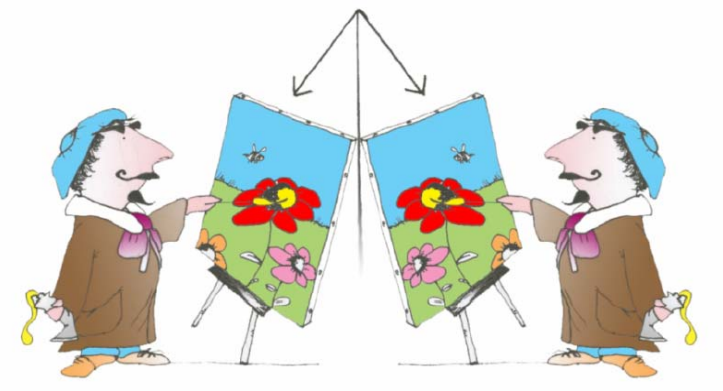
Pre-Proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007)



Satellite Event of **Supported by**

Editors: Karsten Ehrig, Holger Giese

Preface

This volume contains the proceedings of the Sixth International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2007), held in Braga, Portugal on March 31 and April 1, 2007, as a satellite event to the European Joint Conference on Theory and Practice of Software (ETAPS'07).

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), and Vienna (2006).

Due to the variety of languages and methods used in different domains, the aim of the workshop is to promote engineering approaches that starting from high-level specifications and robust formalizations allow for the design and the implementation of such visual modeling techniques, hence providing effective tool support at the semantic level (e.g., for model analysis, transformation, and consistency management). In fact, the workshop series attracts the interest of communities working on popular visual modeling notations like UML, Graph Transformation, Business Process/Workflow Models.

This year's workshop will have an additional focus on application of graph transformation and visual modeling techniques in engineering, biology, and medicine.

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 warmly thank Reiko Heckel for proposing us to organize the workshop in connection with ETAPS 2007. We also thank all our colleagues in the Program Committee and those who helped us as external reviewers. A special thank goes to the GT-VMT 2006 organizers for drawing the workshop logo. We are very grateful to the ETAPS 2007 organizers, for taking care of all the local organization.

The final proceedings will be published in the journal Electronic Communications of the EASST after the workshop. ECEASST is a fully refereed 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.

March 2007 *Karsten Ehrig and Holger Giese*

Program committee

Paolo Baldan (University of Venice, Italy)
Roberto Bruni (University of Pisa, Italy)
Andrea Corradini (University of Pisa, Italy)
Hartmut Ehrig (TU Berlin, Germany)
Karsten Ehrig (University of Leicester, UK) [co-chair]
Gregor Engels (University of Paderborn, Germany)
Reiko Heckel (University of Leicester, UK)
Holger Giese (University of Paderborn, Germany) [co-chair]
Gabor Karsai (Vanderbilt University, US)
Jochen Küster (IBM Zürich Research)
Mark Minas (Universität der Bundeswehr München, Germany)
Jörg Niere (University of Siegen, Germany)
Francesco Parisi-Presicce (University of Rome, Italy)
Arend Rensink (University of Twente, Netherlands)
Andy Schürr (University of Darmstadt, Germany)
Gabi Taentzer (TU Berlin, Germany)
Daniel Varró (TU Budapest, Hungary)
Bernhard Westfechtel (University of Bayreuth, Germany)
Hans Vangheluwe (McGill University in Montreal, Canada)
Martin Wirsing (Ludwig-Maximilians-Universität München, Germany)
Albert Zündorf (University of Kassel, Germany)

List of Referees

As already mentioned, the papers were refereed by the program committee and by the following external referees, whose help is gratefully acknowledged.

Florian Brieler
Pieter Van Gorp
Frank Hermann
Ákos Horváth
Ruben Jubeh
Anneke Kleppe
Thomas Maier
Ulrike Prange
Christian Schneider
Christian Soltenborn
Gergely Varro

Contents and Program for Saturday 31/03/2007

Contents and Program for Sunday 01/04/2007

Invited Session	
09:00 to 10:30	**2nd Invited Talk** **Topic to be announced.**
10:30 to 11:00	Coffee Break
Session on Application of Graph Transformations	
11:00 to 11:30	**Evaluating Workflow Definition Language Revisions with Graph-Based Tools** **(René Wörzberger, Markus Heller, Frank Häßler)** ***page 147***
11:30 to 11:45	**Graph Based Engineering Systems - A Family Of Software Applications And their Underlying Framewor [short talk]** **(Gregor Wrobel, Ralf-Erik Ebert, Matthias Pleßow)** ***page 159***
11:45 to 12:00	**Imposing Hierarchy on a Graph [short talk]** **(Brendan Sheehan, Benoit Gaudin, Aaron Quigley)** ***page 171***
12:00 to 12:15	**The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES [short talk]** **(Ulrike Ranger, Christian Fuß, Christof Mosler, Erhard Schultchen)** ***page 183***
12:15 to 14:15	Lunch
Working Groups	
14:15 to 14:30	**Building of Working Groups**
14:30 to 15:30	**Discussion in Working Groups**
15:30 to 16:00	**General Discussion of the Results**
16:00 to 16:30	Closing and Coffee Break

Membrane Computing [and Graph Transformation]

Gheorghe Păun

Institute of Mathematics of the Romanian Academy, Bucharest, Romania, and
Research Group on Natural Computing, Sevilla University, Spain
george.paun@imar.ro, gpaun@us.es

Membrane computing is a branch of natural computing initiated in [5] which abstracts computing models from the organization and the functioning of the living cell and from the cooperation of cells in tissues, organs (brain included) or other higher order structures. The resulting models, called P systems, can be briefly described as devices which process multisets of abstract objects in the compartments delimited by membranes. According to the arrangement of membranes, there are cell-like P systems (with the membranes embedded hierarchically), tissue-like (with the membranes placed in the nodes of an arbitrary graph), and neural-like P systems (with a special case, of spiking neural P systems).

A P system can be used as a computing device, generating/acccepting sets of numbers, of vectors of numbers, languages, sets of trees or graphs, arrays, etc. Many variants were considered, with biological, mathematical, or computer science motivation, and most of them were proved to be Turing complete. When an enhanced parallelism is available, e.g., by means of membrane division, computationally hard problems (typically, **NP**-complete problems) were solved in polynomial time – by a space-time trade-off.

