba Story Diagrams have potentially high performance fluctuations, the performance of the Story Diagram interpreter is more steady and more scalable compared to the generated Java code.

Keywords: Graph Transformation Systems, Interpreter, Story Diagram

1 Introduction

Story Diagrams [FNTZ00], as supported by the Fujaba Tool Suite[1], are an established graph transformation approach. They have been employed in several applications ranging from behavior specification [FNTZ00], reverse engineering [NSW+02], consistency checking [WGN03, GMW06], and as an implementation technique for model transformations with Triple Graph Grammars [GW09, GH08] (TGG).

In this paper, we present our new interpreter for Story Diagrams, which works directly on Eclipse Modeling Framework (EMF)[2] models. It allows to directly execute Story Diagrams to access and modify arbitrary EMF-based models. This leads to a higher flexibility. On the one hand, because Story Diagrams are now available in Eclipse and EMF. This streamlines our workflows. Currently, EMF models are imported into Fujaba, Story Diagrams are modeled with Fujaba, code is generated, and the code is exported back to Eclipse. One the other hand, Story

[1] http://www.fujaba.de
[2] http://www.eclipse.org/modeling/emf/

Diagrams can be modified and re-executed by the Story Diagram interpreter immediately without recompiling the source code and restarting the application. The additional steps of generating code, compiling code and integrating it into the runtime environment disappear. Modeling Story Diagrams within EMF leverages higher-order transformations [MCG05] because Story Diagrams can be used to modify other Story Diagrams.

Furthermore, the interpreter also supports Dynamic EMF. Dynamic EMF objects are not instantiated from specifically generated code classes but from a generic class. This allows to create and modify meta models and their instances in runtime environments where the application of code generation is not feasible.

During the development of a model transformation system based on Triple Graph Grammars, we encountered performance issues when executing code generated from Story Diagrams by Fujaba. The reason is the static pattern matching strategy used by the generated code, which is occasionally not the optimal pattern matching strategy. The interpreter uses a dynamic pattern matching strategy, that tries to find matches using those instance links with the lowest number of elements, first. This is also the optimal matching strategy in many cases and results in a better scalability compared to Java code with a non-optimal static matching strategy. We have conducted an evaluation that compares the runtime performance of the interpreter with the compiled Java code of Fujaba. As outlined in [VSV05] and [TBB+08], Fujaba has been shown to be one of the most efficient graph transformation engines in comparison to AGG [TÖ00], PROGRES [REEK99], GReAT [BNBK06], and other approaches. Because of that observation, we restrict our analysis to a direct comparison between the new Story Diagram interpreter and the compiled Java code of Fujaba.

The paper is structured as follows: We first describe Story Diagrams as supported by Fujaba in Section 2. Then, we describe our EMF-based meta model of Story Diagrams and the Story Diagram interpreter in Section 3. In Section 4 we discuss the benefits of interpreting Story Diagrams implying a higher flexibility (Section 4.1) and scalability (Section 4.2). The paper closes with some final remarks and an outlook on planned future work in Section 5.

2 Story Diagrams in Fujaba

Story Diagrams extend UML Activity Diagrams by so-called Story Activities to model the behavior of a method of a UML Class. Therefore, they are usually used in conjunction with a UML Class Diagram that describes the structure of a software application. Fujaba is a UML CASE tool that supports Story Driven Modeling (SDM), which is the modeling of Story Diagrams and the generation of Java code from UML Class Diagrams and their accompanying Story Diagrams. This way, it is possible to completely create Java applications using the models provided by Fujaba. Besides Story Activities, Story Diagrams can also contain other kinds of activities like Statement Activities. These activities contain plain Java code. User defined code is inserted into the code which is generated by Fujaba. The user defined code can access objects matched and created in previous Story Pattern executions and it can create objects that can be used in following Story Patterns. More details on Story Diagrams can be found in [FNTZ00].

Figure 1 shows a meta model that reflects a simplified UML Class Diagram and Figure 2 shows an example Story Diagram that describes the *doSomething()* method of the *StoryDiagramTester* class. It operates on instances of the meta model of Figure 2.

Story Activities contain a Story Pattern. A Story Pattern describes a graph transformation rule

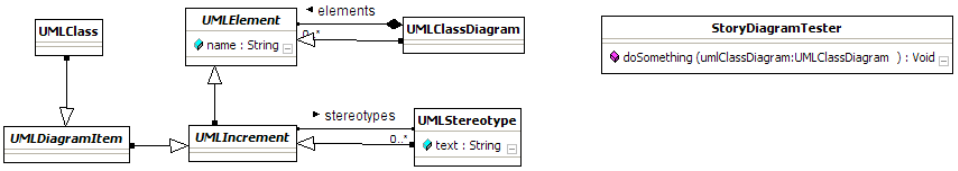


Figure 1: Example meta model of a simplified UML Class Diagram

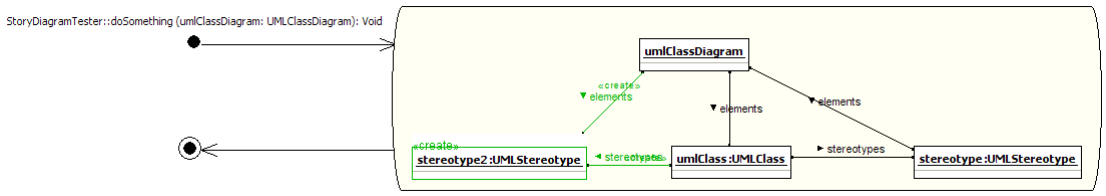


Figure 2: Example Story Diagram

that is executed on the object graph of a running application. Story Patterns can match existing objects, create new objects or delete objects of the running application. For example, the Story Pattern in Figure 2 searches for a *UMLClass* object that is connected to the *umlClassDiagram* via the *elements* link, and to a *UMLStereotype* object (*stereotype*) via a *stereotypes* link. The Story Pattern object *umlClassDiagram* is already bound to the object that was supplied as the method's parameter. The other two Story Pattern objects are unbound. When the Story Pattern is executed, matches for these Story Pattern objects are searched for in the application's object graph. If all Story Pattern objects can be bound to an instance object, a new *UMLStereotype* object is created (indicated by $<<create>>$) and connected to the *umlClass* and the *umlClassDiagram* objects.

3 Story Diagram Interpreter Based on EMF

In this section, we will describe the developed Story Diagram interpreter and briefly describe the meta model of our Story Diagrams, which is based on EMF. The interpreter is implemented as a plug in for the Eclipse framework.

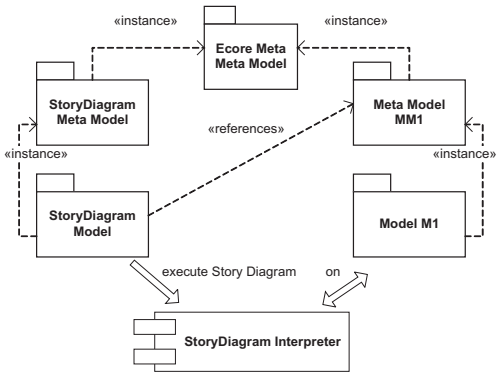


Figure 3: Models used by the Story Diagram interpreter.

EMF provides Ecore as a common meta meta model. All EMF-based meta models are in-

stances of the Ecore meta meta model. This includes the meta model of Story Diagrams. Figure 3 shows these relationships. Story Diagram models are in turn instances of the Story Diagram meta model. Another meta model (MM1 in Figure 3) is required that defines the elements that can be matched and modified by a Story Diagram, e.g., classes, operations and associations. Especially, the definition of the operation is required, whose behavior is modeled by the Story Diagram. Therefore, a Story Diagram model references this meta model. Of course, it is also possible, that a Story Diagram references multiple meta models, including its own meta model.

To execute a Story Diagram, the interpreter needs that Story Diagram, as well as an instance of the meta model that is referenced by the Story Diagram (M1). These are supplied as parameters to the interpreter. During the execution, that model may be modified, depending on the behavior modeled by the Story Diagram. If the operation defined in the meta model (MM1) also has parameters and a return value, these additional parameters can be supplied to the interpreter. The return value is returned when the interpretation is finished.

The use of the common meta meta model Ecore allows to access all EMF-based models in a uniform way. All instance objects provide a generic interface to access their properties and have a reference to their meta class, that provides information about the properties of that object. This allows to work on any EMF-based models without knowing their meta models at design time. Dynamic EMF objects push that concept even further. Usually, code is generated by EMF and objects at runtime are instances of these generated classes. Dynamic EMF objects are not instantiated from specifically generated code classes but from a generic class. Their attributes and associations can only be accessed via the generic interface mentioned above. The Story Diagram interpreter uses only this generic interface to access and modify objects and, therefore, can execute Story Diagrams defined on any EMF-based meta model and can handle normal and dynamic EMF objects.

3.1 Story Diagram Meta Model

Before explaining the interpreter in more detail, we will look at the meta model of Story Diagrams. While Fujaba's meta model of Story Diagrams is intended to be used to generate code, it is unsuitable for interpreting a Story Diagram. This is mainly due to the fact, that statement activities contain plain Java code but Java code cannot be executed directly by our interpreter. We also support OCL for constraints which also requires changes to the meta model. Furthermore, Fujaba uses a proprietary meta meta model that makes integration with other tools difficult. Therefore, we built a new Story Diagram meta model based on EMF.

This meta model is shown in Figure 4. The root node of a diagram, *ActivityDiagram*, contains several *Activities*, each models a method's behavior. Each *Activity* contains several *ActivityNodes* that are connected by *ActivityEdges*. These edges can have guards to conditionally branch the control flow. There are several types of *ActivityNodes* to model the entry and exit points of the method, branches, Story Patterns and imperative calls. This follows the notion of Activity Diagrams of UML 2.0.

InitialNodes, *ActivityFinalNodes*, *DecisionNodes* and *MergeNodes* describe the control flow inside an *Activity*. *CallActionNodes* can be used for imperative calls, *StoryActionNodes* describe Story Patterns.

A *StoryPattern* contains *StoryPatternObjects* that are connected by *StoryPatternLinks*. *StoryPatternObjects* represent an instance object of a meta class. Similarly, *StoryPatternLinks* repre-

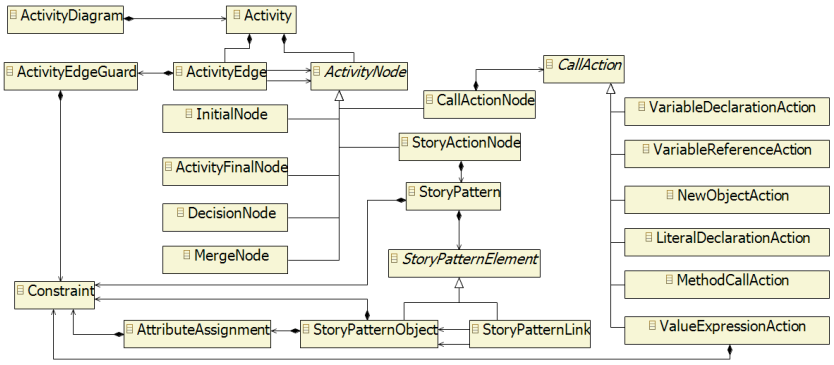


Figure 4: EMF-based Story Diagram meta model

sent instance links of associations. *StoryPatternObjects* can be augmented by *Constraints* and *AttributeAssignments*. *Constraints* define conditions that must be met in order to match that *StoryPatternObject* to an instance object. *AttributeAssignments* assign a new value to an attribute. They are only executed after a valid match for the whole *StoryPattern* could be found. The values of *AttributeAssignments* are calculated by *Constraints*. The *StoryPattern* itself can also have a *Constraint* that is checked when matches for all *StoryPatternElements* could be found. This is useful to specify constraints that include multiple *StoryPatternObjects*. Constraints on *StoryPatternObjects* may not include other elements of the same *StoryPattern* because these other elements might not be bound when the constraint is evaluated.

Constraints are uniformly handled by *Constraint* objects. They contain the constraint expression and the type of the constraint language. Currently, only OCL is supported. Constraints can either evaluate to a Boolean value or an object. The latter case is used for *AttributeAssignments* to compute values.

CallActionNodes try to resemble Fujaba's capability to use arbitrary Java code in statement activities. There are several types of *CallActions*, that can create a new variable and assign a value to it, reference an existing variable, create a new object, define a literal of a primitive type, evaluate an OCL expression and, most importantly, call arbitrary Java methods via Java's reflection mechanism (*MethodCallAction*). This way, user defined code can be integrated into the execution of the Story Diagram.

3.2 Story Diagram Interpreter

Our tool support for modeling Story Diagrams is currently limited to the tree-based editor generated by EMF from the Story Diagram meta model. We are working on a graphical editor using GMF to ease modeling Story Diagrams. Furthermore, we provide a set of basic validation rules using openArchitectureWare's[3] Check language.

The Story Diagram interpreter is also based on Eclipse. It consists of four major parts (cf. meta model in Figure 5): The *StoryDiagramInterpreter*, that manages the interpretation of an activity, the *StoryPatternMatcher*, responsible for executing a single Story Pattern, the *CallActionNodeInterpreter*, responsible for executing call action nodes, and the *InterpreterVariablesManager*, that stores the variables used in the activity along with their instance values. It is also used to evaluate

[3] http://www.openarchitectureware.org

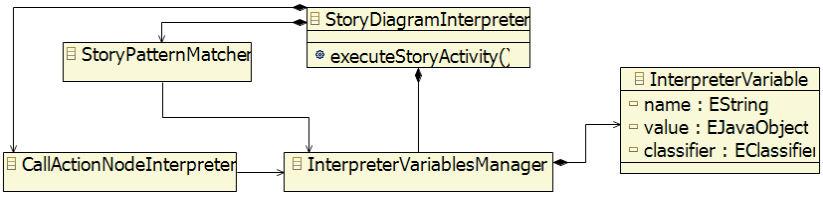


Figure 5: Meta model of the Story Diagram interpreter

OCL constraints using an OCL interpreter.[4]

InterpreterVariables are used to store information about the variables used in a Story Diagram at runtime. They are created for used every variable. These are especially *StoryPatternObjects* but also the parameters of the operation.

To start the interpretation of an activity, the method *executeStoryActivity()* of the *StoryDiagramInterpreter* is called. The parameters of the method are the activity to interpret, a list of values that are used as parameters for the operation modeled by the activity, and the *this* object in whose context the activity will be executed.

The interpreter traverses the activity starting at the *InitialNode*. If a *CallActionNode* or a *StoryActionNode* is encountered, the *CallActionIntepreter* or the *StoryPatternMatcher* are called to execute that node. In case of *DecisionNodes*, constraints on outgoing activity edges are evaluated and the interpreter branches accordingly. If a final node is reached, the execution ends and the return value of the Story Diagram is returned to the caller.

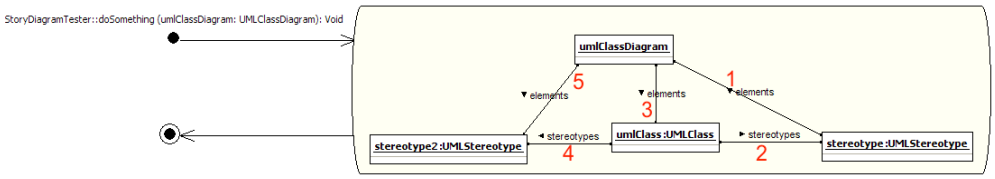


Figure 6: Story Activity of the method *doSomething()*

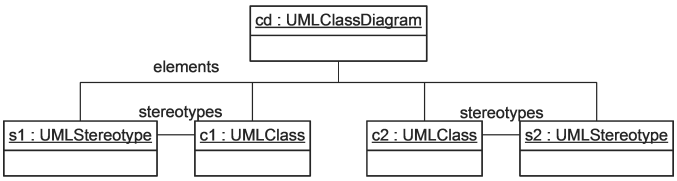


Figure 7: Example instance class diagram

The *StoryPatternMatcher* uses a dynamic pattern matching approach. It tries to find matches for *StoryPatternObjects* using those associations first, that contain the lowest number of elements. Figure 6 shows an example Story Diagram and Figure 7 an instance situation. All references are bidirectional. The instance object *cd* is supplied as a parameter to the activity.

[4] We use the OCL interpreter available at http://www.eclipse.org/modeling/mdt/?project=ocl.

Starting from *umlClassDiagram*, the first story pattern object has to be bound by iterating the *elements* association. Assume, the interpreter matches *stereotype* to *s1*. Now, *umlClass* can be bound by either iterating *elements* a second time, or by following the *stereotypes* link from *s1*. Because the latter contains fewer elements, it is preferred. Also, a match for the last story pattern object *stereotype2* is searched for by following the *stereotypes* link from *c1*. But because the only element *s1* is already bound to another story pattern object, no match can be found. Therefore, the matches for *umlClass* and *stereotype* are discarded and the interpreter tries to find another match for *stereotype*. But this attempt also fails in the example.

To perform this dynamic matching process, the story pattern is analyzed prior execution and *StoryPatternLinks* are grouped into to-one and to-many links. When the interpretation starts, it is checked if a to-one link exists, that starts at a bound *StoryPatternObject* and ends at an unbound one. If such a link exists, it is used to bind the target *StoryPatternObject* of the link. Otherwise, the to-many links are searched. Now, the actual number of elements in the instance association is also checked and the link with the lowest number is followed to bind the next *StoryPatternObject*. After a *StoryPatternObject* was bound, constraints on that object are evaluated and all links are checked, that now have a bound source and target. If these conditions are not met, the match is discarded and another is sought. If they are met, the next link to bind objects is looked up. When all *StoryPatternObjects* could be bound, constraints on the *StoryPattern* are evaluated. If these are fulfilled, *StoryPatternObjects* marked as delete or create are deleted and created, and *AttributeAssignments* are executed.

To keep track of matches, a stack is used. Every time, a story pattern object is bound, an element is put on the stack, that contains lists of all bound and unbound objects, and checked and unchecked to-one and to-many links. If no match can be found for a story pattern object, the top-most stack element is removed and the pattern matching continues using the state of the now top-most stack element. If the stack runs empty, no match could be found for the story pattern.

For debugging purposes, adapters can be registered at the interpreter. Each time, the interpreter performs an action, a notification is send to the adapters. This can be used to print messages to a log or to implement a graphical debugger for Story Diagrams.

The dynamic pattern matching strategy allows to adapt the matching strategy to the instance situation. This is useful, if the optimal matching strategy for a story pattern differs depending on the instance situation. However, the interpreter's matching strategy is not optimal in every case. Cases can be constructed, where traversing a link with many target elements first results in a lower overall execution time. But these cases are rather the exception than the rule.

4 Benefits of the Interpreter

In this section, we outline the benefits of the introduced Story Diagram interpreter. We discuss the improved flexibility in Section 4.1 and the steadier and improved scalability in Section 4.2. On both aspects, we discuss the impact on projects we are currently working on.

4.1 Flexibility

In this section, we discuss the flexibility benefits of the Story Diagram interpreter by means of application areas that we already gained experience from. The main improvements in flexibility are due to the following facts:

- We can improve our workflow because we completely ported SDM to the EMF-based Eclipse platform. Thus, we are able to model and maintain Story Diagrams and further execute them within the same environment.
- We do not need to generate source code from Story Diagrams, which entails the compilation of Story Diagrams and further the integration of the compiled code into the environment for execution.
- We can use other EMF-based tools on Story Diagrams. For example, openArchitectureWare's Check language is used to check well-formedness of Story Diagrams. EMF compare[5] could be applied to compare different versions of Story Diagrams etc.
- We have an explicitly defined meta model of Story Diagrams (Ecore) within Eclipse. This enables to integrate Story Diagrams in the definition of Story Diagrams, which is the prerequisite for higher-order transformations. Fujaba does not allow to reference the Story Diagram meta model within Story Diagrams.
- The support of Dynamic EMF enables to do transformations on meta models without generating code of the mega models. This is most desirable in runtime environments when code generation is not applicable.

Currently, we are working on two projects, where Story Diagrams are frequently used, which are briefly explained in the following. Both projects benefits from the first to facts in the previous listing.

The first project deals with traceability management in an Eclipse-based Model-Driven Engineering (MDE) environment.[6] We have developed a prototypical MDE environment, which is able to model the deployment of software products provided by a company into a model of an IT infrastructure reflecting a customer's IT. Furthermore, the software products, which are modeled in the deployment models, are configured variants of reference models, which contain details of the software product necessary for the deployment domain. Between these models, we have several kinds of relationships tracing certain aspects, which are required to be managed and maintained. The management/maintenance operations for these traceability relationships are expressed by means of Story Diagrams (create and delete operations). Thus, if specific situations in a certain model instantiation exist, there will be Story Diagrams in order to create new relationships between models/model elements and delete existing relationships, which became invalid because of unsatisfied constraints expressed in Story Diagrams.

The first prototype suffered from an uncomfortable workflow we were forced to use. Story Diagrams had to be specified within the CASE tool Fujaba. This required to re-model the meta models of the MDE models in Fujaba in order to specify the Story Diagrams. Further, code for each Story Diagram had to be generated, the code had to be complied and finally integrated into the Eclipse MDE environment. Furthermore, once the MDE environment is deployed to end-users, adding or updating existing Story Diagrams requires an additional mechanism to generate Story Diagram code, compile the code and integrate it into the running MDE environment.

5 http://wiki.eclipse.org/index.php/EMF_Compare

6 This project is funded by CA Labs Inc.

In a subsequent implementation we encountered that the integration of SDM into Eclipse fixed all these issues. We can model Story Diagrams within the same environment, and instantly execute them after specification which safes a lot of time to the user of the environment. Thus, the whole SDM integration brings more flexibility to the user in this project.

In the other project, a model transformation and synchronization system based on Triple Graph Grammars [GH08] (TGG) was developed. The system is also based on Eclipse and EMF. The user specifies a set of declarative TGG rules that describe the model transformation. These rules are translated into Story Diagrams to make them operational. In this step, some operational logic is integrated into the Story Diagrams to support features like incremental transformation and synchronization of the models. Next, Java code is generated from the Story Diagrams. This code is executed by a transformation engine to perform model transformations.

The SDM integration could now improve the usability of the system because it would improve the workflow. After the TGG rules are created by the user and transformed to Story Diagrams, these could be executed instantly without the need to generate code and restart the transformation system. This saves a lot of time when a new set of transformation rules needs to be tested and debugged.

4.2 Scalability

During the development of the TGG-based model transformation system in Eclipse, we discovered that the static matching strategy of the generated code can have a severe impact on the performance of the overall transformation system. The Story Patterns in the Story Diagrams are quite complex and the code generator seldom chooses the optimal matching strategy. Especially in case of large models, this leads to bad scalability of the transformation system. We tried to avoid the problem by splitting complex Story Patterns into simpler ones to *guide* the code generator in choosing the best strategy. However, this does not work in all cases and it increases the complexity of the overall Story Diagrams making debugging and testing of the transformation system more difficult. Therefore, the dynamic matching strategy of the Story Diagram Interpreter would improve the situation. We could use complex Story Patterns (and simpler Story Diagrams) and still be sure to have the best matching strategy in most cases.

To compare the dynamic matching strategy to the fixed matching strategies of compiled code generated by Fujaba, we have conducted a small benchmark[7]. Of course, this is not meant to be an exhaustive performance evaluation. It is only limited to the pattern matching parts. Other performance bottlenecks, like the OCL interpreter, are not considered.

For the benchmark, a simple class diagram model was created conforming to the meta model in Figure 1. In the test models, each *UMLClass* is connected to exactly one *UMLStereotype* object and vice versa, i.e. the number of classes and stereotypes is the same. Figure 7 shows the general scheme. Test models of different sizes ranging from 200 to 100,000 *UMLClass* objects were created, which means a total number of 401 to 200,001 elements. On these test models, the Story Diagram shown in Figure 6 was executed and the time was measured. The test was repeated ten times for each model size and the mean time was calculated. Because the Story Pattern cannot find a match in the instance models, the whole instance models must be traversed.

[7] The benchmarks were run on a PC running on an Intel T5500 Core2 Duo Processor with 1.66 GHz and 2.5 GB RAM under Windows XP SP3. We used Fujaba 5.1 with CodeGen2 5.5 to generate Java code from the Story Diagram. The Story Diagram interpreter runs on Eclipse 3.4.1 and uses EMF 2.4.0. The Java Runtime version is 1.6.

We tested three versions of Java code generated by Fujaba from the Story Diagram, and the interpreter. The interpreter was tested one time using the implementation code generated from the class diagram meta model, the other time using only dynamic objects. This will show the performance penalty when dynamic objects are used.

The code generated by Fujaba uses a fixed matching strategy, that is defined at generation time. The code generator prefers to-one associations to match Story Pattern objects. Surprisingly, the matching order is also influenced by the order in which the links of the Story Pattern are created when the Story Diagram is modeled.

For the example Story Diagram, there are three major categories of matching strategies. The first strategy iterates a single time over the elements association to bind the first Story Pattern object (e.g. *stereotype*). The remaining two Story Pattern objects are bound via the *stereotype* links. This order is depicted in Figure 6 by the numbers. It is the most efficient strategy for the instance models used in the benchmark and is also used by the Story Diagram interpreter. The second strategy iterates the *elements* link two times, the third strategy iterates even three times. Because the iteration over the *elements* association dominates the processing effort, the impact of the model size on the performance can be expected to be much higher than for the first strategy. For each of these categories, we generated code with Fujaba by varying the order in which the Story Pattern links were created.[8]

No. of	Interpreter		Fujaba generated code				
classes	compiled code	dynamic objects	strategy 1	strategy 2	strategy 3	Arithmetic Average	Weighted Average
200	6	12	6	34	4,590	1,543	402
400	12	6	1	121	37,450	12,524	3,182
600	12	12	3	271	126,874	42,383	10,710
800	15	22	6	468	300,781	100,418	25,302
1000	24	25	1	728	587,790	196,173	49,347
2000	59	51	7	2,689	n.a.	n.a.	n.a.
4000	106	110	11	10,703	n.a.	n.a.	n.a.
6000	156	165	11	24,592	n.a.	n.a.	n.a.
8000	236	221	14	43,535	n.a.	n.a.	n.a.
10000	283	276	24	70,393	n.a.	n.a.	n.a.
20000	481	570	28	n.a.	n.a.	n.a.	n.a.
40000	974	1,119	62	n.a.	n.a.	n.a.	n.a.
60000	1,526	1,615	93	n.a.	n.a.	n.a.	n.a.
80000	1,964	2,187	121	n.a.	n.a.	n.a.	n.a.
100000	2,475	2,717	156	n.a.	n.a.	n.a.	n.a.

Table 1: Average execution time of the interpreter and generated code in msec.

The results of the benchmarks are shown in Table 1 and Figure 8. Note the logarithmic scale of the diagram. We also calculated the arithmetic and a weighted average[9] of the Fujaba generated code versions. The weighted average can be seen as the expected value for the execution time of the generated code if the links in the example Story Diagram are created in a random order.

As expected, the performance of the second and third strategies heavily depends on the number of elements in the model. The execution time grows exponentially. The interpreter is generally slower than the first Fujaba code version, but the execution time is still acceptable. The perfor-

[8] An exception is the first strategy. For some reason, Fujaba only generated code that uses the second or third strategies. Therefore, we had to ”force” the code generator by removing two *elements* links from the Story Pattern and inserting the existence check for these links in the generated code manually. This is probably a bug in the code generator.

[9] The weighted average is calculated by giving the first strategy a weight of 0.417, the second a weight of 0.5 and the third strategy a weight of 0.083. These values stem from the theoretical probability that the code generator would choose this matching strategy.

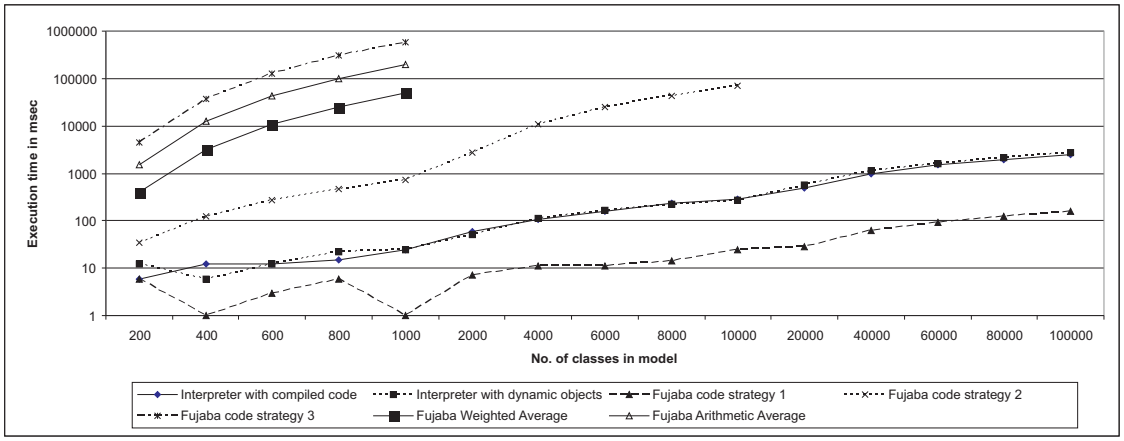


Figure 8: Average execution time of the interpreter and generated code (logarithmic scale)

mance of the interpreter and the first Fujaba generated code strategy depend almost linearly on the model size.

The dynamic pattern matching guarantees a good, and in many cases also optimal, matching strategy. Therefore, the interpreter can make up the performance drawback if the generated code does not use an optimal pattern matching strategy. This is especially useful, if there is no generally optimal matching strategy for a given Story Pattern, but the optimal strategy varies depending on the instance objects. This will definitely be a benefit for the model transformation system mentioned above.

Surprisingly, the use of dynamic objects instead of compiled implementation code does not affect the performance very much. For models up to 10,000 classes the difference to using compiled implementation code for the model elements is not even significant. So the additional flexibility of dynamic objects does almost not come at the expense of performance.

5 Conclusion and Future Work

In this paper, we presented an interpreter for Story Diagrams based on EMF models and Eclipse. The whole SDM implementation improved the flexibility in our research projects because of an improved workflow, the lapse of generating Java code and applying the interpreter for executing Story Diagrams. It further enables the application of EMF-based tools for further validation purposes, as well as higher-order transformations.

Furthermore, the interpreter uses a dynamic matching strategy, which makes the performance of the interpreter scale more steadily. Although the interpreter is generally slower than compiled code, it can be faster in cases where the static matching strategy of compiled code is not optimal. In future work, a dynamic matching strategy may be incorporated into generated code to combine the advantages of both approaches.

Moreover, we want to enhance the usability by improving the visual representation of Story Diagrams using GMF diagrams. We also want to complement concepts from the Story Diagrams in Fujaba that are currently not supported by the Story Diagram interpreter, e.g., object sets and path expressions. Additionally, we plan to improve the control flow in Story Diagrams by supporting the concepts fork and join to model parallelism in Story Diagrams.

Bibliography

[BNBK06] D. Balasubramanian, A. Narayanan, C. van Buskirk, G. Karsai. The Graph Rewriting and Transformation Language: GReAT. *Electronic Communications of the EASST* 1, 2006.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *TAGT'98: Selected papers from the 6th International Workshop on Theory and Application of Graph Transformations*. Pp. 296–309. Springer-Verlag, London, UK, 2000.

[GH08] H. Giese, S. Hildebrandt. Incremental Model Synchronization for Multiple Updates. In *Proceedings of GraMoT"08, May 12, 2008, Leipzig, Germany*. 2008.

[GMW06] H. Giese, M. Meyer, R. Wagner. A Prototype for Guideline Checking and Model Transformation in Matlab/Simulink. In *Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany*. Technical Report tr-ri-06-275, pp. 56–60. University of Paderborn, 2006.

[GW09] H. Giese, R. Wagner. From model transformation to incremental bidirectional model synchronization. *Software and Systems Modeling*, 1 2009.

[MCG05] T. Mens, K. Czarnecki, P. V. Gorp. 04101 Discussion – A Taxonomy of Model Transformations. In Bezivin and Heckel (eds.), *Language Engineering for Model-Driven Software Development*. Dagstuhl Seminar Proceedings 04101. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany, 2005.

[NSW+02] J. Niere, W. Schäfer, J. P. Wadsack, L. Wendehals, J. Welsh. Towards pattern-based design recovery. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*. Pp. 338–348. ACM, New York, NY, USA, 2002.

[REEK99] G. Rozenberg, H. Ehrig, G. Engels, H.-J. Kreowski. *HANDBOOK of GRAPH GRAMMARS and COMPUTING by GRAPH TRANSFORMATION Volume 2: Applications, Languages and Tools*. World Scientific, 1999.

[TÖ00] G. Täntzer. AGG: A Tool Enviroment for Algebraic Graph Transformation. In *Proc. of Applications of Graph Transformation with Industrial Relevance (AGTIVE2000), Kerkrade, The Netherlands*. Lecture Notes in Computer Science (LNCS). Springer Verlag, 2000.

[TBB+08] G. Taentzer, E. Biermann, D. Bisztray, B. Bohnet, I. Boneva, A. Boronat, L. Geiger, R. Geiß, A. Horvath, O. Kniemeyer, T. Mens, B. Ness, D. Plump, T. Vajk. Generation of Sierpinski Triangles: A Case Study for Graph Transformation Tools. Pp. 514–539, 2008.

[VSV05] G. Varro, A. Schurr, D. Varro. Benchmarking for Graph Transformation. In *VLHCC '05: Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 79–88. IEEE Computer Society, Washington, DC, USA, 2005.

[WGN03] R. Wagner, H. Giese, U. Nickel. A Plug-In for Flexible and Incremental Consistency Management. In *Proc. of the International Conference on the Unified Modeling Language 2003 (Workshop 7: Consistency Problems in UML-based Software Development), San Francisco, USA*. Technical Report. Blekinge Institute of Technology, San Francisco, 2003.

Gene Expression with General Purpose Graph Rewriting Systems

Jochen Schimmel, Tom Gelhausen and Christoph Schaefer

Institute for Program Structures and Data Organization, University of Karlsruhe

Abstract: We show that a general purpose graph rewriting system (GRS) nowadays is capable of simulating gene expression, the intra-cellular synthesis of proteins. The model organism we use is one of the best-studied prokaryotic life-forms in genetics, the E. coli bacteria. Our graph representation of the E. coli DNA consists of 23 million graph elements. In our case study we correctly synthesize the proteins of 30 consecutive genes. In this paper we describe our approach as well as our observations. Further on, we discuss some potential extensions to GRSs that would support more sophisticated simulations.

Keywords: Graph Rewriting, Biology, Genetics, Gene Expression

1 Introduction

Current life science research aims at understanding the life cycle process of a human cell[1]. Today, highly optimized string-based approaches exist to compute gene expression products. Since more sophisticated simulations considering chemical behavior can effectively not be done with string-based representations, several authors propose graph-based techniques. The recently increased performance of graph rewriting systems suggests the applicability of this alternative.

In this paper, we present a case study in which we investigated the simulation of gene expression using the general purpose graph rewriting system GrGen.NET [GBG+06]. Simulating the human genome is beyond the capabilities of graph rewriting on current PCs. Therefore, our subject was the (considerably smaller) DNA of E. coli – the simulation steps are similar. In comparison with the natural archetype, our simulation has several simplifications. In fact, our simulation does not outperform the string-based approaches in terms of 'functionality'. However, the intent of this case study was to investigate whether and how such simulations are feasible using graph rewriting techniques. The results show that the graph-based approach is appropriate and performs sufficiently well.

2 Gene Expression Basics

All life forms are based on a large macro molecule called *deoxyribonucleic acid* (DNA). Gene expression is the biological process of creating proteins using the information stored in the DNA as a blueprint. The DNA itself is a concatenation of molecules called *nucleotides*. Four different types of nucleotides occur in the DNA: *Adenin* (A), *Cytosin* (C), *Guanin* (G) and *Thymin* (T). DNA is double stranded: The nucleotides of two strands are kept together by a chemical binding.

[1] For instance, the possibility to entirely simulate the genetic processes within a human cell is expected with the computing power of one ExaFLOP. Intel expects computers with this processing power around the year 2017 [Gel08].

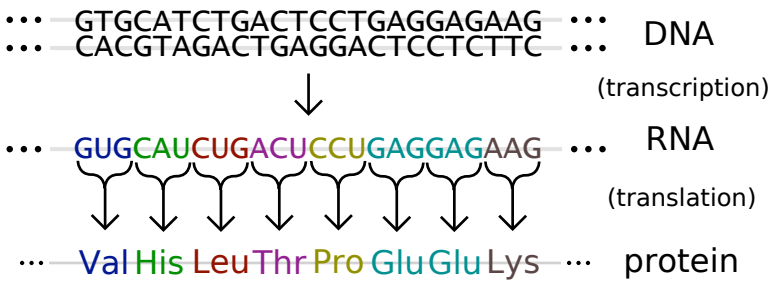


Figure 1: Gene expression depicted[5].

In opposing strands, Adenin binds to Thymin, while Cytosin binds to Guanin. Therefore, the nucleic sequence of one strand can easily be deduced from the other.

Another molecule that is involved in gene expression is the *ribonucleic acid* (RNA). It has the same structure as DNA, except that Thymin is replaced by the nucleotide *Uracil* (U). RNA is (usually) single-stranded.

A protein is built from molecules called *amino acids*. Twenty different types of amino acids exist, which can form to large strings. Such an amino acid sequence is called a protein's *primary structure*. For the protein to be fully functional, the three-dimensional layout of the amino acid chain matters. The three-dimensional layout of a protein is called its *secondary* (and *tertiary*) *structure*. In this study, we only simulate the primary structure[2].

To store the primary structure, each amino acid is encoded using three nucleotides (so called *triplets* of A, C, G, or T). Three consecutive nucleotides enable 64 different combinations, whereas only 20 amino acids exist: Several triplets represent the same amino acid, while others are reserved as special 'commands'[3]. In fact, the additional combinations are chosen in such a way that a) a possible mutation (i. e. alteration of a nucleotide) is likely to represent the original amino acid, or b) the mutation is at least likely to represent an amino acid with a similar chemical behavior, preserving the functionality of the encoded protein. A DNA subsequence that encodes a protein is called a *gene*[4].

The process of actually creating a protein from a gene is called *gene expression* and is done in two steps, *transcription* and *translation*, as depicted in Figure 1. Both will be digested in the following sections.

2.1 Transcription

In order to create a protein from a gene, the gene's code has to be copied. This is performed by an enzyme called *polymerase*. The polymerase attaches to the start of the gene and slides along the

[2] The process of bringing proteins into their correct three-dimensional order is called *folding*. It is not yet thoroughly understood how this process works. A well-known research project in this area is Folding@home (http://folding.stanford.edu/).

[3] Computer scientists would call them 'escape sequences'.

[4] Not all parts of the DNA are genes.

[5] Image from Wikipedia, http://en.wikipedia.org/w/index.php?title=Gene_expression&oldid=258825410.

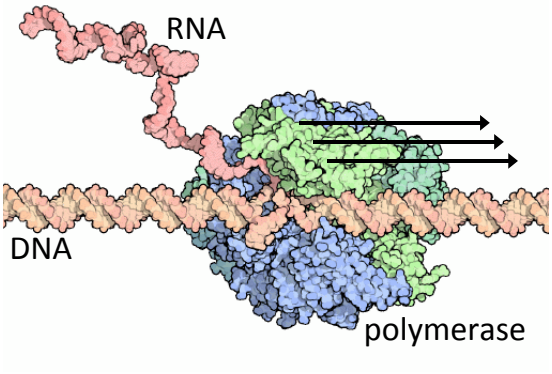


Figure 2: A polymerase, sliding along a DNA molecule and synthesizing RNA[6].

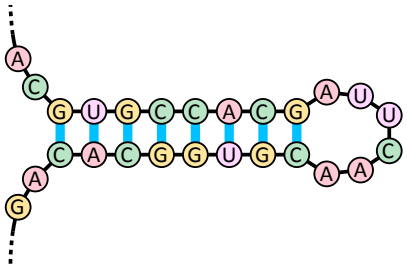


Figure 3: A hairpin structure with a stem-loop[7].

DNA molecule towards the gene's end. In this process, the polymerase creates an RNA molecule that contains the same information as the gene (see Figure 2).

To enable a polymerase to find the exact starting location of a gene, each gene is preceded by a sequence called *promoter*. The promoter sequence has the chemical ability to attract polymerase enzymes and bind them to the DNA. In order to control which genes are transcribed in a cell, a multitude of proteins can attach to promoters: Some proteins enhance the binding behavior of a promoter while others disable a promoter. (However, such proteins are not part of our simulation.) The sequences of different promoters have a common structure, the most obvious one is a region called *TATA-Box*, as its sequence is Thymin-Adenin-Thymin-Adenin. Unfortunately, each promoter differs slightly. As we will show later, this fact is important for simulation with graph rewriting.

One option to terminate the transcription process is the *rho-independent termination*. Here, the last copied nucleotides have an 'inverted' palindromic order. Therefore, the corresponding nucleotides can form a chemical binding which leads to a *hairpin structure*. These nucleotides stick together to form a stem-loop as shown in Figure 3. This stem-loop can slide into the polymerase enzyme and detach it from the DNA. The newly created RNA molecule is then set free.

[6] Image from Wikipedia, http://de.wikipedia.org/w/index.php?title=RNA-Polymerase&oldid=52619006.
[7] Image from Wikipedia, http://de.wikipedia.org/w/index.php?title=Haarnadelstruktur&oldid=39180278.

2.2 Translation

The RNA molecule produced in the preceding transcription step serves as a blueprint for the protein. Like a polymerase attaches to DNA, a *ribosome* enzyme attaches to RNA during translation. As soon as the ribosome detects the triplet 'AUG', it starts to concatenate amino acids according to the triplet pattern found on the RNA strand (c. f. Section 2). The mapping from triplets to amino acids is shown in Figure 4. The translation process stops as soon as the ribosome detects one of the three stop-sequences: 'UAG', 'UAA', or 'UGA'. Together with the start sequence AUG, these are all of the 'special commands' mentioned above.

Ala	Arg	Asn	Asp	Cys	Gln	Glu	Gly	His	Ile	Leu	Lys	Met	Phe	Pro	Ser	Thr	Trp	Tyr	Val
GCA	CGA	AAC	GAC	UGC	CAA	GAA	GGA	CAC	AUA	CUA	AAA	AUG	UUC	CCA	UCA	ACA	UGG	UAC	GUA
GCC	CGC	AAU	GAU	UGU	CAG	GAG	GGC	CAU	AUC	CUC	AAG		UUU	CCC	UCC	ACC		UAU	GUC
GCG	CGG						GGG		AUU	CUG				CCG	UCG	ACG			GUG
GCU	CGU						GGU			CUU				CCU	UCU	ACU			GUU
	AGA									UUA					AGC				
	AGG									UUG					AGU				

Figure 4: Mapping between amino acids and nucleotide triplets [Lew08]. Each triplet is mapped to the amino acid in the first cell of the corresponding column.