Recently, membrane computing was much used as a framework for devising models in biology, economics, linguistics, computer science, optimization.

The talk is intended to be a general introduction to membrane computing, starting by placing it in natural computing, presenting the basic ideas and main types of results and applications, and pointing whenever necessary the interplay with graph theory and graph transformation (graph theory provides ideas/tools for studying P systems, while P systems can be used for handling graphs, e.g., as objects in membranes, or indirectly, as graphs describing membrane structures). Research topics are mentioned. No biological background is necessary.

For further (introductory) details in membrane computing, the reader is referred to the monograph [6], the volume [1], the papers [7], [3], as well as to the web page [10] (a complete bibliography of the domain can be found at this site, many downloadable papers, software, applications, etc.).

Bibliography

[1] G. Ciobanu, Gh. Păun, M.J. Pérez-Jiménez, eds.: *Applications of Membrane Computing*, Springer, Berlin, 2006.

[2] R. Freund, M. Oswald, A. Păun: P systems generating trees. In G. Mauri et al., eds., *Membrane Computing, International Workshop, WMC5, Milano, Italy, 2004, Selected Papers*, LNCS 3365, Springer-Verlag, Berlin, 2005, 221–232.

[3] M. Ionescu, Gh. Păun, T. Yokomori: Spiking neural P systems. *Fundamenta Informaticae*, 71, 2-3 (2006), 279–308.

[4] N. Jonoska, M. Margenstern: Tree operations in P systems and λ-calculus. *Fundamenta Informaticae*, 59, 1 (2004), 67–90.

[5] Gh. Păun: Computing with membranes. *Journal of Computer and System Sciences*, 61, 1 (2000), 108–143 (and TUCS Report 208, November 1998, `www.tucs.fi`).

[6] Gh. Păun: *Membrane Computing. An Introduction*. Springer, Berlin, 2002.

[7] Gh. Păun, G. Rozenberg: A guide to membrane computing, *Theoretical Computer Sci.*, 287, 1 (2002), 73–100.

[8] Gh. Păun, Y. Sakakibara, T. Yokomori: P Systems on graphs of restricted forms. *Publ. Math. Debrecen*, 60 (2002), 635–660.

[9] R. Rama, H. Ramesh: On generating trees by P systems. In *Proc. SYNASC 05*, Timişoara, Romania, IEEE Press, 2005, 462–466.

[10] The P Systems Web Page: `http://psystems.disco.unimib.it`.

Rule-Level Verification of Business Process Transformations using CSP

Dénes Bisztray[1], **Reiko Heckel**[2]

Department of Computer Science, University of Leicester

[1] dab24@mcs.le.ac.uk, [2]reiko@mcs.le.ac.uk

Abstract: Business Process Reengineering is one of the most widely adopted techniques to improve the efficiency of organisations. Transforming process models, we intend to change their semantics in certain predefined ways, making them more flexible, more restrictive, etc.

To understand and control the semantic consequences of change we use CSP to capture the behaviour of processes before and after the transformation. Formalising process transformations by graph transformation rules, we are interested in verifying semantic properties of these transformations at the level of rules, so that every application of a rule has a known semantic effect.

It turns out that we can do so if the mapping of activity diagrams models into the semantic domain CSP is compositional, i.e., compatible with the embedding of processes into larger contexts.

Keywords: Business Process Reengineering, Activity Diagrams, Graph Transformation, CSP, Verification

1 Introduction

A *business process* is a flow of actions, representing the work of an individual, internal system, or external partner company, towards a definite business goal [Hav05]. A *business process model* is a specification of a set of business processes. Of the different aspects addressed by such models we will be interested in the behavioural one, the *workflow specification*.

When organisations adapt to new markets or optimise their business processes, their workflows need to change, too. Business Process Reengineering is concerned with the systematic analysis and redesign of business process models. Depending on the objectives, changes may apply to the internal structure of the process model, preserving its semantics, or to the behaviour itself, making the workflow more flexible or more restrictive, adding new features, etc.

Workflows are often modelled diagrammatically, e.g., by UML activity diagrams [OMG05]. Their semantics, the collection of all workflows conforming to the model, can be formalised as a (potentially infinite) set of action sequences, called *traces*. Rather than specifying these sets explicitly, one can define a mapping into a process calculus like CSP (Communicating Sequential Processes [Hoa85]). This allows the use of theories and tools for analysing properties of processes [Ros97]. In particular it becomes possible to check if a process transformation has the

desired semantic effect, e.g., if the processes before and after the change are equivalent, if one is a restriction or extension of the other.

However, if the workflow models are sufficiently complex, a complete formal analysis of the corresponding CSP processes may be impossible or too costly. Therefore, we are are looking for a formalisation of *local* process transformations, which can be analysed separately for their semantic effect and sequentially composed in order to implement more complex changes.

To define process transformations, we formalise the abstract syntax of activity diagrams in terms of typed graphs, where a type graph TG plays the role of a metamodel for the language and instance graphs G typed over TG represent individual diagrams. Changes to these diagrams can then be specified by typed graph transformation rules, providing us with a formalisation of local workflow redesign steps.

Hence our approach combines two main ingredients: CSP as a semantic domain and analysis technique for workflow models, and rule-based graph transformations for specifying local redesign steps. The main question is about the relation of one with the other: How does a redesign transformation affect the semantics of processes?

It turns out that the question can be answered at the level of rules if the mapping from activity diagrams to CSP is compositional in the sense that the mapping of a sub-activity diagram yields a sub-CSP process at the semantic level. Describing the mapping from activity diagrams to CSP by triple graph grammars [Sch94], we can use results from the theory of graph transformation to verify the compositionality of the mapping.

The paper is structured as follows: in Section 2, we informally introduce the concept of semantics based verification of business process redesign. In Section 3 we present the mapping of activity diagram to CSP. Section 4 formally states the requirements needed to verify the semantic effect of transformations at the rule level and discusses them with respect to the mapping defined before. Section 5 concludes the paper.

2 Business Process Reengineering

For a motivating example, we consider a simple transformation on a workflow that describes the unpacking of a notebook computer.

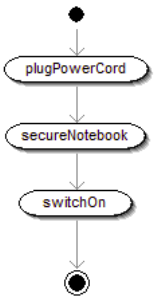


Figure 1: Sample process of unpacking a notebook

As Figure 1 illustrates, the workflow consists in three steps. First, we have to plug in the power

cord. Then, we secure the notebook with a cable lock. Finally, we have to switch on the computer. Analysing this process, we discover that this process could be made more flexible. For reacting to a situation where the cable lock is not available in time we may decide that the notebook can be switched on independently from securing it. The new process model, which allows both activities in either order as shown in Figure 2, represents a semantic extension of the old one.

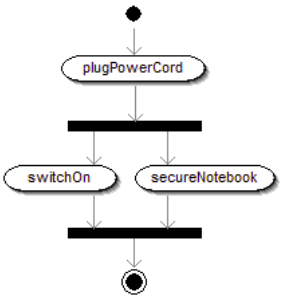


Figure 2: Redesigned notebook unpacking

Formally, the semantics of these processes is defined by standard CSP expressions. Up to substitution of process equations, the result of the mapping introduced in detail in Section 3 is shown below, with definition (1) describing the original process and equation (2-4) for the redesigned process.

$$UNPACK_O = plugPowerCord \rightarrow secureNotebook \rightarrow switchOn \rightarrow SKIP_{UNPACK_O}. \quad (1)$$

$$UNPACK_R = plugPowerCord \rightarrow (S_1 \parallel S_2) \quad (2)$$

$$S_1 = secureNotebook \rightarrow processJoin \rightarrow SKIP_{S_1} \quad (3)$$

$$S_2 = switchOn \rightarrow processJoin \rightarrow SKIP_{S_2} \quad (4)$$

$SKIP_A$ is defined as a process which does nothing but terminating successfully, with alphabet $A \cup \{\checkmark\}$ [Hoa85].

Analysing the processes before and after the transformation, we discover that

- the original process has only one trace $\langle plugPowerCord,\ secureNotebook,\ switchOn \rangle$;
- ignoring the *processJoin* action, in the redesigned process there are the two traces $\langle plugPowerCord,\ switchOn,\ secureNotebook \rangle$, $\langle plugPowerCord,\ secureNotebook,\ switchOn \rangle$.

The behaviour of the old process is thus present in the new one, i.e., $traces(UNPACK_O) \subseteq traces(UNPACK_R)$.

If the activity of unpacking a notebook is embedded into a larger process (e.g., setting up a workplace), it should be possible to derive the global consequences from the changes made to the smaller process. More generally, we want to predict the semantic effect of an operation before even performing it on the real (and potentially large) process. Thus, we formalise the change by

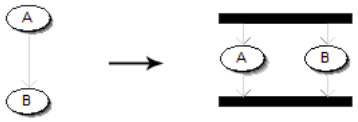


Figure 3: Redesign rule

graph transformation rules as the one sketched in Figure 3 and apply the mapping to CSP on its left- and right-hand side.

$$PROC_L = A \rightarrow B \rightarrow SKIP_{PROC_L} \tag{5}$$

$$PROC_R = (A \rightarrow processJoin \rightarrow SKIP_A \parallel B \rightarrow processJoin \rightarrow SKIP_B) \tag{6}$$

We can indeed observe that, after hiding the *processJoin* action, the traces of $PROC_L$ are included in those of $PROC_R$, and from general results about CSP trace refinement [Hoa85] it follows that this relation is closed under the embedding of CSP processes into context. To benefit from this fact we have to formalise and study the mapping of activity diagrams to CSP, which is described in the following section by means of triple graph grammars.

3 Mapping Activity Diagram to CSP

This section specifies a mapping from activity diagrams to CSP processes. Contrary to previous approaches [LBC00], we follow the approach of triple graph grammars (TGGs), where the abstract syntax of both activity diagrams and CSP processes are represented by typed graphs, i.e., instances of corresponding metamodels. A third metamodel as depicted in Figure 4, is used to capture the relation between corresponding elements of diagrams and processes. A brief introduction to TGGs is provided, however the detailed explanation can be found in [Sch94].

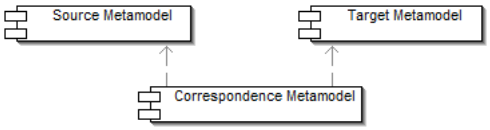
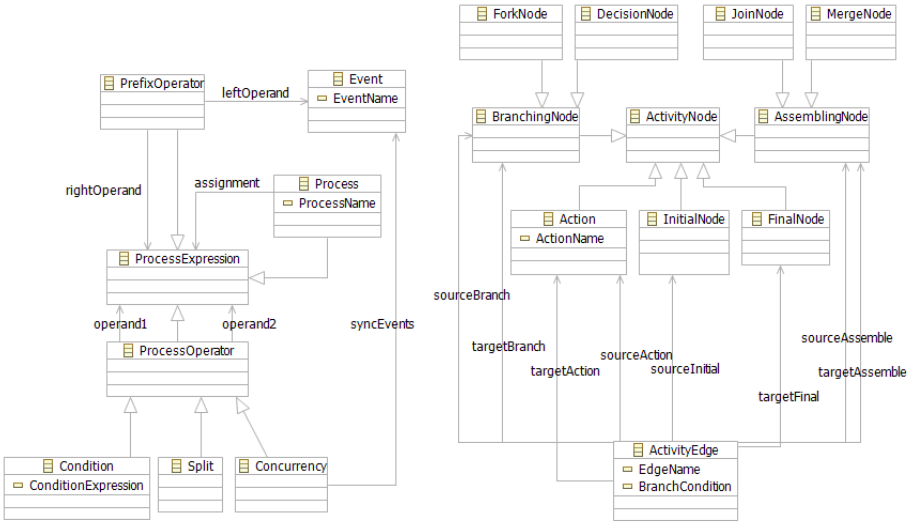


Figure 4: Triple graph grammar concept

3.1 Abstract Syntax

The metamodel for CSP processes, as far as required for our mapping rules, is shown in Figure 5(a). Following the Composite Pattern [BMR+96], a Process Expression either represents a Prefix $(x \rightarrow E)$ of a basic Event x followed by expression E, a Process equation $P = E$ assigning an expression E to a process name P, or a binary Process Operator combining two expressions.