3 Implementation

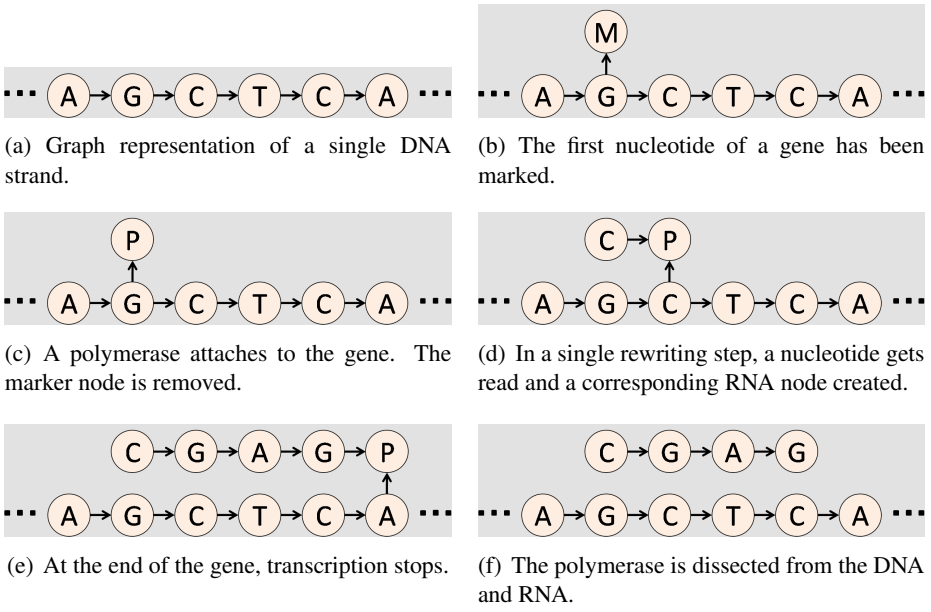
In this section we describe our model of the gene expression process using the GrGen.NET graph rewriting system [GBG+06]. This section can also serve as a guide for implementations on other graph rewriting systems. Even though our simulation considerably simplifies the biological processes, it still produces the proteins found in the natural archetype.

In our graph model, nodes represent molecules or sets of molecules like enzymes. We use the hierarchical node type system of GrGen.NET to express similarities between nodes. For example, Adenin, Guanin, Cytosin, Thymin, and Uracil are represented by node types that extend the abstract node type `Nucleotide`.

Edges represent chemical bindings between molecules or the attachment of enzymes. In our simulation, we simplify complex chemical reactions by introducing special nodes that mark locations of pending reactions. For example, we connect promoters found during our simulation with a marker node in order to abstract and simplify the process of polymerase binding. Furthermore, these promoter marker nodes contain a field that stores the associated gene's length.

Graph rewriting rules represent chemical reactions (processing uses the SPO-approach). The rewriting rules as well as the model can be obtained from http://www.grgen.net/gxb/. The rules we use in our simulation fall in three categories: promoter-identification, RNA-synthesis, and protein-synthesis.

Promoter-Identification. We created a rule for each promoter: These rules add marker nodes at each occurrence of the promoter they search for. A rule matches a promoter's nucleic sequence against the DNA and places a marker node at the first nucleotide of the associated gene. As the



(a) Graph representation of a single DNA strand.

(b) The first nucleotide of a gene has been marked.

(c) A polymerase attaches to the gene. The marker node is removed.

(d) In a single rewriting step, a nucleotide gets read and a corresponding RNA node created.

(e) At the end of the gene, transcription stops.

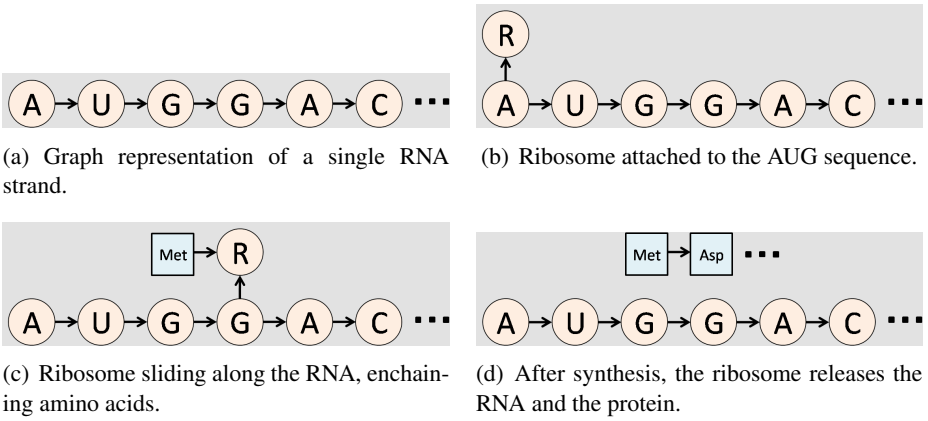
(f) The polymerase is dissected from the DNA and RNA.

Figure 5: Simulation steps for transcription.

promoter sequences are rather long, we developed a tool to generate these graph rewrite rules. The tool can be used to prepare promoter rules for arbitrary genes of arbitrary species.

RNA-Synthesis (Transcription). After inserting the marker nodes, the RNA synthesis process starts and attaches a free polymerase enzyme (i. e. an isolated polymerase node) at a marked nucleotide. Furthermore, the polymerase copies and stores the gene's length from the marker node for the termination of the transcription process. Further rewriting rules move the attached polymerase node along the DNA strand while they create enchained nucleotide nodes that represent the synthesized RNA strand. Each transcription step increments an internal counter of the polymerase node. The transcription process stops as soon as this counter equals the gene's length. The polymerase node detaches from the DNA and the newly created RNA strand. Afterwards, the polymerase node may initiate further transcription processes. This implies, that the total number of available polymerase nodes determines the maximal number of genes that can be transcribed concurrently in our simulation. It is also possible, that several polymerase nodes transcribe a single gene at the same time. Multiple transcription corresponds perfectly to the biological process. The RNA-synthesis is depicted in Figure 5. Listing 1 shows the GrGen.NET Code to detach the polymerase from a DNA strand.

Protein-Synthesis (Translation). The protein-synthesis starts with the application of a rule that accepts as input an RNA strand and a ribosome node that does not yet participate in a translation process. It attaches the ribosome node at the starting sequence (AUG) of an RNA molecule. RNA processing rules match an attached ribosome together with the next three nodes of the RNA strand. They emit and enchain the corresponding amino acids. These rules are



(a) Graph representation of a single RNA strand.

(b) Ribosome attached to the AUG sequence.

(c) Ribosome sliding along the RNA, enchaining amino acids.

(d) After synthesis, the ribosome releases the RNA and the protein.

Figure 6: Simulation steps for translation.

sensitive for the stop sequences 'UAG', 'UAA', and 'UGA'. If an RNA processing rule finds a stop sequence, the process terminates and the ribosome node gets disconnected from the RNA strand and the amino acid chain (i. e. the protein). The protein-synthesis is depicted in Figure 6. A sample screenshot of a simulation result can be found in Figure 7.

3.1 The Sample Organism

As sample data we used the genome of Escherichia Coli (E. coli), a bacterium that is well understood and frequently used as model organism. The genome of E. coli has a length of 4.6 million base pairs. As each base pair is modeled as two nodes that are connected by an edge, the graph representation takes 13.8 million graph objects. The interconnections of the two strands add two more edges per base pair, summing up to a total of 23 million graph objects. Other life forms may have much larger genomes. Table 1 lists some genome lengths of other organisms. The E. coli genome is available at [KRS+02].

To identify genes on the E. coli genome, we used 30 consecutive promoter sequences. The promoter sequences have been extracted using the gene data available at [KRS+02]. They have a length of 50 to 200 nucleotides.

4 Runtime Observations

For a long time, graph rewriting systems have not been an option to simulate complex biological processes, though the benefits of the formal representation of biochemical reactions are widely accepted. Performance is one of the main reasons for the lack of simulation implementations. In this section we present our experiences with GrGen.NET.

As described in Section 3, promoter search is one important part of the gene expression simulation. Our promoter search rules seek large nucleotide node sequences in the graph representation of a DNA molecule. It is obvious, that string based implementations can perform such search tasks faster. A simple string implementation using regular expression takes 0.3 seconds on a

Table 1: Genomes of several viruses, bacteria and organisms.

Name	Basepairs (bp)	Genes
Simian virus 40	5243	6
M13 phage	6407	10
Lambda phage	48502	ca. 50
Helicobacter pylori	1667867	1590
Mycobacterium tuberculosis	4411529	3924
Escherichia coli	4639211	4288
Yeast	12 million	6240
Pinworm	97 million	18240
Fly	180 million	13600
Mouse	3000 million	25000
Human	3000 million	25000
Corn	2400 million	30 - 40000
Rice	440 million	30 - 40000

common PC. GrGen.NET took about 8 seconds on the same PC. The execution time shrinks to 3.5 seconds on an 8-core PC using a preliminary multi threaded implementation of GrGen.NET. Although being slower, this is still fast enough to use graph rewriting approaches.

The generated graph rewriting rules for promoter search are rather large. GrGen.NET translates each rewriting rule into C# source code. The generated C# file for all 30 promoter patterns has a size of about 700 megabytes. This is unwieldy, even for generated code. A solution to this problem could be a generation mode that optimizes code size.

A problem that still persists is memory consumption. Nodes as well as edges are represented as .NET objects. A node requires 28 bytes, while an edge requires 44 bytes. Therefore, the graph representation of the E. coli genome requires 824 megabytes. This is acceptable on current workstation PCs. For a similar graph representation of the human genome 525 gigabytes of memory would be required[8]. As the current representation is very coarse grained the memory consumption for advanced genetic applications will be even higher.

5 Improvements of the Simulation

Even though our simulation produces the correct proteins from the real genes, our simulation relies on several simplifications. In this section, we discuss these simplifications in an attempt to give an outline for the next steps required to reach a more realistic simulation.

A major simplification is the omission of spacial relations. In a real environment, an unbound enzyme and the availability of reactants is not sufficient for most reactions to happen: All reactants need to be physically close to each other and in a certain angle for the reaction to happen. Unfortunately, graph elements do not provide a notion of locality. Locality could be simulated by connecting the nodes of our graph to a three-dimensional grid of nodes representing posi-

[8] Current workstation PCs have about 8 gigabytes of main memory.

tions in an euclidean space. The resulting effort would be immense. However, if the GRS itself had a notion of spatial positioning, internal data representation could possibly benefit from optimizing data structures such as octrees [YKFT84]. An octree is a tree data structure that stores hierarchical three-dimensional data in a compact way.

Another simplification concerns timely relations. No known graph rewriting system implements time constraints, although theoretical approaches exist [GHV02]. The synthesis of an RNA strand in E. coli proceeds with 50 to 100 polymerisation steps per second. Molecules located on the DNA may slow down the process. The speed of current graph rewriting systems depends on the processing speed of software and hardware. For a realtime simulation graph rewriting rules have to be extended by timely relations.

During implementation of the study, we realized that not all details of the gene expression process can be simulated using GrGen.NET. As stated in Section 3, a graph rewriting system needs to know the specific sequence of a promoter in order to attach a polymerase to the corresponding gene. Unfortunately, in reality, this pattern is only imprecisely known and its nucleic sequence differs slightly from instance to instance. Current graph rewriting systems are not able to detect such fuzzy patterns. A fuzzy matching mode would be a helpful extension. For instance, if a search pattern consists of fifty nodes and a match candidate has 48 correctly identified nodes, it could be considered as a valid match.

The inability to fuzzy-match also prevents a more realistic termination of the RNA synthesis: As stated in Section 2, a stem-loop in the newly created RNA-strand initiates the rho-independent transcription termination. The rho-independent transcription termination requires a palindromic sequence. Current graph rewriting systems are in fact able to detect *perfect* palindromic sequences, but nature is a little more tolerant in this concern.

To circumvent the following simplification, additional genetic and software engineering research is required. Our simulation only uses known promoters, so only known genes can be found. In real cells, polymerases react with promoters because of the chemical properties of the promoters' nucleic sequences. Simulating such chemical reactions requires a DNA representation at molecular level and an accurate description (in terms of graph rewriting rules) of organic chemistry. This requires in-depth knowledge of chemistry and is subject to further studies. In [YKS04] a first approach is presented. Nevertheless, rewriting rules could be able to identify new promoters during runtime, as the sequence of a promoter follows a certain pattern.

6 Related Work

Simulating gene expression using graph rewriting systems has already been proposed by McCaskill and Niemann [MN01]. While in their work a graph rewriting system has been created solely for the purpose of genetic simulation, we use a standard multipurpose graph rewriting system. For example, McCaskill and Niemann implemented a specialized mechanism to differentiate between molecules of different types, whereas we can use typed attributed graphs. We see the future of genetic graph rewriting in multi purpose systems, as we can easily benefit from further improvements in graph rewriting technology. Yadav, Kelley, and Silverman propose to simulate chemical reactions at the molecular level [YKS04], too. They provide hints on extending graph rewriting systems for that purpose but their work lacks an implementation. Rosselló

and Valiente propose the computation of signal transduction in cells using graph transformation [RV04]. One can sum up approaches like these to the goal of simulating the inner workings of biological cells as a whole. Rosselló and Valiente give an overview on more approaches to simulate different parts of these fundamentals in [RV05]. We propose to sum up all these approaches into an integrated simulation based on graph rewriting systems. Currently available tools focus on static computation or data storage rather than simulation. Gene expression products, as they are generated in the application presented in this paper, are stored in the ecocyc database [KRS+02]. However no tool tries to simulate the genetic processes as a whole. We propose that graph rewriting may lead to such a software at some point in the future.

7 Conclusion

Even though the processing of string representations is considerably faster, we see the future of genetic simulations in graph representation and graph rewriting. String representations are inflexible in handling multidimensional structures: A polymerase binding simulation at a detailed level involves hundreds of RNA and protein molecules and needs to consider their three dimensional layout and position. Graphs can easily represent multi-dimensional relations containing large numbers of objects.

All genetics simulation approaches aim at simulating the whole life cycle process of a cell at some point. As the size of the human genome does not grow, we are confronted with a fixed problem size. Therefore, we may concentrate on simulation approaches offering a good abstraction level such as graph rewriting. The loss of performance will be overcome by better computing devices in the future. The trend towards parallel computing systems can be used to advantage, as genetic processes offer ample parallelism by nature.

Listing 1: Rule to detach a polymerase from DNA

```
1  rule RNA_Pol_Finish {
2  pattern {
3    pol:Polymerase -cur_conn:PolymeraseToDna->cur_nuc:Nucleotide
4      -:DNA_Backbone-> next_nuc:Nucleotide;
5    pol -rna_conn:PolymeraseToRna->cur_rna_nuc:Nucleotide;
6    if { pol.MaxNucleotide > 0 &&
7      pol.CurrentNucleotide == pol.MaxNucleotide;
8      }
9    }
10 modify {
11   delete(cur_conn); delete(rna_conn);
12   eval { pol.MaxNucleotide = 0; pol.CurrentNucleotide = 0; }
13   }
14 }
```

Table 2: Simulation summary.

Sample organism	Escherichia coli
Total number of DNA base pairs	4,639,211
Total number of DNA base pairs in simulation	4,639,211
Total number of promoters/genes	4,288
Average size of promoters	50-200 nucleotides
Genes used in benchmark	30
Graph rewriting system	GrGen.NET (version 1.4)
Max. number of nodes in simulation	9.3 million
Max. number of edges in simulation	13.9 million
Number of rules	69
Total size of rewriting-rules source code	700 megabytes
Total size of rewriting-rules object code (MSIL-DLL)	13 megabytes
Memory consumption of graph elements	824 megabytes
Runtime (only promoter search), string-based implementation	0.3 seconds
Runtime (only promoter search), GrGen.NET	7.7 seconds
Runtime (only promoter search), parallel GrGen.NET	3.5 seconds
Total runtime of the simulation, GrGen.NET	7.9 seconds

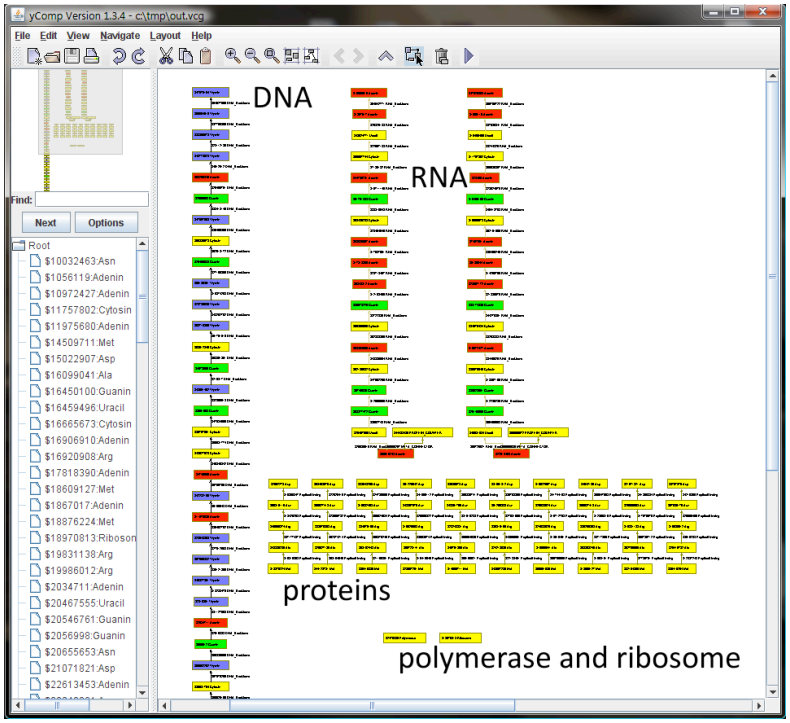


Figure 7: A yComp-visualization of the result graph of our simulation process.

Acknowledgements: We thank Alexander Knoll from the Institute for Botany II, University of Karlsruhe, for guiding us through the bio-molecular jungle.

Bibliography

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. M. Szalkowski. GrGen: A Fast SPO-based Graph Rewriting Tool. In Corradini et al. (eds.), *Graph Transformations - ICGT 2006*. Lecture Notes in Computer Science, pp. 383–397. Springer, September 2006.
http://www.info.uni-karlsruhe.de/papers/grgen_icgt2006.pdf

[Gel08] P. Gelsinger. Keynote at the Intel Developer Forum (IDF), Shanghai 2008. Transcript available at http://download.intel.com/pressroom/kits/events/idfspr_2008/PatGelsinger_keynote_transcript.pdf, Apr. 2008.

[GHV02] S. Gyapay, R. Heckel, D. Varr. Graph transformation with time: Causality and logical clocks. In *Proc. 1st Int. Conference on Graph Transformation (ICGT 02*. Pp. 120–134. Springer-Verlag, 2002.

[KRS+02] P. D. Karp, M. Riley, M. Saier, I. T. Paulsen, J. Collado-Vides, S. M. Paley, A. Pellegrini-Toole, C. Bonavides, S. Gama-Castro. The EcoCyc Database. Volume 30(1), pp. 56–58. January 2002.
http://dx.doi.org/10.1093/nar/30.1.56

[Lew08] B. Lewin. *Genes IX*. 2008.
http://www.biotechnews.com.au/index.php/id;84859370;fp;4194304;fpid;1

[MN01] J. S. McCaskill, U. Niemann. Graph Replacement Chemistry for DNA Processing. In *DNA '00: Revised Papers from the 6th International Workshop on DNA-Based Computers*. Pp. 103–116. Springer-Verlag, London, UK, 2001.

[RV04] F. Rosselló, G. Valiente. Analysis of Metabolic Pathways by Graph Transformation. In *ICGT*. Pp. 70–82. 2004.

[RV05] F. Rosselló, G. Valiente. *Graph Transformation in Molecular Biology*. 2005.
http://www.springerlink.com/content/93ch5yly36nc111w

[YKFT84] K. Yamaguchi, T. Kunii, K. Fujimura, H. Toriya. Octree-Related Data Structures and Algorithms. Jan. 1984.
doi:10.1109/MCG.1984.275901

[YKS04] M. K. Yadav, B. P. Kelley, S. M. Silverman. The Potential of a Chemical Graph Transformation System. In *ICGT*. Pp. 83–95. 2004.

Euler Diagram Transformations

Andrew Fish[1]

[1] andrew.fish@brighton.ac.uk, http://www.brighton.ac.uk/cmis/contact/details.php?uid=agf
School of Computing, Mathematical and Information Sciences
University of Brighton, UK

Abstract: Euler diagrams are a visual language which are used for purposes such as the presentation of set-based data or as the basis of visual logical languages which can be utilised for software specification and reasoning. Such Euler diagram reasoning systems tend to be defined at an abstract level, and the concrete level is simply a visualisation of an abstract model, thereby capturing some subset of the usual boolean logic. The visualisation process tends to be divorced from the data transformation process thereby affecting the user's mental map and reducing the effectiveness of the diagrammatic notation. Furthermore, geometric and topological constraints, called wellformedness conditions, are often placed on the concrete diagrams to try to reduce human comprehension errors, and the effects of these conditions are not modelled in these systems.

We view Euler diagrams as a type of graph, where the faces that are present are the key features that convey information and we provide transformations at the dual graph level that correspond to transformations of Euler diagrams, both in terms of editing moves and logical reasoning moves. This gives a correspondence between manipulations of diagrams at an abstract level (such as logical reasoning steps, or simply an update of information) and the manipulation at a concrete level. Thus we facilitate the presentation of diagram changes in a manner that preserves the mental map. It also provides the ability to realise reasoning systems at the concrete level, thereby providing us with diagrammatic reasoning systems that are inherently different from symbolic logics due to natural geometric constraints. We provide a particular concrete transformation system which preserves the important criteria of planarity and connectivity, which may form part of a framework encompassing multiple concrete systems each adhering to different sets of wellformedness conditions.

Keywords: Euler Diagrams, Graph Transformations, Logical Reasoning

1 Introduction

Euler diagrams are used as the underlying structures in many application areas for the representation of set-based information, such as: non-hierarchical directories [DES03, DF07], complex genetic set relations [KMGB05], ontologies in semantic web applications [HES+05], statistical data [CR03], and as the basis of logical specification and reasoning systems which can be used in software systems development (e.g. Constraint diagrams [FFH05, Ken97]). Logical reasoning systems based on Euler diagrams (e.g. Spider diagrams [HST05]) tend to apply to the abstract level, with the concrete level simply being a visualisation of the abstract level where one exists.

This avoids the complexities of concrete level reasoning, but accordingly denies us the development of a truly diagrammatic reasoning system affected by geometric and topological constraints rather than a visualisation of some natural subset of boolean logical. If we are to define transformation rules at the concrete level then they should *lift* to the abstract level (in the sense that they can be viewed as an instantiation of an application of the abstract rules to their abstract models, but the constraints imposed at the concrete level can restrict the application of the rules). The development of broader transformation systems with more generic manipulations than logical reasoning rules will enable wider applicability of the systems.

We present a brief outline of the methodology used upfront, using Figure 1, for reference purposes; details and explanations of terminology will follow. The left hand part of Figure 1 depicts the generation process for wellformed Euler diagrams (see [FFH08] for details) which involves constructing an abstract labelled graph (called the superdual in [FFH08], or a closeness graph in [Cho]) from an abstract diagram d_1, finding a planar spanning subgraph that satisfies connectivity conditions and embedding it in the plane so that it satisfies certain face conditions, yielding the "dual" of an appropriate concrete diagram. Varying the wellformedness conditions that are imposed on the system affects the conditions imposed on the graphs but the same general approach can be taken (this relaxation of some conditions was performed in [Cho]).

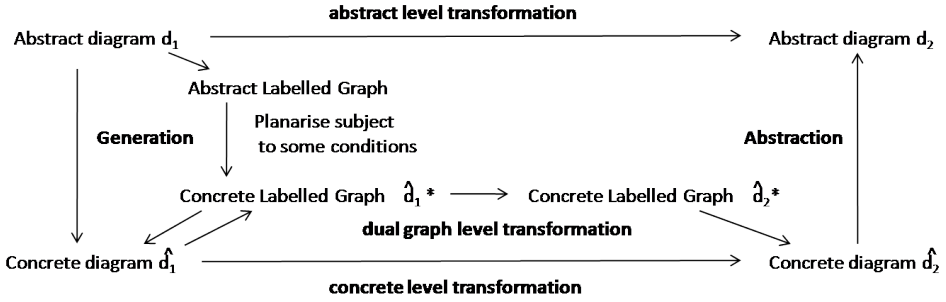


Figure 1: An overview of the generation process and the transformation systems involved.

Transformations of diagrams at the abstract level can only alter the abstract sets of zones and contours, and these are straightforward to describe (typically they would be add or remove abstract contours or zones). However, transformations of diagrams at the concrete level would involve geometric transformations which can be hard to describe (see Section 3 for an example). Also, one must ensure that the transformations do not cause the violation of the wellformedness conditions that are imposed on the system. Therefore, we define a transformation system on the labelled dual graphs of the concrete diagrams[1]. So, as depicted in Figure 1, given an abstract diagram d_1 and a generated concrete diagram $\hat{d}_1$ with labelled dual graph $\hat{d}_1{}^*$, we apply transformations which take $\hat{d}_1{}^*$ and return $\hat{d}_2{}^*$, which is the dual of concrete diagram $\hat{d}_2$. We show that these transformations lift to the abstract level transformations in the sense that the abstraction of the concrete diagram $\hat{d}_2$ is the abstract diagram d_2 obtainable from the abstract level transformation.

In Section 2 we recall background definitions of Euler diagrams at both concrete and abstract

[1] Note that we perform operations on the embedded labelled dual graphs because we want to preserve the mental map, but a similar approach can be taken at an abstract dual graph level; this may provide a system in which rules are more often applicable, but will lose some of the geometric information contained in the concrete dual graphs.

levels, the notion of wellformedness conditions imposed on concrete diagrams, and develop further the notions of the graph of an Euler diagram and its labelled dual graph. We develop transformation systems at the abstract diagram level and then at the concrete dual graph level (corresponding to concrete diagram transformations that lift to abstract diagram transformations), in Section 3; note that some proofs have been sketched or omitted for space reasons. Conclusions and future work plans are discussed in Section 4.

2 Euler Diagrams and Graphs

First of all we recall the definitions of Euler diagrams, separating the abstract and concrete models as usual, and then we provide a set of wellformedness conditions that are often imposed with the intention of reducing human comprehension errors; Definitions 1 and 2 are adapted from those in [FFH08].

Definition 1 An *abstract Euler diagram* is a pair: $d = \langle C(d), Z(d) \rangle$ where: $C(d)$ is a finite set whose members are called *(abstract) contours*, $Z(d) \subseteq \mathscr{P}C(d)$ is the set of *(abstract) zones* of d, where $\mathscr{P}X$ denotes the powerset of set X, and $\bigcup_{z \in Z(d)} z = C(d)$. If $Z(d) \neq \mathscr{P}C(d)$ then the elements of $\mathscr{P}C(d) - Z(d)$ are called *missing zones*.

Definition 2 A *concrete Euler diagram* is a pair $\hat{d} = \langle C(\hat{d}), F_{\hat{d}} \rangle$ where: $C(\hat{d})$ is a finite set of closed curves, called *(concrete) contours*, in the plane, and $F_{\hat{d}} : C(\hat{d}) \to \mathscr{L}$ is a function associating with each contour a label drawn from an infinite alphabet of labels $\mathscr{L}$. The *label set* $\mathscr{L}(\hat{d})$ of $\hat{d}$ is the set of labels associated with $\hat{d}$: $\mathscr{L}(\hat{d}) = \{F_{\hat{d}}(\hat{c}) \,|\, \hat{c} \in C(\hat{d})\}$. A *minimal region* of a concrete Euler diagram $\hat{d}$ is a connected component of $\mathbb{R}^2 - \bigcup_{\hat{c} \in C(\hat{d})} \hat{c}$. Let $X \subseteq C(\hat{d})$ be a set of contours. If the set $\hat{z} = \bigcap_{\hat{c} \in X} interior(\hat{c}) \cap \bigcap_{\hat{c} \in C(\hat{d}) - X} exterior(\hat{c})$ is non-empty, then $\hat{z}$ is a *zone* of $\hat{d}$ (note that a zone is a union of minimal regions); the set of labels associated with the contours in X is the *zone label set* $\mathscr{L}(\hat{z})$ of $\hat{z}$: $\mathscr{L}(\hat{z}) = \{F_{\hat{d}}(\hat{c}) \,|\, \hat{c} \in X\}$.

We say $\hat{d}$ is *wellformed* [2] if all of the following wellformedness conditions (WFCs) hold:

WFC 1 Simple contours: The contours are simple closed curves.

WFC 2 Unique contour labels: Each contour has a unique label; that is, $F_{\hat{d}}$ is injective.

WFC 3 Transverse intersections: Contours meet transversely. This can be subdivided into:

- **WFC 3a** No tangential intersections.
- **WFC 3b** No concurrency; that is contours meet at a discrete set of points.

WFC 4 No multiple points: At most two contours meet at a single point.

WFC 5 Connected concrete zones: Each concrete zone is a minimal region.

Example 1 *The left of Figure 2 shows a concrete Euler diagram $\hat{d}$ with three contours labelled A, B and C, and six zones which can be informally described as: outside all contours; inside A and outside B and C; inside B and outside A and C; inside A and B but outside C, inside A and C*

[2] These are the most commonly considered conditions but other constraints could be imposed, such as using fixed geometric shapes like circles or ellipses (for example, the generation of area proportional Euler diagrams with small numbers of circles was investigated in [Cho]).

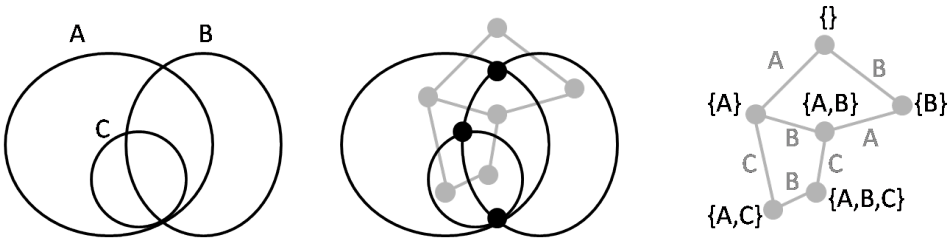


Figure 2: An Euler diagram (left), its graph together with an unlabelled dual graph overlaid in grey (middle) and the labelled dual graph (right) where the vertex labels sets and the induced edge labels are shown (with set brackets on edge labels omitted for readability purposes).

but outside B; inside A, B and C. The associated abstract diagram d is a set of abstract contours, together with a set of abstract zones: $\langle\{A,B,C\},\{\{\},\ \{A\},\ \{B\},\ \{A,B\},\ \{A,C\},\ \{A,B,C\}\}\rangle$, where the abstract zones correspond to the set of contours that the concrete zone is "inside". The concrete diagram $\hat{d}$ fails WFC 3a and WFC 4 since it has a point of intersection of the three curves (i.e. a multiple point) where the contours labelled A and C meet tangentially. The diagram $\hat{d}$ forms a single component and has three branch points (see Definition 4), giving rise to the graph $G(d)$ shown in the middle of the figure. An unlabelled dual graph of $G(d)$ is overlaid, shown in grey, whilst the right of the figure shows $\hat{d}^*$, the dual of the diagram $\hat{d}$.

Definition 3 Let $\mathscr{L}$ be an alphabet of labels. An *abstract labelled graph* G is a vertex-labelled graph, whose vertex labels are set of labels drawn from $\mathscr{L}$. The label set of G, denoted $\mathscr{L}(G)$, is the union of the vertex label sets of G. An abstract labelled graph which has been embedded in the plane is called a *concrete labelled graph* (i.e. this is a drawing in the plane with no edge crossings, which is sometimes called a *plane labelled graph*). A labelling on the edges of the graph is induced by taking the symmetric difference of the label sets of the incident vertices.

One can view a concrete diagram as a graph as follows; this definition generalises the cases given in [Cho, FFH08]: the intuition is that branch points are either points of intersection of the curves or places where concurrent curves separate.

Definition 4 Let $C = \{C_1,\ldots,C_n\}$ be a set of curves in the plane, where we also refer to C_i as the images of the curves, as usual. Let x be a point on any curve in C and let $B_\varepsilon(x) = \{y \in \mathbb{R}^2 : |x-y| < \varepsilon\}$ denote a ball of radius ε around x. If $\exists \varepsilon > 0$ and $i \in \{1,\ldots,n\}$ such that $B_\varepsilon(x) \cap C_i$ is (topologically) a line, but $B_\varepsilon(x) \cap C_j = \emptyset$ for any $j \neq i$ then x is a *non-singular point*. If $\exists \varepsilon > 0, k > 1$ and $i_1 \neq \ldots \neq i_k \in \{1,\ldots,n\}$ such that $B_\varepsilon(x) \cap C_{i_1} = \ldots = B_\varepsilon(x) \cap C_{i_k}$ is a line but $B_\varepsilon(x) \cap C_j = \emptyset$ for any $j \neq i_1,\ldots,i_k$ then x is a *point of k-concurrency*. If x is not a non-singular point nor a point of k-concurrency then x is a *branch point*.

Let d be concrete diagram. Then a *graph of d* is a plane graph $G(d)$ whose vertex set consists precisely of one vertex at every branch point, together with one vertex on each component [3] that has no branch points; and whose edges are the images of the curves joining these vertices. A *dual* of d, denoted d^*, is a concrete labelled graph that is a geometric dual graph of $G(d)$ such that: if vertex v of d^* is placed in zone $\hat{z}$ of d then v is labelled by $\mathscr{L}(\hat{z})$.

[3] i.e. a maximal set of (images of) curves that is connected.

The Euler diagram generation process, in [FFH08], takes an abstract diagram and creates a concrete diagram realising it, utilising a "dual graph" of the Euler diagram as part of the construction process; Definition 5 and Theorem 1 are rephrased from [FFH08].

Definition 5 Let G be a labelled graph. For $l \in \mathscr{L}$, let $G^+(l)$ and $G^-(l)$ denote the subgraphs of G induced by deleting any vertices whose labels contain l, and induced by deleting any vertices whose labels exclude l, respectively. Then G satisfies the *connectivity conditions* if G is connected, and $G^+(l)$ and $G^-(l)$ are connected for all $l \in \mathscr{L}(G)$. We say that G is *well connected* or passes the *connectivity conditions*.

Let G be a well connected plane labelled graph. Then G passes the *face conditions* if each face cycle of G (with distinct vertices and edges) of length $2n$ has crossing index $n-1$, where the crossing index of a face is the number of pairs of labels that appear non-nested in the edge-word around the face, read cyclically.

Example 2 *The dual graph shown on the right of Figure 2 has three faces with edge words BABA, CBCB and BACBCA. All three of these have crossing index 2, with the only non-nested pair of labels being A and C in the outside face word, BACBCA. Therefore the two internal faces pass the face conditions but the outer face fails it, and so there is a multiple point incident with the outer zone of the Euler diagram, as shown.*

Theorem 1 *Let d be an abstract diagram. Then there is a concrete diagram $\hat{d}$ which satisfies all of the WFCs and whose abstraction* [4] *is d if and only if there exists a concrete labelled graph G that has the vertex labeled $\{\}$ incident with the outer face, and the properties that: G is well connected (P1), G has unique vertex labels (P2); vertices of G whose label sets differ by more than one label are not adjacent (P3); and G passes the face-conditions (P4).*

The construction used in [FFH08] did not allow multiple edges between pairs of vertices, and such a concrete labelled graph G may require the addition of extra edges in order to construct the labelled dual graph d^*. In [Cho] a dual graph approach to the generation problem was adopted and similar existence theorems provided.

Proposition 1 *Let d be a concrete diagram and let d^* be a dual of d. If d satisfies (WFC1,2 and 5) then d^* satisfies (P1 and 2). If d^* satisfies (P1 and 2) then there is a concrete diagram d' which has dual d^* and which satisfies (WFC1,2 and 5).*

3 Transformation Systems

We first provide paramaterised transformation rules which enable both the generation and manipulation of diagrams at the abstract level, generalising the logical reasoning rules in [FHJT08].

Definition 6 Let $d = \langle C(d), Z(d) \rangle$ be an abstract Euler diagram. Define:

1. *RemoveContour*(l,d): If $\ell \in C(d)$, then d **with** ℓ **removed** is d' where $C(d') = C(d) - \{\ell\}$ and $Z(d') = \{Y - \{l\} : Y \in Z(d)\}$.

[4] This is the natural mapping from concrete to abstract diagrams; see [FFH08] for details if required.

2. *AddContour*(l, Z_c, Z_s, d): Let Z_c and Z_s denote (possibly empty) disjoint subsets of $Z(d)$, and suppose that $\ell \notin C(d)$. Then, d **with** ℓ **added**, zones Z_c **covered** and zones Z_s **split** is d' where $C(d') = C(d) \cup \{l\}$ and $Z(d') = (Z(d) - Z_c) \cup \{x \cup \ell : x \in Z_s \cup Z_c\}$.

3. *AddZone*(z, d): If $z \in \mathscr{P}C(d) - Z(d)$ then d **with** z **added** is d' where $C(d') = C(d)$ and $Z(d') = Z(d) \cup \{z\}$.

4. *RemoveZone*(z, d): Let $z \in Z(d) - \{\}$ (so z is not the zone outside all contours). Let $X \subseteq Z(d)$ be the set of contours which are in zone z but are not in any other zone in $Z(d)$. Then d **with** z **removed** is d' where $C(d') = C(d) - X$ and $Z(d') = Z(d) - \{z\}$.

Remark 1 *In settings where Euler diagrams represent propositional logic (e.g. see [FHJT08]) such transformations can be restricted to those that induce logical inferences to provide a reasoning system (the transformations are then often called reasoning rules). For instance, the addition of a new contour by AddContour(l, Z_c, Z_s, d) is a reasoning rule if $Z_s = Z(d)$ and $Z_c = \emptyset$; if shading is used in the system, then the reasoning rule for zone removal has the extra precondition that the zone is shaded, whilst zone addition has the extra postcondition that any missing zone that is added is shaded. The effects of altering rule sets within an automated reasoning environment for Euler diagram systems were investigated in [SMF+07].*

Corresponding (geometric) transformations at the concrete level are harder to realise consistently. For example, there are different possible ways of attempting to realise zone removal: if the zone has nice properties such as being star-shaped (i.e. that there is a point p in the region for which every other point is connected to p via a straight line) then one could use radial contraction. More generally, an operation to squash the zone to a point could be used if the region is simply connected, but if the zone is not simply connected (e.g. an annulus) then one may desire a transformation that does not identify all of the boundary of the zone to a point. The application of these operations may cause the violation of some wellformedness constraints (such as the simplicity of the contours), thereby either preventing their application at the concrete level or requiring a different concrete representation to be recreated, if one exists, destroying the mental map and the utility of the visualisation even in the case that such a recreation exists. This contrasts with the effect of zone removal at the abstract level which can always be applied to a (non-outside) zone z in the syntactic transformation system.

Therefore, we wish to build concrete level transformation systems for Euler diagram based on concrete dual graphs manipulations; any instance of a concrete transformation should lift to the appropriate instance of an abstract transformation of Definition 6. Accordingly, we develop a concrete dual graph transformation system in which all of the plane graphs satisfy the connectivity conditions and the vertex label set are unique [5] ; there are many other variations at the plane graph level that could be adopted, or one could consider abstract dual graph level transformations if one is willing to sacrifice some geometric information.

[5] Any abstract diagram has a representation as a concrete diagram which are unions of regions with holes [MF94, RZF08], or Euler-like [Cho]. However, commonly, one wishes to keep the system as straightforward as possible for the users, and here we choose to enforce the use of uniquely labelled simple closed curves and connected zones.

3.1 Adding and removing contours

We define operations at the dual graph level in order to realise contour addition at the concrete level. Intuitively a *collar* of a path, or a cycle, is a thickening of that path or cycle. However, in this context we incorporate the use of labels and we choose to alter the resulting graph so that we retain planarity, connectivity and uniqueness of vertex label sets.

Definition 7 Let d be a concrete diagram and let d^* be a dual of d. Let p be a path of distinct vertices and edges, except possibly for the first and last vertex (i.e. p can be a simple cycle), in d^*. Let p' be a new path (or cycle) which is a copy of p but which has an additional label l added to has all of its vertex label sets. Suppose that $p_1,\ldots,p_n$ is the vertex sequence of p and $p'_1,\ldots,p'_n$ is the vertex sequence of p', where the label of p_i differs from the label of p'_i by the label l. Then $collar(p;d^*,l)$, the *collaring* [6] of p, is the graph obtained from d^* by:

1. embedding p' in the plane such that:
   (a) for each i, the vertex p'_i is disjoint from every vertex and edge in d^*.
   (b) the edges of p' are disjoint from the edges of p and the vertices of d^*.
   (c) each vertex p'_i is in a neighbourhood of vertex p_i (that is, if p_i has coordinates (a_i,b_i), then $\exists\varepsilon>0$ such that $p'_i \in B_\varepsilon(p_i)=\{(x,y)\in\mathbb{R}^2 \mid (x-a_i)^2+(y-b_i)^2<\varepsilon\}$), and
   (d) if p is a cycle then p' is in the interior of the bounded region that is bounded by p [7].
   (e) if p is not a cycle, then there is a vertex of d^*, labelled by $\{\}$, which is not in any region of the plane bounded by p' and d^*.
2. adding an edge labelled l between vertex p_i and vertex p'_i, for each $i\in\{1,\ldots,n\}$.
3. for each edge $e=(p_i,v)\in d^*$ which crosses the path p',
   (a) if $v\in V(d^*)-\{p_1,\ldots p_n\}$ then delete e and add an edge from the vertex p'_i to v.
   (b) if $v=p_j\in\{p_1,\ldots p_n\}$ then delete e and add an edge from the vertex p'_i to p'_j.

Example 3 *The dual graph d_1^* at the top left of Figure 3 has a simple cycle p highlighted using dashed edges. The effect of collaring p is shown in the middle dual graph d_2^*; in this case no edges of d_1^* were crossed by the insertion of p' and so no edges of d_1^* were deleted. The effect of the application of collaring on the highlighted cycle in d_2^* is shown by d_3^* at the bottom right; three edges of d_2^* were crossed by the insertion of the path p' and so these were deleted and three new edges were added (the alterations are shown in grey). However, note that the label D has also been added to the vertex labelled $\{A,B,C\}$, which was in the interior of the cycle p in d_2^*, after the collaring operation described; this corresponds to the new contour covering the zone in the diagram (see Theorem 2). The corresponding Euler diagrams are shown, starting at the top left with d_1, performing a transformation that lifts to $AddContour(C,\emptyset,Z(d_1),d_1)$ to give d_2, shown in the top right, and then performing a transformation that lifts to $AddContour(D,\{\{A,B,C\}\},\{\{A\},\{A,C\},\{C\},\{B,C\},\{B\},\{A,B\}\},d_2)$ to give d_3 shown at the bottom left.*

[6] note that, as stated this operation is non-deterministic; any choices involved when edges are to be crossed by the insertion of the collar can lead to different diagram layouts, but here we concentrate on the fundamental properties of planarity and connectivity.

[7] note that d^* is embedded in the plane and so the cycle p splits the plane into two regions by the Jordan curve theorem; of course, one can adapt the theory to the non-embedded dual graph level.

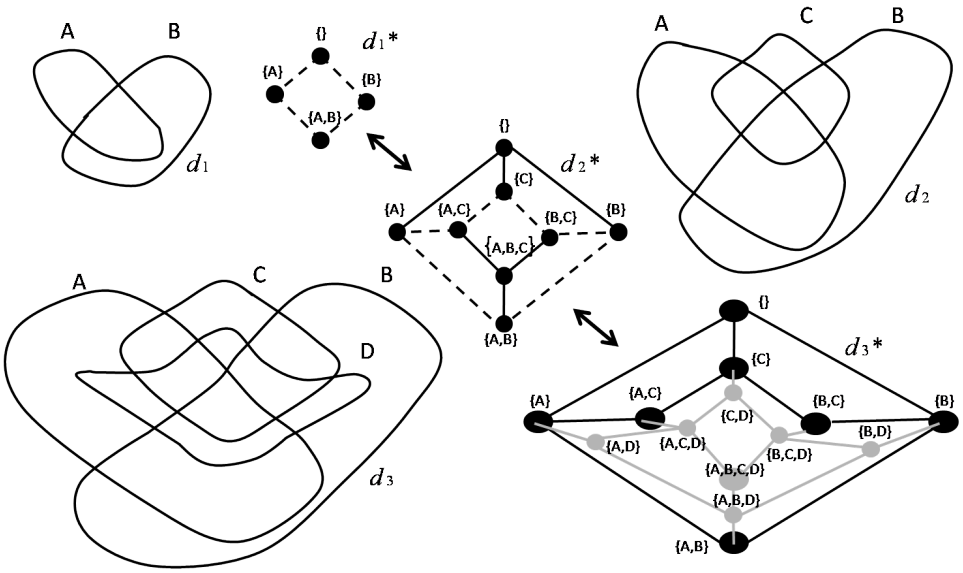


Figure 3: Transformations of the dual graph, and the corresponding Euler diagrams.