(a) Abstract syntax for CSP (b) Abstract syntax for Activity Diagram

Figure 5: Abstract Syntax

- If b is a Boolean and E and F are process expressions, a Condition is an expression $E \not< b \not> F$ (*if b then E else F*);
- if E and F are expressions, Split is their sequential composition $E;F$ (*upon termination of E, continue as F*);
- if E and F are expressions Concurrency yields their synchronous parallel composition $E \parallel F$ (*perform E and F in lock-step synchronisation of shared events*).

A simplified metamodel for activity diagram based on [OMG05] is shown in Figure 5(b). Nodes not present in the standard document include the BranchingNode and AssemblingNode. The BranchingNode is the superclass of the fork-like nodes that have one incoming edge and several outgoing edges. The AssemblingNode is the superclass for the join-like nodes, that have several incoming edges and one outgoing edge. Without loss of generality we restrict Action nodes to have only one incoming one outgoing edge.

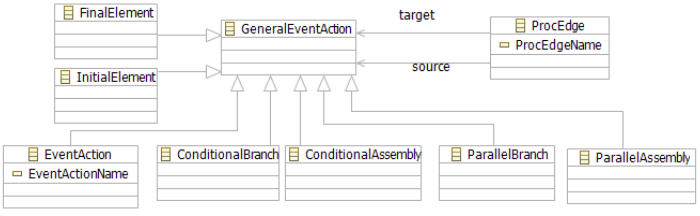


Figure 6: Correspondence metamodel

In order to be able to specify triple graph grammar rules, we require a correspondence metamodel as given in Figure 6. The EventAction is connected via associations to both Event in the Activity Diagram metamodel and Action in the CSP metamodel. The same is assumed for ProcEdge as the intermediary of Process and ActivityEdge, etc. Such associations are omitted from the illustrations to simplify the layout.

3.2 Transformation Method

Next we illustrate the design of our transformation rules, concentrating on a single rule for a detailed representation while using a semi-formal notation based on the concrete syntax for the others.

Consider the following simple example rule for transforming an Action node. The concrete syntax of the transformation rule is depicted in Figure 7.

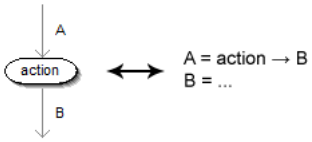


Figure 7: Action rule with concrete syntax

The idea behind the mapping is to relate an Edge in the activity diagram to a Process name in CSP. A previously introduced edge/process name *A* is defined in terms of a new prefix expression, and the continuation edge/process name *B* is introduced.

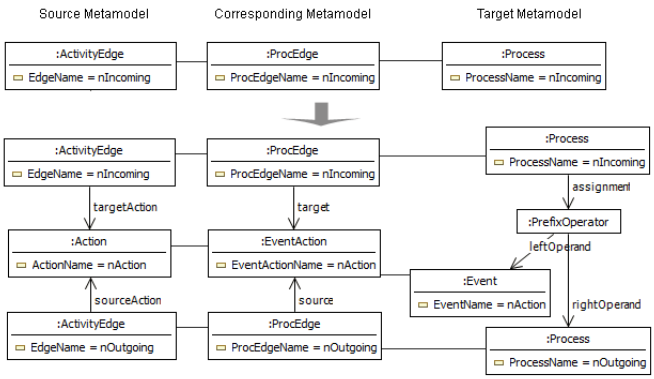


Figure 8: TGG Rule

The formal TGG rule is shown in Figure 8, to be read from top to bottom. In the top (the left-hand side of the rule) a triplet of ActivityEdge, ProcEdge and Process is matched. The bottom (right-hand side) generates simultaneously the Action and outgoing ActivityEdge of the activity diagram, the PrefixOperator with the Event and continuation Process and the relational elements between them.

In order to be used for a transformation from activity diagrams to CSP (rather than a symmetrical specification of their correspondence), the translation of the TGG rule to ordinary graph transformation rule is illustrated in Figure 9. There are various tools that enable the automatic generation of graph transformation rules from TGG rules. We match the pattern of an Action with incoming and outgoing edges, and create the corresponding CSP elements. A negative application condition [EEPT06] is defined for the relational element to prevent us from applying the same rule twice.

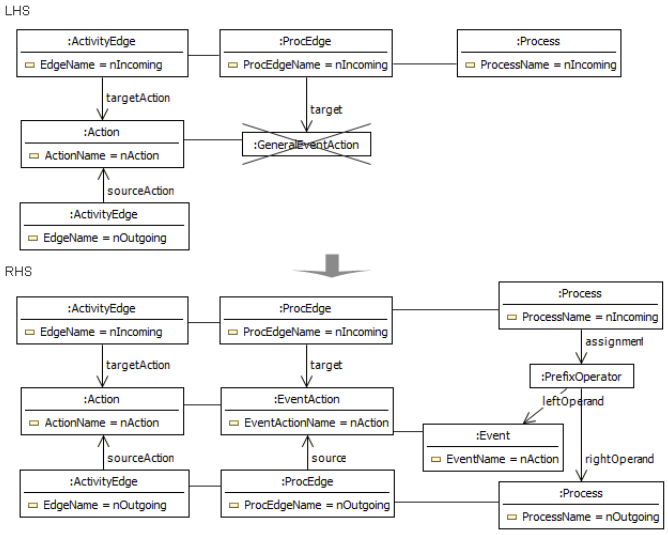


Figure 9: Graph Transformation rule

Indeed, a pragmatic benefit of TGGs consists in the use of the correspondence model for controlling the progress of transformation. The creation of relational elements, in combination with negative application conditions on the correspondence and target model, allow us to retain the original model and restrict ourselves to non-deleting rules. These two properties will be important later.

3.3 Transformation Rules

In this section we define the remaining transformation rules based on the concrete syntax of CSP and activity diagrams.

First, we consider the InitialNode depicted in Figure 10. Although this node is not mapped to anything directly, its outgoing ActivityEdge is related to a process definition with the same name. This will be the first process.