Theorem 2 *Let d^* be a well connected plane labelled graph with unique vertex label sets which is the dual of a concrete Euler diagram d, let p be a path of distinct vertices and edges, or a simple cycle, in d^*, l be a label which is not in $\mathscr{L}(d^*)$ and let H denote $collar(p;d^*;l)$. Then:*

1. *if no edge of d^* is missing from H, then H is a well connected plane labelled graph.*
2. *if there are edges of d^* that are missing from H, then collaring p yields a labelled plane graph which is wellconnected if and only if $H^-(l)$ is connected.*
3. *if c is a simple cycle of d^* with no vertices in $int(c)$, the interior of the bounded region bounded by c, and the path p lies entirely on the cycle c [8], then collaring p yields a well connected plane labelled graph.*
4. *Let Z_s denote the set of vertex label sets of p and let Z_c denote the set of vertex label sets of the vertices in $int(p)$ if p is a cycle (and $Z_c = \{\}$ otherwise). Then $collar(p;d^*;l)$ followed by the addition of label l to all vertices in $int(p)$, if p is a cycle, yields a graph K which is the dual of a concrete diagram d' that differs from d by the addition of the new contour labelled l, covering zones Z_c and splitting zones Z_s; that is, the composite transformation lifts to $AddContour(l,Z_c,Z_s,d)$. In particular, if p is not a cycle then collaring p lifts to $AddContour(l,\emptyset,Z_s,d)$.*

Proof. A path p of distinct edges and vertices in d^* corresponds to a sequence of adjacent zones in d. Adding a collar of p splits each of these zones into two adjacent zones, one inside l and one outside l, where l is the new label. The definition of collaring ensures that the resultant graph is planar. If no edges of d^* were removed upon collaring then the connectivity conditions also hold: the subgraphs $H^+(l) = p$ and $H^-(l) = d^*$ are connected, whilst the other induced subgraphs do not become disconnected by the addition of the collar, and case 1 holds.

However, if any edges of d^* were removed during collaring then the connectivity conditions for H could be broken. Now $H^+(l) = p$ is connected by definition, but $H^-(l)$ could be disconnected. The extra edges added during collaring ensure that there are no other obstructions to H

[8] this includes the case that p is the entire cycle c.

being wellconnected, as follows. Suppose that we have a vertex labelled x adjacent to a vertex labelled y in d^* and this edge is removed by the collaring operation. Then, without loss of generality, there is either a path $x-xl-y$ or a path $x-xl-yl-y$ in H. Thus, if x and y had a label k in common in d^*, then all of the vertices in this new path in H have label k, and similarly if they both exclude a label m in d^* then so do all of the vertices in this new path in H. Therefore, the connectivity of $G^+(k)$ and $G^-(m)$ are preserved upon collaring, and part 2 holds.

Part 3 follows since if p lies entirely on a simple cycle c which has no vertices in its interior then $H^-(l)$ is connected. In fact, since the only obstruction to the collaring operation preserving the connectivity conditions is the connectivity of $H^-(l)$, we can ensure that it is preserved in the case of the path being a simple cycle by adding the new label l to any vertices in $int(p)$; this operation does not affect the other connectivity conditions. This composite transformation of dual graphs corresponds to the addition of a new contour, where the zones to be split into two correspond to vertices in the simple cycle p, whilst those to be covered lie in $int(p)$. The effect of lifting to the abstract level can be checked by considering the label sets of the vertices that were present before and are present after the transformation, and so part 4 follows. □

Definition 8 Let G be a concrete labelled graph, and let l be a label in $\mathscr{L}(G)$. Define the operation *RemoveLabel*(l) to be the contraction of every edge labelled by exactly l, together with the identification of the corresponding vertices [9], followed by the removal of the label l from all vertex label sets.

Proposition 2 *Let d^* be a well connected plane labelled graph with unique vertex label sets which is the dual of a concrete diagram d, and suppose that c is a contour of d with label l. Then the operation RemoveLabel(l) on d^* corresponds [10] to the removal of the contour c from d.*

Proof. (sketch). Contraction of edges labelled by exactly l corresponds to the merging of each pair of adjacent zones whose label sets differ by the label l. The removal of the label l from all vertex label sets suitably updates all vertices. □

Example 4 *The dual graphs in Figure 3 from the bottom right to top left show the removal of labels D and C corresponding to Euler diagram contour deletion. In this case the RemoveLabel operation actually leaves multiple edges between vertices such that there are no other vertices in the interior of the region bounded by these vertices and edges. However, since such multiple edges have no significant effect on the diagrams constructed we can assume that these excess edges can be discarded.*

3.2 Adding and removing zones

When considering the operations of adding or removing zones at the concrete level, the natural addition or deletion of vertices of the dual graph may break the connectivity conditions. Since

[9] The label set taken is the union of the two label sets; the intuition is that this edge contraction then corresponds to stretching the contour labelled l so that it also covers the zones that it previously split.

[10] Note that the remove contour operation in our system is not applicable if the removal of the contour leaves a disconnected zone. This corresponds to the existence of vertices differing by label l which are not adjacent in d^*. Relaxing the zone connectedness condition would mean that contour removal is always applicable.

we wish to ensure that planarity and wellconnectedness are preserved, we consider extra edge additions on the neighbourhood of the vertices to be added or removed.

Definition 9 Let G be a labelled graph and let E be a set of edges in the complement of G. Then E is a *wellconnecting edge set for* G if G together with the edges in E is wellconnected. If G is a plane labelled graph then a *plane wellconnecting edge set for* G is a wellconnecting edge set E for G together with an embedding of E such that $G \cup E$ is a plane graph.

Definition 10 Let G be a well connected plane labelled graph, let v be a vertex of G, and let $G - v$ denote G with v removed. Suppose that E is a plane wellconnecting edge set for $G - v$. Then let $RemoveVertex(v, G)$ denote the operation of removing vertex v from G and adding E [11].

Lemma 1 *If v is a vertex of a well connected plane labelled graph, G, and v has vertex degree at most 3, then the edges of a complete graph on the set of vertices incident with v in G is a plane wellconnecting edge set for $G - v$.*

Definition 11 Let G be a well connected plane labelled graph and let Y be a set of labels which is not the label set of any vertex of G [12]. Suppose that G has a face F whose incident vertices have label sets that contain all of the labels of Y, and there is a plane wellconnecting edge set E for $G \cup v$, where v is a new vertex labelled by Y embedded in the interior of the face F. Then define $AddVertex(Y, G)$ to be the operation which inserts the vertex v with label Y in face F, and adds E.

Remark 2 *There is flexibility in the choice of rules developed, which could be chosen according to system requirements or user preference. The rule $RemoveVertex(v, G)$ is not applicable if there is no plane wellconnecting edge set for $G - v$, but if it is applicable then it preserves planarity and wellconnectivity. If one wanted a more relaxed system which always enabled the application of the rule then one could alter the rule so that the vertex v is always deleted. Furthermore, to attempt to improve the layout one could add a subgraph of the clique on the vertices incident with v that maintains planarity whilst minimsing the number of connectivity conditions that are broken. Similarly, one could generalise the $AddVertex(Y, G)$ rule to be always applicable and to attempt to preserve the connectivity conditions for as many labels as possible.*

Proposition 3 *Let d^* be a well connected plane labelled graph with unique vertex label sets which is the dual of a concrete diagram d. Let Y be a set of labels corresponding to a zone z which is missing from d and let w be a vertex of d^*. Suppose that there is a face F which has every label of Y appearing in its incident vertices and there is a plane wellconnecting edge set E for $G \cup v$, where v is a new vertex labelled by Y embedded in the interior of the face F. Then $AddVertex(Y, d^*)$ is a well connected plane labelled graph with unique vertex label sets, and the operation corresponds to the addition of zone z with label set Y to d yielding d'; i.e. when applicable, this lifts to the operation $AddZone(z, d)$. Also, $RemoveVertex(w, G)$ is a well connected plane labelled graph with unique vertex label sets and the operation corresponds to*

[11] The choice of edge set used effects the layout but we are primarily concerned with planarity and connectivity.
[12] This condition can be relaxed if the system allowed disconnected zones.

the removal of zone z with label set Y from d; this lifts to the operation RemoveZone(z,d).

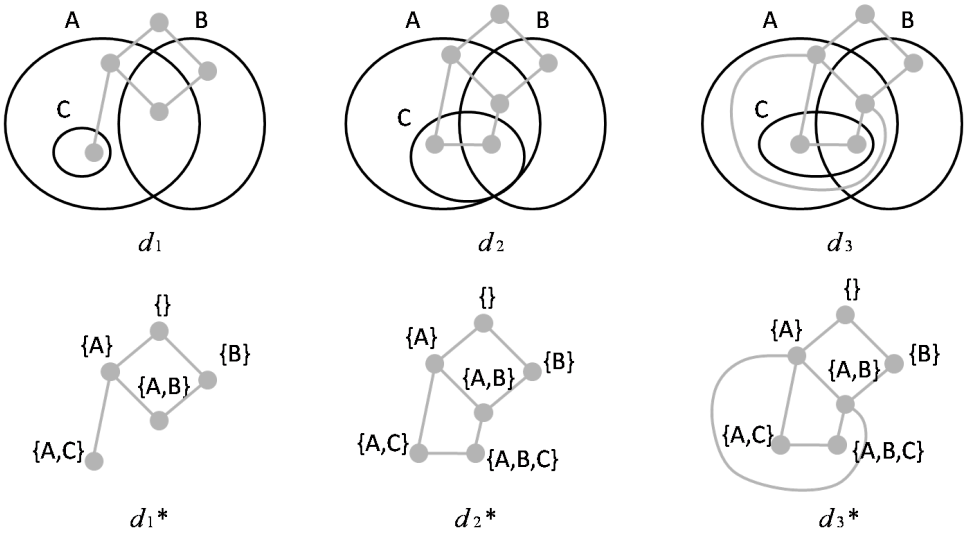


Figure 4: Left/middle: Addition/removal of a vertex to the dual graph and a zone to the diagram. Middle/Right: a post-processing graph transformation step for removing tangential intersections.

Example 5 *In Figure 4, the middle dual graph d_2^* shows the effect of AddVertex$(\{A,B,C\},d_1^*)$ applied to the left hand dual graph d_1^*, using the outer face F and wellconnecting edge set as shown. This corresponds to the addition of the zone $\{A,B,C\}$ to d_1 to give d_2. Since d_1^* is wellconnected, the application of RemoveVertex(v,G^*) where v is labelled by $\{A,B,C\}$, to d_2^* returns d_1^* corresponding to the removal of the zone $\{A,B,C\}$ of the diagram d_2 giving d_1.*

The incorporation of techniques to alter the dual graph to remove various breaks in well-formedness conditions, or to exchange them, would facilitate the construction of different systems. For instance, in the right hand side of Figure 4, we see the insertion of an extra edge into the dual graph between $\{A\}$ and $\{A,B\}$ surrounding $\{A,C\}$ and $\{A,B,C\}$. Since we obtain exactly one vertex in each face when taking a dual, this has the effect of moving the contour C away from the intersection point of A and B in the Euler diagram, thereby removing the tangentiality and the multiple point.

4 Conclusion

Logical reasoning systems based on Euler diagrams are commonly constructed at an abstract level, and the concrete level is merely a visualisation of the abstract level. Then, changes at the abstract level are not consistently reflected by changes at the concrete level that preserve the mental map (i.e. the usual approach is the regeneration of new diagrams after some transformation that do not take into account of the previous diagrams prior to the transformation). Building transformation systems at the concrete level which are realisations of the abstract level transformations addresses this concern, enabling the presentation of diagrams that reflect change in a local manner when it is possible to do so. This enables strong control of the topological

and geometric properties of the diagrams allowed in order to assist with user comprehension and preferences. In this paper we have viewed concrete diagrams as graphs and provided a concrete level dual graph transformation system which can be utilised to transform concrete diagrams. Altering the graph theoretic properties of the system will enable the realisation of transformation systems of concrete diagrams satisfying different sets of wellformedness conditions and should facilitate interplay between such systems. Furthermore, abstract level dual graph transformation systems could also be constructed; these will enable rules to be applicable more often than at the concrete level, but they will lose some geometric information.

In the future, specialisations of this theory could be used to provide fast computations of restricted classes of diagram transformations. This could be utilised in a graph transformation based library system for generation and manipulation of diagrams for instance, where a collection of initial graphs together with a set of transformation rules is used to attempt to generate a diagram requested by a user, whether that user be human or a system request.

Acknowledgements: Funded by UK EPSRC grant EP/E011160: Visualisation with Euler Diagrams. Thanks to the anonymous reviewers and to John Taylor for their helpful comments.

Bibliography

[Cho] S. C. Chow. *Generating and Drawing Area-Proportional Euler and Venn Diagrams.* PhD thesis, University of Victoria.

[CR03] S. Chow, F. Ruskey. Drawing Area-Proportional Venn and Euler Diagrams. In *Proceedings of Graph Drawing 2003, Perugia, Italy*. LNCS 2912, pp. 466–477. Springer-Verlag, September 2003.

[DES03] R. DeChiara, U. Erra, V. Scarano. VennFS: A Venn Diagram file manager. In *Proceedings of Information Visualisation*. Pp. 120–126. IEEE Computer Society, 2003.

[DF07] R. DeChiara, A. Fish. EulerView: A non-hierarchical visualisation component. In *Proceedings of IEEE Symposium on Visual Languages and Human Centric Computing*. IEEE 134, pp. 145–152. 2007.

[FFH05] A. Fish, J. Flower, J. Howse. The Semantics of Augmented Constraint Diagrams. *Journal of Visual Languages and Computing* 16:541–573, 2005.

[FFH08] J. Flower, A. Fish, J. Howse. Euler Diagram Generation. *Journal of Visual Languages and Computing* 19:675–694, 2008.

[FHJT08] A. Fish, J. Howse, C. John, J. Taylor. A normal form for Euler diagrams with shading. In *Proceedings of Diagrams 08*. LNAI 5223, pp. 206–221. Springer, 2008.

[HES+05] P. Hayes, T. Eskridge, R. Saavedra, T. Reichherzer, M. Mehrotra, D. Bobrovnikoff. Collaborative Knowledge Capture in Ontologies. In *Proceedings of the 3rd International Conference on Knowledge Capture*. Pp. 99–106. 2005.

[HST05] J. Howse, G. Stapleton, J. Taylor. Spider Diagrams. *LMS Journal of Computation and Mathematics* 8:145–194, 2005.

[Ken97] S. Kent. Constraint Diagrams: Visualizing Invariants in Object Oriented Modelling. In *Proceedings of OOPSLA97*. Pp. 327–341. ACM Press, October 1997.

[KMGB05] H. Kestler, A. Muller, T. Gress, M. Buchholz. Generalized Venn Diagrams: A New Method for Visualizing Complex Genetic Set Relations. *Journal of Bioinformatics* 21(8):1592–1595, 2005.

[MF94] E. C. M. Egenhofer, P. di Felice. Topological Relations between Regions with Holes. *International Journal of Geographical Information Systems* 8:129–144, 1994.

[RZF08] P. Rodgers, L. Zhang, A. Fish. General Euler Diagram Generation. In *Proceedings of 5th International Conference on Diagrams 2008*. LNAI 5223, pp. 13–27. Springer-Verlag, 2008.

[SMF+07] G. Stapleton, J. Masthoff, J. Flower, A. Fish, J. Southern. Automated Theorem Proving in Euler Diagrams Systems. *Journal of Automated Reasoning* 39(4):431–470, 2007.

A Generic Graph Transformation, Visualisation, and Editing Framework in Haskell

Scott West[1], **Wolfram Kahl**[2]

[1]saynte@gmail.com [2]kahl@cas.mcmaster.ca
Department of Computing and Software, McMaster University, Hamilton, Ontario, Canada

Abstract: Graph transformation, visualisation, and editing are useful in many contexts, and require domain-specific customisation. However, many general-purpose graph solutions lack customisability in at least one area.

We present a framework that aims to allow polished customisation in all three areas, using the powerful abstraction capabilities of the pure functional programming language Haskell. The design of our framework integrates and adapts time-tested object-oriented designs into a purely functional framework, and uses current user-interface libraries (GTK+ and Cairo) to achieve polished presentation.

Our framework provides both a low-level programmed approach to graph transformation, and, on top of this, high-level approaches including SPO and DPO, which are implemented using categorical abstractions in an intuitive and flexible way.

Keywords: Programmed graph transformation, Algebraic graph transformation, Pure functional programming, Generic graph editor

1 Introduction

Although graphs are a mathematically simple concept, the pragmatics of providing tool support for graph manipulation is surprisingly complex. The lack of general frameworks for graph manipulation tools has as a consequence that even successful tools for special-purpose graph manipulation get away with surprisingly poor user interfaces.

The more widely known graph transformation tools, such as AGG, DiaGen, Progress, and DaVinci (now uDraw) tend to concentrate on the particular aspects that are related to the research direction they have grown out of. This means that these tools frequently introduce limitations in other aspects that cannot be overcome by a reasonable effort on the side of the framework user, i.e., of the developer who implements a more customised application.

In our framework, we strive to implement the following requirements:

(1) Visual presentation of and interaction with graphs must be customisable to an extremely large degree, and in a "natural" way.

(2) Arbitrary graph transformation needs to be programmable.

(3) High-level graph transformation approaches need to be expressible in a natural way, and in a way that results in reasonably efficient implementations.

(4) A programmer willing to invest more effort into optimisation should have the tools available to implement more efficient transformations by reverting to lower-level primitives.

The problem here is that requirement (3) implies, in particular, availability of categorical abstractions, which tend to be blissfully unaware of efficiency concerns, and is also normally understood to contradict (2), or tends to accommodate (2) in an only partial way that, however, still compromises the principled and declarative nature of the high-level approaches. That apparent conflict is best resolved by embedding the transformation capabilities into a language that provides powerful abstraction mechanisms, and allows abstraction barriers to be enforced. The latter strongly suggests the use of a pure programming language that controls side effects via its type system; together with the former, and with general availability and library support requirements, essentially only the pure functional programming language Haskell fits this bill.

However, for visual presentation of and interaction with graphs (requirement 1) there is no good functional paradigm available yet, while it is essentially *the* case study of object-oriented design [Joh92]. Therefore it is natural to satisfy requirement (1) by exposing an object-oriented interface for visual interaction purposes. This does not even contradict the choice of Haskell — Kiselyov and Lämmel have recently catalogued [KL05] a number of ways to satisfactorily implement object-oriented abstractions in Haskell. Using this, we created a framework that, in its kernel (Sections 3 and 4), encapsulates the presentational and interactive aspects of graphs and graph items (i.e., nodes and edges) in an object-oriented class hierarchy closely emulating that of [Joh92], but still provides a purely functional interface to item-level read-only graph access, and also provides a safe monadic interface to item-level graph manipulation, enabling a programmed graph transformation approach not too different from that of PROGRESS [SWZ99].

On top of this kernel, we use the abstraction mechanisms of Haskell to provide a high-level interface that includes an abstract datatype of partial graph homomorphisms with both algebraic and item-level access functions (Sect. 5), and basic category-theoretic constructions, and standard algebraic approaches to graph transformation (DPO and SPO) implemented on top of those (Sect. 6). In Sect. 7, we discuss some related work.

2 A Few Quick Haskell Notes

The pure functional programming language Haskell [P+03] features a relatively lean, mathematical syntax; it uses indention to indicate components which are part of the same term, **case** alternatives, and utilises monadic **do** statements to program "with an imperative flavour".

Function application is normally denoted by juxtaposition, as in "f x", has highest precedence, and associates to the left. To avoid parentheses, there is also a low-priority infix operator $ for function application; this allows to write e.g. "f $ g x" instead of "f (g x)". Another infix operator stands for function composition: $(f \circ g)\ x = f\ (g\ x)$.

Binary functions in Haskell can be converted to infix operators using a pair of back-ticks; for example, "x `someFunction` y" is just another notation for "someFunction x y".

Haskell features of *type classes*, which are similar to Java interfaces. In the Eq (equality) type class, instances must supply at least one of $\equiv$ and $\not\equiv$ to be override the default implementation.

$$\begin{aligned} \textbf{class}\ \mathsf{Eq}\ a\ \textbf{where}\ (\equiv), (\not\equiv) &:: a \to a \to \mathsf{Bool} \\ a \equiv b &= \neg\,(a \not\equiv b) \\ a \not\equiv b &= \neg\,(a \equiv b) \end{aligned}$$

Type classes may have class constraints, such as **class** Eq a $\Rightarrow$ Ord a **where**..., which specifies

that instances of Ord must necessarily also be instances of Eq. The typing annotation

$$\mathsf{lookup} :: \mathsf{Eq\ a} \Rightarrow \mathsf{a} \rightarrow [(\mathsf{a}, \mathsf{b})] \rightarrow \mathsf{Maybe\ b}$$

specifies that the function lookup is to be applied to a value of some type, a, for which equality must be implemented (type class constraints appearing to the left of the $\Rightarrow$), and to a list of pairs as second argument; results are then of type Maybe b, which contains the value Nothing and the values Just y for each value $\mathsf{y} :: \mathsf{b}$.

Monads, which are often used to model sequential instructions in the otherwise functional language, are a very versatile concept. The related idea of monad transformers is used in this work to provide a clear definition of the sequential context in which we are operating in.

If we operate in the Maybe monad, we know that the results of the computation will either be success (Just) or failure (Nothing). To add some integer-variable lookup environment such as a Map String Integer to the Maybe monad, we could use the ReaderT monad transformer to construct a new monad instance. This new instance would have an underlying read-only state which can be accessed by using the ask function anywhere within the monad. When the computation is run, the results are still of type Maybe. Similar write-only and read-write transformers are defined as WriterT and StateT transformers — explicitly composing complex monads in this way has the advantage that imperative effects are precisely constrained by the type system.

3 Object-Oriented Abstractions for Interaction and Visualisation

The framework presented in this paper strives to make generating fully-featured graph editors very simple. The built-in features include the ability to select and drag nodes around, as well as resize the extents of the nodes. Edges are represented as Bezier curves, allowing for smooth transitions between nodes. Additionally, the view can be modified by using linear transformations such as zooming and panning.

At a low-level, these abilities are implemented in an object-oriented manner, as there are known, working solutions in the object-oriented world [Joh92]. We realist this in Haskell using a method of translating OO designs into a functional setting which is largely inspired by [KL05].

In Haskell, programming to a concept of updatable state is one instance of the use of *monads*, which allows an imperative programming style embedded into pure functional programs. Since object-oriented designs naturally employ also the concepts of object creation, we essentially have the choice between the two predefined monads providing dynamic creation of references to updatable memory, namely ST, in which reference allocation and updates are the only possible side effects, and IO, which also allows input and output as side-effects. We implement our object-oriented data structures parametrised over such monads, will always use the type variable m for this monad parameter. As far as objects are used to support a GUI, we will be compelled to use IO for m, but the possibility of using an ST monad instead makes it feasible to create and transform graphs within pure functional computations.

A *canvas* for us is an area for graphical representation that is not unlike a magnet-board: Objects can be placed, moved, and modified in an interactive way. This is unlike a drawing-surface which allows write-only operation, i.e., where marks, once made, cannot be easily changed or deleted. The classes supporting our canvas abstraction include Figure for elements placed on the

canvas, ConnectionFigures that form connections between other figures, aggregations of figures called Drawings, and DrawingView, the visualisation of the drawing as it appears on-screen.

For deep changes to the way the framework operates interactively and visually by default, appropriate subclasses to these classes will need to be defined. For more superficial changes, much can be accomplished through the use of node and edge labels. In our setting, we use labels to define particular kinds of graphs. For example, a Petri net can be considered as a graph with node labels that indicate whether a node is a place (with some number of tokens) or a transition.

```
data Petri = Place Integer | Transition deriving (Show, Typeable)
```

To allow these elements to be used in an interactive and visual setting, we demand that labels support type reïfication via Typeable (necessary for the class casting mechanisms) and provide an instance of the FigureLabel type class:

```
class (Show label, Typeable label) ⇒ FigureLabel label where
  draw       :: label → Rect → Render ()
  size       :: label → Maybe Point
  parseLabel :: Parser label
  interfaceIO :: LabelFig label IO → DView IO → IO ()
```

The members of this type-class should provide the following functionality:

- draw – The visual representation of the label adapted to its physical size, specified as a rectangle. The resulting drawing operations are encapsulated in the Render monad.
- size – Specify that the label should be drawn either with variable size (Nothing), or with fixed size by producing Just pt for a Point value pt which will be interpreted as a size.
- parseLabel – The framework also handles the saving and loading of graphs in files, and uses the Parsec [LM01] parser combinator library. To allow the labels to be restored from a string, a parsing function needs to be implemented that acts as inverse to the Show instance required by the superclass constraint.
- interfaceIO – We can additionally allow interactive actions to be performed when the node carrying the label is activated (currently by double-clicking) through a GUI view. These actions could be anything including popping up a dialog-box to modify the label, or performing a graph transformation.

Using about two pages of literate Haskell code, we defined a FigureLabel instance for the Petri label type introduced above, and with two additional pages defining the GUI wrapper around the drawing canvas, we created the editor and animator shown in Figure 1, where interfaceIO will fire the clicked transition if possible — this is implemented using the programmed approach described in Sect. 4.

Since the framework's visual and interactive components aim to be cross-platform and appealing to the eye, they use a Haskell binding to the well-known and portable GTK+ library. Graph display uses a vector graphics library associated with GTK+, the Cairo rendering engine (http://cairographics.org), which can draw on various kinds of "surfaces". Aside from the "screen" surface, one can also draw on more abstract surfaces, such as PDF and PostScript surfaces. The result is that any graph drawings produced in the editing framework can be easily reproduced in a high-quality format for use in printable documents.

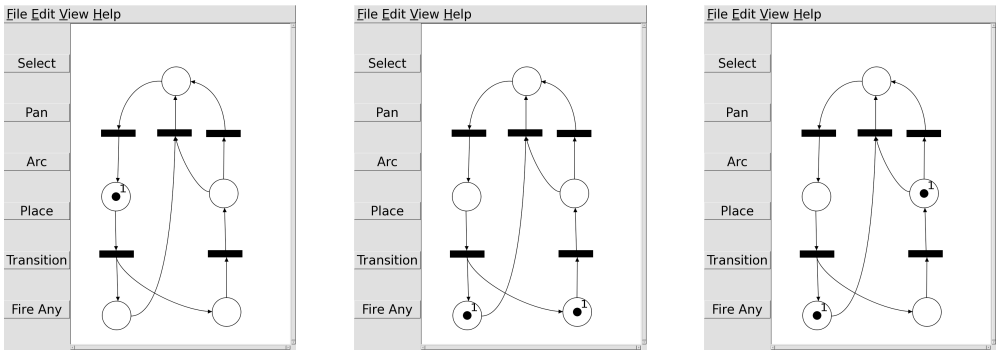


Figure 1: Petri net editor and animator, defined using labels of type Petri.

4 Programmed Transformations

Graph transformation steps typically divide into two parts: A matching phase, which does not need more than read-only access to the graph that is to be transformed, and a modification phase. Haskell programmers will naturally expect that the type system can prevent modification steps to be used during the matching phase — this makes reasoning about graph transformation programs, and activities like code refactoring much easier. Since matching is conceptually non-side-effecting, it is only natural to expect that matching can be programmed without performing monadic computations. However, one of the problems of programming to a graph view built on the object-oriented programming principles outlined above is that all graph access has to be performed as part of a monadic computation.

Therefore we start graph transformations by adding a "pure view" to the object-oriented view of a drawing representing a graph. This "pure view" allows pure functions to access the graph for matching and analysis purposes, but also connects to the object-web in a way that monadic updates can be performed on both together. For both purposes we provide low-level and higher-level interfaces, described in the remainder of this section.

4.1 Low-Level Interfaces

All the graph transformation machinery in our framework is built on top of two interfaces.

A *graph inspection* interface ReadLGraph provides *pure* (i.e., non-monadic) functions for read-only access to the graph structure, including node and edge sets, edge incidence, node and edge labels. This interface not only includes basic functions, but also more advanced facilities, as for example a function calculating strongly connected components — although such functions could be implemented on top of the basic functions, in general only custom implementations by the interface instances can achieve satisfactory performance.

A *monadic graph modification* interface MonadLGraph, containing item-level graph update functions, like addition and deletion of labelled nodes and edges, label updates, updates of source or target of edges, etc. The unusual aspect of the modification interface is the typing of the

exposed functions: A pure node addition function would typically have the type[1] $a \to g \to (n, g)$, meaning that given a node label and a graph, a new graph is produced that differs from the argument graph only by including a new, appropriately labelled node, which is returned together with the new graph. (Our inspection interface contains *no* functions with graph results.)

A typical monadic interface would instead use the type $a \to g \to m\ n$, for a monad type constructor m, i.e., the function would produce a computation of type $m\ n$, that is, a computation returning the new node, with the understanding that the graph reference passed in as argument now refers to the updated graph when it is be used in subsequent computations.

For our graph modification interface, we intentionally allow "the worst of both worlds" in order to give the implementation more liberties:

$$\mathsf{grNewLNode} :: a \to g \to m\ (n, g)$$

At first sight, this appears to imply a very awkward style of programming to this interface. However, the ordering of arguments and results has been chosen so that this function can be embedded directly into a standard-library state monad transformer [Jon95], so that we obtain $\mathsf{StateT} \circ \mathsf{grNewLNode} :: a \to \mathsf{StateT}\ g\ m\ n$. From this typing, we see that this is now a function that, given a node label, returns a computation in the "state-enriched" monad $\mathsf{StateT}\ g\ m$, which keeps a current graph (of type g) as its state while performing computations in monad m. This StateT view of the modification interface then allows a normal imperative programming style, and also forms the basis for the primitive operations of the GraTra monad, see Sect. 4.3 below.

4.2 The Select Monad

Matching, i.e., identifying redexes for graph transformation rules, has two aspects that invite application of "pre-fabricated abstractions": All matching activity happens in the context of a fixed, "current" graph, and matching steps are frequently non-deterministic, and failure requires backtracking. The first can be implemented using a Reader monad, while the second is most easily handled by a list monad; using monad transformers [Jon95, LHJ95], the two can be combined:

$$\textbf{type}\ \mathsf{Select}\ g = \mathsf{ListT}\ (\mathsf{Reader}\ g)$$

This is automatically a monad, and allows an intuitive programming style for matching purposes, as the following utility operator demonstrates: starting from a node selector $\mathsf{sel} :: \mathsf{Select}\ g\ n$, the selector $\mathsf{sel} \rightsquigarrow p$ finds nodes that sel generates whose outgoing edges target nodes which have a label satisfying the node label predicate $p :: a \to \mathsf{Bool}$.

```
(~>) :: (ReadLGraph g n e a b) => Select g n -> (a -> Bool) -> Select g (e,(n,n))
sel ~> p = do
  n1 <- sel                      -- select a first node
  e  <- sel_out1 n1              -- select an outgoing edge
  n2 <- sel_trg e                -- obtain target node of that edge
  sel_node_label n2 >>= guard p  -- backtrack if target node label does not satisfy p
  return (e,(n1,n2))
```

[1] In the Haskell code fragments in the remainder of this paper, we use the type variables g for graphs, n for nodes, e for edges, a for node labels, and b for edge labels.

4.3 The GraTra Monad

For higher-level graph modification, we have to start from the monad m required by the low-level modification interface, and we add backtracking as for Select, and an updatable state containing the current graph together with its "history" represented as a chain of graph homomorphisms. The history is necessary for cases where, for example, a lot of information is collected during matching, and then applied in separate modification steps: Nodes used in later steps may have been identified with other nodes, and taken on their identity; the history is used to implement the necessary indirection in a modular way that mixes well with backtracking. The history also allows to form an overall partial graph morphism as a result of a series of operations.

Due to the necessity of history concatenation, the standard StateT monad transformer [Jon95] is not sufficient here, so the GraTra monad is implemented directly, and exported abstractly, so that its state invariants can be guaranteed.

The GraTra monad provides primitive operations such as adding and removing graph elements, as well as updating the node labels. In addition, inside GraTra computations, Select computations can be performed by embedding them using the following function:

$$\mathsf{select} :: (\mathsf{MonadLGraph\ m\ g\ n\ e\ a\ b}) \Rightarrow \mathsf{Select\ g\ x} \rightarrow \mathsf{GraTra\ g\ n\ e\ m\ x}$$

All these, together with the abstraction capabilities of Haskell, constitute a powerful toolkit to express graph transformations in a programmatic way.

5 Abstractions for Categorical Rewriting Approaches

Low-level graph transformations in a high-level programming language are of course unsatisfactory, so we provide a number of abstractions to enable programming of and with high-level graph transformations approaches.

For any graph type g implementing the ReadLGraph g n e a b interface, the data type SubGraph g n e encapsulates a graph of type g together with the node and edge sets of a subgraph (so the node set includes all nodes incident to edges in the edge set). On such SubGraphs, both item-level operations, like insertion, deletion, membership test, are available, and lattice operations like join (union), meet (intersection), and pseudo-complement (the subgraph lattice of a non-discrete graph is not complemented).

Similarly, a GraphMor g n e encapsulates two graphs and two finite maps representing a partial graph homomorphism between these graphs. Again, item-level operations are provided, including membership tests, and insertion and deletion of item pairs from the maps. More importantly, a categorical interface is provided, including morphism source, target, composition $***$, identities, and also operations returning domain and range of a morphism as SubGraphs, and restricting a morphism on either side with a (compatible) SubGraph.

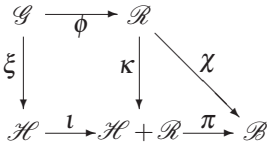
Categorical constructions involve creation of new graphs, and therefore require access to the monadic modification interface MonadGraph; we provide for example production of isomorphic copies via copyGraph, disjoint union of graphs via directSum, and division modulo graph congruences via quotientProj, which is used to implement co-equalisers of total graph homomorphisms — we just list the type signatures for these:

$$\mathsf{copyGraph} :: \mathsf{g} \rightarrow \mathsf{m\ (GraphMor\ g\ n\ e)}$$

```
directSum    :: g →              g              → m (GraphMor g n e, GraphMor g n e)
quotientProj :: [[n]] → [[e]] → g              → m (GraphMor g n e)
coEqualiser  :: GraphMor g n e → GraphMor g n e → m (GraphMor g n e)
```

It is well-known that pushouts of total graph homomorphisms can be calculated from direct sums and co-equalisers where those exist; in Haskell, the category-theoretic diagram can be almost directly transliterated for this purpose:

```
pushout :: GraphMor g n e → GraphMor g n e → m (GraphMor g n e, GraphMor g n e)
pushout xi phi = do
  (iota, kappa) ← Mor.directSum (Mor.trg xi) (Mor.trg phi)
  proj ← coEqualiser (xi *** iota) (phi *** kappa)
  return (iota *** proj, kappa *** proj)
```



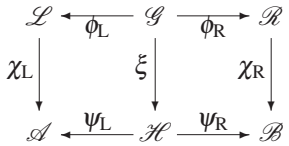
In the next section we show how to use these abstractions to program DPO and SPO graph rewriting at the level of categorical descriptions.

6 Implementing DPO and SPO Graph Transformation

The *double-pushout approach* is the "classical" variant of the "algebraic approach" to graph rewriting, going back to [EPS73]. In this approach, a rewriting rule is a *span* $\mathscr{L} \xleftarrow{\phi_L} \mathscr{G} \xrightarrow{\phi_R} \mathscr{R}$ of morphisms, which we represent using a record datatype:

```
data Span g n e = Span{spanLMor, spanRMor :: GraphMor g n e}
```

Performing a DPO rewrite step on an application graph $\mathscr{A}$ involves identifying a redex, i.e., a suitable "matching" morphism χ_L from the left-hand side $\mathscr{L}$ into the source graph, completing the left square via the construction of a pushout complement, including the host morphism ξ from the gluing graph $\mathscr{G}$ to the host graph $\mathscr{H}$, and completing the right square by constructing a pushout.



One of the two parts of the "gluing condition" necessary for the existence of a pushout complement is the "dangling condition", which is conventionally given as follows:

Definition 6.1 *For two graph morphisms $\phi : \mathscr{G} \to \mathscr{L}$ and $\chi : \mathscr{L} \to \mathscr{A}$, the* dangling condition *holds iff whenever an edge connects a node outside the image of χ with a node x inside the image of χ, then x has in its pre-image via χ only nodes in the image of ϕ.* □

Kawahara has shown that the gluing condition can be formulated directly in the language of relations in the topos of graphs [Kaw90]; the following equivalent definitions [Kah01, Def. 5.4.1], all for morphisms $\phi : \mathscr{G} \leftrightarrow \mathscr{L}$ and $\chi : \mathscr{L} \leftrightarrow \mathscr{A}$, directly use the setting of relational graph homomorphisms, and can easily be implemented using our Morphism and SubGraph libraries:

- The *identification condition* holds for ϕ and χ iff χ is almost-injective besides $\mathsf{ran}\,\phi$:

$$\chi ; \chi^{\smile} \subseteq \mathbb{I} \cup (\mathsf{ran}\,\phi) ; \chi ; \chi^{\smile} ; \mathsf{ran}\,\phi \quad .$$

  For almost-injectivity, we have a morphism predicate:

```
identCond ranPhi chi = Mor.almostInjectiveBesides chi ranPhi
```

- The *dangling condition* holds for ϕ and χ iff $\chi ; \mathsf{ran}\,\chi^{\sim} \subseteq (\mathsf{ran}\,\phi) ; \chi$.

  The dangling condition uses semicomplement $S^{\sim}$ of a subgraph S of G. This is the least subgraph C such that $S \cup C = G$. The subgraph $S^{\sim}$ therefore overlaps with S exactly in the nodes inside S that are connected to nodes outside S.

```
danglingCond ranPhi chi =
              (chi `Mor.ranRestr` SubGraph.semiComplement (Mor.ran chi))
  `Mor.leq` (ranPhi `Mor.domRestr` chi)
gluingCond phi chi = danglingCond ranPhi chi ∧ identCond ranPhi chi
  where ranPhi = Mor.ran phi
```

- χ is called *conflict-free for* ϕ iff $\mathsf{ran}\,(\phi ; \chi ; \chi^{\smile}) \subseteq \mathsf{ran}\,\phi$.

  This property is important in the single-pushout approach; we can replace the occurrence of a conversion with a pre-image operator, since $\mathsf{ran}\,(\phi ; \chi ; \chi^{\smile}) = \mathsf{ran}\,(\mathsf{ran}\,(\phi ; \chi) ; \chi^{\smile})$.

```
conflictFree phi chi = Mor.preImg chi (Mor.ran (phi ⁂ chi)) `SubGraph.leq` Mor.ran phi
```

For implementing the low-level Def. 6.1_8 of the dangling condition directly, one may pre-calculate the pre-images via χ, by calculating the converse of χ considered as a relation; with that, it should be straight-forward to see how the following code implements the above definition (a more concise danglingCond is defined below):

```
danglingCond' :: forall g n e ∘ ReadGraph g n e ⇒ GraphMor g n e → SubGraph g n e → Bool
danglingCond' chi ranPhi = let
  g          = Mor.trg chi                    -- the target of the matching
  ranChi     = SubGraph.nodeSet $ Mor.ran chi -- the node range of chi
  chiConverse :: MRel n n   -- a relation represented as set-valued map
  chiConverse = converseToMRel $ Mor.nodeMap chi
  safeNode :: n → Bool   -- holds iff x has in its χ-pre-image only nodes in the image of φ.
  safeNode x = lookupMRel x chiConverse `Set.isSubsetOf` SubGraph.nodeSet ranPhi
    -- Remember that the ordering ⩽ on Bool is implication:
  safeEdge (s,t) = if   s `Set.member`    ranChi
                   then t `Set.notMember` ranChi ⩽ safeNode s
                   else t `Set.member`    ranChi ⩽ safeNode t
  in all safeEdge $ mapMaybe (grIncidence g) (grEdges g)
```

This serves to show how categorical abstractions can lead to more concise code, but also how concepts that break the categorical abstraction (like Def. 6.1_8) can still be implemented in a mathematically accessible way.

The host graph $\mathscr{H}$ in a DPO rewriting step is always a subgraph of the application graph $\mathscr{A}$; Kawahara's construction for the DPO approach is also useful for the SPO approach. We also provide a variant that preserves nodes incident to dangling edges and can be used for Parisi-Presicce's "restricting derivations" [PP93]; both variants directly transliterate [Kah01, Def. 5.4.6]:

```
straightHostSG phi chi = Mor.ran chi `SubGraph.implication` Mor.ran (phi *** chi)
sloppyHostSG phi chi =
  SubGraph.semiComplement (Mor.ran chi) `SubGraph.join` Mor.ran (phi *** chi)
```

While SubGraph denotes a subgraph via subsets of the carrier sets of the underlying graph, the function Mor.subGraph produces an independent graph resulting from restriction to these subsets, and returns a pair consisting of a total injection morphism from the newly constructed subgraph to the original graph, and it converse, which is a partial (univalent) homomorphism, and which can be used to calculate the host morphism ξ.

```
constructHost chiL sp = do
  psi@(_, psiLC) ← Mor.subGraph $ straightHostSG phiL chiL
  return (psi, phiL *** chiL *** psiLC)
  where phiL = spanLMor sp
```

For reducing a DPO redex, we just need to put this together with the pushout construction for the right-hand side. We choose to return, together with the embedding of the right-hand side, the partial morphism $(\psi_L^\smile ; \psi_R) : \mathscr{A} \to \mathscr{B}$ along the bottom of the DPO diagram; one could of course also choose to return the constituant morphisms, or even the whole diagram.

```
reduceDPOredex rule chiL = do
  ((_psiL, psiLconv), xi) ← constructHost chiL rule
  (psiR, _chiR) ← pushout xi $ spanRMor rule
  return (psiLconv *** psiR, xi)
```

This redex reduction is an essentially deterministic computation (in the monad m for which no backtracking capabilities are assumed). A DPO rule induces a non-deterministic reduction relation by permitting arbitrary redexes; we implement this as a backtracking computation into which we lift the redex reduction:

```
applyDPO rule = do
  let phiL = spanLMor rule
  chiL ← select ∘ matchSel $ Mor.trg phiL     -- backtrack over possible matchings
  guard $ gluingCond phiL chiL                -- prunes illegal redexes
  (m, xi) ← lift $ reduceDPOredex rule chiL   -- DPO construction
  doMorphism m                                -- updates "current" graph
  return xi                                   -- return RHS embedding
```

For many applications, matchSel will not be an appropriate choice for determining redexes, either for efficiency reasons, or because of the presence of some strategy. The function applyDPO is therefore only an example how our building blocks can be assembled at a high level to easily produce reasonable implementations of high-level graph transformation approaches.