The transformation of a DecisionNode depicted in Figure 12 is a more complicated case. The concrete syntax is obvious, but Condition is a binary operator. Thus, we have to build a binary tree bottom-up as depicted in Figure 11. First we match the *else* branch with an arbitrary edge and create the lowest element of the tree. Then we build a tree adding the elements one-by-one, gluing its top to the Process that represents the incoming edge.

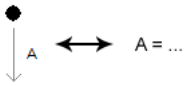


Figure 10: InitialNode

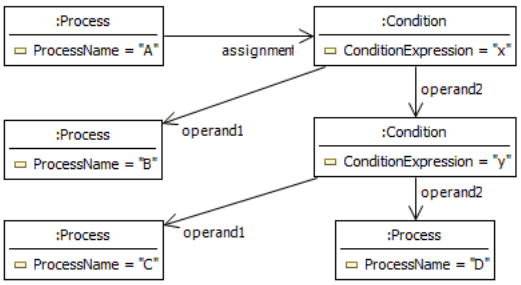


Figure 11: Abstract syntax tree for the result of DecisionNode transformation

Note that this transformation, which creates non-determinism at the syntactic level, leads to semantically equivalent processes. According to [OMG05], *the order in which guards are evaluated is undefined* and *the modeler should arrange that each token only be chosen to traverse one outgoing edge, otherwise there will be race conditions among the outgoing edges*. Hence, if guard conditions are disjoint, syntactically different nestings are semantically equivalent.

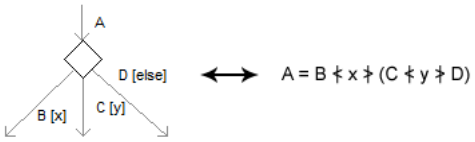


Figure 12: DecisionNode

The MergeNode is a simpler case, as illustrated in Figure 13. It is mapped to an equation identifying the processes corresponding to the two incoming edges.

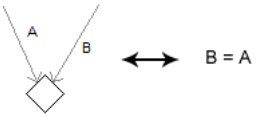


Figure 13: MergeNode

ForkNode and JoinNode represent the most complex cases. Before describing the transformation, we discuss some observations. If in an activity diagram the names of Action nodes are unique, the intersection of the alphabets of the corresponding processes is empty. This is partly intended because in this way the processes will not get stuck while waiting for some random

other process that accidentally has events with similarly names. On the other hand we need synchronisation points in order to implement the joining of processes. Thus we add an event *processJoin* to the alphabet of every participating processes. Since events that are in the alphabets of all participating processes require simultaneous participation, this fact is used to join concurrent processes by blocking them until they can perform the synchronisation event.

The rule for the ForkNode is shown in Figure 14. The Concurrency operator is binary, so by processing the nodes one-by-one, we create a binary abstract syntax tree of Concurrency nodes like we did in Figure 11 for the Condition operator. Since $P \parallel (Q \parallel R) = (P \parallel Q) \parallel R$, the different trees are semantically equivalent.

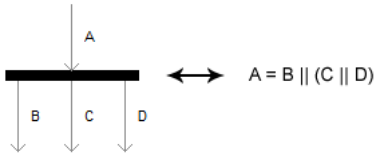


Figure 14: ForkNode

The transformation of JoinNode is depicted in Figure 15. The first edge that meets the JoinNode is chosen to carry the continuation process, while the others terminate in a *SKIP*. As we mentioned, CSP expressions 2-4) are based on this transformation. The process corresponding to D is substituted, since it terminates after the syncronisation.

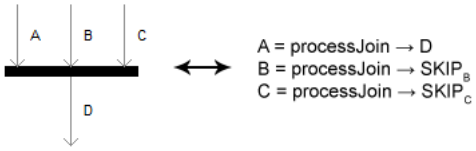


Figure 15: JoinNode

4 Rule-Level Verification

In this section we provide a formalisation of the notions required to define the compositionality condition of the mapping and state the main objective of the approach as a theorem. As mentioned in Section 1, we handle business processes as typed graphs, and reengineering steps as graph transformations. Since the contributions in this section are based on equivalence and refinement of CSP processes, we summarise the necessary definitions based on [Hoa85].

A *trace* is a finite sequence of symbols recording the events in which the process has engaged up to some point in time. The set of all traces of a process P is denoted by $traces(P)$. Processes P and Q are trace equivalent ($P \equiv Q$) if $traces(P) = traces(Q)$. P is a refinement of Q ($P \sqsubseteq Q$) if $traces(P) \subseteq traces(Q)$. A context is a process expression $E(X)$ with a single occurrence of a

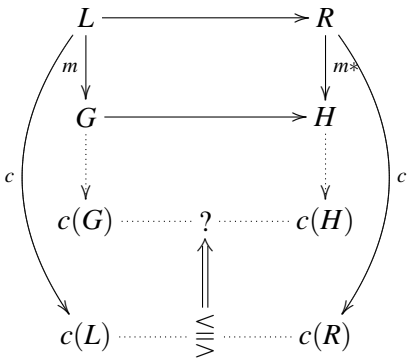


Figure 16: CSP correspondence for behaviour verification

distinguished process variable X. The relations of trace equivalence and refinement are closed under context, i.e., $P \equiv Q \implies E(P) \equiv E(Q)$ and $P \sqsubseteq Q \implies E(P) \sqsubseteq E(Q)$.

Denoting the mapping from activity diagrams to CSP by c, the idea of rule-level verification is illustrated in Figure 16. The original business process is given by graph G, the redesigned one by the resulting graph H of the application of rule $p : L \to R$ at match m and comatch m^*. Applying the mapping c to the rule's left- and right hand side, we compare the corresponding CSP expressions $c(L)$ and $c(R)$, checking, e.g., that $c(L) \sqsubseteq c(R)$ (the right process has more traces than the left one). From this relation at the level of the rule we hope to conclude that the same holds for all its transformations, i.e., $c(G) \sqsubseteq c(H)$. For such a property to hold, we make the following assumption on the mapping c.

Definition 1 (compositionality) A mapping c from graphs to CSP expressions is compositional if for all injective graph morphisms $m : L \to G$ there exists a context E such that $c(G) \equiv E(c(L))$. Moreover, this context is uniquely determined by the part of G not in the image of L, i.e., given a pushout diagram as below with injective morphisms only, and a context F with $c(D) \equiv F(c(K))$, then E and F are equivalent.

$$\begin{array}{ccc} K & \xrightarrow{l} & L \\ {\scriptstyle d}\downarrow & & \downarrow{\scriptstyle m} \\ D & \xrightarrow{g} & G \end{array}$$

Definition 1 applies particularly where L is the left hand side of a rule and G is the given graph of a transformation. In this case, the CSP expression generated from L contains the one derived from G up to equivalence, while the context is uniquely determined by $G \setminus m(L)$.

Theorem 1 *If a mapping c from graphs to CSP expressions is compositional, for all transformations $G \overset{p,m}{\Longrightarrow} H$ via rule $p : L \to R$ with injective match m, it holds that $c(L) \triangleq c(R)$ implies $c(G) \triangleq c(H)$, where $\triangleq$ is any relation in $\{\equiv, \sqsubseteq, \sqsupseteq\}$.*

Proof. By assumption the match m, and therefore the comatch $m^* : K \to H$ are injective. Since the mapping c is compositional, according to Definition 1 there are contexts E and F such that

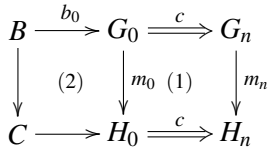


Figure 17: Extension diagram

$c(G) \equiv E(c(L))$ and $c(H) \equiv F(c(R))$. Now, $E(c(L)) \triangleq E(c(R))$ since $c(L) \triangleq c(R)$ and the relation is closed under context. Finally, $E(c(R)) \equiv F(c(R))$ by the uniqueness of the context. □

In the following we discuss how our mapping from activity diagram to CSP satisfies the compositionality condition. Since we did not introduce our mapping formally, we only provide a sketch of a proof. The main argument is based on the Embedding Theorem [Ehr77] and its extension to conditional graph transformations [Hec95]. In the basic version we assume a graph H_0 including a smaller one G_0 with inclusion morphism m_0. For a sequence of transformation $c: G_0 \stackrel{*}{\Longrightarrow} G_n$ we create a boundary graph B and a context graph C. The boundary graph is the smallest subgraph of G_0 which contains the identification points and dangling points of m_0. Since (2) is a pushout, the context graph can be determined. If none of the productions of c deletes any item of B, then m_0 is consistent with c and there is an extension diagram over c and m_0. This basically means that H_n is the pushout complement of c and m_0, thus can be determined without applying the transformation c on H_0. Hence, the compositionality condition holds for c.

Adding context and boundary graph to our picture, we get the extension diagram in Figure 17. The initial graph G_0 is either the LHS or the RHS of a production rule in the reenginering transformation. With our previous notation, G_n equals $c(G_0)$, i.e., our graph transformation implements the mapping c and our inclusion graph morphism is the match m_0 for the actual rule.

In our case, consistency of the embedding is trivial because, due to the use of triple graph grammars, our rules never delete nodes. Moreover, the embeddings we are interested in only add new context to the source model part of the graph, while negative conditions are only concerned with the correspondence and target part. This ensures that the embedding never violates any of the application conditions, i.e., the embedding theorem holds for the conditional transformations, too [Hec95].

Uniqueness of the context follows from the fact that the mapping c is deterministic.

5 Conclusion

In this paper we have studied the relation between two dimensions of model transformations: a semantic dimension representing a mapping of models into another formalism, and a dimension of change capturing the evolution of models. We have investigated an example and defined a condition which ensures that the effect of change on the semantics of models is predictable at the level of rules.

The approach differs from previous work on semantics-preserving transformation [BEH06] by the use of denotational rather than operational semantics. This makes is necessary to introduce a "semantic" formalism like CSP, but it allows the use of theories and tools of the semantic

domain for analysing models. A similar approach has been proposed in [EGHK02], but without formalising it.

Future work consists in completing the definition of the mapping and the proof that it satisfies the compositionality condition. In relation to business processes we intend to add a concept of observable vs. hidden actions to make more flexible use of existing notions of process equivalence.

Acknowledgements: This work has been partially sponsored by the project SENSORIA, IST-2005-016004. The authors also wish to thank Hartmut Ehrig for pointing out an improvement of the main theorem of the paper.

Bibliography

[BEH06] L. Baresi, K. Ehrig, R. Heckel. Verification of model transformations: A case study with BPEL. In Bruni et al. (eds.), *Proc. 2nd Symposium on Trustworthy Global Computing (TGC 2006), November 2006, Lucca, Italy*. LNCS. November 2006.

[BMR+96] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. *A System of Patterns*. Pattern-Oriented Software Architecture Volume 1. John Wiley and Sons, 1st edition, August 1996.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation (Monographs in Theoretical Computer Science)*. An EATCS Series. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[EGHK02] G. Engels, L. Groenewegen, R. Heckel, J. Küster. Consistency-preserving model evolution through transformations. In Jezequel et al. (eds.), *Proc. UML 2002, Dresden, Germany*. LNCS 2460. Springer-Verlag, Oct. 2002.

[Ehr77] H. Ehrig. Embedding Theorems in the Algebraic Theory of Graph Grammars. In *Fundamentals of Computation Theory, Proceedings of the 1977 International FCT-Conference*. Pp. 245–255. Poznan-Kórnik, Poland, September 1977.

[Hav05] M. Havey. *Essential Business Process Modeling*. Theory In Practice. O'Reilly Media, August 2005.

[Hec95] R. Heckel. Embedding of Conditional Graph Transformations. In Valiente Feruglio and Rosello Llompart (eds.), *Proc. Colloquium on Graph Transformation and its Application in Computer Science*. 1995.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International Series in Computer Science. Prentice Hall, April 1985.

[LBC00] G. Luttgen, M. von der Beeck, R. Cleaveland. A compositional approach to statecharts semantics. In *Foundations of Software Engineering*. Pp. 120–129. 2000.

[OMG05] OMG. Unified Modeling Language, version 2.0. Website, August 2005. http://www.omg.org/technology/documents/formal/uml.htm

[Ros97] A. W. Roscoe. *Theory and Practice of Concurrency*. Prentice Hall, 1st edition, November 1997.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In Tinhofer (ed.), *Proc. WG'94 Int. Workshop on Graph-Theoretic Concepts in Computer Science*. LNCS 903, pp. 151–163. Springer-Verlag, 1994.