For a further example, we re-use some of the material above to implement the single-pushout approach. A partial morphism is easily converted into a span of total morphisms:

```
spanFromPMor mor = do
  inj ← Mor.subGraphInj $ Mor.dom mor
  return $ Span inj (inj *** mor)
```

With that, we can use [Kah01, Thm. 5.4.11] (which is a generalised version of the result by Löwe [Löw90, Cor. 3.18.5] that single-pushout squares for conflict-free matchings have total embeddings of the right-hand side) to employ reduceDPOredex to implement single-pushout rewriting for conflict-free matchings.

```
applySPO rule = do
  chiL ← select ∘ matchSel $ Mor.src rule
  guard $ conflictFree rule chiL
  (m, xi) ← lift $ spanFromPMor rule >>= flip reduceDPOredex chiL
  doMorphism m
  return xi
```

7 Related Work

The DiaGen diagram generation tool [MK00, Min02] aims in a similar direction as our framework. The similarities include the idea of combining graph transformation and visualisation into a single framework, to generate interactive editors. However, the DiaGen system uses a different fundamental transformation system, namely hyper-edge grammars. Our approach is not limited in this way, offering a programmed approach that can be expanded upon to utilise DPO and SPO if required. Also, our approach does not require the preprocessing that DiaGen does to generate the templates for developers to expand upon to create their editor.

Another influential graph rewriting system is the PROgrammed Graph REwriting SyStem (PROGRESS) [SWZ99]. PROGRESS combines textual and graphical representations to specify graph productions, tests and paths. Simple productions can be used to construct more complicated programmed transformations by using the PROGRESS control structures. The organisation of PROGRESS is similar to our approach, including the ability to backtrack programmed transformations in the case of failure. However, the inclusion of a graphical language to specify graph transformations is one large difference. Also, PROGRESS seems to be less concerned with user-interaction, customisability, and algebraic approaches.

In the same spirit as our framework, the AToM3 tool [LVM03] offers an environment to create interactive editors for graphs. AToM3 is able to generate Python scripts which can then be loaded back into AToM3 to offer customised model creation. For example, the base-tool offers a Petri net model that can be loaded at runtime to start editing Petri nets. The AToM3 approach differs from ours in a few fundamental ways. Firstly, the visualisations are markedly more primitive, not offering any compositing of graphical primitives, nor any high-quality rendering options like anti-aliasing and PDF and PostScript export. We do not yet match the export facilities of AToM3 to GXL and GML, but these would be easy to add even for a user of our framework.

Tiger [EEHT04] is an ambitious project offering full graphical descriptions of editors. It is built on top of the Eclipse IDE framework, extended by Eclipse GEF (Graphical Editing Framework) as well as the EMF (Eclipse Modeling Framework); graph transformation facilities are handled by the AGG-engine. The heavy usage of the Eclipse framework leads to some UI confu-

sion, as the custom editors still sport many of the default Eclipse menus and interface elements, leading to an arguably less focused user experience. Transformation rules and the appearance of nodes and edges are customisable visually.

On the Haskell side, the most notable graph abstractions are the inductive graph interfaces of Erwig's Functional Graph Library FGL [Erw97]. These interfaces allow to decompose non-empty graphs by separating from the remaining graph the context (label, and incoming and outgoing edges) for an either given or arbitrary node. Since edges are simply triples of source node, target node, and edge label, these interfaces do not directly support the concept of edge identity essential for most categorical approaches. Our implementation actually uses FGL for the "pure view" explained in Sect. 4, and therefore has to embed edge identity information in the FGL edge labels.

The depth-first search tree abstraction of King and Launchbury [KL95] is entirely geared towards graph analysis, where it provides a collection of useful tools. It provides an array-based representation for unlabelled graphs, and includes no facilities for graph modification.

8 Conclusion

We have outlined a framework that allows developers to easily create interactive graph editors that offer polished user interfaces and include powerful graph transformation capabilities.

Graph transformations can be expressed in different ways, with primary support offered for a programmed approach to graph transformation, and derived implementations of the categorical DPO and SPO approaches.

In addition to the graph transformation abilities, the framework has progressed far enough that the direct interaction mechanism works in an reasonably intuitive manner. Several small editors have been constructed for models including Petri nets and Hasse diagrams, and some preliminary work has started on a code-graph editor as well. The project is still in an early stage, but already offers interactive resizing and positioning of canvas items, diagrams can be exported to PDF and PostScript, some rudimentary layout facilities are offered by way of GraphViz, with all of these features available to any graphs which the framework produces. A prototype of functionality for undoing basic operations is also available.

The ease with which the high-level approaches are implemented on top of the low-level programmed approach owes much to the power of the abstraction mechanisms provided by Haskell. The result is a unique environment both for experimentation with novel approaches to graph transformation and interaction, and for easy creation of polished high-quality graph manipulation applications.

Bibliography

[EEHT04] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Towards Graph Transformation based Generation of Visual Editors using Eclipse. In *Visual Languages and Formal Methods, Elec. Notes Theor. Comp. Sci. vol. 127*. Pp. 127–143. Elsevier, 2004.

[EPS73] H. Ehrig, M. Pfender, H. J. Schneider. Graph Grammars: An Algebraic Approach. In *Proc. IEEE Conf. on Automata and Switching Theory, SWAT '73*. Pp. 167–180. 1973.

[Erw97] M. Erwig. Functional Programming with Graphs. In *ICFP '97*. Pp. 52–65. ACM, 1997. See also http://web.engr.oregonstate.edu/~erwig/fgl/.

[Joh92] R. E. Johnson. Documenting Frameworks Using Patterns. In *OOPSLA '92*. Pp. 63–76. ACM, 1992.

[Jon95] M. P. Jones. Functional Programming with Overloading and Higher-Order Polymorphism. In Jeuring and Meijer (eds.), *Advanced Functional Programming*. LNCS 925, pp. 97–136. Springer, 1995.

[Kah01] W. Kahl. A Relation-Algebraic Approach to Graph Structure Transformation. 2001. Habil. Thesis, Fakultät für Informatik, Univ. der Bundeswehr München, Techn. Report 2002-03, http://www.cas.mcmaster.ca/~kahl/Publications/RelRew/.

[Kaw90] Y. Kawahara. Pushout-Complements and Basic Concepts of Grammars in Toposes. *Theoretical Computer Science* 77:267–289, 1990.

[KL95] D. J. King, J. Launchbury. Structuring Depth-First Search Algorithms in Haskell. Pp. 344–354 in [POP95].

[KL05] O. Kiselyov, R. Lämmel. Haskell's overlooked object system. Draft, submitted for journal publication. Online since 30 Sept. 2004; full version released 10 Sept. 2005, http://homepages.cwi.nl/~ralf/OOHaskell/, 2005. (last accessed 19 Dec. 2008).

[LHJ95] S. Liang, P. Hudak, M. Jones. Monad Transformers and Modular Interpreters. Pp. 333–343 in [POP95].

[LM01] D. Leijen, E. Meijer. Parsec: Direct Style Monadic Parser Combinators for the Real World. Technical report UU-CS-2001-27, Department of Computer Science, Universiteit Utrecht, 2001. See also: http://www.cs.uu.nl/~daan/parsec.html.

[Löw90] M. Löwe. Algebraic Approach to Graph Transformation Based on Single Pushout Derivations. Technical report 90/05, TU Berlin, 1990.

[LVM03] J. de Lara Jaramillo, H. Vangheluwe, M. A. Moreno. Using Meta-Modelling and Graph Grammars to Create Modelling Environments. *Electr. Notes Theor. Comput. Sci.* 72(3), 2003.

[Min02] M. Minas. Specifying graph-like diagrams with DiaGen. In *Proc. International Workshop on Graph-Based Tools (GraBaTs'02)*. ENTCS 72(2), pp. 16–25. 2002.

[MK00] M. Minas, O. Köth. Generating Diagram Editors with DiaGen. In Nagl et al. (eds.), *Applications of Graph Transformations with Industrial Relevance, Proc. AGTIVE'99, Kerkrade, The Netherlands, Spt. 1999*. LNCS 1779, pp. 443–440. Springer, 2000.

[P+03] S. Peyton Jones et al. *The Revised Haskell 98 Report*. Cambridge Univ. Press, 2003. Also on http://haskell.org/.

[PP93] F. Parisi-Presicce. Single vs. double pushout derivation of graphs. In Mayr (ed.), *Graph Theoretic Concepts in Computer Science, WG '92*. LNCS 657, pp. 248–262. Springer, 1993.

[POP95] *POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. acm press, 1995.

[SWZ99] A. Schürr, A. J. 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: Applications, Languages and Tools*. Chapter 13, pp. 487–550. World Scientific, Singapore, 1999.

Aspects for Graph Grammars

Rodrigo Machado, Luciana Foss and Leila Ribeiro

rma, lfoss, leila @ inf.ufrgs.br
Instituto de Informática
Universidade Federal do Rio Grande do Sul
Porto Alegre, Brasil

Abstract: Aspect-oriented programming (AOP) is an extension to the object oriented paradigm that aims to provide better modularity for code that is scattered across object oriented systems, such as logging, authentication and distributed object handling. Aspect weaving is a novel way to compose systems, focusing on the integration of system-wide policies through pattern-action rules. While there are several semantic proposals for representing aspects over source code and programs, aspect weaving for visual models such as graph rewriting systems are still not established. In this work, we propose the definition of aspect-oriented graph grammars, an extension to conventional graph grammar where aspects are represented as transformations rules over the structure of a base graph grammar.

Keywords: aspect-oriented software development, graph grammars, double-pushout approach.

1 Introduction

Aspect-oriented programming (AOP) [KLM+97] is an extension to the object oriented paradigm that aims to provide better modularity for code that is scattered across object-oriented systems, such as logging, authentication and distributed object handling. The main idea of the paradigm is to encapsulate the statements that deal with such situations in a module called aspect. Inside the aspect there are rules (advices) that describe how these statements should be weaved into the base code. Every advice actuates over a specific set (pointcut) of system execution points (join points), executing some action *before*, *after* and/or *in place of* the join point.

Aspect-orientation can be seen as a kind of *meta-programming* that allows one to describe system-wide behaviors in a compact notation. Since its proposal in the late 90's, the paradigm has been gaining acceptance and development tools. Although there are several proposals to describe the operational effect of aspect weaving over programs, visual models such as graph rewriting systems are still not established.

The fact that several visual languages can be naturally modeled using graphs makes graph grammars an appealing formalism to define the semantical models for such languages. A graph grammar (GG) [Roz97] is a model in which the state is represented by a graph and system evolution is represented by graph rewriting productions. Several interesting models for computation and software development, such as UML diagrams [HET08], have a natural graph-based interpretation, and thus can be modeled by means of GGs.

In this work we address the issue of crosscutting concerns in graph grammars, and propose the definition of *graph aspects* to modularise their treatment. Our main contribution is the definition of aspect-oriented graph grammars (AOGG), where aspects are represented by a second-order transformation over the productions of a base GG. We also specify how aspects are combined to a base grammar, resulting in a *weaved* graph grammar. By defining formally aspects and aspect weaving over graph grammars, we also provide a semantic interpretation for aspect-oriented concepts over other models that are specific instances of GGs.

The rest of the text is organized as follows. Initially, in Section 2, we informally present the main concepts of the aspect-oriented paradigm. In Section 3, we review graph grammars and introduce our working example. Then, in Section 4, we discuss how to modify the example in order to implement system-wide policies such as logging. In Section 5, we provide a description of aspects over graph grammars and formally define aspect-oriented graph grammars. Finally, in Section 6, we compare our approach to other proposals, state our final remarks and present future work.

2 Aspect-Oriented Paradigm

The main purpose of using AOP is to spread some behavior automatically over the whole source code (or bytecode) of the application. The fundamental abstractions of the paradigm are the following: *i) join points:* execution points that can be affected by aspects; *ii) pointcuts:* specific sets of join points; *iii) advices:* rules that, given a pointcut, define some behavior to be triggered when the system reaches some of its join points; *iv) inter-type declarations:* extensions to the static structure of the system, which may be needed by the behavior introduced by the advices. *v) aspects:* modules containing all advices and inter-type declarations for dealing with a specific crosscutting concern.

In aspect-oriented programming languages, join points are generally defined has a subset of the system named transitions, like method calls and attribute accesses. Pointcuts are specific sets of join points usually defined by means of a *pointcut language*, which defines expressions for join point matching. Advices substitute the join points that match its associated pointcut with some programmed behavior, which can also include the original behavior. The module that combines the aspects over the original base code is called *aspect weaver*. As a simple example of aspect weaving, consider the AspectJ source code depicted in Figure 1 (AspectJ is the most popular AOP extension for the Java programming language). The AspectJ weaver receives both the base code and the aspect code. Then, it applies the advices within the aspect, inserting the commands provided in the advices every time it finds their pointcuts in the base code. In the example of Figure 1, the aspects simply introduces a print command right before the start of the execution of any method without parameters sent to an object of class `A`. Although the result of the combination is shown as a source-code transformation, the AspectJ compiler actually performs byte-code level weaving, i.e. the aspect weaving occurs after the compilation of both base code and aspects.

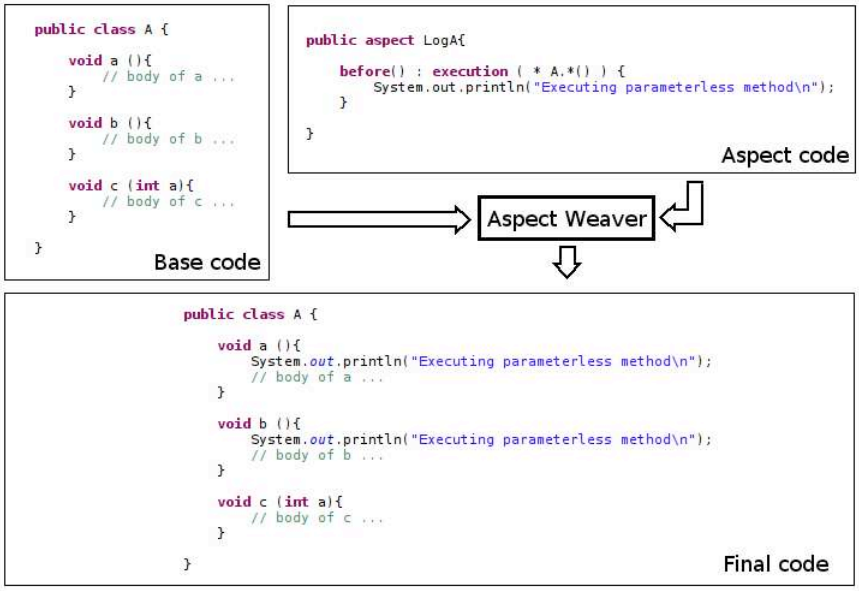


Figure 1: Example of aspect weaving in AspectJ

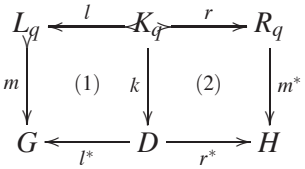
3 Graph Grammars

A graph grammar (GG) is a visual model to represent systems. In a GG, the states of the system are graphs and the system behavior is defined by an starting graph together with a set of graph rewriting rules. In this section, we recall the basic concepts of GGs, according to the DPO (double-pushout) approach [C+97], and provide the working example to be used in the rest of the paper. We will use *typed graph grammars*, i.e. grammars where all states and rules are typed.

Definition 1 ((Typed) Graph and Graph Morphisms) A *graph* is a tuple $G = \langle V_G, E_G, s^G, t^G \rangle$, where V_G and E_G are sets of vertices and edges, and $s^G, t^G : E_G \to V_G$ are the source and target function. A *(total) graph morphism* $f : G \to G'$ is a pair of functions $(f_V : V_G \to V_{G'}, f_E : E_G \to E_{G'})$ such that $f_V \circ s^G = s^{G'} \circ f_E$ and $f_V \circ t^G = t^{G'} \circ f_E$. The category of graphs and total graph morphisms is called **Graph**. Let $T \in$ **Graph** be a fixed graph, called type graph, a *T-typed graph* G^T is given by a graph G and a (total) graph morphism $t_G : G \to T$. A morphism of T-typed graphs $f : G^T \to G'^T$ is a (total) graph morphism $f : G \to G'$ that satisfies $t_{G'} \circ f = t_G$. A typed graph G^T is called *injective* if the typing morphism t_G is injective. The category of T-typed graphs and T-typed graph morphisms is the comma category **Graph**$\downarrow T$, shortened by T-**Graph**.

Definition 2 (Graph Productions and Graph Grammars) A *T-typed (graph) production* (or graph rule) is a tuple $q : L_q \overset{l_q}{\hookleftarrow} K_q \overset{r_q}{\hookrightarrow} R_q$, where q is the name of the production, L_q, K_q and R_q are T-typed graph, l_q and r_q are injective morphisms. The class of all T-typed graph production is denoted by T-**Prod**. A T-typed *graph grammar* is a tuple $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, where T is a type graph, P is a set of production names, π is a function mapping production names to productions in T-**Prod**, and G_0 is a T-typed graph, named the *initial graph*.

Definition 3 (Direct derivation and Derivations) Given a T-typed graph G, a T-typed graph production $q = L_q \overset{l}{\leftarrow} K_q \overset{r}{\rightarrow} R_q$ and a match (i.e. an injective T-typed graph morphism) $m : L_q \rightarrow G$, a *direct derivation* from G to H using q (based on m) exists if and only if the diagram on the right can be constructed, where both squares are pushouts in T-**Graph**. In this case the direct derivation is denoted by $\delta : G \overset{q,m}{\Rightarrow} H$ or $\delta : G \overset{q}{\Rightarrow} H$ if we do not make explicit m. Elements in L_q which are not in the range of l are said to be deleted by q, while elements in R_q which are not in the range of r are said to be created by q. Given a graph grammar $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, a *derivation* $\rho : G_0 \overset{p_1,m_1}{\Rightarrow} G_1 \overset{p_2,m_2}{\Rightarrow} G_2 \cdots$ of $\mathscr{G}$ is a finite or infinite list of direct derivations $\delta_i : G_i \overset{p_i,m_i}{\Rightarrow} H_i$, where $G_{i+1} = H_i$ and $i \geq 0$. If a derivation $\rho : G_0 \overset{p_1,m_1}{\Rightarrow} \cdots \overset{p_n,m_n}{\Rightarrow} G_n$ is finite we call G_n the final graph.



Example 1 (Graph grammar) *Figure 2 shows a graph grammar thats models a client-server scenario. The type graph* `T` *represents the possible kinds of nodes: clients (stylized persons), content servers (cylinders), addresses (pentagons), data (rectangles with sharp angles), signaling messages (rectangles with rounded angles), and connections between clients and servers (circles with the letter* `C`*). There are basically two kinds of interactions in this system: clients can recover information from servers providing an address as parameter, and clients can store information in servers, passing both the address and the desired information as parameters*[1]*. In order to retrieve or store information, the client must first connect with a server that provides the required address. After the connection, the information is exchanged and, finally, the connection is released. The graph productions* `ConnectGet`, `TransferGet` *and* `CloseGet` *perform the information retrieval from servers, and the graph productions* `ConnectSend`, `TransferSend` *and* `CloseSend` *perform the information update. Inside the rules, the items annotated with small* `D`*'s are the ones being deleted, and the ones with small* `C`*'s are the ones being created. The initial graph of the system consists of two clients and three servers. One of the clients comes with an initial send message for address* `A2` *(the "updater" client), while the other one has two get messages for addresses* `A2` *and* `A3` *(the "reader" client). According to the order in which the productions are applied, the reader client can retrieve information about the address* `A2` *before or after it is updated by the updater client. Also, the reader client can get connected to any server that provides address* `A3`*, retrieving different results according to the server it connects to.*

Graph grammars provide a natural and visual way to represent distributed and nondeterministic systems, such as the one shown in Example 1. Distribution is naturally represented by the graph topology. The semantics of graph grammars is based on production applications. If there are matches for more than one production in one state (graph), they may all be applied in parallel, if there are no conflicts. Conflicts exist if two (or more) productions try to delete the same portion of a graph at the same time. In such a situation, the choice of which production will be actually be applied is non-deterministic.

[1] in this example, it would be necessary to have attributes in order to properly represent addresses and numeric data. Since our main focus is in the crosscutting concerns, for now we left attributes out of the theoretical development.

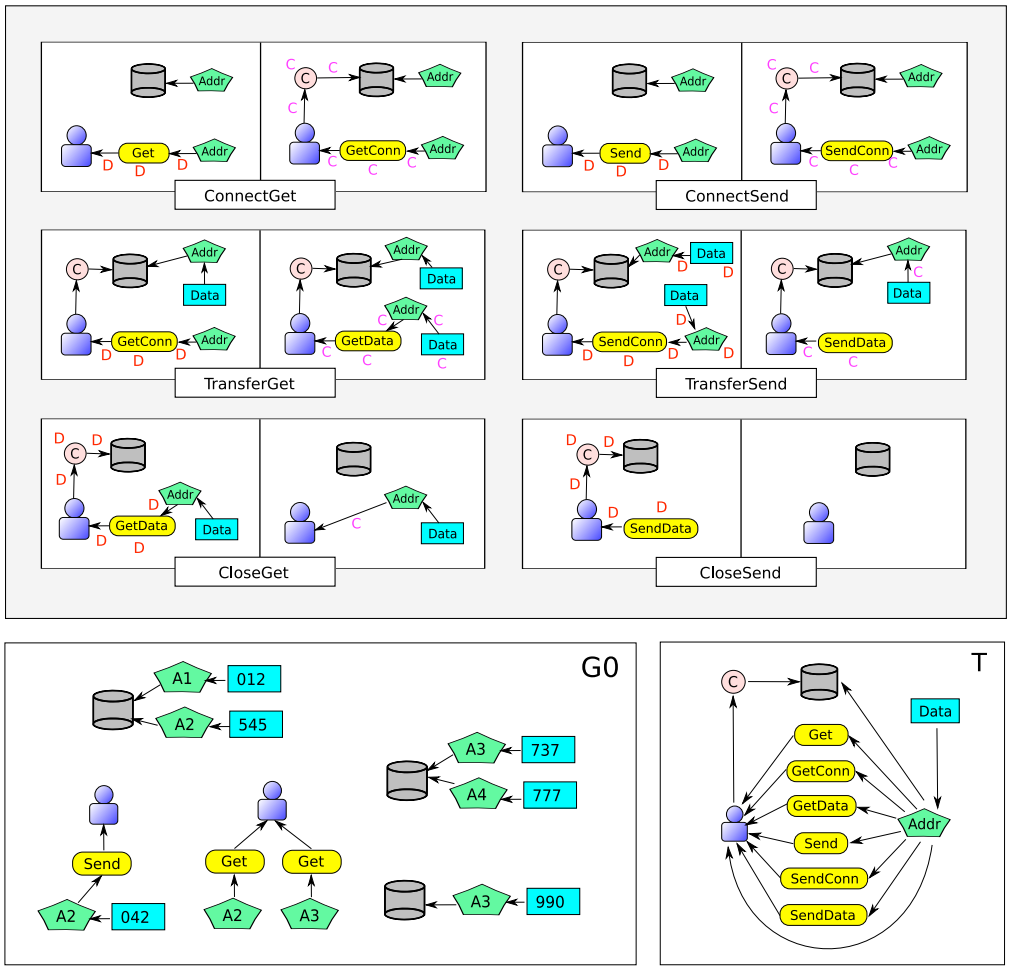


Figure 2: Example of graph grammar for clients and servers

4 Crosscutting Concerns in Graph Grammars

The aspect-oriented paradigm's main purpose is to solve the problem of lack of modularity for the code that handles crosscutting concerns. The classical examples of such requirements in object-oriented systems are logging policies, authentication and distributed object handling. In order to illustrate the concept of crosscutting concerns in the context of graph grammars, we propose two simple modifications to the system of Figure 2: the inclusion of a *logging* object (to log executions) and of a security policy for server access.

4.1 Logging Execution Steps

Suppose we want to register every execution step within the system in order to have access to the execution history. For instance, it is very common that servers store information about the start and the end of each client connection, both for profiling and security reasons. In the context of

GGs, this would mean that we have to record each production application, or derivation step. In our example, the changes that have to be performed to introduce such a log object are:

1. the type graph would have to be extended to introduce the new kind(s) of element(s);
2. the initial graph should be populated with initial instances of the new elements (if any). In the case of log, we must introduce one global instance of the log object;
3. all relevant productions must be modified in order to reflect the desired behavior. The left-hand side of every rule should have an additional element (the log register), and some information related to the effect of a production application on this log shall be included.

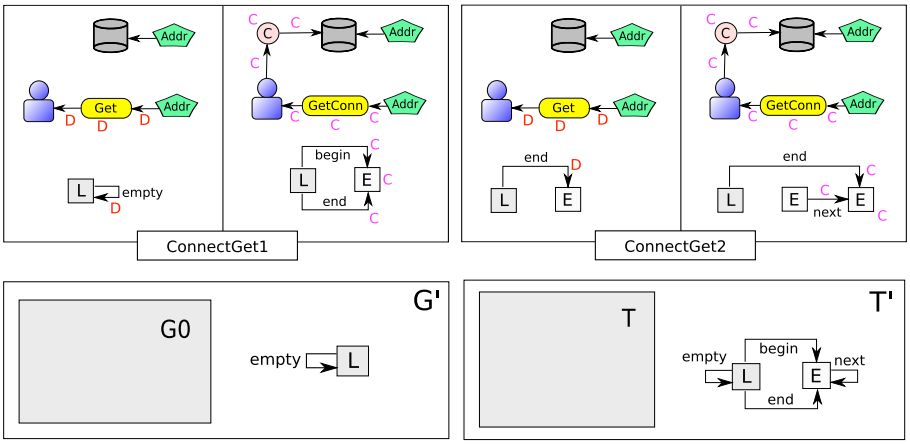


Figure 3: New type graph, initial graph and variations of the original rule `ConnectGet` to implement log.

Figure 3 depicts the required modifications over the GG presented in Example 1 in order to implement a simple log policy. The square node with an `L` represents the global log object. The square node with an `E` represents a log entry, which carries information about the production application. In order to keep the example simple, we omitted this information from log entries – they actually only represent the number of applied productions (this abstraction is fine, since our purpose is not to show how to model logs, but rather how to model transformation of specifications, that is, how one specification is transformed into another by considering an aspect). Log entries are connected to each other in a way that resembles a linked list structure, represented by the arrows `begin`, `end` and `next`. The empty list is represented by the endoarrow `empty`. The modification to the initial graph would be only the addition of one empty log object, i.e. one with a unique `empty` arrow. The modification that has the a bigger impact concerns the productions, since all of them must be altered to cope with two different situations: when the log list is empty, and when it has at least one element. For instance, the rule `ConnectGet` must be rewritten as a pair of productions, as shown in Figure 3. This should be done for every production, duplicating the total number of productions of the graph grammar. This very small example shows how structural patterns for rules do not scale well in the usual definition of graph grammars.

One interesting effect of this log model concerns the graph grammar execution. Since we have a global log object which is updated by all productions, we lose the possibility of simultaneous application of productions, even if they refer to different client and servers. Thus, this implementation of logging modifies the concurrent semantics of the system, although the sequential semantics is not changed at all.

4.2 Security policy for server access

Another system requirement that is a crosscutting concern is the implementation of *security policies*. Suppose it is important to distinguish between two kinds of users: *content administrators*, the ones that have write and read access to the servers, and ii) *plain users*, who can only read information. Every time a user tries to connnect to a server, its type should be taken into account to decide if the connection should be allowed. A very simple implementation of such policy is depicted in Figure 4, which depicts a new type graph and new versions for rules `ConnectGet` and `ConnectSend`. The user attribute `R` represents *read* privilege and `W`, *write* privilege. Both user marks are preserved by the productions. Unlike the log policy, which affected all the productions, these may be the only rules affected by the security policy, since the permissions may be verified only when the connections are being made.

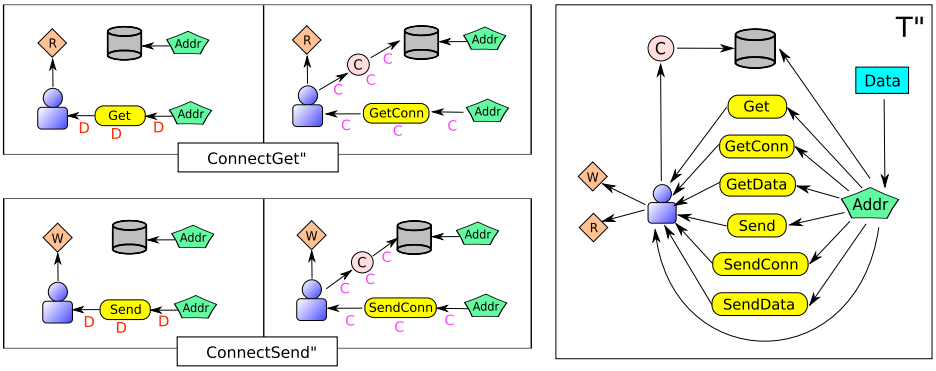


Figure 4: Modified rules and type graph for security policy.

Note that both the log and the security implementations are modelled as modifications of both the structure (type graph) and the behavior (initial graph and graph productions) of the original GG. If both crosscutting concerns are needed in our specification, the productions may become excessively complex and difficult to understand, since they may have to treat several crosscutting concerns. One of the original motivations for using visual methods such as GG is its ease of use, and such lack of modularity can difficult its adoption for modeling large systems. In the next sections we introduce aspect-oriented graph grammars (AOGG) as an extension of traditional graph grammars. In AOGGs, the modifications needed to treat every crosscutting concern is encapsulated into an *aspect*, allowing clearer specifications.

5 Aspect-Oriented Graph Grammars

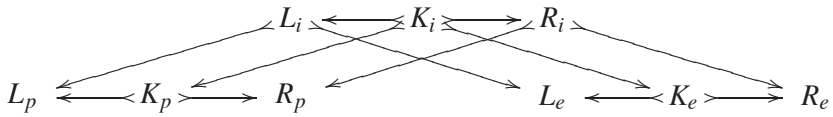
In this section, we describe formally how to define aspects over graph grammars, leading to the definition of aspect-oriented graph grammars (AOGG).

Graph advices may be seen as meta-productions defining how the original graph productions should be modified in order to implement a given crosscutting concern. Therefore, we employ the same mechanism for graph rewriting in order to describe production rewritings. First, we define a notion of how to relate productions (production morphism), that will be used to formally define graph advices.

Definition 4 (Production morphism) Let $p : L_p \overset{l_p}{\hookleftarrow} K_p \overset{r_p}{\rightarrowtail} R_p$ and $q : L_q \overset{l_q}{\hookleftarrow} K_q \overset{r_q}{\rightarrowtail} R_q$ be T-typed graph productions. A *production morphism* $f : p \to q$ is a triple $\langle f_L, f_K, f_R \rangle$ of T-typed graph morphisms between the left-hand side, interface and right-hand side of the productions such that the following diagram commutes. The production morphism $f = \langle f_L, f_K, f_R \rangle$ is *injective* iff all its components are injective. The category of T-typed productions and T-typed production morphisms is denoted T-**MSpan**.

$$\begin{array}{ccccc} L_p & \overset{l_p}{\hookleftarrow} & K_p & \overset{r_p}{\rightarrowtail} & R_p \\ {\scriptstyle f_L}\downarrow & & \downarrow{\scriptstyle f_K} & & \downarrow{\scriptstyle f_R} \\ L_q & \underset{l_q}{\hookleftarrow} & K_q & \underset{r_q}{\rightarrowtail} & R_q \end{array}$$

Definition 5 (Graph advice) A T-typed *graph advice* a is a production over T-typed productions, i.e. it is a monic span $p \hookleftarrow i \rightarrowtail e$ in T-**MSpan**. In terms of T-typed graphs, a graph advice has the structure depicted below, where all squares commute:



Given a T-typed advice $a : p \hookleftarrow i \rightarrowtail e$, the production p is called the *advice pointcut*, i, the *advice interface*, and e, the *advice effect*.

Definition 6 (Graph aspect) Given a graph grammar $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$, we define a *graph aspect* A over $\mathscr{G}$ as a triple $\langle D, t, g \rangle$, where D is a set of T'-typed *graph advices* (see Definition 5), and $t : T \hookrightarrow T'$ and $g : G_0 \hookrightarrow G'_0$ are graph inclusions. The graphs T' and G'_0 are called, respectivelly, the *type graph* and *initial graph* of the aspect A.

Example 2 (Graph aspect) *Figure 5 depicts a graph aspect for the graph grammar of Figure 2, implementing an execution log. The regions* `T` *and* `G0` *refer to the original type graph and initial graph, respectively. The advices* `a1` *and* `a2` *implement the modifications over the original productions as presented in Section 4. The fact that the pointcut is empty makes them match all the original productions, as will be shown later. Figure 6 shows a graph aspect implementing the security policy, which can be applied to* `ConnectGet` *and* `ConnectSend` *productions, since there is an occurrence of the pointcut in that productions.*

Definition 7 (Aspect-oriented graph grammar) An aspect-oriented graph grammar (AOGG) is a pair $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$, where $\mathscr{G}$ is a graph grammar, and Δ is a (possibly empty) finite sequence $[A_1, A_2, \ldots, A_n]$ of graph aspects over $\mathscr{G}$.

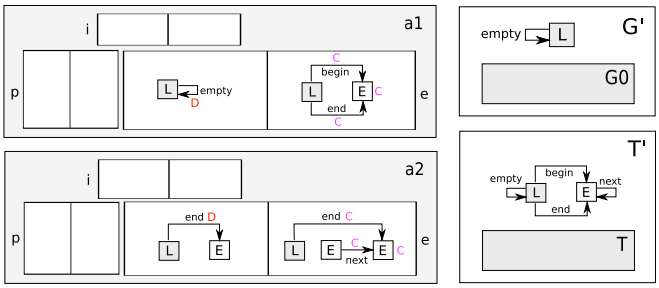


Figure 5: Example of a graph aspect implementing execution log.

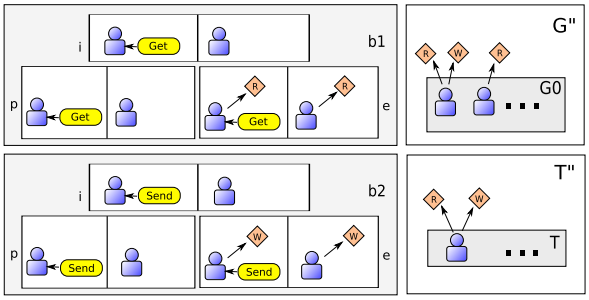
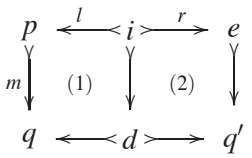


Figure 6: Example of a graph aspect implementing security policy.

The behavior of an AOGG $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$ is given by its *weaved graph grammar*, i.e. the graph grammar resulting from the combination of all aspects in Δ over $\mathscr{G}$. We start by defining how a single advice modifies one production (advice weaving), then how an aspect is weaved to a graph grammar (aspect weaving), and finally how one obtains the weaved graph grammar from a given aspect-oriented graph grammar (AOGG weaving).

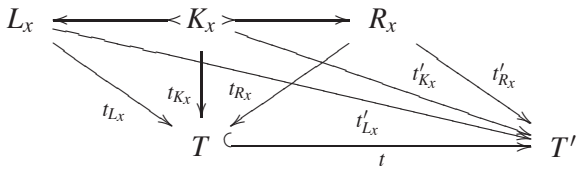
Definition 8 (Advice weaving) Given a T-typed graph production q, a T-typed graph advice $a : p \hookleftarrow i \rightarrowtail e$ and a production monomorphism $m : p \rightarrowtail q$ (called a *production match*), an *advice weaving* from q to q' using a (based on m) exists if and only if the diagram on the right can be constructed, where both squares are pushouts in T-**MSpan**. In this case the advice weaving is denoted by $q \overset{a,m}{\Rightarrow} q'$.



An advice applied to a production rewrites it to another one, if there is an occurrence of its pointcut in the production. Then, the production is updated by a double pushout construction applied componentwise in their left- and right-hand sides and in its interface. Intuitively, the elements that are in the poitcut production (for each graph component L, K and R) but not in the effect production are consumed and those that are in the effect but not in the pointcut are created.

Definition 9 (Aspect weaving) Let $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$ be a graph grammar, and $A = \langle D, T \overset{t}{\hookrightarrow} T', G_0 \overset{g}{\hookrightarrow} G' \rangle$ and aspect over $\mathscr{G}$. Then the *aspect weaving* of A over $\mathscr{G}$, denoted by $W_{\text{ASP}}(\mathscr{G}, A)$, is a graph grammar $\mathscr{G}' = \langle T', P', \pi', G' \rangle$, where P' and π' are calculated as follows:

1. all T-typed productions $x \in range(\pi)$ are *retyped* for T' by composing their respective typing morphisms with the inclusion t. This generates the set $Q^{T'}$ of T'-typed productions:



2. the set Q' is defined as the smallest set which the following holds: for all $y \in Q^{T'}$,

   (a) if does not exist an advice $a \in D$ and a match m such that $y \overset{a,m}{\Rightarrow} y'$, then $y \in Q'$

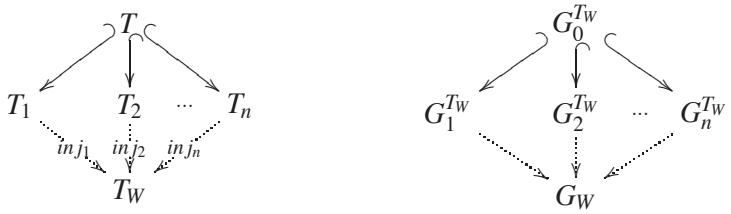
   (b) if $y \overset{a,m}{\Rightarrow} y'$ for some a and m, then $y' \in Q'$

3. The set P' of rule names and the function $\pi' : P' \to Q'$ are chosen arbitrarily, respecting the restriction that π' must be a bijection.

The application of an aspect weaving in a AOGG generates a GG consisting of the type and initial graph of the aspect and all productions obtained by applying all advices based on all possible matches over all productions of the AOGG. The productions that are not updated by any advice are kept in the resulting GG.

Definition 10 (AOGG weaving) Let $\mathscr{A} = \langle \mathscr{G}, \Delta \rangle$ be an AOGG, such that $\mathscr{G} = \langle T, P, \pi, G_0 \rangle$ and $\Delta = [\langle D_i, T \overset{t_i}{\hookrightarrow} T_i, G_0 \overset{g_i}{\hookrightarrow} G_i \rangle \mid 1 \leq i \leq n]$. The weaved graph grammar $\mathscr{G}_W^\Delta$ is calculated as follows:

1. the type graph T_W of $\mathscr{G}_W^\Delta$ is the object of the colimit (in **Graph**) of the diagrams containing all type graph inclusions in Δ, as shown in the diagram below in the left-hand side.



2. in order to relate the initial graphs and productions in all aspects, we need to re-type the original graph grammar $\mathscr{G}$ and all aspects in Δ by composing all their typing morphisms with the respective injections over T_W. This generates the retyped AOGG $\langle \langle T_W, P^{T_W}, \pi^{T_W}, G_0^{T_W} \rangle, \Delta^{T_W} \rangle$, where all type graph inclusions $t_i^{T_W} : T_W \hookrightarrow T_W$ (in all aspects) become identities.

3. the initial graph G_W of $\mathscr{G}_W^\Delta$ is the object of the colimit (in T_W-**Graph**) of the diagram containing all T_W-retyped initial graph inclusions in Δ^{T_W}, as shown in the diagram above in the right-hand side.

4. the graph grammar $\mathscr{G}_W^{\Delta}$ is obtained as the result of $W_{\text{AOGG}}(\langle T_W, P^{T_W}, \pi^{T_W}, G_W\rangle, \Delta^{T_W})$. The operation W_{AOGG} is defined inductively, combing all aspects in Δ^{T_W} according to the order they appear.

$$W_{\text{AOGG}}(\mathscr{G}', []) = \mathscr{G}'$$
$$W_{\text{AOGG}}(\mathscr{G}', [A_1, A_2, \ldots A_n]) = W_{\text{AOGG}}(W_{\text{ASP}}(\mathscr{G}', A_1), [A_2, \ldots, A_n])$$

Finally, the AOGG weaving is done applying all aspects in order of occurrence: the first aspect is applied over the original grammar and the subsequent ones are applied over the grammar resulting of the previous aspect weaving.

The AOGG weaving model has the following characteristics: *i) positive pointcut match:* the pointcut matching is given by a single production monomorphism; *ii) non-reentrant weaving:* our weaving model combines one advice and one rule at most once for every possible match; *iii) deterministic aspect combination:* by using a sequence instead of a set, we enforce a canonical ordering for the aspects in a AOGG. Although these properties allow us to easily express the aspects for our example, they also may not be the most expressive choices. In aspect-oriented languages, usually there is a rather complex *expression language* for defining pointcuts, which also includes negative expressions such as "all methods whose return type is *not* void". Without negative matches, we can not differ created elements from preserved elements in the pointcut, since we can not test their absence from the interface of the production. It would also be of interest to define how pointcuts should be composed in our graph-based setting. Concerning the non-reentrant weaving, this brings both advantages and drawbacks. The positive effect is that non-deleting advices (the ones where the left-hand side is isomorphic to the interface) pose no problem to the weaving, since they will never start a non-terminating rewriting. On the other hand, it may not suffice to describe more complex aspects.

6 Concluding Remarks

One of the first connections between graph rewriting systems and aspect-oriented programming was made in [AL99], where graph rewriting mechanism was proposed has a tool for describing aspects over graph based models. Some proposals, such as the MATA framework [WJ07], follow this principle, characterizing aspect weaving as a special kind of model transformation. Most of these works do not extend the theory of graph rewriting for aspects, employing it as a language for specifying diagram transformations. As far as we know, there is no other formal approach for defining aspects and aspect weaving over graph grammars.

In this work, we addressed the problem of the lack of modularity for crosscutting concerns in graph-grammars, and claimed that aspects for graph grammars are an interesting approach for the modularisation of such requirements. We provided a formal definition for aspects over graph grammars, leading to the definition of aspect-oriented graph grammars (AOGG). We also defined the aspect weaving process that combines all the aspects over the base grammar in an AOGG, resulting in a (weaved) graph grammar. We showed by means of an example how the use of AOGGs may reduce the size of GG-based specifications that must deal with crosscutting concerns.

Our approach differs from others that relate aspects and graph rewriting systems mainly because it propose the definition of aspects over graph grammars, and not graph grammars as rewriting models for aspects. On the technical side, there is still room for improvements on the proposed theory, such as extending the pointcut matching model and defining composition operations for pointcuts and advices. It would be interesting to confirm that this theory holds for other kinds of graph rewriting models, such as attributed graph grammars. Other topics of investigation include the study of aspect interference over the execution of the base grammar and the possible conflicts between aspects.

Acknowledgements: The authors would like to thank the anonymous referees for their helpful comments and suggestions. This work was partially supported by CNPq – Conselho Nacional de Desenvolvimento Científico e Tecnológico.

Bibliography

[AL99] U. Aßmann, A. Ludwig. Aspect Weaving with Graph Rewriting. In Czarnecki and Eisenecker (eds.), *GCSE*. LNCS 1799, pp. 24–36. Springer, 1999.

[C+97] A. Corradini et al. 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, chapter 3, pp. 163–245. World Scientific, River Edge, February 1997.

[HET08] F. Hermann, H. Ehrig, G. Taentzer. A Typed Attributed Graph Grammar with Inheritance for the Abstract Syntax of UML Class and Sequence Diagrams. *ENTCS* 211:261–269, 2008.

[KLM+97] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. marc Loingtier, J. Irwin. Aspect-oriented programming. In *Eropean Conference on Object-Oriented Programming, ECOOP'97*. LNCS 1241, pp. 220–242. Springer, Finland, June 1997.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1. World Scientific, River Edge, February 1997.

[WJ07] J. Whittle, P. K. Jayaraman. MATA: A Tool for Aspect-Oriented Modeling Based on Graph Transformation. In Giese (ed.), *MoDELS Workshops*. LNCS 5002, pp. 16–27. Springer, 2007.

Evolution of Model Transformations by Model Refactoring

Hartmut Ehrig, Karsten Ehrig and Claudia Ermel

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
ehrig@cs.tu-berlin.de, karstene@cs.tu-berlin.de, claudia.ermel@tu-berlin.de

Abstract: Model-to-model transformations between visual languages are often defined by typed, attributed graph transformation systems. Here, the source and target languages of the model transformation are given by type graphs (or meta models), and the relation between source and target model elements is captured by graph transformation rules. On the other hand, refactoring is a technique to improve the structure of a model in order to make it easier to comprehend, more maintainable and amenable to change. Refactoring can be defined by graph transformation rules, too. In the context of model transformation, problems arise when models of the source language of a model transformation become subject to refactoring. It may well be the case that after the refactoring, the model transformation rules are no longer applicable because the refactoring induced structural changes in the models. In this paper, we consider a graph-transformation-based evolution of model transformations which adapts the model transformation rules to the refactored models. In the main result, we show that under suitable assumptions, the evolution leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to a well-known model transformation from statecharts to Petri nets.

Keywords: model transformation, graph transformation, model refactoring

1 Introduction

Model-driven software development (MDD) is a discipline that relies on models and that aims to develop, maintain and evolve software by performing model transformations [BBG05]. The basic idea of model transformations is to more or less automatically derive models of a certain target language from models of a source language, e.g. by mapping the source language components of a domain specific language to Petri nets, where model properties can be analyzed formally.

An intrinsic property of software (and their models) in a real-world environment is their need to evolve. As the model is enhanced, modified and adapted to new requirements, it becomes more and more complex and drifts away from its original design. *Refactoring* [Fow99, MT04], originally used in the industry for source code re-structuring, aims at reducing the software complexity by "changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure" [Fow99]. Recently, approaches for refactoring have been lifted to the more abstract level of design models (*model refactoring*),

supporting in particular the refactoring of UML diagrams like class diagrams, statecharts and activity diagrams [BSF02, SPLJ01].

In this paper we tackle the problem which arises when model refactoring operations are applied to a model (or a modelling language) which is transformed by a model transformation. Problems arise if the refactoring operations induce structural model changes which cannot be handled by the model transformation. In order to solve this problem, we propose a strategy for a systematic evolution of model transformation specifications in accordance to the refactoring operations.

Model transformations between visual languages are conveniently defined in a formal way by typed, attributed graph transformation [EEPT06, MVVK05, Kön05, EE05]. To execute model transformation rules and to check functional properties of model transformations (termination and confluence), the graph transformation engine AGG [AGG] is available.

On the other hand, various approaches exist using graph transformation to provide a formal specification of model refactorings [MTM05, MTR07, GGZ+05]. Basically, a refactoring operation is defined by a set of graph transformation rules typed over the modelling language of the models to be refactored.

In our approach, we consider a construction allowing us to apply the refactoring operation not only to models of the source or target language of a model transformation, but also to the model transformation rules. The approach is based on the work of Parisi-Presicce who defined the transformation of graph grammars in [Par95]. In our main result, we show that under suitable assumptions, such an evolution of the model transformation rules leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to a well-known model transformation from statecharts to Petri nets, when the statechart becomes subject to a refactoring.

The paper is structured as follows: After introducing our case study for refactoring and model transformation in Section 2, we consider the notion of consistency of a model transformation step and a refactoring step in Section 3, where the steps are defined as single rule applications of the respective graph rules to a model state. In Section 4, we extend this basis to sequences of rule applications and state our main result for the consistent evolution of model transformations. Section 5 compares our approach to related work, and in Section 6 we conclude the paper with an outlook to future work. Our technical report [EEE09] contains full proofs for the technical results, discusses possible extensions of our formal framework, and presents additional refactoring operations applied to our sample model transformation.

2 Example: Transforming and Refactoring Statecharts

2.1 Model Transformation *State2PN* from Statecharts to Petri Nets

In this section, we review the model transformation from a simple version of statecharts into Petri nets, given in [EEPT06].

Example 1 (Type Graph of the *SC2PN* Model Transformation) *The statechart type graph TG_S is shown in the left part of Fig. 1 and explicitly introduces several ideas from the area of statecharts that are only implicitly present in the standard UML metamodel (such as state configurations).*

We consider a network of state machines StateMachine. *A single state machine captures the behavior of any object of a specific class by flattening the state hierarchy into* state configurations *and grouping parallel transitions into* steps. *A* Configuration *is composed of a set of* States *that can be active at the same time. A* Step *is composed of non-conflicting* Transitions *(which are, in turn, binary relations between states) that can be fired in parallel. A step between two configurations is* triggered *by a common* Event *for all its transitions. The effect of a step is a set of* Actions.

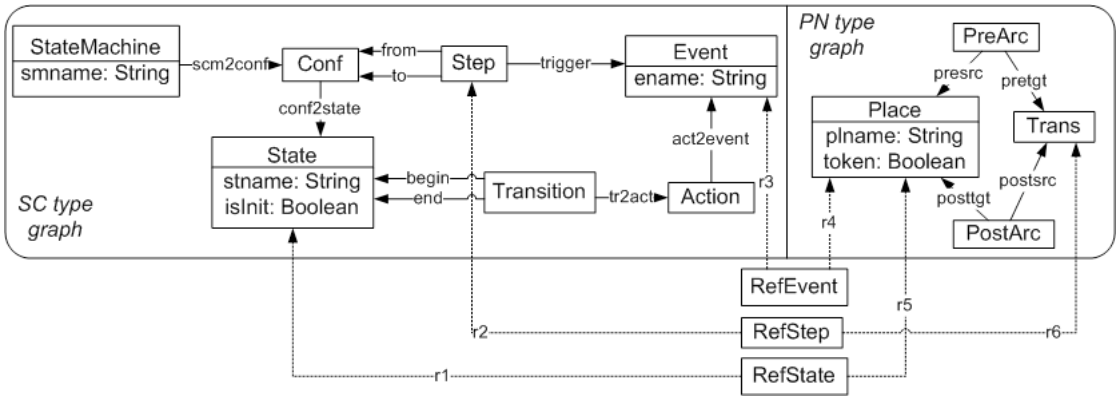


Figure 1: Integration of Attributed Type Graphs for the Model Transformation *SC2PN*

The target modelling language are Petri nets. The Petri net type graph TG_T is shown in the right part of Fig. 1. In fact, we use elementary net systems [Rei85], where each place contains at most one token. *In order to interrelate the source and target modeling languages, we use reference types to construct an integrated attributed type graph, as shown in Fig. 1. For instance, the reference node type* RefState *relates the source type* State *to the target type* Place.

The model transformation from statecharts into Petri nets is fully given by the transformation rules defined in [EEPT06]. In this paper, we concentrate on the rules constructing the integrated model which contains elements of both source and target language, and do not consider explicitly the restriction of the integrated model to the target language of Petri nets.

The main model transformation rules are shown in Fig. 2. Note that we use a shortcut notation for our rules where the left- and right-hand sides of each rule are depicted in one graph. Nodes which exist only in the right-hand side (i.e. they are generated by the rule) are coloured, and their adjacent arcs are also generated by the rule. Moreover, all model transformation rules are *non-deleting*, and each rule has a negative application condition (NAC) which equals the right-hand rule side and prevents the rule to be applied more than once at the same match as before.

Example 2 (*SC2PN* Model Transformation Rules) *Each state in the statechart is transformed to a corresponding place in the target Petri net model, where a token in such a place denotes that the corresponding state is active initially (rules* InitState2Place *and* State2Place*). A separate place is generated for each valid event in rule* Event2Place. *Each step in the statechart is transformed into a Petri net transition (rule* Step2Trans*). Since the Petri net should simulate how to exit and enter the corresponding states in the statechart, input and output arcs of the transition have to*

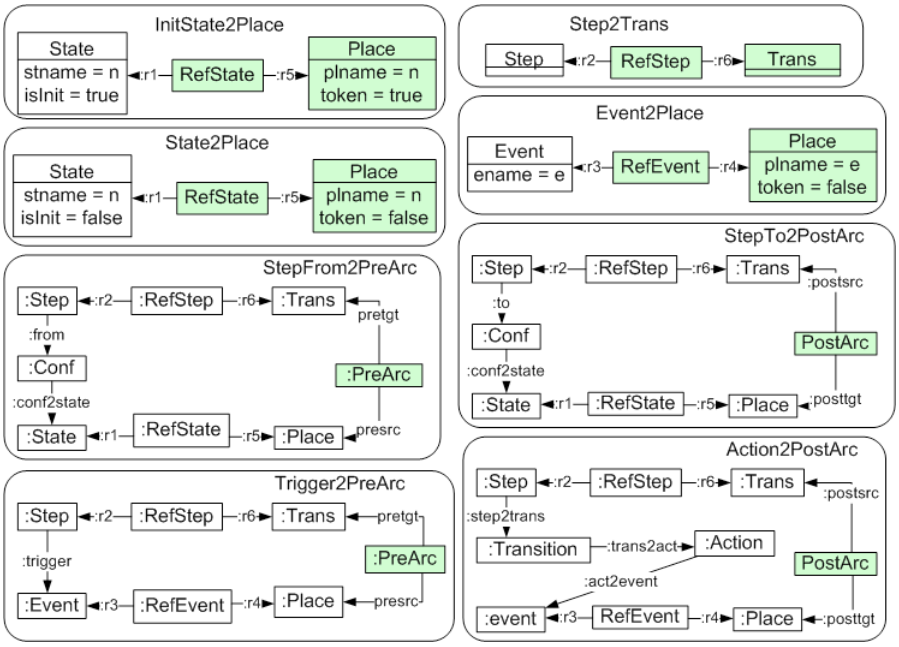


Figure 2: Model Transformation Rules for *SC2PN*

be generated accordingly (see rules StepFrom2PreArc *and* StepTo2PostArc*). Furthermore, firing a transition should consume the token of the trigger event (rule* Trigger2PreArc*), and should generate tokens on (the places related to) the target event indicated as the action (*Action2PostArc*).*

2.2 Refactoring Operation for Statecharts

Not all possible model refactorings make it necessary to adapt the model transformation rules. One well-known refactoring is the so-called *Pull-Up-Attribute* which removes an attribute type from all subtypes of a supertype and adds the attribute type to the common supertype, instead. This kind of refactoring (changing only the inheritance relation of a meta model) does not induce changes on the instance models which remain valid as they are. Hence, model transformation rules remain applicable after the refactoring, too. On the other hand, there are refactorings which induce structural changes of the instance models. This kind of critical refactorings make an adaption of the model transformation rules necessary and are considered here.

We present a refactoring operation for statecharts, where the representation of initial states is changed from an attribute to a new node type. The motivation for this refactoring is to simplify the definition of a concrete syntax for statecharts, where node types are mapped to figures. We use this example later on to illustrate the evolution of a model transformation from statecharts to Petri nets when such a model refactoring on statecharts has taken place. Further refactoring operations which induce model transformation adaptation are considered in our technical report [EEE09] and concern e.g. the renaming of node or attribute types.

Example 3 (Refactoring Operation for Statecharts) *Let the type graph for statecharts be the one*

depicted in the left part of Fig. 1. For the definition of our refactoring operation, this type graph is extended by two new node types Initial *and* Normal*, which are linked to the* State *node type. The refactoring operation* markState *is modelled by the two graph rules in Fig. 3, where an* Initial *node is added to a state whose* isInit *attribute is true (rule* markInitial*), and, vice versa, a* Normal *node is added to a state whose* isInit *attribute is false (rule* markNormal*). Note that the* isInit *attribute is deleted by the refactoring rules.*

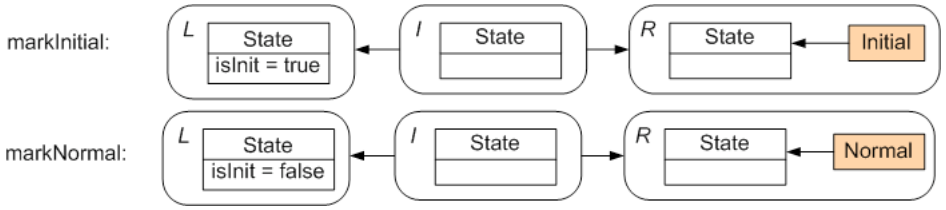


Figure 3: Rules for Statechart Refactoring Operation markState

3 Consistency of Stepwise Model Transformation and Refactoring

In this section, we give the formal definition how to adapt a model transformation to a refactoring operation (Def. 1) and consider the relation of a model transformation step and a refactoring step in Lemma 1.

A model transformation rule $p_1 \in P$ is adapted to a refactoring (given by refactoring rule $q \in Q$), by applying refactoring rule q to all rule graphs of model transformation rule p_1, resulting in the adapted model transformation rule p_2. Note that the construction of applying rules to rules is based on [Par95] and extended to rules with NACs in [EE08].

Definition 1 (Application of Q-Productions to P-Productions) Production $q = (L_q \leftarrow I_q \rightarrow R_q)$ is applicable to $p_1 : L_1 \rightarrow R_1$ with $nac_1 : L_1 \rightarrow N_1$ leading to $p_2 : L_2 \rightarrow R_2$ with $nac_2 : L_2 \rightarrow R_2$ if we have $m : L_q \rightarrow L_1$ leading to the following DPOs, written $p_1 \overset{q,m}{\Longrightarrow} p_2$, where all morphisms are injective:

$$\begin{array}{ccccc} L_q & \leftarrow & I_q & \rightarrow & R_q \\ {\scriptstyle m}\downarrow & (1) & \downarrow & (2) & \downarrow \\ L_1 & \leftarrow & D & \rightarrow & L_2 \end{array} \qquad \begin{array}{ccccc} L_1 & \leftarrow & D & \rightarrow & L_2 \\ {\scriptstyle p_1}\downarrow & (3) & \downarrow & (4) & \downarrow{\scriptstyle p_2} \\ R_1 & \leftarrow & E & \rightarrow & R_2 \end{array} \qquad \begin{array}{ccccc} L_1 & \leftarrow & D & \rightarrow & L_2 \\ {\scriptstyle nac_1}\downarrow & (5) & \downarrow & (6) & \downarrow{\scriptstyle nac_2} \\ N_1 & \leftarrow & F & \rightarrow & N_2 \end{array}$$

Example 4 (Applying a Refactoring Rule to a Model Transformation Rule) *Fig. 4 shows the application of refactoring rule* markInitial *from Fig. 3 to model transformation rule* InitState2Place *from Fig. 2, according to Def. 1.*

General Assumption: Let a visual modeling language VL be given by all models (graphs) typed over a type graph. As basis for model transformation and refactoring, we assume a com-

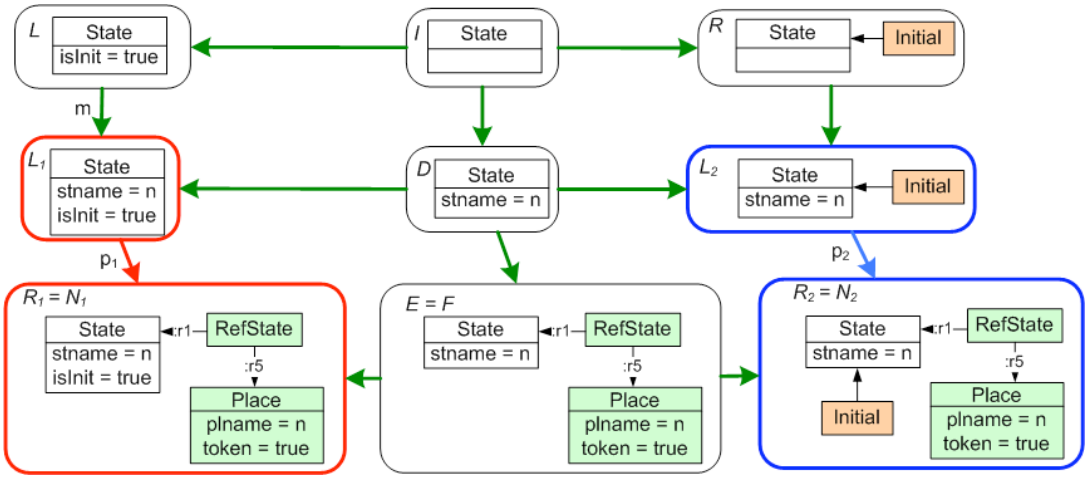


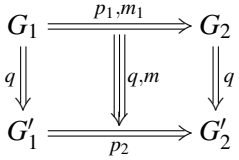
Figure 4: Applying Refactoring Rule *markInitial* to Model Transformation Rule *InitState2Place*

mon type graph TG which includes the type graphs for the source and the target languages of the model transformation, as well as the extended type graph for the refactoring. Let $(MT,P): VL_1 \to VL_2$ be a model transformation (with P non-deleting with NACs), $(MR_1,Q): VL_1 \to VL_1^*$ be a model refactoring (with Q bijective on nodes, without NACs), and $(MR,Q): P \to P^*$ be a model refactoring of rules according to Def. 1, and let TG be the common type graph for VL_1, VL_2, VL_1^*, P and Q. All over, we assume injective rules and injective matches. For simplicity, we do not handle the corresponding refactorings of the different type graphs in this paper.

The following lemma shows the compatibility of a model transformation step transforming source model $G_1 \in VL_1$ into target model $G_2 \in VL_2$ by applying rule $p_1 \in P$, and a refactoring step, changing $G_1 \in VL_1$ to $G_1' \in VL_1^*$ by applying rule $q \in Q$, where the refactored source model G_1' is transformed by the refactored model transformation rule $p_2 \in P^*$, resulting in model G_2'.

Lemma 1 (Direct Transformation and Refactoring Steps)

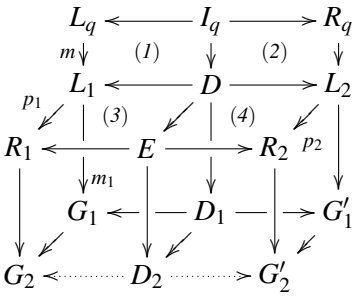
Given $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ *with* $p_1 \in P$ *and* $p_1 \overset{q,m}{\Longrightarrow} p_2$ *with* $q \in Q$, *we have* $G_1 \overset{q}{\Longrightarrow} G_1', G_2 \overset{q}{\Longrightarrow} G_2'$ *and* $G_1' \overset{p_2}{\Longrightarrow} G_2'$.



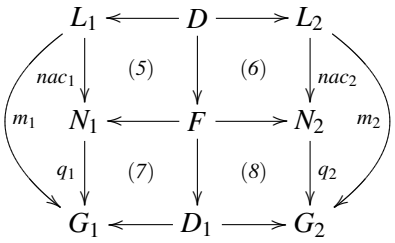
Proof. Given $p_1 : L_1 \to R_1$ with $nac_1 : L_1 \to N_1$, we obtain $p_2 : L_2 \to R_2$ with $nac_2 : L_2 \to N_2$ with pushouts $(1)-(6)$ as in Def. 1.

Furthermore, we obtain from $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ the pushout in the left square in the diagram below, with pushouts $(1)-(4)$, as shown in Def. 1. Next, we construct D_1 as pushout complement in the left back square – using that $I_q \to L_q$ and hence $D \to L_1$ is bijective on nodes, which implies that the gluing condition is satisfied – and then G_1' as pushout in the right back square. Then, D_2 and G_2' are constructed as pushouts in the middle and right square, respectively, leading to induced morphisms $D_2 \to G_2$ and $D_2 \to G_2'$ such that all squares commute.

In the left cube, the left, right, back and top squares are pushouts by construction. This implies that also the front and bottom squares are pushouts by pushout composition and decomposition. Hence, all squares of the left cube and, similarly, also of the right cube are pushouts. This leads to the DPOs of the direct transformations $G_1 \overset{q}{\Longrightarrow} G_1'$, $G_2 \overset{q}{\Longrightarrow} G_2'$ and $G_1' \overset{p_2,m_2}{\Longrightarrow} G_2'$.



It remains to show that $m_2 : L_2 \to G_1'$ satisfies $nac_2 : L_2 \to N_2$, defined by pushouts (5) and (6) in Def. 1, using that $m_1 : L_1 \to G_1$ satisfies $nac_1 : L_1 \to N_1$. Assume that $m_2 \not\models nac_2$, then we have injective $q_2 : N_2 \to G_1'$ with $m_2 = q_2 \circ nac_2$. Pushout-pullback decomposition allows us to construct pushouts (7) and (8) from the outer DPO, leading to an injective q_1 with $q_1 \circ nac_1 = m_1$. This contradicts $m_1 \models nac_1$. Hence, we have $m_2 \not\models nac_2$.



□

Example 5 (Model Transformation Step and Refactoring Step) *Fig. 5 shows the diagram relating the source and target model of the model transformation step and the changed source and target models of the refactoring step where p_1 and p_2 are given in Fig. 4.*

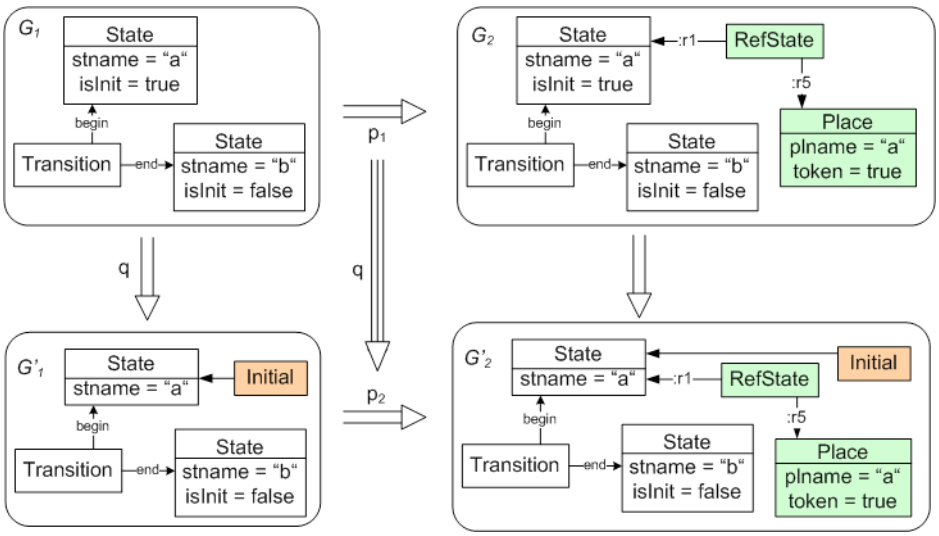


Figure 5: Relating Refactoring and Model Transformation Step

4 Sequences of Rule Applications

In this section, we extend our result from Lemma 1 on the compatibility of model transformation and refactoring steps to sequences with rule sets Q, P and P^* according to the general assumption in Section 3. Our main result in Thm. 1 states that under certain compatibility assumptions which can be decided at rule level, a complete model transformation sequence can be refactored, leading to a compatibility diagram similar to the one in Lemma 1, but where now sequences of rule applications are considered instead of single steps. For the proof of Thm. 1, we require compatibility of model transformation and refactoring rules, defined in Def. 2. Furthermore, we use a lemma stating that a terminating transformation at rule level leads to a terminating transformation at model level, as well (Lemma 2). We say that graph G (resp. rule p^*) is terminal wrt. Q if no rule $q \in Q$ can be applied to G (resp. p^*).

Definition 2 (Q– (P, P^*–) Compatibility) Q is (P, P^*)-compatible if we have:

1. *Independence Compatibility*:
   Given terminal p^* wrt. Q, $G_1 \overset{p^*}{\Longrightarrow} G_2$ and $G_1 \overset{q}{\Longrightarrow} G_1'$ (resp. $G_2 \overset{q}{\Longrightarrow} G_2'$) with $p^* \in P^*$ and $q \in Q$, we have parallel (resp. sequential) independence including NACs of $G_2 \overset{p^*}{\Longleftarrow} G_1 \overset{q}{\Longrightarrow} G_1'$ (resp. $G_1 \overset{p^*}{\Longrightarrow} G_2 \overset{q}{\Longrightarrow} G_2'$ for terminal G_1 wrt. Q).
2. *Termination Compatibility*:
   For each G terminal wrt. P and $G \overset{Q!}{\Longrightarrow} G^*$, also G^* is terminal wrt. P^*, where $Q!$ means to apply rules in Q as long as possible.

Example 6 (Compatibility of the *SC2PN* Model Transformation and the *markState* Refactoring) *We continue our case study introduced in Examples 1 - 5. Fig. 6 shows the refactored model transformation rules* InitState2Place *and* State2Place. *Note that all other model transformation rules from Fig. 2 remain unchanged because the refactoring rules cannot be applied to them.*

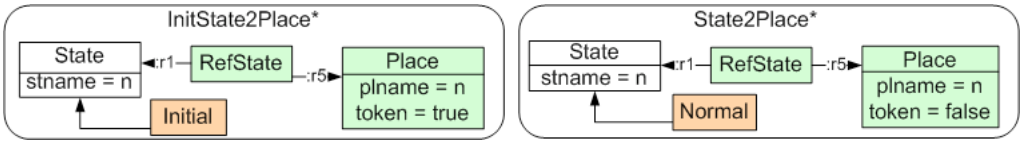


Figure 6: Refactored Model Transformation Rules for *SC2PN*

We now show that we have independence and termination compatibility as defined in Def. 2:

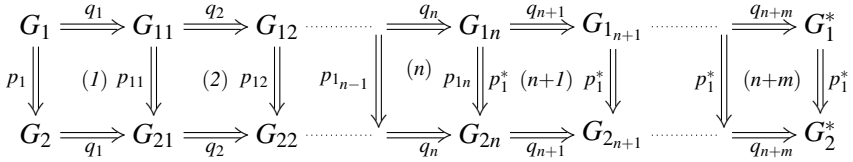
1. *Independence compatibility: Given terminal p^* wrt. Q and $q \in Q$ with $G_1' \overset{q}{\Longleftarrow} G_1 \overset{p^*}{\Longrightarrow} G_2$, we have parallel independence because the matches can only overlap in* State *which is a gluing point for both rules. Moreover, we have NAC compatibility because the nodes and edges generated by the rules in Q are of different types from those generated by p^*. Analogously, we can show sequential independence.*
2. *Termination compatibility: Given terminal G wrt. P and $G \overset{Q!}{\Longrightarrow} G^*$, then the markState refactoring rules have been applied to all initial state nodes occuring in a rule in P, and to*

all initial state nodes in G. So there is no match from a rule $p^* \in P^*$ to G^* where the NAC of p^* would not prevent its application, and hence, G^* is terminal wrt. P^*.

The following lemma states that a terminating transformation at rule level leads to a terminating transformation at model level.

Lemma 2 (Direct Transformation and Terminating Refactoring) *Given $G_1 \overset{p_1,m_1}{\Longrightarrow} G_2$ with $p_1 \in P$ and $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ terminating, we construct $G_1^* \overset{p_1^*}{\Longrightarrow} G_2^*$ and terminating $G_1 \overset{Q!}{\Longrightarrow} G_1*$ and $G_2 \overset{Q!}{\Longrightarrow} G_2^*$, provided that we have termination of (MR_1, Q) and independence compatibility (see Def. 2.1).*

Proof. Let $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ terminate via $(q_1, .., q_n)$ and $G_1 \overset{p_1}{\Longrightarrow} G_2$, then we apply Lemma 1 in each step, leading to diagrams (1) – (n).



If G_{1_n} is not yet terminal wrt. Q, we can extend $G_1 \overset{*}{\Longrightarrow} G_{1_n}$ by $G_{1_n} \overset{Q!}{\Longrightarrow} G_1^*$ via $(q_{n+1}, .., q_{n+m})$ with terminal G_1^* wrt. Q, using termination of (MR_1, Q). Parallel independence of $G_{1_n} \overset{p_1^*}{\Longrightarrow} G_{2_n} \overset{q_{n+1}}{\Longrightarrow} G_{1_{n+1}}$ according to independence compatibility allows us to construct diagram $(n+1)$ by the Local Church-Rosser Theorem with NACs, and, similarly, diagrams $(n+2), .., (n+m)$. But now also $G_2 \overset{*}{\Longrightarrow} G_2^*$ via $(q_1, .., q_{n+m})$ is terminating because $G_2^* \overset{q}{\Longrightarrow} G_2^{**}$ would imply $G_1^* \overset{q}{\Longrightarrow} G_1^{**}$ by sequential independence of $G_1^* \overset{p_1^*}{\Longrightarrow} G_2^* \overset{q}{\Longrightarrow} G_2^{**}$ according to independence compatibility. □

Now we state our main result saying that under certain compatibility assumptions which can be decided at rule level, a complete model transformation sequence can be refactored, leading to a compatibility diagram similar to the one in Lemma 1, but where now sequences of rule applications are considered instead of single steps.

Theorem 1 (Evolution of Model Transformations by Model Refactoring) *Given a model transformation $(MT, P) : VL_1 \to VL_2$ (with P nondeleting with NACs), a model refactoring $(MR_1, Q) : VL_1 \to VL_1^*$ (with Q bijective on nodes, without NACs), and a model refactoring $(MR, Q) : P \to P^*$ according to Def. 1 with common type graph TG for VL_1, VL_2, VL_1^*, P and Q, such that*

1. *$(MT, P), (MR_1, Q)$ and (MR, Q) are terminating,*
2. *Q is locally confluent,*
3. *Q is (P, P^*)-compatible (see Def. 2),*

then we have VL_2^* typed over TG with extended

4. *terminating model refactoring $(MR_2, Q) : VL_2 \to VL_2^*$, and*
5. *terminating model transformation $(MT^*, P^*) : VL_1^* \to VL_2^*$ with*
6. *commutativity of the diagram to the right.*

$$\begin{array}{ccc} VL_1 & \xrightarrow{(MT,P)} & VL_2 \\ {\scriptstyle (MR_1,Q)}\downarrow & & \downarrow{\scriptstyle (MR_2,Q)} \\ VL_1^* & \xrightarrow[(MT^*,P^*)]{} & VL_2^* \end{array}$$

Proof. Given $G_1 \in VL_1$, $G_1 \overset{Q!}{\Longrightarrow} G_1^*$, $G_1 \overset{P!}{\Longrightarrow} G_2$ via $(p_1,..,p_n)$, and $p_i \overset{Q!}{\Longrightarrow} p_i^*$ for $(i = 1,..,n)$, where termination is given by assumption 1. Now, we use Lemma 2 above to construct the following sequence $(1) - (n)$:

$$\begin{array}{ccccccccc} G_1 & \overset{p_1}{\Longrightarrow} & G_{11} & \overset{p_2}{\Longrightarrow} & G_{12} & \Longrightarrow & \cdots\cdots & \overset{p_n}{\Longrightarrow} & G_{1_n} = G_2 \\ {\scriptstyle Q!}\Downarrow & (1) & \Downarrow{\scriptstyle Q!}\ \Downarrow{\scriptstyle Q!} & (2) & \Downarrow{\scriptstyle Q!}\ \Downarrow{\scriptstyle Q!} & & & (n) & \Downarrow{\scriptstyle Q!} \\ G_1^* & \underset{p_1^*}{\Longrightarrow} & G_{11}^* = G_{11}^+ & \underset{p_2^*}{\Longrightarrow} & G_{12}^* = G_{12}^+ & \Longrightarrow & \cdots\cdots & \underset{p_n^*}{\Longrightarrow} & G_{1n}^* = G_2^* \end{array}$$

Note that $G_{11} \overset{Q!}{\Longrightarrow} G_{11}^*$ and $G_{11} \overset{Q!}{\Longrightarrow} G_{11}^+$ are in general defined by different Q-sequences induced by $p_1 \overset{Q!}{\Longrightarrow} p_1^*$ and $p_2 \overset{Q!}{\Longrightarrow} p_2^*$, respectively. But termination and local confluence of Q by assumptions 1 and 2 implies unique normal forms and hence, $G_{11}^* = G_{11}^+$ (up to isomorphism), and similarly $G_{12}^* = G_{12}^+, .., G_{1_{n-1}}^* = G_{1_{n-1}}^+$. Finally, $G_1^* \Longrightarrow G_2^*$ via $(p_1^*,..,p_n^*)$ is terminating by termination compatibility according to assumption 3. Hence, we have diagram (A) for each $G_1 \in VL_1$, with $G_2 \in VL_2, G_1^* \in VL_1^*$ and $G_2^* \in VL_2^*$, where $VL_2^* = \{G_2^* | \exists G_2 \in VL_2 : G_2 \overset{Q!}{\Longrightarrow} G_2^*\}$, which implies terminating $(MR_2, Q) : VL_2 \to VL_2^*$ and $(MT^*, P^*) : VL_1^* \to VL_2^*$ with commutativity of diagram (B):

$$\begin{array}{ccc} G_1 & \overset{P!}{\Longrightarrow} & G_2 \\ {\scriptstyle Q!}\Downarrow & (A) & \Downarrow{\scriptstyle Q!} \\ G_1^* & \underset{P^*!}{\Longrightarrow} & G_2^* \end{array} \qquad \begin{array}{ccc} VL_1 & \xrightarrow{(MT,P)} & VL_2 \\ {\scriptstyle (MR_1,Q)}\downarrow & (B) & \downarrow{\scriptstyle (MR_2,Q)} \\ VL_1^* & \xrightarrow[(MT^*,P^*)]{} & VL_2^* \end{array}$$

□

Remark 1 *If (MT,P) and (MT^*,P^*) are not functional, then commutativity of diagram (B) means that for each $G_1 \overset{P!}{\Longrightarrow} G_2$ exists a corresponding $G_1^* \overset{P^*!}{\Longrightarrow} G_2^*$ such that diagram (A) commutes.*

Example 7 (Refactoring of the *SC2PN* Model Transformation) *In order to apply Theorem 1, we have to show the* required properties*:*

1. *The original model transformation $(MT,P) = SC2PN$ is terminating by [EEPT06]. The refactoring operation* markState *is terminating, because rules* markInitial *and* markNormal

delete one attribute each, and therefore each rule is only applicable once at a match to a State *node. The refactoring of the model transformation rules* (MR,Q) *is terminating, because at most one rule* $q \in Q$ *with* $Q = \{markInitial, markNormal\}$ *is applicable once.*

2. *The refactoring rules in* Q *are locally confluent: rules* markInitial *and* markNormal *are parallel independent because their left-hand sides overlap in gluing point* State *only. Moreover, there is at most one match of* markInitial *resp.* markNormal *at the same* State.

3. Q *is* (P,P^*)*-compatible as shown in Example 6.*

According to the application of Theorem 1, we obtain the terminating model refactoring (MR_2,Q)*, and the terminating model transformation* (MT^*,P^*) *for each possible statechart which is transformed to a Petri net using* (MT,P)*, i.e. the rules in* P*, and which is refactored using the refactoring* (MR_1,Q)*, i.e. the rules in* Q*. As result we have the commutative diagram below, where* VL_1 *is the visual language of statecharts,* VL_1^* *is the statechart language, extended by the new node types* Initial *and* Normal *for the* markState *refactoring,* VL_2 *is the integrated language of statecharts and Petri nets (defined by the type graph in Fig. 1), and* VL_2^* *is the integrated language of extended statecharts and Petri nets.*

$$\begin{array}{ccc} VL_1 & \xrightarrow{(MT,P)} & VL_2 \\ {\scriptstyle (MR_1,Q)}\downarrow & & \downarrow{\scriptstyle (MR_2,Q)} \\ VL_1^* & \xrightarrow[(MT^*,P^*)]{} & VL_2^* \end{array}$$

5 Related Work

Refactoring of information systems is a common technique for software evolution through transformation [LKPS06, MT04]. Automated transformation within domain specific languages including version support has been considered in [Bel07, GSA07].

Refactoring by graph transformation rules plays an important role for software system refactoring by providing a graphical way for rule definition and an underlying algebraic framework for analyzing refactoring dependencies [MTR07] and to assure behavior preservation in model refactoring using transformations with borrowed contexts [RLK+08]. Moreover suitable verification techniques are available, e.g. architectural refactoring by rule extraction [BHE08].

From a technical point of view, in this paper we apply model refactoring rules Q deleting (on edges) to non-deleting transformation rules P, which is in some sense dual to the *S2A*-construction of animation rules P_A from simulation rules P_S in [EE08], where non-deleting rules Q are applied to deleting rules P_S. Both kinds of rule transformations are based on the construction in [Par95] but have been extended by NACs and by the possibility to transform generated or deleted rule objects, as well.

Within the Eclipse Modeling Framework [EMF08] model refactoring has already been implemented using graph transformation concepts [BEK+06]. While software refactoring is a common technique, a general theory for refactoring of model transformations has still been missing.

6 Conclusion

In this paper, we consider a graph-transformation-based evolution of model transformations which adapts model transformation rules to refactored models. In the main result, we show

that under suitable assumptions, the evolution leads to an adapted model transformation which is compatible with refactoring of the source and target models. In a small case study, we apply our techniques to refactor a model transformation from statecharts to Petri nets. Further refactoring examples and an extension of the presented theory to model refactoring rules with NACs are given in our technical report [EEE09] which is available online.

As future research, we intend to consider refactoring operations at type graph level based on our approach on transformations of type graphs with inheritance [EEH09]. Moreover, up to now, we have studied model transformations resulting in an integrated model which contains both source and target language elements. A restriction to the target model presently means that we get the same target model as before refactoring the source model and the model transformation rules. Additionally, we plan to handle target language refactorings analogously to refactorings of the source language.

Bibliography

[AGG] AGG Homepage. http://tfs.cs.tu-berlin.de/agg.

[BBG05] S. Beydeda, M. Book, V. Gruhn (eds.). *Model-Driven Software Development*. Springer-Verlag, Heidelberg, 2005.

[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. In *Proc. Model Driven Engineering Languages and Systems (MoDELS'06)*, 2006.

[Bel07] P. Bell. Automated Transformation of Statements within Evolving Domain Specific Languages. In *Proc. Workshop on Domain-Specific Modeling (OOPSLA'07)*. Comp. Science Reports TR-38, University of Jyväskylä, Finland, 2007.

[BHE08] D. Bisztray, R. Heckel, H. Ehrig. Verification of Architectural Refactorings by Rule Extraction. In *Proc. Fundamental Approaches to Software Engineering (FASE'08)*. LNCS 4961, pp. 347–361. Springer Verlag, 2008.

[BSF02] M. Boger, T. Sturm, P. Fragemann. Refactoring Browser for UML. In *Proc. Conf. on eXtreme Programming and Flexible Processes in Software Engineering*. pp. 77–81. 2002.

[EE05] H. Ehrig, K. Ehrig. Overview of Formal Concepts for Model Transformations based on Typed Attributed Graph Transformation. In *Proc. Workshop on Graph and Model Transformation (GraMoT'05)*. ENTCS 152. Elsevier Science, 2005.

[EE08] H. Ehrig, C. Ermel. Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation. In *Proc. Conf. on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 194–210. Springer Verlag, 2008.

[EEE09] H. Ehrig, K. Ehrig, C. Ermel. Evolution of Model Transformations by Model Refactoring: Long Version. Technical Report 2009-04, Fak. IV, TU Berlin, 2009. http://www.eecs.tu-berlin.de/menue/forschung/forschungsberichte/2009

[EEH09] H. Ehrig, C. Ermel, F. Hermann. Transformation of Type Graphs with Inheritance for Ensuring Security in E-Government Networks. In *Proc. Fundamental Aspects of Software Engineering (FASE'09)*. LNCS. Springer Verlag, 2009.

[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

[EMF08] Eclipse Consortium. Eclipse Modeling Framework (EMF) – Version 2.4. 2008. http://www.eclipse.org/emf.

[Fow99] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[GGZ+05] L. Grunske, L. Geiger, A. Zündorf, N. Van Eetvelde, P. Van Gorp, D. Varro. Using Graph Transformation for Practical Model Driven Software Engineering. In *Model-driven Software Development*. pp. 91–118. Springer, 2005.

[GSA07] G. de Geest, A. Savelkoul, A. Alikoski. Building a framework to support Domain Specific Language evolution using Microsoft DSL Tools. In *Proc. Workshop on Domain-Specific Modeling (OOPSLA'07)*. Comp. Science Reports TR-38, University of Jyväskylä, Finland, 2007.

[Kön05] A. Königs. Model Transformation with Triple Graph Grammars. In *Model Transformations in Practice. Proc. Satellite Workshops of MODELS'05*. 2005.

[LEPO08] L. Lambers, H. Ehrig, U. Prange, F. Orejas. Embedding and Confluence of Graph Transformations with Negative Application Conditions. In *Proc. Conf. on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 162–177. Springer Verlag, 2008.

[LKPS06] M. Löwe, H. König, M. Peters, C. Schulz. Refactoring Information Systems. In Favre et al. (eds.), *Proc. Workshop on Software Evolution through Transformations: Embracing the Chance (SeTra 2006)*. Volume 3. EC-EASST, 2006.

[MT04] T. Mens, T. Tourwé. A Survey of Software Refactoring. *Transactions on Software Engineering* 30(2):126–139, February 2004.

[MTM05] T. Mens, G. Taentzer, D. Müller. Model-Driven Software Refactoring. In Rech and Bunse (eds.), *Model-Driven Software Development: Integrating Quality Assurance*. Pp. 170–203. Idea Group Inc., 2005.

[MTR07] T. Mens, G. Taentzer, O. Runge. Analysing Refactoring Dependencies Using Graph Transformation. *Software and System Modeling* 6(3):269–285, 2007.

[MVVK05] T. Mens, P. Van Gorp, D. Varrò, G. Karsai. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *Proc. Workshop on Graph and Model Transformation*. ENTCS 152, pp. 143–159. Elsevier Science, 2005.

[Par95] F. Parisi-Presicce. Transformation of Graph Grammars. In *Proc. Workshop on Graph Grammars and their Application to Computer Science*. LNCS 1073, Springer Verlag, 1996.

[Rei85] W. Reisig. *Petri Nets: An Introduction*. EATCS Monographs on Theoretical Computer Science 4. Springer Verlag, 1985.

[RLK+08] G. Rangel, L. Lambers, B. König, H. Ehrig, P. Baldan. Behavior Preservation in Model Refactoring using DPO Transformations with Borrowed Contexts. In *Proc. Conf. on Graph Transformation (ICGT'08)*. LNCS 5214. Springer Verlag, 2008.

[SPLJ01] G. Sunyé, D. Pollet, Y. LeTraon, J.-M. Jézéquel. Refactoring UML Models. In *Proc. UML 2001*. LNCS 2185, pp. 134–138. Springer-Verlag, 2001.

Resource-based enactment and adaptation of workflows from activity diagrams

Paolo Bottoni[1], **Andrea Saporito**[1]

[1] bottoni@di.uniroma1.it, anersap@tin.it,
http://w3.uniroma1.it/dipinfo/scheda_docente.asp?cognome=Bottoni&nome=Paolo
Department of Computer Science - "Sapienza" - University of Rome, Italy

Abstract: Workflow management deals with different types of dependencies among tasks, in particular data- and policy-driven. The ability to reason on dependencies of different types allows workflow designers to consider different alternatives, or to define customized flows, reducing non-determinism. We propose a resource-centered view, in which both data-dependency between tasks and plan-dependent ordering of tasks are expressed as production and consumption of resources. This view is translated into a rule-based formalism, expressed in terms of multi-set rewriting for workflow enactment. In turn, rules are themselves seen as resources, so that they are prone to the same rewriting process, in order to redefine process schemas. We show how workflows expressed as activity diagrams can be translated to the proposed formalism, exploiting enforced generative patterns applied to triple graph grammars, and how redefinition of workflow processes can occur through typical patterns of adaptation. We also discuss possible concrete syntaxes for the obtained rules.