Bisimulation Verification for the DPO Approach with Borrowed Contexts

Guilherme Rangel[1], Barbara König[2] and Hartmut Ehrig[3]

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

[2] barbara_koenig@uni-due.de
Institut für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany

Abstract:

Bisimilarity is the most widespread notion of behavioral equivalence and hence algorithms for bisimulation checking are of fundamental importance for verifying that two systems are behaviorally equivalent (seen from the perspective of the environment). We investigate this problem in the context of behavioral equivalences of graphs and graph transformation systems, where the extension of the DPO approach to borrowed contexts provides us with a formal basis for reasoning about bisimilarity of graphs. In this paper we extend Hirschkoff's on-the-fly algorithm for bisimulation checking, enabling it to verify whether two graphs are bisimilar with respect to a given set of productions. We then apply this framework to refactoring problems and verify instances of a model transformation which describes the minimization of deterministic finite automata.

Keywords: Bisimulation, Graph Transformation, Refactoring, Automata

1 Introduction

Model transformation [MV05] concerns the automatic generation of models from other models according to a transformation definition, which describes how a model in the source language can be transformed into a model in the target language. Such transformations can take place between different models or, more specifically, inside one single model (refactoring). Software refactoring is a modern software development activity to cope with the internal modification of source code to improve system quality, without changing the observable behavior.

Graph transformation systems (GTS) are well-suited to model not only refactorings but also model transformation (see [MT04] for the correspondence between refactoring and GTS). A GTS specifies model transformation by defining graph transformation rules to translate one model into another. The general idea is to have a graph describing an

instance of the source model as a start graph and to apply graph productions until no further production can be applied and the resulting graph is an instance of the target model. Model transformations via GTS can be found in [EE06, EW06, VVE+06]. A crucial question that must be asked is whether a given refactoring (or model transformation) is behavior-preserving, which means that transforming one model into another model does not change the original external behavior. In practice, the proof of behavior-preserving transformations is not an easy task and therefore one normally relies on test suite executions and informal arguments in order to improve confidence that the behavior is preserved. In a recent paper Narayanan and Karsai [NK06] proposed a method for checking bisimilarity in model transformations using GTS which is similar to ours. The new contribution of our paper is to present an efficient bisimulation checking algorithm, which works on the fly for infinite state spaces, and to develop the theory in the very general framework of borrowed contexts [EK04].

In this paper we give a formal treatment of the question of behavior-preserving refactoring. We define model refactoring by graph productions in the Double Pushout Approach (DPO) [CMR+97], which is one of the standards for GTS. Our goal is to show that instances of one model are bisimilar to their refactored counterparts, which implies behavior preservation. We employ the extension of DPO to borrowed contexts [EK04], which provides the means to reason about bisimilarity. We also extend Hirschkoff's [Hir01] on-the-fly bisimulation checking algorithm to deal with our setting. A case study of refactoring is presented in terms of minimization of deterministic finite automata (DFA), where we can test if a given DFA is bisimilar to its minimal refactored version. Since the procedure to check bisimilarity by hand is quite tedious we have implemented a tool in Objective Caml [OCa] to support this activity.

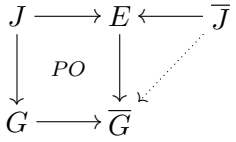
2 Graphs with Interfaces and Borrowed Contexts

In this section we recall the DPO approach to graph rewriting and its extension with borrowed contexts.

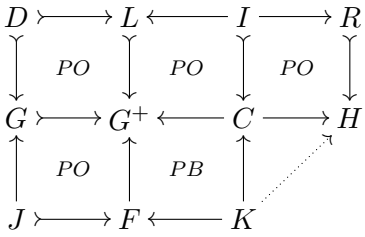
Definition 1 (**Graph and graph morphism**) A *graph* $G = (V, E, s, t, l_v, l_e)$ consists of a set V of nodes, a set E of edges, two functions $s, t\colon E \to V$ (source and target) and two labeling functions for nodes and edges $l_v\colon V \to \Omega_V$, $l_e\colon E \to \Omega_E$, where Ω_V and Ω_E are node and edge labels. A *graph morphism* $f\colon G_1 \to G_2$ is a pair of functions $f = (f_E\colon E_1 \to E_2, f_V\colon V_1 \to V_2)$, which is compatible with source, target and labeling functions of G_1 and G_2, i.e., $f_V \circ s_1 = s_2 \circ f_E$, $f_V \circ t_1 = t_2 \circ f_E$, $l_{e_2} \circ f_E = l_{e_1}$ and $l_{v_2} \circ f_V = l_{v_1}$.

In the standard DPO approach, graph productions rewrite graphs with no interaction with any other entity than the graph itself and the production. In the DPO with borrowed contexts [EK04] graphs have interfaces and may borrow missing parts of left-hand sides from the environment via the interface. This leads to open systems which take into account interaction with the outside world.

Definition 2 (**Graphs with interfaces and graph contexts**) A *graph G with interface* J is a morphism $J \to G$ and a context consists of two morphisms $J \to E \leftarrow \overline{J}$. The *embedding*[1] of a graph with interface $J \to G$ into a context $J \to E \leftarrow \overline{J}$ is a graph with interface $\overline{J} \to \overline{G}$ which is obtained by constructing $\overline{G}$ as the pushout of $J \to G$ and $J \to E$.



Definition 3 (**Rewriting with borrowed contexts**) Given a graph with interface $J \to G$ and a production $p\colon L \leftarrow I \to R$, we say that $J \to G$ reduces to $K \to H$ with transition label[2] $J \to F \leftarrow K$ if there are graphs D, G^+, C and additional morphisms such that the diagram below commutes and the squares are either pushouts (PO) or pullbacks (PB) with injective morphisms. In this case a *rewriting step with borrowed context* (BC for short) is feasible: $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$.