Keywords: Workflow, Activity Diagrams, Resources, Multiset rewriting

1 Introduction

Workflow specifications increasingly have visual representations, either in some domain specific language [Swe94], or exploiting general purpose languages for process specification, typically Petri nets [EKR95, AM00a, AM00b, AH01, AB02]. In general, these diagrammatic notations have to provide a precise syntax and semantics in order to allow the specification of correct workflows and their translation to some enactment mechanisms.

The increasing popularity of UML, and the introduction of the action semantics, have made activity diagrams a suitable notation for the precise specification of workflows [Dt01]. With respect to Petri nets, activity diagrams offer the advantage of making the existence of distinct control and data flows explicit, and of making parallelism more apparent, through the use of *fork* and *join* nodes, thus gaining in expressivity. Activity diagrams also offer a more widely known and general language for specification, without the need to acquire additional competence in some language, and favoring interoperability and integration of independent specifications. On the other hand, some aspects of the semantics of activity diagrams are not completely defined and some syntactic variants are still allowed. For example, one can enforce pairing and correct nesting of *fork-join* or *choice-merge* nodes, or allow several forked sequences to have independent terminations or to be joined on single nodes. In this paper, we adopt a version of activity

diagrams suitable for workflows, by assuming the existence of a single initial activity and a single final activity, as in WF-nets. We propose a translation mechanism associating multiset rewriting rules with activity nodes, according to an execution model where activities are defined by the (multi)sets of resources they produce or consume. Separate translation processes can be defined for control and data flows. This offers greater opportunities for independent reasoning on data dependencies and synchronisation policies. For example, it could be possible to explore the parallelism allowed by causal data dependencies and to assess the compatibility of a control policy with the induced partial order. Moreover, data and control rules can be formally composed, or be kept separated. The translation is proposed towards an abstract syntax for multiset rewriting, to which different concrete syntaxes, visual or textual, can be associated.

Under the additional assumption that all parallel activities are defined by paired and correctly nested *fork* and *join* nodes, and that each *fork* (*join*) node has only two outgoing (incoming) edges, a mechanism for dynamic reconfigurations of workflows can be defined, where workflow changes are immediately reflected by changes in the set of rules. The class of activity diagrams which can be manipulated in this way corresponds to *well-structure workflows* [KtB00].

The translation mechanism exploits triple graph grammars to establish the correspondence between activity nodes and rules, while the reconfiguration process is based on the adoption of a view of rules as resources on their own, subject to specific transformation processes.

The rest of the paper proceeds as follows. We explore related work on the use of activity diagrams as workflow specifications and on adaptation processes in Section 2, and provide some formal background on the types of rewriting involved in the paper in Section 3. Section 4 discusses the triple metamodel relating activity diagrams and multiset rewriting rules, and illustrates the translation process, while Section 5 presents the basic mechanisms for coherent modification of diagrams and rules. Finally, Section 6 illustrates two possible concrete syntaxes for the multiset rewriting model, and Section 7 draws conclusions and points to related work.

2 Related work

The use of activity diagrams as a way to specify workflows has been illustrated in [Dt01], showing how some interesting workflow patterns can be captured by them, but also pointing to the limitations of the then current definition of activities as a submodel of state machines.

A formal operational semantics for activity diagrams, still in the UML 1.4 version, is given in terms of Abstract State Machines in [KLN+05], extending diagrams with timing information. In this paper we do not consider the timing information and focus on a unifying concept of resource to express pre- and post-conditions of activities.

In a series of papers, van der Aalst *et al.* propose several patterns for control [AHKB03] and data [RHEA05] flows and for usage of resources [RvtE05] in workflows, specifying such patterns thorugh Coloured Petri Nets. They do not deal with composition of patterns related to different aspects, while we propose to treat it exploiting pushouts as described in [BMWY08].

A distinct advantage in the use of activity diagrams with respect to Petri nets for the expression of workflows is pointed at in [EW03], with respect to the possibility of opening the workflow to external signals. Our approach can indeed be augmented with the definition of specific communication resources, related to the presence of signals.

The approach presented here only considers an abstract view of transitions, so that it can be mapped to specific rule-based languages, with suitable translators. A quite straightforward translation can be devised to the WIPPOG language [BDD+04]. WIPPOG provides an operational semantics and an executable language, based on production and consumption of resources. It has been used to map different diagrammatic languages, based on some notion of transformation, to a common language, thus allowing the interoperability of diagrammatic transformations expressed with different notations. WIPPOG rules express rewriting of multisets of resources, distinguishing between resources which are internally produced or consumed, and resources which can be exchanged among different agents. Moreover, it is also possible to denote that some (contextual) resources are required for a transformation to take place, but are not consumed, or that some resources must not be present, thus expressing negative application conditions.

Multisets have been proposed as a way to express semantics of Petri Nets, viewing the marking of a net as a multiset of elements corresponding to the places in it [MM90]. Hence, the abstract definition of transformations proposed here could be mapped to a Petri Net specification.

3 Formal background

The approach followed here exploits three different rewriting models. On the one hand, we consider attributed typed graph rewriting as a way to provide an abstract syntax and semantics for activity diagrams. Second, we use multiset rewriting as the basis for the definition of an enactment mechanism, modeling the production and consumption of data and synchronisation resources. Finally, we exploit triple graph transformations [Sch94] as a formal device to relate the two metamodels for activity diagrams and multiset rewriting, exploiting the recently proposed notion of enforced generative pattern [BGL08] to automate the generation of operational triple rules. In particular, we adopt a version of triple graphs where only nodes can be put in correspondence, i.e. a triple graph $TrG = (G_s, G_c, G_t, c_s, c_t)$ has three graphs G_i, $i \in \{s,c,t\}$, and two functions $c_j \colon V_{G_c} \to V_{G_j}$, $j = s,t$.

A multiset M over an alphabet Γ is defined by a characteristic function $m_M : \Gamma \to \mathcal{N}$ such that only a finite number of elements from Γ is assigned a non-zero function value. Membership in M is defined as $a \in M \Leftrightarrow m_M(a) > 0$. In the following, we omit the distinction between a multiset and its characteristic function, when no ambiguity arises. An alternative way to represent a multiset is as $\bigcup_{a\in\Gamma}\{a\} \times \{[m(a)]\}$, where $[n]$ is the initial segment of the naturals of length n and $[0] = \emptyset$. Γ can thus be regarded as a (flat) type system, while the natural numbers identify type instances. We are thus actually reduced to a particular type of set. We define a category **MSet** with multisets as objects, while its morphisms are the monomorphisms between multisets preserving the element types. In particular, let m and m' be two multisets on Γ and $\mu : m \to m'$ a morphism between them. Then we have $\mu((a,k)) = (a,j)$ for all $a \in \Gamma$, for some $k \in [m(a)]$ and $j \in [m'(a)]$. The case when m and m' are defined on different alphabets can be managed by taking their union. The pushout is then constructed in an analogous way to the construction of the coproduct in **Set**. Although **MSet** is not weak adhesive (as **Set** is not), we can write rules in DPO form, and adopt the MPOC approach to rewriting [BB08], where the pushout complement $K \xrightarrow{m'} D \xrightarrow{l'} G$ of $K \xrightarrow{l} L \xrightarrow{m} G$ is taken as the minimal object (and associated pair of morphisms) such that the resulting diagram is a pushout, while the pushout of $D \xleftarrow{m'} K \xrightarrow{r} R$ is constructed as

before. By minimal, we intend that for any other D' which defines a pushout complement for $K \xrightarrow{l} L \xrightarrow{m} G$, there is a unique monomorphism $D \to D'$ making the resulting diagram to commute.

Actually, we use multisets of terms formed by attributed symbols on some finite alphabet Γ with attributes taking values on simple domains. Given a collection of activities available to the workflow, the set of admissible values for synchronisation is indeed finite, while for data resources we consider that the values characterizing their descriptions are either finite or they are string names, on which only equality or inequality can be checked.

4 Relating activity diagrams and multiset rewriting

The translation process from activity diagrams to rule-based rewriting exploits Triple Graph Grammars and is based on the metamodel triple of Figure 1. The source metamodel is derived from the metamodel of UML Activity Diagrams [OMG07], where we have introduced a new type, called `SynchNode`, to provide a missing common abstraction for `ControlNode` and `ExecutableNode`, keeping them distinct from `ObjectNode`[1]. Note that by inheriting from `NamedElement`, an `ActivityNode` has a name to identify it.

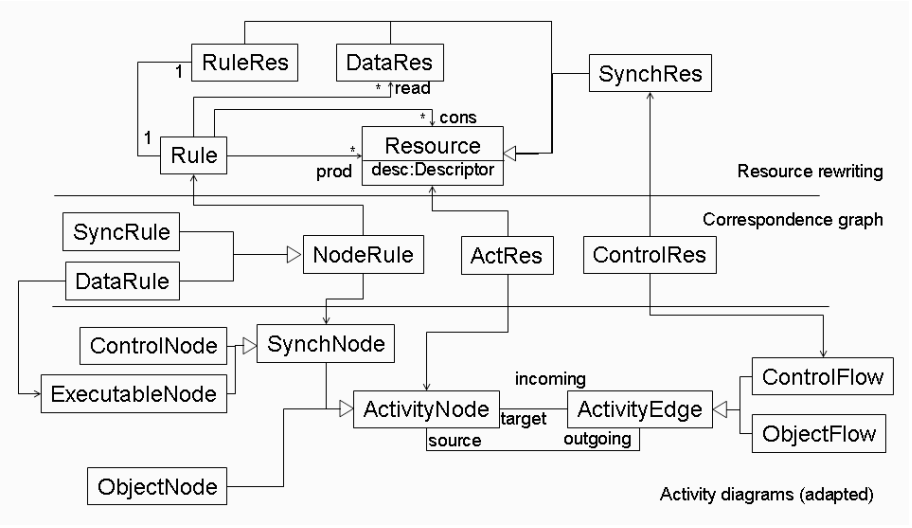


Figure 1: The metamodel triple relating activity diagrams and multiset rewriting.

The target metamodel provides a definition of multiset rewriting rules based on the notion of `Resource` as some distinguishable entity which can be produced or consumed in a transformation. Each resource is defined by a `desc` attribute, coding a suitable description of it. In particular, in this context we are interested in `SynchRes`, used to model the flow of control, and `DataRes`, used to model object flow. A `Rule` is composed of three collections of resources: those which are *cons*umed or *prod*uced by the rule execution and those which are simply *read*, i.e. they must be present, but they are not consumed. Typically, data rules do not consume their

[1] This can also be achieved without modifying the metamodel, by inserting type checks in the triple rules.

input ObjectNodes, unless they explicitly transform data. Each rule is modelled as a resource in turn, via the RuleRes type, so that rules are subject to transformation processes.

Finally, the correspondence metamodel identifies the relations between activity nodes and resources, between control flow edges and synchronisation resources, and between synchronisation nodes and rules. In particular, an ExecutableNode, besides being related to a SynchRule through the correspondence with NodeRule inherited from SynchNode, will also be related to a DataRule. Such correspondence element is mapped, in the metamodel for resource rewriting, to a rule which is only concerned with the transformation of objects, but not with modification of control flow. The advancement of the control flow as the effect of the completion of the activity will be modelled, if need be, by a distinct Rule related to the SynchRule for that node. Not indicated in Figure 1 is the restriction of ObjectNode to correspond to DataRes only.

In order to define the transformation rules, we exploit *triple patterns* [BGL08], as a mechanism to generate triple graph operational rules, coupling syntactic and semantic roles, starting from the definition of syntactic rules. Figure 2 presents the basic patterns relating nodes to rules managing synchronisation or object transformation, while Figure 3 relates activity edges and synchronization resources. This latter pattern states that for each control flow edge in the activity diagram there is a synchronisation resource which is produced by the rule associated with the activity node which is the source of the edge, and which is consumed by the target activity node. In the following, we will use the name of the target rule as an attribute of the synchronisation resource and derive the name of the rule from the name of the corresponding activity node. Analogous patterns can be defined for object flow edges, so that an object at the end of the edge will be produced and consumed, or simply read, by the rules corresponding to the nodes related to the object.

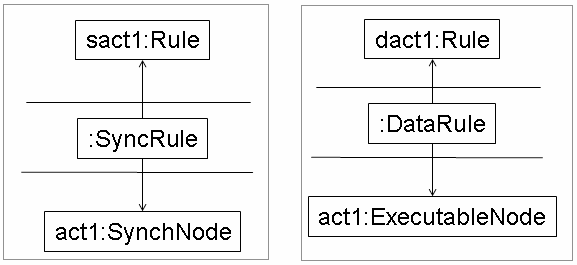


Figure 2: The triple patterns relating activity nodes and rules.

Figure 4 illustrates the result of the application of the triple pattern of Figure 3 to an editing rule adding a control flow between two existing SynchNodes (this is actually an abstract rule to be instantiated for the different specializations of SynchNode), in order to produce an operational triple graph rule which maintains the consistency between the activity diagram and the rule set. A match from the editing rule to the triple pattern causes the construction of the L' (and the omitted identical K') component of the rule, by creating the corresponding nodes, and then the completion of the R' component, according to the process described in [BGL08]. In order to keep the figures illustrating the rules compact, we adopt the following convention. The $L \setminus K$ component of the rule, i.e. those nodes and edges which have to be present for the match to

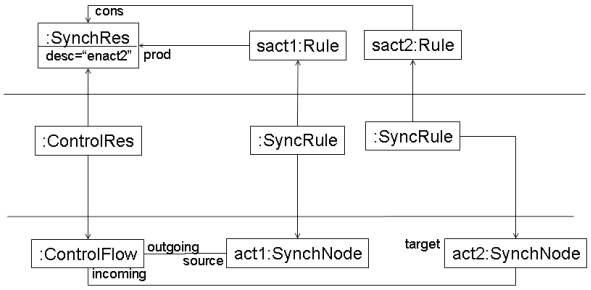


Figure 3: The triple patterns relating activity edges and synchronization resources.

succeed, but which are deleted by rule application, is identified by drawing light grey regions around them and tagging them with the {del} label. In an analogous way, the $R \setminus K$ component of the rule, denoting the elements which are created by rule application, are surrounded by dark grey regions tagged with {new}. As it is easy to see, the $L = K$ part for the correspondence graph contains only the RuleRes nodes, while the R part adds the SyncRes node and its associated edges. The correspondence mappings are also generated according to the pattern.

In a similar way, the triple patterns of Figure 2 are used to create operational rules generating the resource rewriting rules whenever a SynchNode is added to the diagram.

We can therefore assume that when the rule of Figure 4 is applied to a triple graph containing two instances of SynchNode in its source graph, the correspondence and target graphs already contain the corresponding RuleRes and Rule nodes, so that these rules are enriched with the correct definition of production and consumption for SynchRes nodes. An analogous effect updates the data rule associated with an ExecutionNode to reflect production or consumption of data resources according to the direction of object flow edges,

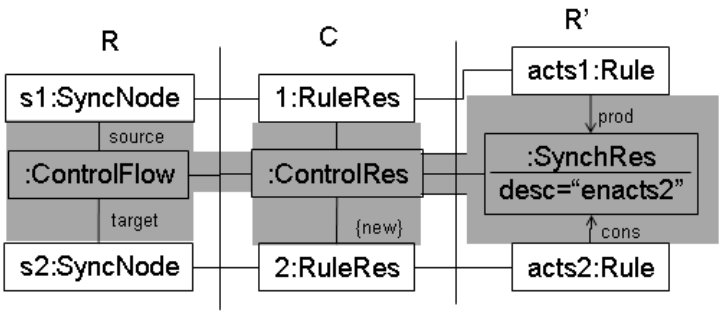


Figure 4: Construction of a triple rule from an editing rule adding a control flow.

Whenever the rule of Figure 4 is applied to generate a control flow edge from a fork node, the rule associated with the fork will accordingly be updated to produce a new ControlRes to enable the rule corresponding to the target of the edge. Symmetrically, the addition of a control flow leading to a join node will add a new resource to the L component of the rule for the join node, which will therefore require that a sufficient number of such resources are produced by its

predecessors.

Figure 5 shows the rule for inserting a fork-join pair between two existing nodes and the corresponding updates on the rules according to the triple pattern construction. The pair is identified by an attribute `pair`, which is an addition to the activity diagram metamodel and is computed to produce a unique new value with each pair creation. To show the effect on the set of rules, we have used a specific representation of a `RuleRes`, listing the multisets of enabling resources produced and consumed by each rule. The actual names in the rule description depend on the values of the description attribute. The rule descriptions in the rule resources and the actual rules are maintained consistent by the patterns.

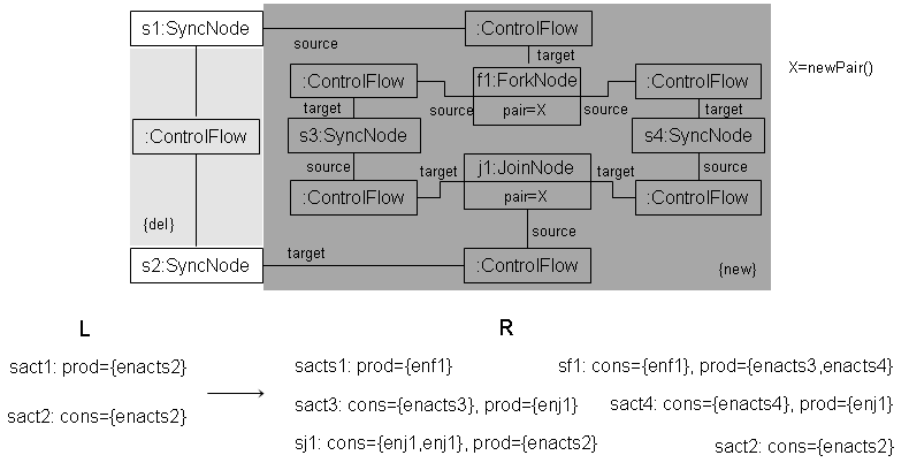


Figure 5: Insertion of a fork-join pair and consequent modification of rules.

A similar construction holds for choice and merge nodes, where the rule for the merge node will usually be connected also to a data resource which is simply queried (i.e. read without being consumed). The abstract representation of resource rewriting rules can actually be translated to several concrete rule-based languages, as discussed in Section 6.

The relation between synchronization and data transformation rules can be established following the construction presented in [BMWY08], to relate control and data flows on spatial structures, and based on the composition of pushouts as shown in Figure 6. Note that, while in [BMWY08] the construction was performed on typed attributed graph rules, we use here triple graph rules, so that each L, K, or R component in Figure 6 is actually a triple graph.

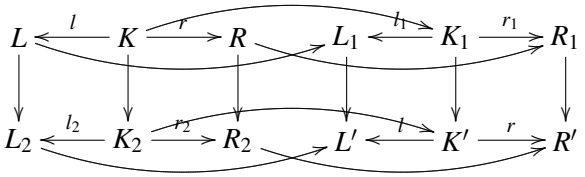


Figure 6: The construction for rule composition.

In particular, the intersection will result from the identification of the `ExecutableNodes`

involved both in the control and the data flow specification. As all other nodes and edges types are different for control and data flows, the pushout will simply result by the union of all other components of the rules and of their associations with the identified nodes.

5 Patterns of transformation

The transformation of control policies in workflows can redefine sequentialization or parallelization of activities for which there is no specific order required by causal (data) dependency. Hence, these modifications should not affect the definition of the data transformation part of the activities, but only the enabling mechanism.

From this point of view, it could be useful to redefine the synchronization enactment rules in an incremental way, as the transformation of the activity diagram takes place. In particular, one could thus maintain a continuous connection between the specification of the workflow and its enactment mechanism. We do not address here the problem of dynamic change – occurring when modifications are performed on workflow regions which are processing workflow instances – which can be dealt with with standard methods [EKR95].

In particular, we consider the two basic adaptation patterns for control flows, i.e. sequentialization of parallel activities and parallelization of sequential activities, under the assumption that forks and joins are paired and correctly nested. To this end, we supplement the metamodel for activity diagrams with a pair of marker node types, called `MoveMark` and `StayMark`. The first is used to follow a chain of activity nodes descending from the fork node for which we want to sequentialize activities. The second marks the beginning of the second chain. Figure 7 shows the rules in the transformation unit for sequentializing activities included in a *fork-join* pair.

In particular, rule *I* starts the transformation by marking the nodes immediately below the fork node: one node is assumed to start the chain of activities, while the other will start a sequence of activities to be performed after the first one. This rule removes the fork node and the control flows associated with it and creates a `ControlFlow` to the node marked with a `MovingMark` from the node which preceded the fork node. The marking nodes have an attribute which identifies the fork node from which we have started. Rule *II* simply moves the marking down the chain. This rule is equipped with a negative application condition (not shown here for simplicity), which prevents the propagation of the marking if node 1 is the *join* node paired with the originating fork node. Rule *II* is therefore performed as long as possible until this join node is reached, at which moment Rule *III* is executed. This rule eliminates the join node and its associated control flows, relates the terminal node for the chain which will have to end the sequence with the node which descended from the join node, attaches the terminal node of the first sequence with the initial node (marked with `StayMark`) of the second sequence, and removes the markings.

Only the modifications depending from rules *I* and *III* need be performed at the level of the enactment rules, while rule *II* does not have effect on the control structure. All in all, this results in the removal of 6 control flows and the insertion of 3 new ones. By applying the synchronization pattern of Figure 3 to these rules, one can specify the operational triple rules required to keep the enactment rules consistent with the modified diagram. Note that the resulting transformation unit can be equipped with parameters, to indicate the *fork-join* region to be sequentialized, or modified so as to start new sequentialization processes if nested *fork-join* regions are present. In

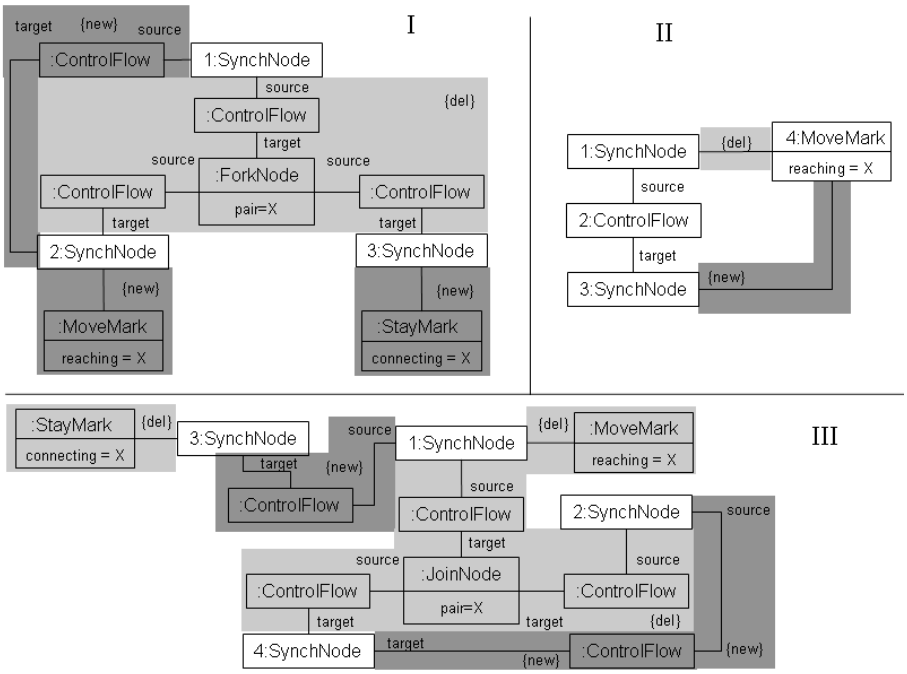


Figure 7: The three rules for sequentialization of activities.

this case the quest resumes for a fork from the node which is now starting the sequence.

The opposite process of parallelization can be specified through the rule of Figure 8. In this case a pattern of 4 nodes in sequence has to be found (this is guaranteed by the presence of the initial and final node and the requirement that at least two activities must be performed for them to be parallelized). The rule removes all the existing control flows and inserts a *fork-join* pair within which the two intermediate activities can now be performed concurrently. Again, the process can be iterated, and a negative application condition can be used to check that no causal dependency exists between the two activities. Similar to the case before, the operational triple rules can be derived from the patterns to ensure the incremental update of the enactment rules.

6 From abstract to concrete syntax

The translation process introduced in Section 4 leaves us with a graph defining an abstract syntax, which could be presented to the user or translated towards an executable syntax in several forms.

Figure 9(b) shows a possible visual representation of the rule derived from the fragment of activity diagram in Figure 9(a) and associated with the `Action` node named *makePayment* by combining the data and control parts. The representation of the rule is based on the containment relation, so that a rule is a container with three compartments showing the context, equivalent to the *K* component of a DPO rule, the left and right sides of the rule, with the usual interpretation, a condition compartment and an assignment compartment, where values of the attributes in the

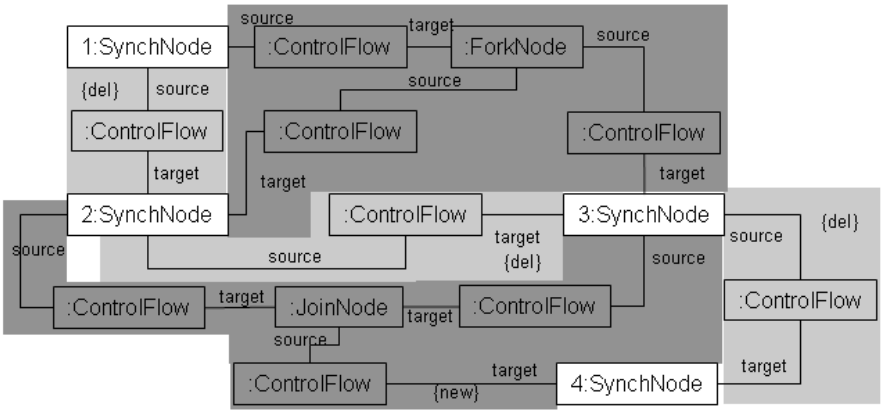


Figure 8: The rule for parallelization of activities.

right-hand side can be evaluated. One can note that this representation is equivalent to a graph rewriting rule for typed attributed discrete graphs, in which no edges exist between entity nodes.

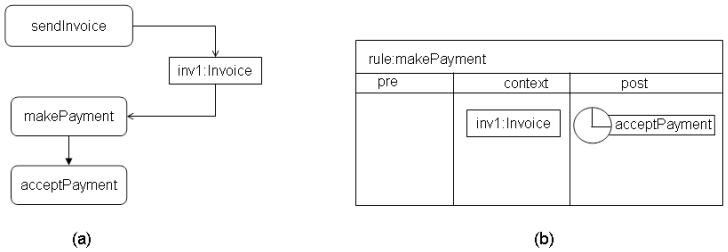


Figure 9: A fragment of an activity diagram (a) and a visual rule derived from it (b).

This visual representation is equivalent to the WIPPOG rule given by:

CONTEXT: invoice(id=”inv1”)
PRODUCES: synchronize(desc=”acceptPayment”)

Note that this rule is specialized to fire only when the specific invoice with name *inv1* has been generated, hence the execution of a workflow coded through WIPPOG rules of this type can exploit indexing of rules according to the required resources.

7 Conclusions

Activity diagrams are increasingly used to express workflows, exploiting users’ familiarity with the use of UML for process specification. While some formal semantics have been provided for activity diagrams, the generation of concrete enactment mechanisms ensuring the coherence of the execution with the specification is still an area of research.

In this paper, we have presented an approach to this problem, based on a simplified model of activity diagrams, suitable for the expression of non-iterative workflows, where activity nodes correspond to rules in a resource production-consumption setting.

The approach allows the separate specification of control and data flows, which results in different types of rules, which can then be integrated exploiting pushouts. The correspondence between nodes and rules, or between flows and enabling resources, is modelled through triple graph grammars, thus allowing an incremental construction and maintenance of the rules, as the diagram is edited or transformed. Although we have used patterns to generate operational rules with source in the activity graph and target in the resource rewriting metamodel, the approach could be used in a bidirectional way, so that modifications in the abstract representation of rules could be reflected to the activity diagrams. Also, the possibility of seeing rules as resources allows the execution of transformations via reflection.

While we have focused only on control and data flow, several dimensions of activity diagrams could also be explored under the resource perspective. For example, distribution could be modelled through the use of distribution resources associated with partition nodes, so that rules are constrained to occur only at some locations or be executed by some organizational roles. Loops could be modelled associating data resources, either already defined or suitably created, with loop variables. Alternatively, transformation units, here exploited only to perform adaptation tasks, could be extended to model some complex activities.

Acknowledgements: Partially funded by Ministry of Research, Project ”CHAT”.

Bibliography

[AB02] W. van der Aalst, T. Basten. Inheritance of Workflows: an approach to tackling problems related to change. *TCS* 270, 2002.

[AH01] W. van der Aalst, K. van Hee. *Workflow Management: Models, Methods, and Systems*. MIT Press, 2001.

[AHKB03] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases* 14(1):5–51, 2003.

[AM00a] A. Agostini, G. D. Michelis. Improving flexibility of workflow management systems. In *Proc. BPM 2000*. LNCS 1806. Springer, 2000.

[AM00b] A. Agostini, G. D. Michelis. A light workflow management system using simple process models. *Int. J. Collab. Comp.* 9(3-4), 2000.

[BB08] B. Braatz, C. Brandt. Graph Transformations for the Resource Description Framework. In *Proc. GT-VMT 2008*. ECEASST 10. 2008.

[BDD+04] P. Bottoni, M. De Marsico, P. Di Tommaso, S. Levialdi, D. Ventriglia. Definition of visual processes in a language for expressing transitions. *JVLC* 15(3):211–242, 2004.

[BGL08] P. Bottoni, E. Guerra, J. de Lara. Enforced generative patterns for the specification of the syntax and semantics of visual languages. *Journal of Visual Languages and Computing* 19(4):429–455, 2008.

[BMWY08] P. Bottoni, N. N. Mirenkov, Y. Watanobe, R. Yoshioka. Composing control flow and formula rules for computing on grids. *ECEASST* 10, 2008.

[Dt01] M. Dumas, A. ter Hofstede. UML Activity Diagrams as a Workflow Specification Language. In *Proc. UML'01*. LNCS 2185, pp. 76–90. 2001.

[EKR95] C. Ellis, K. Keddara, G. Rozenberg. Dynamic Change Within Workflow Systems. In *Proc. COOCS'95*. ACM Press, 1995.

[EW03] R. Eshuis, R. Wieringa. Comparing Petri Net and Activity Diagram Variants for Workflow Modelling - A Quest for Reactive Petri Nets. In *Petri Net Technology for Communication-Based Systems*. LNCS 2472, pp. 321–351. 2003.

[KLN+05] E.-J. Ko, S.-Y. Lee, H.-M. Noh, C.-J. Yoo, O.-B. Chang. Workflow Modeling Based on Extended Activity Diagram Using ASM Semantics. In *Proc. ICCSA 2005*. LNCS 3482, pp. 945–953. 2005.

[KtB00] P. Kiepusziewski, A. ter Hofstede, C. Bussler. On Structured Workflow Modeling. In *Proc. CAISE 2000*. LNCS 1789, pp. 431–445. Springer, 2000.

[MM90] J. Meseguer, U. Montanari. Petri nets are monoids. *Information and Computation* 88(2):105–155, 1990.

[OMG07] OMG. Unified Modeling Language: Superstructure. OMG, 2.1.1 edition, Feb 2007.

[RHEA05] N. Russell, A. H. M. ter Hofstede, D. Edmond, W. M. P. van der Aalst. Workflow Data Patterns: Identification, Representation and Tool Support. In *Proc. ER 2005*. LNCS 3716, pp. 353–368. 2005.

[RvtE05] N. Russell, W. van der Aalst, A. ter Hofstede, D. Edmond. Workflow Resource Patterns: Identification, Representation and Tool Support. In *Proc. Advanced Information Systems Engineering*. LNCS 3520, pp. 216–232. 2005.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *WG*. LNCS 903, pp. 151–163. Springer, 1994.

[Swe94] K. Swenson. Collaborative planning: Empowering the user in a process environment. *Collaborative Computing* 1(1), 1994.

Generating Correctness-Preserving Editing Operations for Diagram Editors

Steffen Mazanek[1], **Mark Minas**[2]

[1] steffen.mazanek@unibw.de
[2] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: In previous work it has already been shown that syntax-directed and free-hand editing can be gainfully integrated into a single diagram editor. That way, the user can arrange diagram components on the screen without any restrictions in free-hand editing mode, whereas syntax-directed editing operations provide powerful assistance. So far, editing operations had to be specified or programmed by the editor developer. In contrast, this paper proposes an approach where diagram-specific editing operations are generated on the fly during the editing process and without any additional specification effort. These operations provably preserve the correctness of the diagram. The proposed approach requires a specification of the visual language by a hypergraph grammar.

Keywords: syntax-directed editing operations, diagram editors, correctness preservation, hypergraph grammars

1 Introduction

Generally, two kinds of diagram editors are distinguished: A structure editor offers the user operations that transform correct diagrams into (other) correct diagrams. Users like this kind of guidance, because that makes editing much easier. But they also like to draw their diagrams freely just following the flow of their uninhibited associations. A free-hand editor allows them to arrange diagram components on the screen without any restrictions. Using syntax analysis, the free-hand editor decides whether the drawing conforms to the visual language and what structure the user intended to define.

A method for the combination of free-hand and syntax-directed editing has already been proposed and realized previously [Min02]. Thereby, free-hand editing is supported by a hypergraph parser [Min97], whereas syntax-directed editing is realized using hypergraph transformation [KM00]. That way, a wide range of operations can be implemented. For instance, operations for diagram execution (like firing of a transition in a petri net or processing the input of a finite state machine) can be defined. But one can also define more local operations, i.e. syntax-directed editing operations in the narrower sense, which require the user to select certain diagram components in advance (like adding a token to the selected place or connecting two selected states with an arrow). These are the operations considered in the following. Concerning such operations, the approach of [Min02] still has some weak points:

- Specifying syntax-directed editing operations is a tedious task: For each operation an additional hypergraph transformation rule has to be defined.
- Specifying *correct* syntax-directed editing operations is difficult: The editor developer has to ensure that his operations do not destroy correct diagrams of the user, i.e. all operations have to comply with the grammar.
- Specifying all possible syntax-directed editing operations is infeasible: In fact, there might even be infinitely many possible editing operations – at least if the number of diagram components that can be inserted at one go is not restricted. The editor developer might not know which of them the users actually need.

The approach presented in this paper offers solutions for these three weak points. Users still can draw their diagrams with maximal freedom. At the same time, they have access to powerful syntactical assistance whenever required. The provided assistance provably cannot do harm, i.e. the correctness of the user's diagram is preserved. These benefits come for free, i.e., the editor developer does not need to define the syntax-directed editing operations anymore.

Concretely, the editor user has to select the diagram components in whose context additional components are to be inserted. On request, meaningful editing operations are computed as follows: First, the selection-induced part of the diagram's hypergraph representation is separated. The resulting hypergraph is analyzed by an error-correcting hypergraph parser, which tries to complete it again by adding hyperedges and gluing nodes. Only those completions are presented to the user as editing operations that meet some language-independent relevance criteria.

Outline: The following sections 2 and 3 briefly introduce the running example and previous work. Sect. 4 then presents the main result of this paper, i.e., how syntax-directed editing operations can be generated on demand during the editing process. The proposed solution is discussed in Sect. 5. Related work is reviewed in Sect. 6, and Sect. 7 concludes the paper.

2 Running Example

Throughout this paper, the simple visual language of Nassi-Shneiderman Diagrams (NSDs) is used as a running example. However, the approach also has been applied successfully to several other visual languages. Its overall applicability is discussed in Sect. 5.

Fig. 1 shows an example NSD and a corresponding Abstract Syntax (Hyper-)Graph (ASG). Hyperedges are represented by boxes with the particular label inside. Nodes are represented as black dots. Lines indicate that a hyperedge visits a node. The correspondence between the diagram and the hypergraph is obvious: Components are mapped to hyperedges. Each corner of a component is represented by an attachment node of the corresponding hyperedge. Hyperedges visit the same node if the respective corners of their components touch each other.

The hypergraph language of NSDs can be recursively defined using a hyperedge replacement grammar (HRG) [DHK97] as shown in Fig. 2. A BNF-like notation is used here. Nonterminal symbols are highlighted. The first two rules (i.e., the upper row) basically state that an NSD is a non-empty chain of successive statements. A statement in turn either is a primitive statement, a condition, a while or an until loop. The body of a loop and the branches after a condition have to be NSDs again.

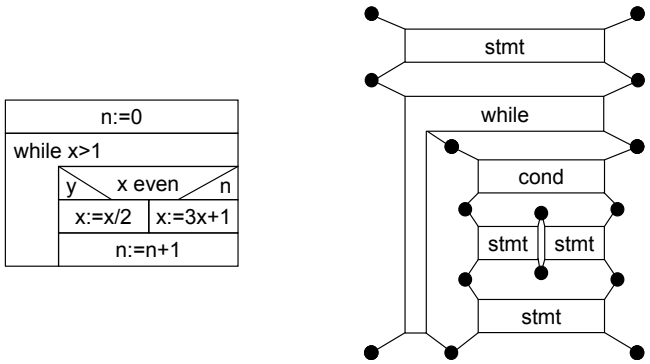


Figure 1: Example NSD and corresponding abstract syntax hypergraph

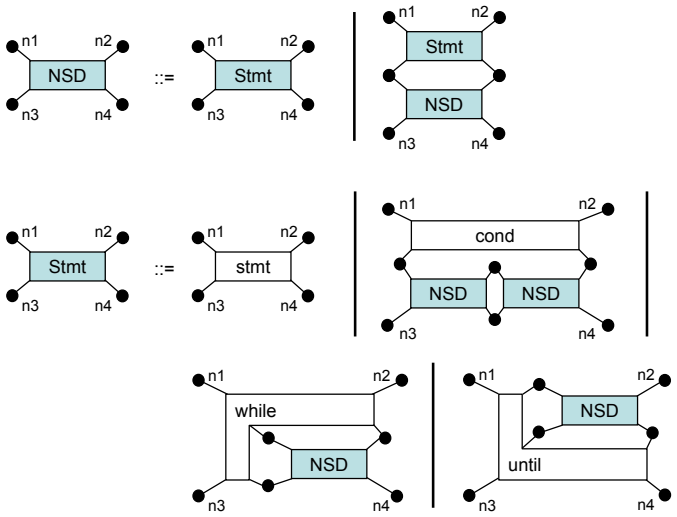


Figure 2: Hyperedge replacement grammar of NSDs

Syntax-directed editing operations are very handy in order to manipulate NSDs. For instance, the user might want to insert a statement `write x` right before the statement `n:=n+1` in the NSD shown in Fig. 1. This task cannot be done conveniently in free-hand mode, because a lot of editing is required to make room for the new statement. In this situation, a syntax-directed editing operation would be preferable: The user could just select the statement `n:=n+1` and call an operation "insert statement before".

3 Basic Formalism and Previous Work

Formally, a hypergraph $H = (V_H, E_H, att_H, lab_H)$ over a set C of labels consists of a set V_H of nodes, a set E_H of hyperedges, a mapping $att_H : E_H \to V_H^*$ that assigns a sequence of attachment nodes to the hyperedges of H, and a labeling function $lab_H : E_H \to C$ for the hyperedges of H. The HRG formalism as used in the following is extensively described in [DHK97].

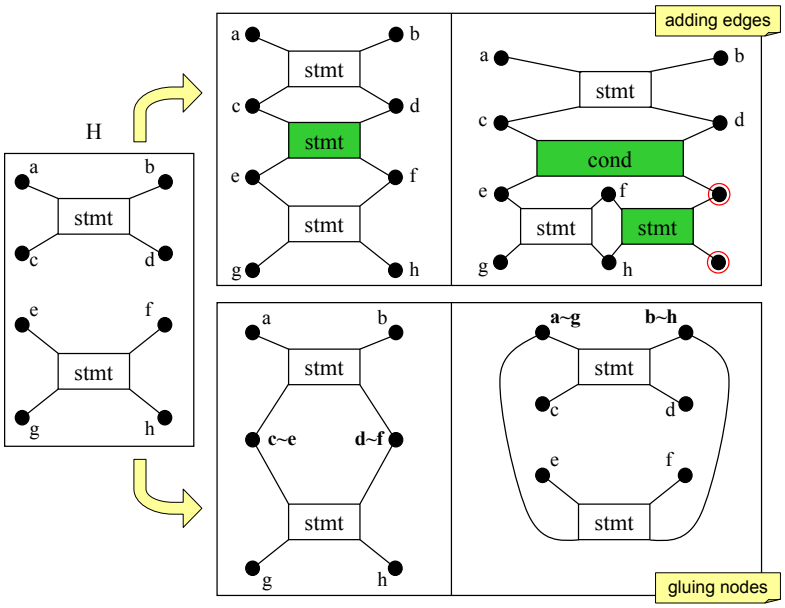


Figure 3: Application of hypergraph patches

Furthermore, this work relies on [MMM08a], where an algorithm for *hypergraph completion* with respect to HRGs has been proposed. Given a hypergraph H and an HRG G, this algorithm embeds additional hyperedges into H in a way, such that the resulting hypergraph H' is a member of the language defined by G. Besides hyperedges it might also introduce some fresh nodes (incident to these hyperedges). This is shown in Fig. 3 (top). The fresh nodes are highlighted by an extra circle. Since there might be an infinite number of possible completions, their size (i.e., the number of additional hyperedges) has to be restricted though.

This algorithm has been extended recently, so that it (optionally) can also glue existing nodes where required. This is also shown in Fig. 3 (bottom). As expectable, the two isolated statements of the given hypergraph H can be combined in two different ways (orders). The extended parsing algorithm returns so-called *hypergraph patches* as a result. Formally, a patch $P_H = (\sim, V, E, att, lab)$ for a hypergraph H consists of an equivalence relation $\sim \subseteq V_H \times V_H$ on the nodes of H, a set V of additional nodes, and a set E of additional hyperedges with corresponding attachment and labeling functions. Applying a patch then basically means to construct the quotient hypergraph (a hypergraph whose nodes actually are equivalence classes of the original nodes of H) and to embed the additional hyperedges. Note that all patches computed by the parser can be used to transform the given hypergraph into a correct one.

Since hypergraphs have appeared to be well-suited as a model for diagrams [Min02], hypergraph patches can be naturally used in diagram editors. In this manner the DIAGEN toolkit has been extended to support diagram correction and completion [MMM08b].

The conventional DIAGEN editing process (as marked in Fig. 4) consists of several steps [Min02]: The modeler first creates a so-called Spatial Relationship (Hyper-)Graph (SRG) corresponding to the diagram. Thereafter, the reducer simplifies the SRG (similar to lexical analysis

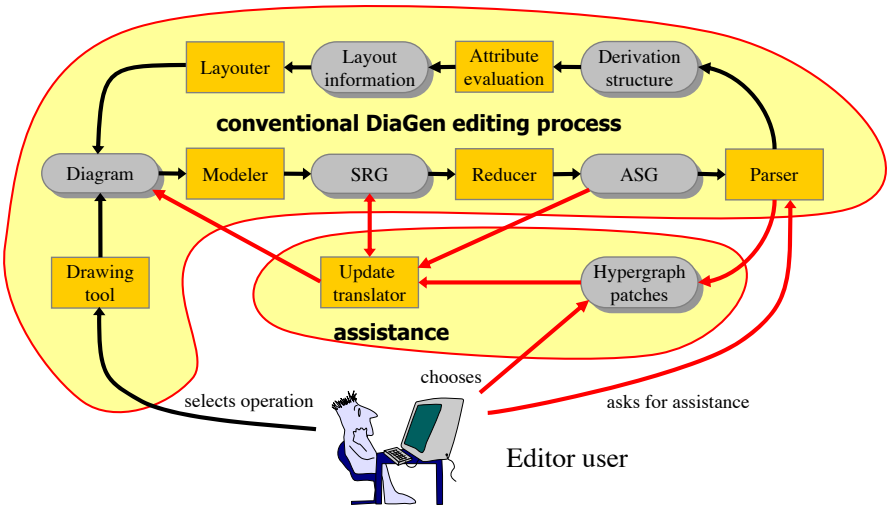


Figure 4: Extended DIAGEN editing process

in the string setting). This results in an abstract representation of the diagram, the ASG. The parser analyzes the ASG and constructs the derivation structure (if any). Finally, the layouter computes a layout for the diagram (using derivation information if required).

In [MMM08b] this process has been extended as follows (see also Fig. 4): If a user explicitly asks for assistance, the parser is triggered with the desired size of completions as a parameter. It computes all possible hypergraph patches up to this size [MMM08a]. From those, the user has to choose. Next, the selected patch is applied and embedded into the SRG using a language-specific update translator component. The editor then calls the reducer and parser again, so that the layouter can arrange the new components within the actual diagram and adapt existing components if necessary.

4 Generating Syntax-Directed Editing Operations

Since an editing operation might be applicable to several parts of a diagram, the user normally has to select the context in which additional components are to be inserted. For instance, the already mentioned operation “insert statement before” requires the selection of the statement where a new statement should be inserted before. If operations are predefined by the editor developer it is easy to specify as a precondition what has to be selected by the user. This approach cannot be used if operations are to be generated at runtime. However, for a generic approach the user’s selection can be interpreted as follows: A selection should induce editing operations that separate the user-selected diagram part, add new diagram components, and finally paste the extended diagram part back into the remaining diagram such that it is correct again.

Fig. 5 shows some example operations following this idea. On the left-hand side four example NSDs are given. The components selected by the user are surrounded by a thick border. To the right of the arrows, the figure shows extended diagrams resulting from the application of certain editing operations to the input diagram. All new diagram components are highlighted.

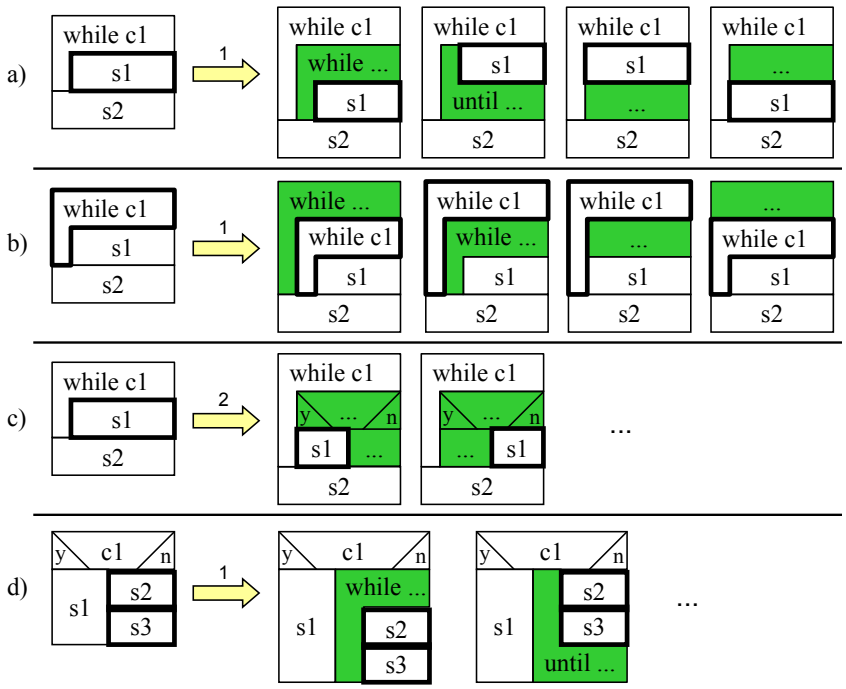


Figure 5: Example editing operations

The numbers above the arrows indicate the size of the operations, i.e. the number of components to be added. Note that all shown operations preserve the correctness of the respective input NSD. All of them can be generated following the approach presented next.

In Fig. 5a, the statement s1 is selected and one new component should be inserted. In this case four different operations of size one are possible: s1 could either be enclosed by a *while* or *until* loop or, alternatively, a primitive statement could be inserted below or above. In Fig. 5b, a *while* component is selected. Four operations of size one are possible: Another *while* component could be inserted outside or within the selected one. Alternatively a primitive statement could be inserted within or above the selected *while*.

The selection in Fig. 5c is equal to Fig. 5a. However, this time operations of size two are requested. Two useful operations are shown. Actually, both cannot be simulated by just repetitive application of operations of size one, because intermediate results are required to be correct. This means, each operation has to yield a correct diagram. Note that many more reasonable operations of size two exist. However, in contrast to the insertion of a *cond*, these could also be constructed successively. The last row, Fig. 5d, demonstrates that sometimes it is even necessary to allow the selection of several diagram components at once. Otherwise, it would not be possible to insert a *while* or an *until* component around a correct sub-NSD (here just two successive statements). Again there are some more solutions, but those can already be realized by selecting just one of the existing components and, hence, have been omitted.

In order to generate such syntax-directed editing operations the visual language's hypergraph grammar can be exploited. The basic idea is to reuse the patch-computing parsing algorithm

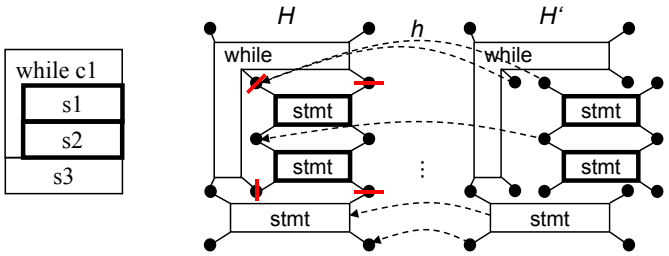


Figure 6: Separation of selected components

described in Sect. 3. The intuition of a user selection on the diagram level has already been described. On the level of a diagram's ASG, the separation of the selection means breaking up the ASG into two disjoint hypergraphs H_1 and H_2 where H_2 corresponds to the user-selected diagram part. Breaking up the ASG generally means splitting up some of its nodes (cf. Fig. 6). Adding new diagram components and re-merging the diagram parts just means to find and apply a hypergraph patch using the disjoint union of H_1 and H_2 as input. However, not every hypergraph patch constitutes a meaningful editing operation. Language-independent *relevance criteria* can be used to discard hypergraph patches that are inappropriate as editing operations.

Next, the generation of editing operations is described more formally and the relevance criteria are defined. Let H be the ASG of the diagram and $E_s \subseteq E_H$ the set of selected hyperedges of H. Let H_s and $H_{\bar{s}}$ be the sub-hypergraphs of H induced by the sets E_s resp. $E_H \setminus E_s$ of hyperedges. Finally, let H' be the disjoint union of H_s and $H_{\bar{s}}$. H' differs from H since nodes in H being visited by selected as well as non-selected hyperedges are "split" in H'. An epimorphism h maps H' to H, cf. Fig. 6. Let V_{split} be the split nodes of H', i.e., those nodes that are merged by h (in Fig. 6 there are 8 split nodes).

H' is incorrect in general, so that the application of the patch-generating parser normally yields a wide range of solutions. Not all of them form meaningful editing operations though. This issue is illustrated in Fig. 7 using the diagram of Fig. 5a as an example. The trivial patch of size 0, which just glues the separated statement back to its original position, is omitted in Fig. 7. Fig. 7 rather shows all patches of size 1 and their resulting hypergraphs. However, only 4 of these 10 patches constitute meaningful editing operations; the other 6 are crossed out. They are not meaningful since either the selected statement has been glued back to its original position and a new component has been added at a remote position, or the selected statement has been moved to a remote position and its original position has been "filled" by a new component. Such a behavior is not meaningful for NSD or any other diagram language. This observation motivates the definition of *relevant* hypergraph patches that describe meaningful editing operations independent of the specific diagram language:

A patch $P_{H'} = (\sim, V_P, E_P, att_P, lab_P)$ is *relevant* if and only if

1. $\forall e \in E_P : sequenceToSet(att_P(e)) \subseteq V_{split} \cup V_P$, i.e., additional hyperedges do not visit any nodes that are not related to the selection, and

2. $\forall n_1, n_2 \in V_{H'} : n_1 \sim n_2 \Rightarrow h(n_1) = h(n_2)$, i.e., only the split nodes can be glued, but only to the respective nodes they have been separated from.

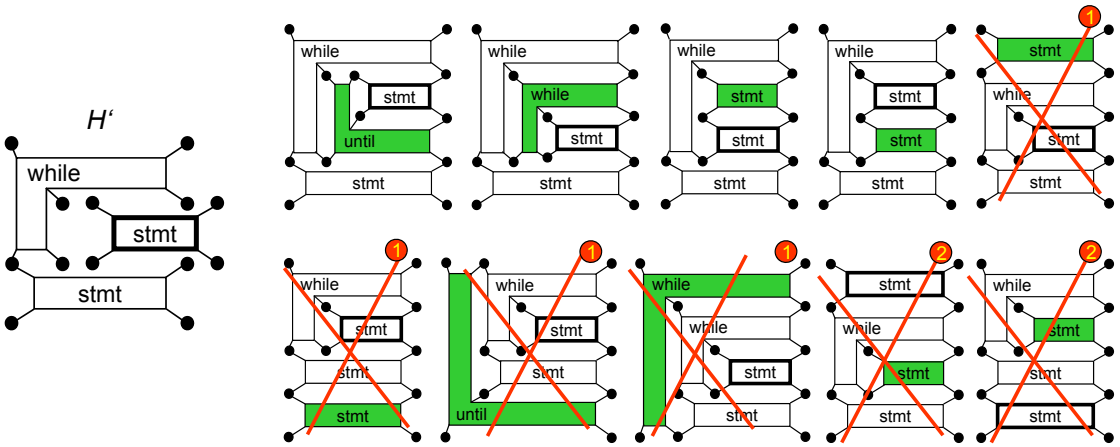


Figure 7: Relevance criteria, input diagram Fig. 5a

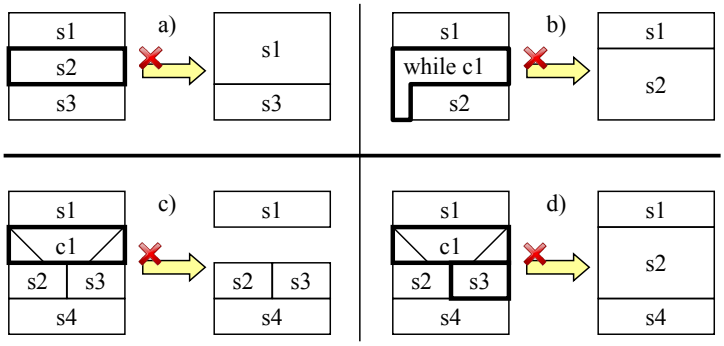


Figure 8: Intelligent component removal

Fig. 7 shows which criterion excludes a particular patch. This is indicated by numbers in the upper right corners of the resulting hypergraphs.

Intelligent Remove and Replace

Syntax-directed editing operations generated by the previous process do not delete any diagram component. However, generating *intelligent remove* operations (and replacement similarly) is also straightforward. Conventional removal of one or more components does not modify the remaining diagram which may become invalid after the removal. In contrast, *intelligent remove* adjusts the remaining diagram so that it becomes valid again. It is performed in two steps: First, a conventional remove of the selected components is performed. Subsequently, corrections are computed. Again, not each correction of the remaining diagram is meaningful. A relevant hypergraph patch here is neither allowed to add new hyperedges, nor is it allowed to glue remote nodes, i.e., nodes that have not been visited by the hyperdeges deleted previously.

Fig. 8 shows four examples of intelligent removal. In Fig. 8a the middle *stmt* in a chain of three successive *stmt*s is selected. Intelligent removal deletes this *stmt* and glues the other *stmt*s

together preserving their order. In Fig. 8b, a *stmt* s1 is followed by a *while* component that encloses another *stmt*, s2. Intelligently removing the *while* component glues the *stmt*s s1 and s2 together (again preserving their order). Fig. 8c shows an example where intelligent removal cannot help. In this situation, intelligent removal corresponds to a conventional remove. Indeed, a diagram cannot be kept correct when just a *cond* component is to be removed. However, if one of the branches is selected, too, intelligent removal works as expected, cf. Fig. 8d.

All in all, intelligent removal is a useful function for such visual languages where the removal of components is likely to yield incorrect diagrams. Otherwise, it just converges to the conventional remove function, i.e. it simply removes the selected components.

5 Discussion

Following the presented approach, the user can quickly access local editing operations. In most cases, selecting just a single component is sufficient already. In fact, the selection of several components usually makes sense only if they "share nodes". This behavior most likely conforms to the user's intuition. Nevertheless, there are still some challenges that are addressed next.

Performance: It is important to stress that the presented approach does not generate generic operations at "compile time". The generated operations are rather specific to the current diagram and completely generated on the fly. As a consequence, each time the user asks for operations the whole diagram needs to be analyzed again. This additional effort is not necessary if the operations are predefined. However, a basic requirement for free-hand editors is an efficient parser, since the diagram has to be reanalyzed after every single modification. The current implementation of the patch-generating parser is a prototype with reasonable speed. For instance, the computation of operations of size one for an NSD of size 20 takes less than a second on standard hardware. Further performance improvements are subject of current work.

Information Overload: When increasing the possible size of operations their number might explode. The problem is that the user can hardly distinguish between the really new solutions and the solutions that he could also get by successively applying smaller operations. It might be useful to apply a special filter to avoid this issue. But this has not been realized yet.

Understandability: In certain (rare) situations knowledge of the abstract syntax seems to be necessary to understand why a particular operation currently is not possible. For instance, if a *cond* is selected, a *stmt* can only be inserted above, but not in the branches below. However, we have not found a meaningful operation yet that cannot be generated at all. The user just needs to select the "right" components (in the example, the first statement in the respective branch).

Applicability: Since the patch-computing hypergraph parser relies on context-free HRGs, the applicability of this approach naturally is restricted. However, the approach can also be applied to hypergraph grammars that contain so-called *embedding rules* [Min02]. Indeed, all practically relevant visual languages can be described that way. Operations then are only computed for the context-free part of the particular diagram. So, if a language exhibits a significant context-free core, the presented approach can still be used. For instance, it has been applied to sequence diagrams where only the arrows need to be embedded. The generated syntax-directed editing operations have appeared to be helpful even for this non-context-free diagram language. In addition, the approach has been applied to other examples like flowcharts, logic gates, and trees.

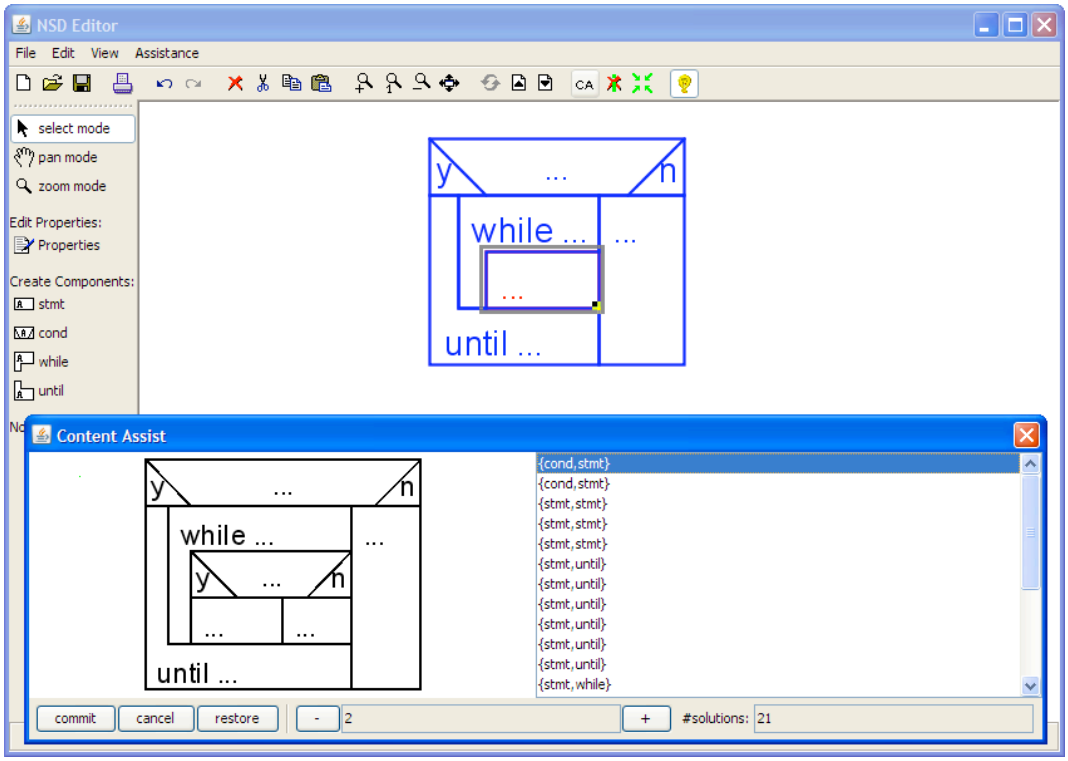


Figure 9: Screenshot of the user interface

The proposed approach has been integrated into the DIAGEN toolkit. Fig. 9 shows a screenshot of a DIAGEN editor for NSDs. The user has selected a primitive statement and called the assistance function (the "Content Assist" dialog). Here, he can set up the size of operations he is interested in using the buttons at the bottom. There are already 21 operations of size two as can be seen in the lower right corner. Selecting one of them updates the preview pane on the left-hand side. Both the application of the operation and refocusing of the diagram are animated, so that the user can easily see what is going on. Committing finally applies the selected operation to the actual diagram. Intelligent remove of selected components can be invoked via a shortcut.

6 Related Work

In TIGER [EEHT05] editing operations are specified by means of graph transformation rules. These operations directly define the language, so that the editor developer does not have to ensure that they comply with a grammar. Unfortunately, TIGER does not support free-hand editing. The predecessor of TIGER, GENGED [BST01], had supported some kind of free-hand editing. It even had generated some initial editing operations from the type graph at compile time. Those, however, only allowed the insertion of nodes and edges in graph-like languages. Recently, the TIGER developers have extended the popular Eclipse GMF framework with support for complex editing commands [TCSE08] – at the price of additional specification effort though.

CIDER [JMM04] supports both free-hand and syntax-directed editing. Its transformation mechanism is fully integrated with an incremental parser. Thus, transformations can be defined in terms of high-level diagram components. Here, the basic idea behind a transformation is to change the parse forest of a diagram from one valid state to a different valid state. This is similar to the conventional DIAGEN approach to syntax-directed editing, where information from the derivation can also be accessed. In CIDER all transformations have to be specified manually.

The grammar-based system VLDESK [CDPR05] provides support for so-called symbol prompting. Here, the parsing table is exploited to extract information about possible contexts of a particular symbol. That way, local suggestions can be computed without additional specification effort (similar to our approach). This kind of assistance is efficient and permissive, but does not ensure the correctness of the resulting diagram from an overall perspective.

For widely used and highly relevant languages specific tool support is still implemented by hand. For instance, Gschwind et al. have proposed a set of powerful operations on business process models [GKW08]. Their approach has been realized as a plugin for the well-known WebSphere Business Modeler. As long as generic assistance mechanisms are not powerful enough this is a reasonable approach to improve the usability of modeling tools. Our approach, where applicable, can help to reduce the burden of implementing language-specific syntactical assistance. Business process models are context-free to a large extent, so that our approach is applicable.

7 Conclusion

The approach proposed in this paper can be used to generate powerful, correctness-preserving editing operations for free-hand editors and, at the same time, is easy to apply and understand by users: They just have to select one (or more) components and ask for assistance.

The editor user can trust the generated operations, because they provably cannot do harm. He can use an animated preview to inspect all possible operations of a particular size. Thereby, he is likely to get new insights into the visual language at hand. Although the improvement of user support has been our primary goal, the burden for the editor developer is also significantly reduced. He does not need to specify editing operations in compliance with the grammar anymore. Rather he can focus on special-purpose operations and diagram execution. Since our approach is just complementary, the previous flexibility of DIAGEN operations is fully preserved.

In the future we will try to relax the precondition “correctness of the input diagram” that has been assumed throughout this paper. Indeed operations would be also very useful for correct sub-diagrams. Furthermore, we want to improve the support for the non-context-free parts of languages.

Screencasts of the NSD example editor can be found at http://www.unibw.de/inf2/DiaGen/assistance/. The editor can be downloaded from there, too.

Bibliography

[BST01] R. Bardohl, T. Schultzke, G. Taentzer. Visual Language Parsing in GenGEd. *Electronic Notes in Theoretical Computer Science* 50(3):289 – 294, 2001.

[CDPR05] G. Costagliola, V. Deufemia, G. Polese, M. Risi. Building syntax-aware editors for visual languages. *Journal of Visual Languages and Computing* 16(6):508–540, 2005.

[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.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual editors as eclipse plug-ins. In *ASE '05: Proc. of the 20th IEEE/ACM International Conference on Automated Software Engineering*. Pp. 134–143. ACM, New York, NY, USA, 2005.

[GKW08] T. Gschwind, J. Koehler, J. Wong. Applying Patterns during Business Process Modeling. In Dumas et al. (eds.), *BPM*. LNCS 5240, pp. 4–19. Springer, 2008.

[JMM04] A. R. Jansen, K. Marriott, B. Meyer. Cider: A Component-Based Toolkit for Creating Smart Diagram Environments. In *Diagrams*. LNCS 2980, pp. 415–419. Springer, 2004.

[KM00] O. Köth, M. Minas. Generating diagram editors providing free-hand editing as well as syntax-directed editing. In Ehrig and Taentzer (eds.), *Proc. Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems*. Technical Report 2000-2, pp. 32–39. Technical University, Berlin, 2000.

[Min97] M. Minas. Diagram Editing with Hypergraph Parser Support. In *VL '97: Proceedings of the 1997 IEEE Symposium on Visual Languages (VL '97)*. P. 226. IEEE Computer Society, Washington, DC, USA, 1997.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[MMM08a] S. Mazanek, S. Maier, M. Minas. An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars. In Ehrig et al. (eds.), *Proc. of the 4th International Conference on Graph Transformations*. LNCS 5214, pp. 39–53. Springer, 2008.

[MMM08b] S. Mazanek, S. Maier, M. Minas. Auto-completion for Diagram Editors based on Graph Grammars. In Bottoni et al. (eds.), *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 242–245. IEEE Computer Society Press, 2008.

[TCSE08] G. Taentzer, A. Crema, R. Schmutzler, C. Ermel. Generating Domain-Specific Model Editors with Complex Editing Commands. In Schürr et al. (eds.), *Proc. Third Intl. Symposium on Applications of Graph Transformation with Industrial Relevance (AGTIVE 2007)*. LNCS 5088, pp. 98–103. Springer, 2008.

Syntactical Analysis of Exploded VL Diagrams

Steffen Mazanek[1], **Mark Minas**[2]

[1] steffen.mazanek@unibw.de
[2] mark.minas@unibw.de
Institut für Softwaretechnologie
Universität der Bundeswehr München, Germany

Abstract: A free-hand diagram editor allows the user to place diagram components on the pane without any restrictions. This increase in flexibility often comes at the cost of editing performance, though. In particular it is tedious to manually establish the spatial relations between diagram components that are required by the visual language. Therefore, in this paper an approach is proposed that permits to deduce the intended concise diagram from a rough arrangement of diagram components. Such an arrangement actually looks like an exploded view and can be drawn much more rapidly. The proposed feature, *diagram contraction*, considers and mostly preserves the layout of the existing diagram components. An important special case are (certain) graph-like visual languages, where diagram contraction corresponds to linking the node components appropriately. Such auto-linking is considered useful. It even has been integrated in first commercial modeling tools – by manual programming, though. This effort can be avoided to a large extent.

The proposed approach can be applied to visual languages that are specified by means of hypergraph grammars. For syntax analysis an error-tolerant hypergraph parser is used, which computes a cost function by attribute evaluation. That way, unfavorable derivation (sub-)trees can be excluded at an early stage, and combinatorial explosion is mostly prevented.

Keywords: exploded diagram, diagram contraction, auto-link, least cost parsing

1 Introduction

Exploded views frequently occur in descriptive manuals. Normally, they are used to show the components of a system and their placement relative to each other. Therefore, the components of the system's model are slightly separated by distance as if there had been a small, controlled explosion emanating from the middle of the model. This technique is all the more effective for the visualization of complex 3D systems. Given a model, existing tool support allows the convenient creation of exploded views, e.g. [LACS08].

In the context of diagram editors, the other way round appears to be quite as interesting. Given an *exploded diagram*, i.e., the exploded view of some (yet unknown) diagram, from which concise diagram is it most likely an exploded view? This diagram can be deduced most effectively, if there is an underlying visual language and a notion of syntactical correctness. In the following, the language of Nassi-Shneiderman Diagrams (NSD) serves as a running example. Fig. 1 shows an exploded NSD and the corresponding concise diagram of the language.

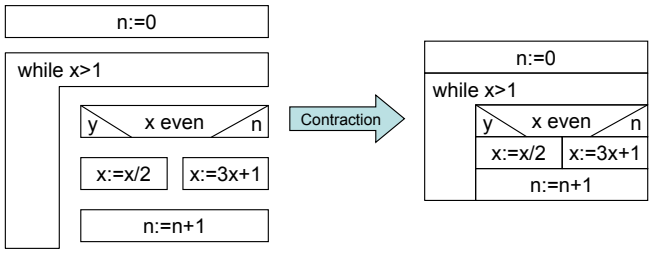


Figure 1: Exploded NSD and contraction

The process of finding the (concise) diagram corresponding to an exploded diagram is called *diagram contraction* in the following. Diagram contraction is quite a useful operation in the context of free-hand diagram editors. Such a free-hand editor allows the user to place the diagram components on the pane without any restrictions. However, in order to create a correct diagram the components have to be spatially related in a language-specific way. In case of NSDs the respective corners of components have to touch each other. It is quite tedious though, to place the components such that their corners indeed touch (at least without snap to grid). Therefore, a lot of precision work is required (even if slight correction of geometric errors is provided as in CIDER [CM03] or DIAGEN [Min02]). That way, the accessibility of the editor is restricted, and both the editing performance and freedom suffer. With diagram contraction available the editor user rather just roughly arranges the diagram components, so that it looks as if it had been exploded (most likely in an imperfect way, though). Then he can invoke contraction and the editor creates the corresponding concise diagram automatically. Initial experiments with a prototypical NSD free-hand editor have shown that indeed an exploded NSD can be created more than 1.5 times faster than a concise one, cf. Sect. 5.[1]

The method presented in this paper exploits least cost parsing in order to compute diagram contractions in an efficient way. An error-tolerant hypergraph parser is used that computes costs for all possible derivations. Therewith, unfavorable (sub-)derivations can be excluded at an early stage. Combinatorial explosion is effectively prevented. At the same time, a flexible control of the parsing process is possible via the cost function.

Finally, an important special case of contraction, namely auto-link, will be discussed. Thereby, missing connectors are automatically derived from the layout of node components of a graph-like visual language. Where applicable, auto-link can greatly increase the editing performance.

Outline: In Sect. 2 previous work is briefly reviewed. Sect. 3 presents the method for contraction. The scope of this solution is discussed in Sect. 4. Empirical results of a user study are presented in Sect. 5. Related work is discussed in Sect. 6, and Sect. 7 concludes the paper.

2 Previous Work

The proposed approach is based on hypergraphs as a model for diagrams [Min02], which have proven to be well-suited for that purpose. The syntax of the particular visual language then can

[1] Note, that by providing syntax-directed editing operations for frequent editing tasks a similar improvement can be achieved. That way, however, the users are quite restricted while editing.

be defined by a hyperedge replacement grammar (HRG) [DHK97]. Fig. 2 shows the HRG of NSDs. Nonterminal symbols are highlighted. The first two rules (i.e., the upper row) basically state that an NSD is a non-empty chain of successive statements. A statement in turn either is a primitive statement, a condition, or a while loop. The body of a loop and the branches below a condition have to be NSDs again, i.e. the language is recursively defined.

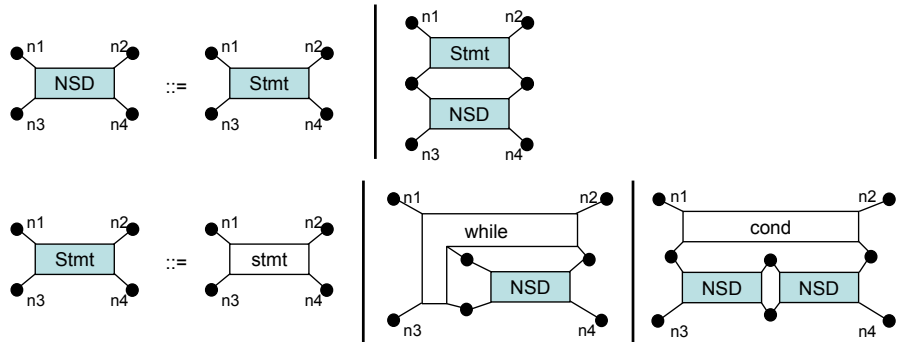


Figure 2: Hypergraph Grammar of NSDs

In [MMM08a] an algorithm for *hypergraph completion* with respect to HRGs has been proposed. Given a hypergraph H and an HRG G, this algorithm embeds additional hyperedges into H in a way, such that the resulting hypergraph H' is a member of the language defined by G. This algorithm has been extended recently, so that it (optionally) can also glue existing nodes where required. Technically, it constructs quotient hypergraphs whose nodes actually are equivalence classes of the original nodes of H. This is shown in Fig. 3. The two isolated statements of the given hypergraph H can be combined in two different ways. An equivalence relation on the nodes of H together with a set of additional hyperedges is called a *patch* of H. The parser, hence, computes hypergraph patches that, when applied to H, yield correct hypergraphs.

Such hypergraph patches can be naturally used in diagram editors [MMM08b]. In this manner the DIAGEN toolkit has been extended to support diagram correction and completion. The conventional DIAGEN editing process (as marked in Fig. 4) consists of several steps [Min02]: The modeler first creates a so-called Spatial Relationship (Hyper-)Graph (SRG) corresponding to the diagram. Thereafter, the reducer simplifies the SRG (similar to lexical analysis in the string setting). This results in the Abstract Syntax (Hyper-)Graph (ASG), an abstract representation of

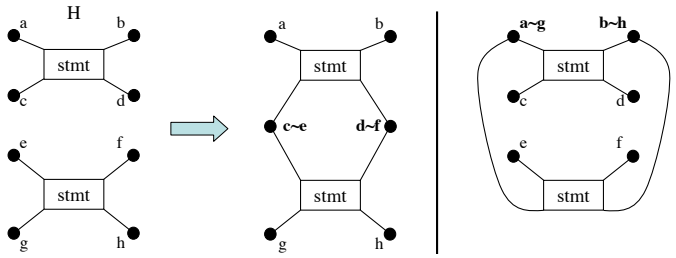


Figure 3: Application of hypergraph patches

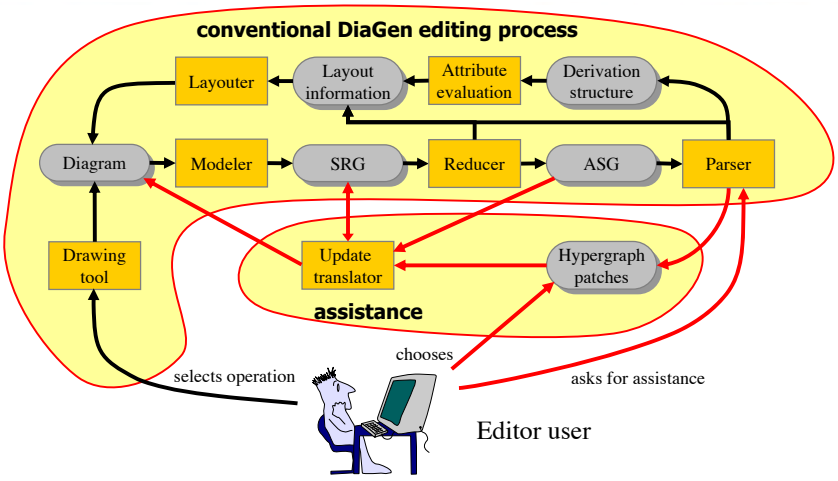


Figure 4: Extended DIAGEN editing process

the diagram. The parser analyzes the ASG and constructs the derivation structure. Finally, the layouter uses this derivation information in order to compute a layout for the diagram.

In [MMM08b] this process has been extended as follows (cf. Fig. 4): If a user explicitly asks for assistance, the parser is triggered again. It computes hypergraph patches of some user-defined size (determining the number of additional components). From those, the user has to choose. The selected patch is applied and embedded into the SRG using a language-specific update translator. The editor then calls the reducer and parser again, so that the layouter can arrange the new components within the actual diagram and adapt existing components if necessary.

In [MMM08c] a method has been suggested that supports – as a special case – a limited form of diagram contraction. There, possible corrections are computed on the abstract syntax level. The quality of these corrections is evaluated afterwards using the layout engine. Finally, that correction is applied that causes minimal changes to existing diagram components. The main challenge of this approach is the possible (likely) combinatorial explosion. For instance, n NSD statements, where no two corners touch, can be arranged in $n!$ ways. If other kinds of components are also involved it can get even worse. The example diagram of Fig. 1 already can be corrected in 1800 [sic!] different ways. Moreover, the assessment procedure (layout) is also quite expensive. So, contraction of practical diagrams is infeasible that way. Nevertheless, the editing performance has already been improved: Diagrams can be created incrementally by sketching just a few diagram components a time and integrating them into the main diagram by contraction. Following this process, precise spatial relations need not be established manually anymore. On the other hand, such a restricted solution certainly is not desirable.

3 Realization of Diagram Contraction

This combinatorial explosion can be avoided if undesirable corrections are excluded as early as possible in the parsing process. For instance, the user is unlikely to be interested in corrections

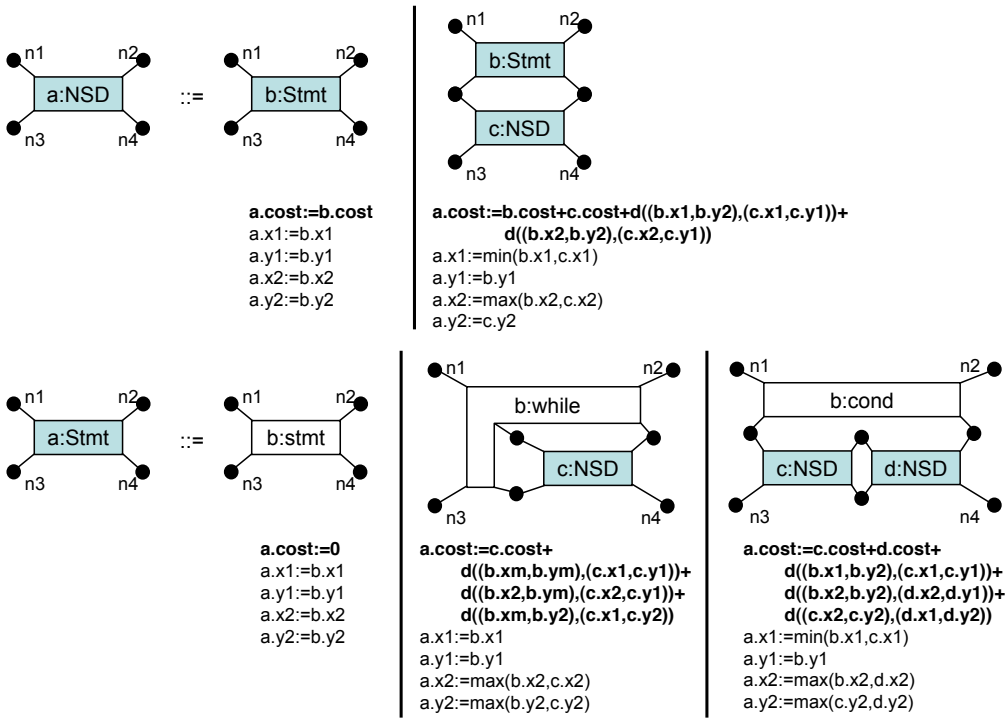


Figure 5: Cost grammar of NSDs

where the vertical order of his NSD statements is not preserved. So, positional information of the diagram components has to be considered while parsing the abstract representation. That way, derivations that preserve the order of statements cross out derivations that don't.

The approach proposed next exploits attribute evaluation, which is normally used to compute a semantic representation, in order to compute a *cost* for each possible derivation. This cost is an estimation of the changes that have to be performed when this correction should be applied. For this purpose a cost attribute is assigned to each nonterminal hyperedge. Its value gets computed bottom-up by evaluation rules that are assigned to the grammar's productions.

An example of such a *cost grammar* is given in Fig. 5. In case of NSD nonterminal edges additionally are characterized by attributes `x1,y1,x2,y2` that describe the bounding box of the respective sub-diagram. Whenever a reduction is performed during parsing, the attributes `x1,y1,x2,y2` and `cost` are set as indicated by the assignments below the used production. The attributes `xm,ym` of the while edge describe the inner bend of a while component. Note, that we address the edges in the assignment part using labels, e.g. `a:Stmt`.

The goal of a cost model is to punish undesirable contractions with high costs. In Fig. 5 distances between points (that ideally should coincide) are used (`d(p1,p2)` denotes the Euclidean distance between `p1` and `p2`). However, there is some freedom in choosing a cost function. In general, it is not possible to define a cost function that precisely reflects the costs finally caused by the layout engine. Indeed, this would lead to the combinatorial problem of [MMM08c] again. A reasonable estimation already yields acceptable results in practical time instead.

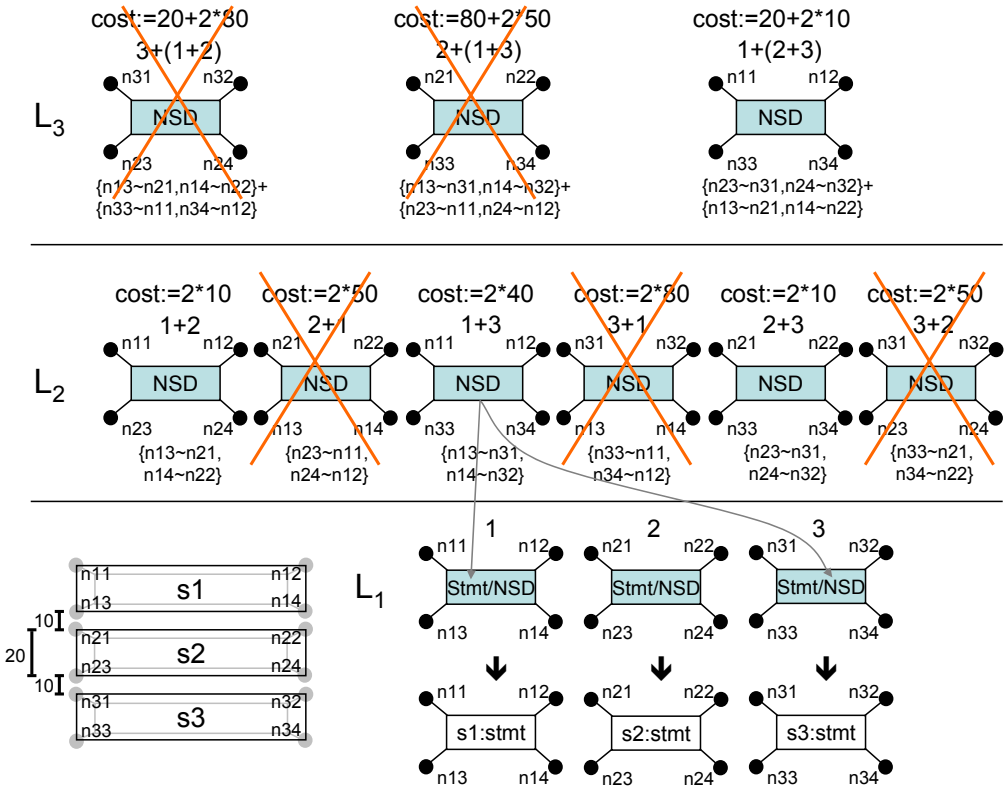


Figure 6: Example parser run

Let us recall the details of error-tolerant parsing as described in [MMM08a]. First, as in the string setting, the HRG has to be transformed to Chomsky normalform. Thereby, chain productions with a single nonterminal edge on the right-hand side are eliminated. For NSD an additional terminal production `NSD::=stmt` is introduced. Then possible derivation trees are constructed bottom up by reverse application of productions, i.e. reductions. Thereby, nodes can be glued where required, i.e. an equivalence relation on nodes is computed successively. Derivations are computed layer by layer where derivations of layer i result in hypergraphs of i terminal hyperedges. The layers are constructed as follows:

- Layer 1 is created by reverse-application of terminal productions, i.e. productions with a single terminal hyperedge as right-hand side.
- Layer i $(i > 1)$ is created by selecting every pair of layers j $(j < i)$ and $i - j$, and combining derivations thereof.

3.1 Extension of the Parser

This parser has been extended by attribute evaluation in a straightforward way in order to support the computation of the cost attribute. Moreover, and this is the major improvement, a clean-up phase has been incorporated that follows each time after a layer has been computed.

layer	old parser	new parser					
	finally	after processing of layer					
		1	2	3	4	5	6
1	10	10	10	10	10	10	10
2	24		24/19	14	8	8	8
3	112			87/29	10	5	5
4	468				135/12	6	5
5	1512					42/9	4
6	2760						13/1

Table 1: Savings due to cleaning up layers, number of elements in layers

Fig. 6 shows an example run of this extended parser. In the lower left corner an exploded NSD is given. It consists of just three statements (whose intended order can easily be recognized by a human observer). Sizes and distances relevant for the computation of the cost function are marked. In the lower right corner the first derivation layer L_1 is shown. The terminal productions `Stmt::=stmt` and `NSD::=stmt` are applied three times each (their derivation nodes are combined indicated by the label `Stmt/NSD`). In layer L_2 there are six ways of combining a `Stmt` and an `NSD` at a time. Thereby, nodes have to be glued appropriately as indicated by $\sim$. The numbers on top of the derivation nodes (a derivation node is a node of a derivation tree) indicate which derivation nodes from layer L_1 have been combined. The costs following from the cost function and the distances in the input diagram are also shown. After layer L_2 is computed those derivation nodes are filtered out that consume a subset of edges of another derivation node and have the same root label, but cause higher costs. That way, `2+1`, `3+1` and `3+2` can be directly removed from L_2, but not `1+3`, because the cheaper derivations consume other edges. In the top layer L_3 then there are only three valid combinations possible. The best one is `1+(2+3)`. This derivation is even better than just `1+3`. Consequently, for higher layers, `1+3` and all derivation nodes that have `1+3` as a sub-tree would also be removed after the computation of layer L_3 is completed. In this simple example L_3 is the top layer though.

So, in the clean-up phase undesirable derivation nodes are removed across lower (and equal) layers, i.e., after layer i is processed, layers 1 to i are cleaned up. The savings thanks to this improvement are significant. Reconsider the exploded NSD shown in Fig. 1. Tab. 1 opposes the old and the extended parser regarding the filling level of layers. It can clearly be seen that after processing a higher layer elements from lower layers are removed. The notation x/y is used for the currently processed layer to also provide the number of elements in this layer before clean-up. Removing derivation nodes in lower layers in turn further increases the performance for filling higher layers. In the example this boils down to just a single derivation in layer 6.

3.2 Discussion

The proposed approach significantly reduces the effort for contracting diagrams compared to [MMM08c]. However, for the contraction of large diagrams the performance is still not sufficient. The reason is that lots of undesirable derivations are kept quite long, because a better derivation

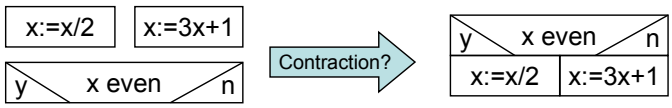


Figure 7: Undesirable contraction?

at a higher layer is found too late. There are two effective ways to deal with this issue:

- Introduce a special cost ∞ that directly triggers the removal of the respective derivation node. For instance, in case of NSD it could be stated that way, that the order of primitive statements must not be changed or that a condition always has to be above its branches. As a positive effect of this approach unexpected contractions like the one shown in Fig. 7 are directly excluded. However, it has to be accepted that completeness of contractions is lost that way, i.e. there might be a correction but contraction does not yield a result.
- Introduce a cost bound per component. This cost bound is multiplied with the number of components processed by a derivation and has to be adhered in order to actually perform a reduction. This is very effective. However, it might be a balancing act to set a reasonable cost bound that improves performance but does not lose valuable solutions.

With either of these approaches (or a combination thereof) practical performance can be achieved. Note that the presented parser still supports completion, i.e., the insertion of new edges if required. Thus, a cost can also be assigned to the introduction of each new hyperedge. In case of NSD, depending on this cost the parser would evaluate whether it is better to insert a new statement or to relocate an existing one. However, this has not been investigated yet.

4 Scope

The proposed approach actually has a wider scope than just "NSD-like languages". To understand this one has to consider how (certain, context-free) graph-like diagrams can be modeled with hypergraphs. In this context, connecting lines and arrows often can be reduced (in DIAGEN by the reducer component, cf. Fig. 4) and, thus, are not part of the abstract syntax anymore. As an example consider the flowchart shown in Fig. 8:1a. Its hypergraph model is shown in Fig. 8:1b (for further information regarding this example consult [Min02], where flowcharts are used as a running example to clarify the DIAGEN approach). The reduction of the arrows has caused certain nodes to be glued. The same diagram without the connecting arrows is given in Fig. 8:2a. Again, the corresponding ASG is shown in Fig. 8:2b. A low cost correcting patch for this ASG would glue the nodes such that its application just results in Fig. 8:1b. Following the extended editing process as shown in Fig. 4, the update translator then would introduce new arrows between the respective components where nodes have to be glued. So, diagram contraction here means, connecting particular components by lines or arrows.

Such contraction improves the editing performance even more. Whereas in case of NSD just the precision work is saved, this time several new diagram components, i.e. the missing arrows, are inserted automatically. Usually two (quite precise) mouse clicks are necessary for drawing

a single arrow manually (bend points not even included). It is worthwhile to save those, at least where possible. Unfortunately, this approach cannot be applied to context-sensitive languages like class diagrams: First of all, such languages do not exhibit a recursive structure as required. Furthermore, edges like associations are first-class citizens of the abstract syntax.

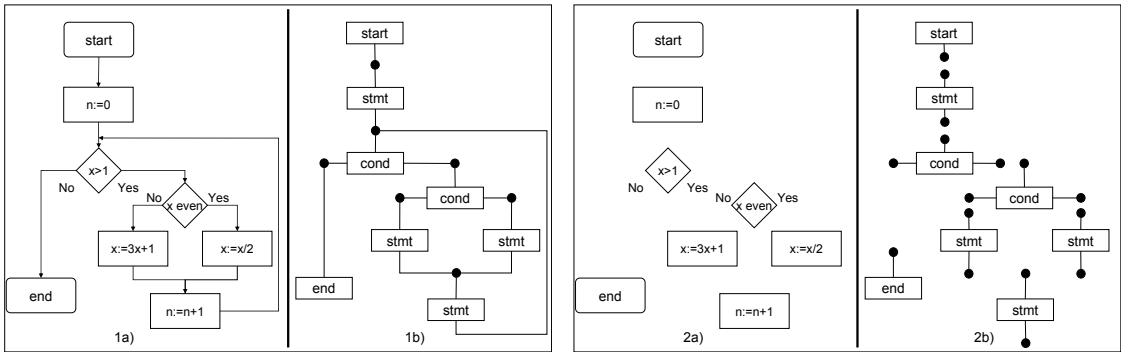


Figure 8: Flowchart with and without connectors plus corresponding abstract syntax hypergraphs

Such a feature has already been realized (by manual programming though) for the WebSphere Business Modeler. There, it is called "auto-link" [WGK+08][2]. It is described as "some magic where you simply place the activities of your business process in some approximate arrangement and your modeling tool connects them for you". This transformation enables users to automatically create connections between existing tasks and subprocesses. This means, the user only places the desired tasks and subprocesses on the canvas following some simple layout guidelines and the transformation does all the tedious connection drawing. It can be applied to both unconnected and partially connected models to quickly create complex business processes.

DIAGEN editors for logic gates and flowcharts have already been realized that support this kind of auto-linking as a special case of contraction. Indeed, this is absolutely the same editor function. Business process models are context-free to a large extent, so that the proposed approach to diagram contraction indeed should be applicable.

5 User Study

In this section the results of a first user study are provided. It has been conducted in order to verify the assumption that it is much faster to draw an exploded diagram than a concise one. Test users had to proceed as follows:

- learning stage: getting used to the NSD editor
- creation of the concise NSD given in Fig. 9 (left-hand side)
- creation of a corresponding exploded NSD and contraction thereof; the intermediate result of one user is shown in Fig. 9 (right-hand side)

[2] Unfortunately, this is an internal report. The auto-link feature has been demonstrated at the BPM 2008 conference together with the results of [GKW08].

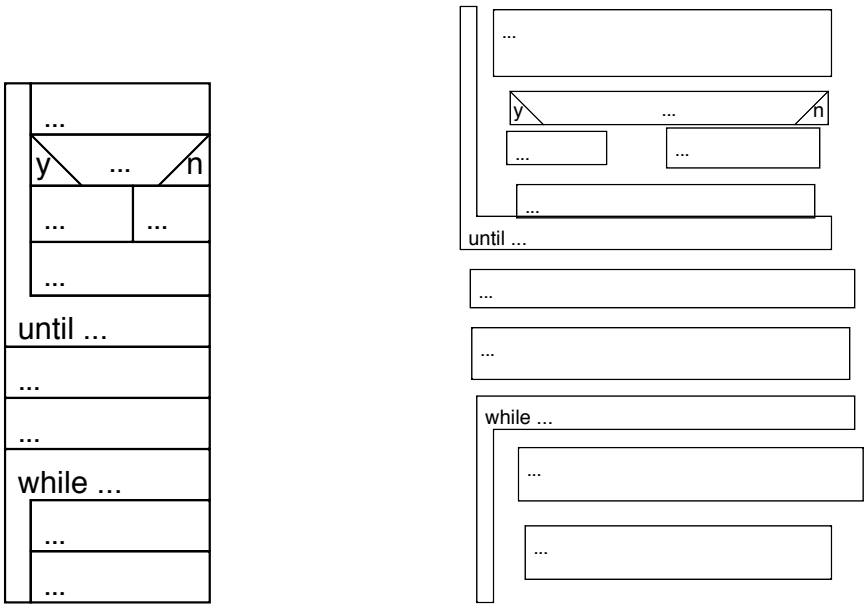


Figure 9: NSD to be created and the one actually created by a user

The empirical results shown below indeed support the claim of this paper. The task performance is increased by a factor of more than 1.5 (at least in this scenario). All drawn diagrams had been contracted by the editor as intended. Remarkably, it can be observed that users even get better in drawing exploded diagrams. For instance, in a second run several users had not created the diagram from top to botton anymore, but rather by kind of component (i.e., many statements had been added at once). That way, the time for switching the component kind had been saved.

User	time (s) for concise NSD	time (s) for exploded NSD	improvement factor
1	80	45	1.78
2	55	33	1.67
3	63	33	1.91
4	58	32	1.81
average	64	35.75	1.79

Table 2: Empirical results

6 Related Work

Cost grammars in the context of diagram editors have been successfully used previously to reason about the quality and interpretation of sketches, cf. [BM08]. There, the resolution of disambiguities, which frequently occur in hand-drawn diagrams, is effectively supported by syntax analysis in order to achieve a more reliable recognition. Since the authors achieved practical performance straight away no clean-up phase as ours had to be introduced.

Quite some research has been done regarding the creation and manipulation of exploded views. Most of these approaches keep the original diagram as an internal model though, so that an explosion can be collapsed again in an efficient way. In [LACS08] a powerful set of semi-automatic authoring tools and an interactive viewing interface for exploded views have been presented. The primary challenge there had been the specification how the parts of a system interact with each other. For this purpose, so-called explosion graphs are used that encode how parts explode with respect to each other. Furthermore, blocking constraints can be defined that must not be violated. A part can explode as long as all of its descendants in the explosion graph have been moved out of the way. Additionally, this graph stores the explosion direction of a part and its current offset from its initial position. These graphs can be computed automatically to a large extent using part hierarchies (to be defined by the user). In our approach, information from the grammar is exploited to gain information how parts might interact. In contrast to [LACS08] we can deal with whole classes of diagrams.

Finally, it has to be admitted that the practical importance of some kind of diagram contraction, namely auto-link, has been recognized already in [WGK+08]. However, the corresponding tool support most likely has been achieved by manual programming effort and, thus, cannot easily be used beyond business process models.

7 Conclusion

The proposed approach to diagram contraction is promising for several reasons: First of all, it can be used to improve the editing performance and accessibility of diagram editors. In addition the attribution of the grammar with a cost function has also appeared to be very handy. By this means the editor developer has much finer control of the parsing process. Undesirable derivations, hence, can be excluded at an early stage, and the performance possibly can be improved.

An editor with support for diagram contraction allows even freer editing. The distraction caused by the tedious precision work can be significantly reduced. First measurements have been conducted and the results are very promising: The editing performance for drawing a given NSD is improved by a factor of more than 1.5. The recognition rate is acceptable (there is the typical trade-off with performance). No additional specification effort is required except for the cost function. As an important special case of diagram contraction the auto-link feature has been discussed. With it the diagram components of certain graph-like languages can be connected automatically.

The proposed approach does not only support the contraction of a fully exploded diagram. It also respects already correct sub-diagrams. That way, exploded fragments can be integrated into the main diagram successively. All in all, diagram editing can be a more joyful task.

In the future this approach could be extended to 3D. Actually, most practically used exploded views are 3D, so this is a logical next step. Extending the scope of the proposed approach beyond context-free languages would also be important.

Screencasts of the NSD and flowchart example editors (including demonstrations of diagram contraction and auto-link) can be found at http://www.unibw.de/inf2/DiaGen/assistance/. The editor can be downloaded from there, too.

Bibliography

[BM08] F. Brieler, M. Minas. Ambiguity Resolution for Sketched Diagrams by Syntax Analysis Based on Graph Grammars. In Ermel et al. (eds.), *Proc. 7th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT '08)*. Electronic Communications of the EASST 10. European Association of Software Science and Technology, 2008.

[CM03] S. S. Chok, K. Marriott. Automatic generation of intelligent diagram editors. *ACM Trans. Comput.-Hum. Interact.* 10(3):244–276, 2003.

[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.

[GKW08] T. Gschwind, J. Koehler, J. Wong. Applying Patterns during Business Process Modeling. In Dumas et al. (eds.), *BPM*. LNCS 5240, pp. 4–19. Springer, 2008.

[LACS08] W. Li, M. Agrawala, B. Curless, D. Salesin. Automated generation of interactive 3D exploded view diagrams. *ACM Trans. Graph.* 27(3):1–7, 2008.

[Min02] M. Minas. Concepts and Realization of a Diagram Editor Generator Based on Hypergraph Transformation. *Science of Computer Programming* 44(2):157–180, 2002.

[MMM08a] S. Mazanek, S. Maier, M. Minas. An Algorithm for Hypergraph Completion According to Hyperedge Replacement Grammars. In Ehrig et al. (eds.), *Proc. of the 4th International Conference on Graph Transformations*. LNCS 5214, pp. 39–53. Springer, 2008.

[MMM08b] S. Mazanek, S. Maier, M. Minas. Auto-completion for Diagram Editors based on Graph Grammars. In Bottoni et al. (eds.), *2008 IEEE Symposium on Visual Languages and Human-Centric Computing*. Pp. 242–245. IEEE Computer Society Press, 2008.

[MMM08c] S. Mazanek, S. Maier, M. Minas. Exploiting the Layout Engine to Assess Diagram Completions. In Störrle and Fish (eds.), *Proc. 2nd International Workshop on Layout of (Software) Engineering Diagrams (LED 2008)*. Electronic Communications of the EASST 13. European Association of Software Science and Technology, 2008.

[WGK+08] J. Wong, T. Gschwind, W. Kleinoeder, J. Koehler, K. Muhidini, A. Maystrenko. Pattern-based Editing, Transformation and Refactoring of Business Process Models for WebSphere Business Modeler 6.1.1 and 6.1.2. Sept. 2008. IBM Internal Beta Release.

A Meta-Model-Based Approach for Specification of Graphical Representations

Merete Skjelten Tveit

merete.s.tveit@uia.no,
Faculty of Engineering, University of Agder
Grooseveien 36, N-4876 Grimstad, Norway

Abstract: Meta-models are widely used for the specification of the internal structure of graphical modelling languages, and well-established standards (e.g. MOF) exist for this. For the graphical representation there is not the same agreement and no related standards. This paper presents a new meta-language for an *independent* specification of graphical representations. A diagram from the domain-specific language *Service* is used as a running example to show how this meta-model-based approach is appropriate for specifying the graphical representation in a precise way, but still on a high level of abstraction.

Keywords: language specification, graphical representations, meta-models, mapping descriptions

1 Introduction

Graphical modelling languages are an important part of information technology and in the software development process as they are very suitable for visualising structures, compositions and relationships between elements. Unfortunately, there is not the same broad understanding and agreement of how these graphical languages are specified in a formal way. A formal language specification for a graphical language normally consists of a structure definition (also known as abstract syntax), a graphical representation definition (concrete syntax) and a definition of the semantics. A formal specification of the language, including the graphical representation is important both for the users and for the tool developers that intend to build appropriate tool support for the language.

Meta-models are well-known as specification approach for the *structure* of a language, and there already exist standards like MOF [OMG03] for this purpose. MOF is a meta-language, also called a meta-meta-model, and defines all the concepts that are necessary to specify the structure of a language. A language structure that is specified based on the concepts in MOF is said to *conform to* MOF. The UML 2.0 [OMG04] meta-model is an example of a language structure definition that conforms to MOF. The structure meta-model is in turn instantiated when models in the language are made. This kind of instance hierarchy is one of the fundamental concept in a meta-model-based approach.

For graphical representations, there still does not exist any standards similar to MOF. This paper presents a new meta-model-based approach defining a meta-language for specification of *graphical representations* which based on the same principles as MOF, that is importing and

reusing concepts from the UML Infrastructure [OMG07]. The approach follows the same three levels in the meta-model hierarchy as the structure specification, with the meta-language for specification of graphical representations at the same level as MOF. The meta-language defines all the concepts that are necessary to specify the representation of graphical languages at a very high level of abstraction, but still in a precise way. The graphical representation in this approach is treated and specified as a complete language aspect. The approach aims to describe the graphical aspects, at both the language level and the diagram level, as complete *constructions*, in a similar way that we know from structure specifications (i.e. the UML meta-model). The placement of the approach in the hierarchy is illustrated at the left-hand side of Figure 1. It is important to note that in this paper, the layers are not seen as absolute reference points, but relative to each other.

The main difference between the meta-language presented in this paper and already existing model-based approaches for specification of graphical representation is the combination of *independence*, *completeness* and *expressiveness*. The *independence* implies that the graphical representation is specified independent of the structure. This independence is a clear advantage when the languages are complex and when it is necessary to have more than one graphical representation for the structure. Because of this independence, it is important to have precisely defined mapping relations between the representation and the structure description. In this approach, the mapping is defined based on a mapping meta-meta-model. In addition to this, the meta-language gives the possibility to describe the graphical representation of both languages and diagrams as *complete* constructions. Features to describe all kinds of spatial relationships between elements in a graphical language also make the meta-language more *expressive* than many other approaches. The approach that is most similar to ours regarding independence is GMF [gmf], but this approach has weaknesses both regarding completeness and expressiveness. Section 5 presents these differences in more details.

The focus in this paper is mainly on the conceptual parts of the approach, nevertheless, a prototype that can be used to define graphical languages and generate graphical editors based on the description is implemented on the GMF platform, and is outlined in Section 2. The following sections will give a bottom-up description of the approach, starting at the diagram level. Section 3 presents the meta-model-based approach both for a specific diagram and for the language aspects (Section 3.2). Section 3.3 presents the most important concepts from the meta-language. Section 4 describes how the relationship between the structure and the representation is handled in this approach. The concluding remarks are found in Section 6.

2 An overview of the Approach and its Implementation

The biggest difference between a string language and a graphical language is the dimensional space of the sentences in the language. While sentences in a string language are linear, the sentences (i.e. the diagrams) in a graphical language have a minimum of two dimensions. The differences are in the concrete representation. We consider a diagram expressed in a graphical language as a collection of graphical elements that are arranged in various ways. There are different methods for describing how the graphical elements are located and arranged to form valid diagrams. One way is to specify the placement of an element *physically* by using concrete coordinates. The coordinates will give the exact position of an element. Another method is

to specify the placement *logically*. A logical placement is normally described by using spatial relationships to describe where an element is placed relative to other elements. A deeper outline on how graphical elements are spatially related and arranged in diagrams is found in [BG04]. In the approach presented in this paper, the arrangements of the graphical elements are also specified in a logical way using *spatial relationships*.

At the lowest level in the approach, M1, we have the *diagram description model* (see left-hand side in Figure 1), which describes the graphical representation of one particular *diagram*. This includes instances of the graphical elements, their properties and how they are related. The model is an *abstraction of a diagram*, and presents a complete construction of it. The diagram description model conforms to the *graphical representation meta-model* (M2) for the *language* the diagram belongs to. The graphical representation meta-model consists of two parts, *the graphical representation description* and the *basis library* which is language independent. The *graphical representation description* specifies all the graphical elements in the language, their *role* in the diagram and how they are spatially *related* to form well-formed diagrams. The *basis library* consists of a set of pre-defined geometrical shapes, which the graphical elements inherit from. These two parts are described more detailed in Section 3.2. At the upper-most level (M3) we have the *graphical representation meta-meta-model* which defines all the *role concepts* that are necessary for specifying the representation of a graphical language in a precise way. This meta-language re-uses, in its conceptual form, and extends, some concepts from UML Infrastructure. The three different levels are explained in more detail in Section 3.

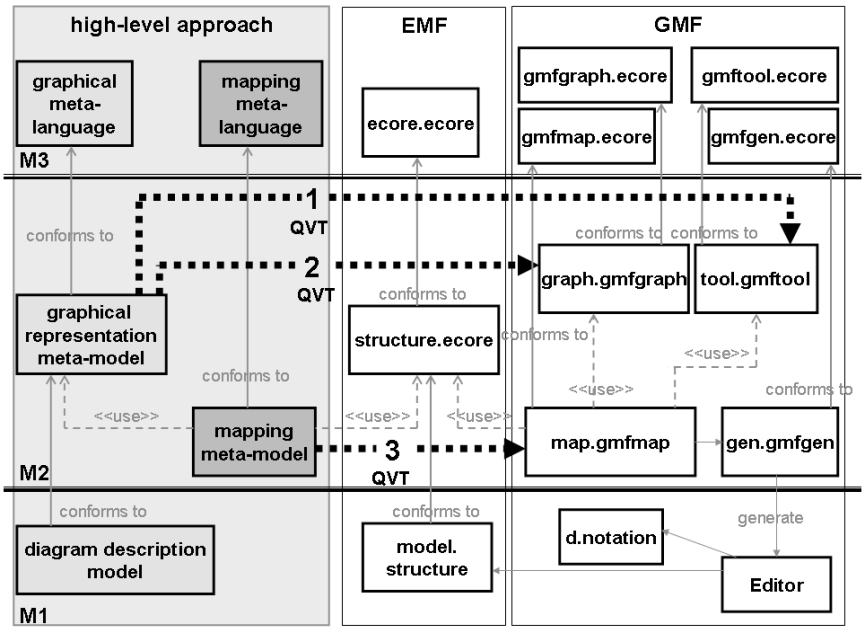


Figure 1: An overview of the high level approach (left-hand side) and its transformation to the GMF platform

The prototype for this approach is based on the Eclipse framework. The Eclipse Modeling Framework (EMF) [emf] makes it possible to easily create editors by generating code from meta-

models. The meta-language for graphical representation is implemented in the EMF framework, and by using the editors generated from the meta-language, it is possible to define the graphical representation for a specific language. This *high-level* description is then *transformed* to the lower level GMF framework [gmf] which is responsible for generating the graphical editor features that are necessary. The model-to-model transformations between the high-level specification and the GMF models are completely handled using QVT-R [OMG08], implemented using the tool *ikv++ medini QVT*. The transformations are illustrated in Figure 1 which shows the high-level approach on the left-hand side, the GMF framework on the right hand side (with the EMF in the middle since the structure meta-model is used by both approaches) and the transformation arrows in between. This transformation consists of three steps: first, from the graphical representation meta-model to graph.gmfgraph, second, from the graphical representation meta-model to tool.gmftool, and third, from the mapping meta-model (high-level approach) to map.gmfmap. These three gmf-models are sufficient for generating a GMF editor.

Currently, the users specify the graphical representation within the high-level approach in a textual editor, implemented for the meta-language using TEF [Sch08]. This is to make it easier and more user-friendly to create the meta-models for the graphical representations and to generate corresponding GMF graphical editors.

The following section will focus on the conceptual sides of the approach, presented from a bottom-up approach, starting with the diagram description model. There are some minor differences between the conceptual aspects and the practical aspects covered by the prototype. These differences will be described in the sections for the current aspect.

3 Specification of the Graphical Representation

As an example for the article, a diagram from the domain specific language *Services* will be used. The language is used to model service devices with plugs, and their connections. The example diagram in Figure 2 includes the following graphical elements: two *device symbols* with names “PC” and “Keyboard”, one *female plug symbol* with name “USB_in” and one *male plug symbol* named “USB_out”, one *connection point symbol* (the filled ellipse) and two *connector symbols*. It is not only important to identify the graphical elements in a diagram, it is also necessary to

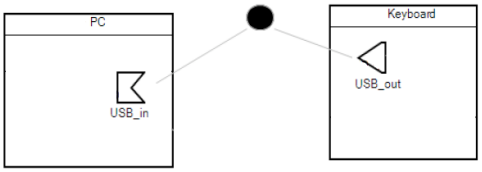


Figure 2: The service diagram used as running example

describe how the elements could legally be related to each other. The relations between the elements are what actually create the diagrams in a language. For the service diagram we can recognise the following relations between the graphical elements: The *device symbols* have two *compartments* placed **inside**, one *name compartment* and one *plug compartment*. A horizontal line, also placed **inside** the device symbol, is separating the two compartments. The *device name*

is placed **inside** the *name compartment*. The *plug symbols* are placed **inside** the *plug compartments* of the devices. The *plug names* are **associated with** the *plug symbols*. The *connector symbols* are **connected to** a *plug symbol* at their source end and to a *connection point* at their target end.

The next sections will give a bottom-up description of how the graphical elements and their spatial relationships can be specified in a meta-model-based way. We start with the description of the *diagram* in Section 3.1, the language graphical concepts in the service *language* are presented in Section 3.2 and finally the most important concepts in the graphical *meta-language* are presented in Section 3.3.

3.1 The Diagram Description Model (M1)

The graphical elements and how and where they are related form the most important aspects of the graphical representation of the service diagram, and are also what we would like to describe in the *diagram description model* in Figure 3. The graphical elements are represented as objects and the relations as plain links with role names representing the spatial relationships between the elements involved in the relation. They are all instances of the graphical representation specification for the language, specified on the level above (refer Section 3.2). Three kinds of

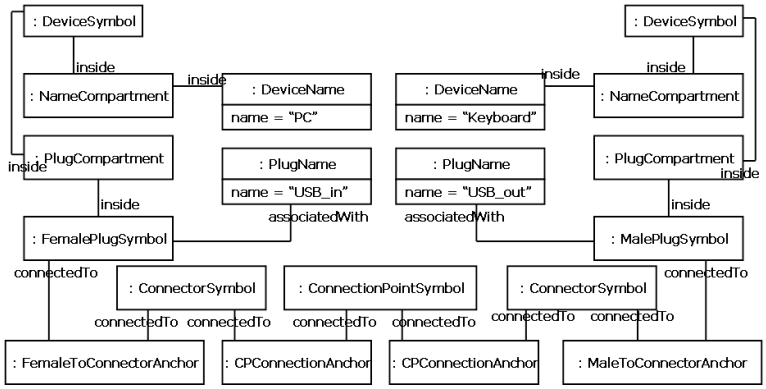


Figure 3: The diagram description model (M1) specifying a complete abstraction of the service diagram

relations were identified in the service diagram, and these three kinds of relationships are also presented in the model: *inside*, *connectedTo* and *associatedWith*. The *inside* relation represents one graphical element placed inside another graphical element, and is given in the model as a link between the two objects that are involved in the relationship, e.g. the *device name* "PC" inside the *name compartment*.

The *associatedWith* relationship describes a special kind of relation since it is not directly visible in the diagram. An example of use is the instance of a *plug name* which is associated with an instance of a *male* or *female plug*. In the diagram it is not possible to see that these elements have a concrete relationship in between, but they are still related with an "invisible" association.

The *connectedTo* relationship is used to describe that two (or more) graphical elements are

connected physically to each other. In the service diagram we have the *connector symbol* which at each target end is connected to the border of a *connection point symbol*. The connectedTo relationship (and also the associatedWith in some cases) also involves an *anchor* object (e.g. *cp connection anchor*) that specifies where the connecting appears. The anchor specifies a single point or a set of points that represent the concrete intersection point/area between the graphical elements that are involved. In the diagram description model (Figure 3), the *cp connection anchors* specify one single point at the border of the connection point where it intersects with the *target end* of a *connector symbol*.

The three spatial relationships, *inside*, *connectedTo* and *associatedWith*, are the only ones that are found necessary. Combining them with anchors gives possibilities to specify more complex relationships, and also more specific connection areas, than for the connectors and connection points.

Since the approach is implemented using EMF and transformed to GMF, the diagram description model is not a part of the prototype. Instead it is replaced by an instance of the notation model in GMF.

3.2 The Graphical Representation Meta-model (M2)

The model in Figure 3 is an abstraction of how the elements are arranged in the particular *diagram*. The objects in the model are instances of the graphical elements which are specified within the *language*. One particular diagram is just *one*, among many, legal representations in a language. At the language level in the meta-model hierarchy (see Figure 1) we have the *graphical representation meta-model* describing all the graphical elements in the language and their legal, spatial relations.

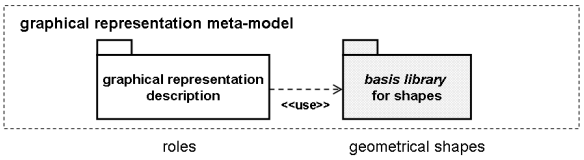


Figure 4: The aspects covered in the graphical representation meta-model at the M2 level

The *graphical representation meta-model* is separated into two parts: the *graphical representation description* and the *basis library for shapes* (language independent). The *graphical representation description* consists of an identification of all the graphical elements in the language, their roles (connection, container shape etc.) and how they are related to form well-formed diagrams. These elements are instances of the roles that are specified in the meta-language. The *basis library* contains a number of pre-defined shapes that are considered as the most important shapes for graphical languages. The abstract descriptions of the geometrical shapes are all instances of a UML::Class (refer Section 3.3). If necessary, the basis library can be extended. The relationship between the graphical representation package and the basis library package is expressed by the UML dependency *use* as illustrated in Figure 4, and the relationship between the graphical elements and the related pre-defined shapes is expressed by *inheritance* (also UML). This implies that the graphical elements get the semantics for their role in the language by in-

stantiation and their geometrical shape by inheritance.

A *device symbol* in the service language plays the role as a *container shape* and is shaped like a rectangle. Figure 5 illustrates how this is expressed in the meta-model: the *device symbol* which is an instance of container shape inherits from *rectangle* in the basis library. Being an instance of a container shape implies that the device symbol gets its semantics, i.e. it can have other graphical elements inside itself, from the container shape defined in the meta-language. This is opposed to simple shapes which can not act as a container for other graphical elements. The properties *bounding* and *attachment area* are inherited from the actual geometrical shape, in this case rectangle, since they depends on the geometry. The bounding is a *point set* representing the outer border (visible or invisible) of the geometrical shape. The attachment area is also a point set, representing the point or area where the shape could be legally related to other graphical elements through spatial relationships. In many cases, the bounding and attachment area are equal, like for the *connection point*. The classes Point and PointSet are part of the basis library. The connector plays the role of a *connection* and is shaped like an *arrow*. We use the connection

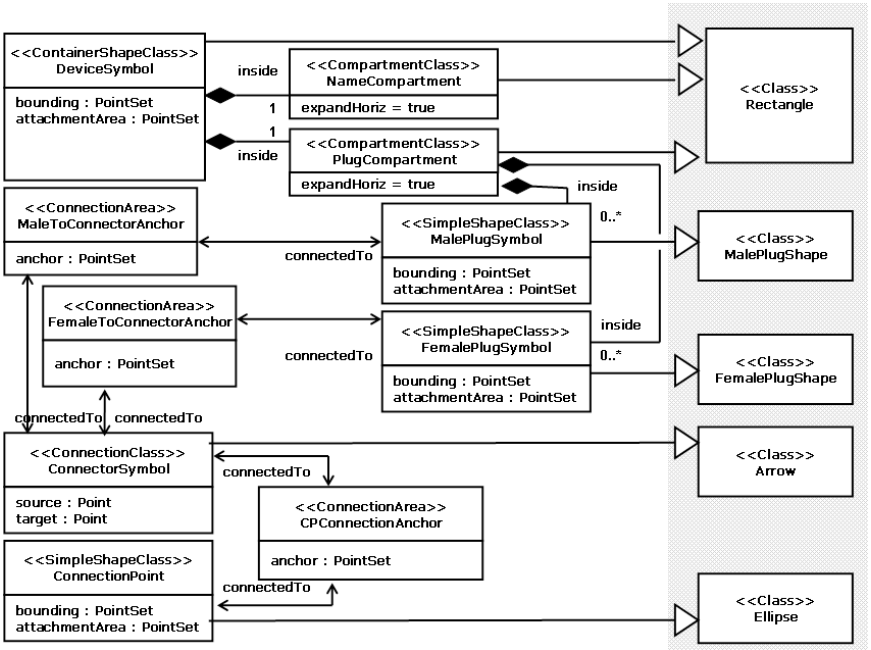


Figure 5: A subset of the *graphical representation meta-model* (M2) for the Service language

role when we have simple relationships like the connector symbol connecting to a shape in each end. The advantage of using *connection* is the possibility to easily state the positions of the source and the target as they are properties in the element. These values act as legal attachment areas for connections. The *anchor* property in the *connection area* is derived from the *attachment* properties for the graphical elements involved in the relationship, and represents the possible intersection point/area. For the *cp connection anchor*, the anchor is specified (by constraint) to be all the possible points where the *target* of the *connection symbol* intersects the *attachment area* of the *connection point*. On this level, the anchor is relative to the involved elements.

The *compartment* role is used for specification of compartments within a *container shape*. *Name compartment* and *plug compartment* are examples, and they both *expand* the container, the *device symbol*, horizontally.

Using a combination of spatial relationships (*inside*, *connected to* and *associated with*) and the connection areas for describing relationships between graphical elements makes this approach different from most other meta-model-based approaches. The advantage by using these features is the possibility to express complex spatial relationships on a high level of abstraction. The specification of the legal attachment areas for a graphical element is not a new idea itself, and especially in the traditional grammar-approaches (e.g. DiaGen [Min06]), attachment areas are used. In meta-model-based approaches on the other hand, these kinds of features are omitted in many cases, which makes it difficult to specify attachment areas that are unequal to the shape boundings. The meta-model-based approaches that are most similar to the one presented in this paper, are described in more detail in Section 5.

3.3 The Graphical Meta-meta-model (M3)

The *graphical meta-meta-model* is the meta-language at the M3 level in the meta-model hierarchy (see Figure 1), and defines all the concepts that are necessary for specification of graphical representations. This section presents a slightly simplified version of the meta-meta-model, with focus on the concepts that are used in earlier sections.

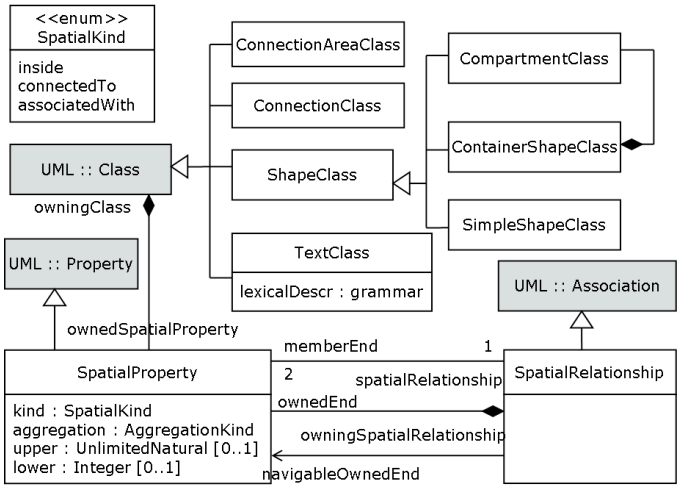


Figure 6: The meta-language (M3) for graphical representations

The basis element in the meta-meta-model is UML::Class which acts as super class for all the conceptual roles in the meta-language as illustrated in Figure 6. There are four special kinds of roles: *shape class*, *connection class*, *connection area class* and *text class*. A *container shape class* is a special kind of shape that acts as container for other graphical elements. A container shape can contain *compartments*, which are used to arrange their contents. The properties for the different roles are semantically generated from OCL constraints that are defined for the meta-language. The explicit constraints are omitted from the paper.

One of the interesting features in this approach is the possibility to describe spatial relationships between graphical elements. For this to be possible it is necessary to include concepts for describing complex relationships in the meta-language. Also for these features, UML Infrastructure is used as basis. The most important concept is the *spatial property*, which is an extension of UML *property*. This concept gives the possibility to express *spatial relationships* (extension of UML association) with aggregation, multiplicity and navigable end. The new feature presented in spatial property is the *spatial kind*. This property is used to specify which kind of spatial relationship (*inside*, *connectedTo*, *associatedWith*) that exist between graphical elements. All these features give the possibility to express the graphical representation for languages and diagrams as complete construction, which also is an aspect where this approach differs from other meta-model-based approaches (see also Section 5). As we can see, the meta-language for specification of graphical representations is not very different from MOF, but it contains some additional semantics which are especially related to conceptual roles and spatial relationships.

In Section 2, the use of EMF and GMF as implementation technologies for the meta-language was described. It is in this point the conceptual ideas and the implementation differs most, since the conceptual part of the approach are described re-using concepts from UML Infrastructure, while the implementation is based on EMF. These differences are solved as follows: The *UML::Class* is replaced by *EClass*. For the *spatial relationship*, which extends UML::Association, and *spatial property*, which extends UML::Property, the challenge is bigger. While UML uses associations, with properties for the association ends, EMF uses references to express relationships between classes. In the implementation, this is solved by merging the *spatial relationship* and *spatial property* into one unit, and let it extend *EReference*. The property *EOpposite* from *EReference* is used on the level below to express bidirectional spatial relations between graphical elements. By adding the *property kind* of type *spatial kind* to the new unit all the necessary semantics are kept.

4 Relating the Representation and the Structure

As we have seen in Section 3, the graphical representation is specified completely independent from the structure definition. This means there needs to be some kind of mapping defined between the two syntactic aspects to keep them synchronized. The complete separation is an important feature in this approach as it gives the possibility to have several representation definitions for the same structure and vice versa. The importance of keeping the structure and representation separated is discussed more widely in [Fon07]. The mapping meta-language that is defined within this approach is based on a study of the graphical languages SDL [ITU99] and UML [OMG04] and a categorisation of their relationships between the structure and the representation. From this study, three important mapping patterns were identified. The *one-to-one pattern* is the simplest kind of mapping, and also the most common. The *merge pattern* is used to map graphical concepts which have several notation options. A well known example is signal declarations in SDL. At last, the *partial description pattern* is used to map graphical concepts which have a partial representation in addition to a complete. Well-known examples are references and duplications. These patterns are also described with some more examples in [Tve08]. The patterns are implemented as a mapping meta-meta-model as presented in Figure 1, and a mapping

meta-model that conforms to it is transformed to the gmfmap model in the GMF framework. In

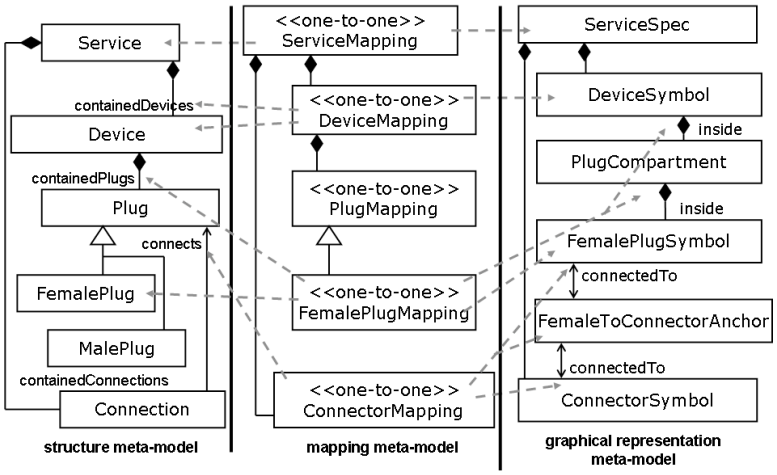


Figure 7: Illustration of mapping relationships in the Service Language

the Service language, all mapping relationships are described using instances of the *one-to-one* pattern. This is illustrated in figure 7.

5 Related Work

There exist a number of meta-tools that generate graphical editors from language specifications which involve meta-models to some degree: XMF-Mosaic [Cet], The Graphical Modeling Framework (GMF) [gmf], MetaEdit+ [Met05], The Generic Modeling Environment (GME) [LBM07] and Tiger [EEHT05] to mention some. All of these meta-tools present their own approach for handling and specifying the graphical representation. While most of these approaches (e.g. MetaEdit+ and GME) specifies the graphical representation on top of the structure meta-model and with this keep the graphical information as properties in the structure elements, GMF and XMF-Mosaic presents new, independent meta-languages for the graphical representation. The graphical representation description is then related to the structure meta-model by a syntactic mapping description. Of these two, GMF is the approach that shares most with the approach presented in this paper as it has a strong focus on separating the structure and the representation.

The biggest differences between the Eclipse-based GMF and the approach presented in this paper are related to two aspects: *expressiveness* and *completeness*. It is not possible to explicitly specify spatial relationships between graphical elements or specific attachment areas in GMF, except compartments, which make it possible to handle inside relationships. The relationships between nodes and connections are handled implicitly. This implicit specification makes the approach less expressive than the one presented in this paper. The other difference is related to the completeness of the graphical representation. The meta-language presented in this approach aims to specify the graphical representation as a complete construction (model), and not only single graphical elements collected in a container (canvas in GMF) without any relationships in between. In GMF, the relationships between a compartment and its content, and between

a connection and its connected node, are specified in the mapping description, and not in the graphical representation itself.

This is also the case for another approach [Fon07] that is worth mentioning. This approach is different from both GMF and XMF as it does not provide a new meta-language for the graphical representation, but instead proposes to extend the already existing structure meta-model by an extra layer of visual objects. This approach is interesting as it presents features for specification of spatial relationship (contains, overlap, connects and nearby) between graphical elements. The weakness with this approach, as with GMF, is that the spatial information is specified as a part of the mapping description, and not in the graphical representation itself.

6 Conclusion

This paper presents an approach for meta-model-based graphical representation. The central part of the approach is a new meta-language based on re-use and extensions of concepts from UML Infrastructure, which provides concepts for specification of graphical elements and their arrangements. The graphical representation is specified completely independent of the structure definition. The introduction of *connection areas* together with three kinds of *spatial relationships* (*inside*, *connected to* and *associated with*) gives the possibility to specify both simple and more sophisticated relationships between and within diagram elements on a high level of abstraction. The expressiveness, the possibility to describe the graphical representation for both the language and its diagram as a complete construction, and the complete independence from the structure specification are the main advantages with the approach presented.

For now, the conceptual aspects are the strength of this approach. Nevertheless, a prototype that is based on EMF and transformed to the GMF platform is implemented. With this, the approach can be used to specify the graphical representation of a graphical language, which together with a structure specification and the mapping description make it possible to generate a GMF editor from the specification. At this moment the meta-language has a textual representation and a textual editor implemented using TEF [Sch08]. The future plan is to also specify a graphical representation for the meta-language and generate a graphical editor for it, using the approach and the meta-language itself.

Bibliography

[BG04] P. Bottoni, A. Grau. A Suite of Metamodels as a Basis for a Classification of Visual Languages. *2004 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC 2004), 26-29 September 2004, Rome, Italy*, pp. 83–90, 2004.

[Cet] Ceteva. XMF. http://www.ceteva.com/index-resources.html

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual editors as eclipse plug-ins. In *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. Pp. 134–143. 2005.

[emf] Eclipse Modeling Framework.
http://www.eclipse.org/modeling/emf

[Fon07] F. Fondement. *Concrete Syntax Definition For Modeling Languages. PhD Thesis*. Ecole Polytechnique Federale De Lausanne, 2007.
http://library.epfl.ch/en/theses/?nr=3927

[gmf] Graphical Modeling Framework.
http://www.eclipse.org/gmf

[ITU99] ITU-T. SDL - ITU-T Specification and Description Language (SDL-2000). ITU-T Recommendation Z.100, 1999.

[LBM07] A. Ledeczi, D. Balasubramanian, Z. Molnár. An Introduction to the Generic Modeling Environment. In *Proceedings of MDD-TIF07, Model-Driven Development Tool Implementers Forum*. 2007.
http://www.dsmforum.org/events/MDD-TIF07/GME.2.pdf

[Met05] MetaCase. MetaEdit+. Version 4.0. Evaluation Tutorial. Technical report, MetaCase, 2005.
http://www.metacase.com/support/40/manuals/eval40sr2a4.pdf

[Min06] M. Minas. Syntax Definition with Graphs. *Electronical Notes in Theoretical Computer Science* 1:19–40, Feb. 2006.

[OMG03] OMG. *Meta Object Facility (MOF) 2.0 Core Specification*. Object Management Group, Oct. 2003. ptc/03-10-04.

[OMG04] OMG. *UML 2.0 Superstructure Specification*. Object Management Group, Oct. 2004. ptc/04-10-02.

[OMG07] OMG. *OMG Unified Modeling Language (OMG UML) Infrastructure, V2.1.2*. Object Management Group, Nov. 2007. ptc/06-10-06.

[OMG08] OMG. *Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification*. Object Management Group, Apr. 2008. ptc/07-07-08.

[Sch08] M. Scheidgen. Textual Modelling Embedded into Graphical Modelling. In *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*. Lecture Notes in Computer Science 5095, pp. 153–168. Springer, 2008.

[Tve08] M. S. Tveit. Towards Diagrammatic Patterns. In *Diagrammatic Representation and Inference, 5th International Conference, Diagrams 2008, Herrsching, Germany, September 19-21, 2008. Proceedings*. Lecture Notes in Artificial Intelligence 5223, pp. 427–429. Springer, 2008.