Consider the diagram above. The upper left-hand square merges L and the graph G to be rewritten according to a partial match $G \leftarrow D \to L$. The resulting graph G^+ contains a total match of L and can be rewritten as in the standard DPO approach, producing the two remaining squares in the upper row. The pushout in the lower row gives us the borrowed (or minimal) context F, along with a morphism $J \to F$ indicating how F should be pasted to G. Finally, we need an interface for the resulting graph H, which can be obtained by "intersecting" the borrowed context F and the graph C via a pullback. Note that the two pushout complements that are needed in Definition 3, namely C and F, may not exist. In this case, the rewriting step is not feasible. Let us also remark that the arrows depicted as $\to$ in the diagram above can also be non-injective (see [SS05]).

Note that our notion of labels exactly coincides with labels derived by relative pushouts [LM00, SS05].

3 Bisimilarity

Here we show how to use transition labels in order to check bisimilarity between two graphs with interfaces. A bisimulation is an equivalence relation between states of transition systems, associating states which can simulate each other.

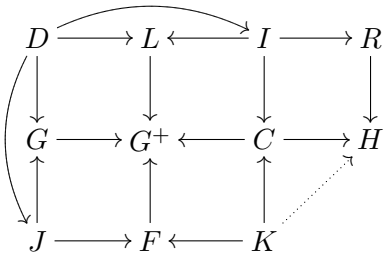
[1] The embedding is defined up to isomorphism since the pushout object is unique up to isomorphism.
[2] Transition labels, derived labels and labels are synonyms in this paper.

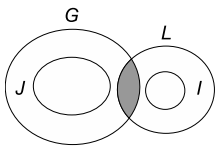
Definition 4 (**Bisimulation and Bisimilarity**) Let $\mathcal{P}$ be a set of productions and $\mathcal{R}$ a symmetric relation containing pairs of graphs with interfaces $(J \to G, J \to G')$. The relation $\mathcal{R}$ is called a *bisimulation* if, whenever we have $(J \to G)\,\mathcal{R}\,(J \to G')$ and a transition $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$, then there exists a graph with interface $K \to H'$ and a transition $(J \to G') \xrightarrow{J \to F \leftarrow K} (K \to H')$ such that $(K \to H)\,\mathcal{R}\,(K \to H')$.

We write $(J \to G) \sim (J \to G')$ whenever there exists a bisimulation $\mathcal{R}$ that relates the two graphs with interface. The relation $\sim$ is called *bisimilarity*.

In the graph setting not all labels that can be derived from a graph and a set of productions are relevant for the bisimulation. We will distinguish two kinds of transition labels.

Definition 5 Let $(J \to G) \xrightarrow{J \to F \leftarrow K} (K \to H)$ be a transition of $(J \to G)$. We say that the transition is *independent* whenever we can add two morphisms $D \to J$ and $D \to I$ to the diagram in Definition 3 such that the diagram below commutes, i.e., $D \to I \to L = D \to L$ and $D \to J \to G = D \to G$. We write $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ if the transition is not independent and we call it *dependent*.





An independent label has a borrowed context F that provides the entire left-hand side L for G and hence G does not contribute to the rewriting. (A trivial example is a label derived with $D = \emptyset$.) The figure above on the right schematically depicts this situation where the partial match occurs only in the overlap of the interfaces J and I leading to an independent label.

The bisimulation game for graphs mainly takes dependent labels into account. That is, if we modify Definition 4 in such a way that only dependent transitions $(J \to G) \xrightarrow{J \to F \leftarrow K}_d (K \to H)$ have to be simulated (either by a dependent or independent transition), then the resulting bisimilarity $\sim$ is unchanged (see [EK04]).

One of the main advantages of the borrowed contexts technique is that the derived bisimilarity is automatically a congruence, which means that whenever one graph with interface is bisimilar to another, one can exchange them in a larger graph without effect on the observable behavior. This is very useful for model refactoring since we can replace one part of the model by another bisimilar one.

Theorem 1 (**Bisimilarity is a Congruence** [EK04]) *The bisimilarity relation $\sim$ is a congruence, i.e., it is preserved by contextualization as given in Definition 2.*

Bisimulation proofs often yield infinite relations. Hence up-to techniques [San95] for bisimulation are useful to relieve the onerous task of bisimulation proofs by reducing the size of the relation needed to define a bisimulation. Bisimulation up-to is defined by replacing $(K \to H)\,\mathcal{R}\,(K \to H')$ by $(K \to H)\,\mathcal{F}(\mathcal{R})\,(K \to H')$ in Definition 4, where $\mathcal{F}$ is a function from relations to relations that defines the up-to technique (for details see [EK04]). We use for instance $\mathcal{F}^{iso}$ which generates all isomorphic copies of every pair in $\mathcal{R}$. A more powerful up-to technique is given by $\mathcal{F}^{context}$, which embeds all pairs into the same contexts (as in Definition 2), for all pairs and all compatible contexts.

The search of dependent labels among several partial matches might lead to cases where the pushout complement F or C (see Definition 3) does not exist and so the borrowed context step is not feasible. In [BGK06] a technique, based on initial pushouts, is defined to check if a partial match allows the existence of F and C.

Proposition 1 *[BGK06] Let $p\colon L \leftarrow I \to R$ be a production and $f\colon D \to L$ a monomorphism such that the diagram below on the left is the initial pushout of f. The pushout complement F of Definition 3 exists if and only if there is a monomorphism $D \to G$ and a morphism $J_D \to J$ such that the diagram on the right commutes.*

$$\begin{array}{ccc} J_D & \longrightarrow & F_D \\ {\scriptstyle g}\downarrow & IPO & \downarrow \\ D & \xrightarrow[f]{} & L \end{array} \qquad \begin{array}{ccc} J_D & \xrightarrow{g} & D \\ \downarrow & = & \downarrow \\ J & \longrightarrow & G \end{array}$$

Symmetrically one can check that the pushout complement C exists by taking the initial pushout over $D \to G$.

4 Partial Match Finding

Here we propose an algorithm that takes as input a graph with interface $J \to G$ and a set $\mathcal{P}$ of productions of the form $p\colon L \leftarrow I \to R$ to find all possible partial matches $G \leftarrow D \to L$ that will lead to dependent labels. We first need to introduce partial morphisms.

Definition 6 (Partial graph morphism) Let $G = (V, E, s, t, l_v, l_e)$ be a graph as in Definition 1. A *subgraph* S of G, written $S \subseteq G$, is a graph with $V^S \subseteq V^G$, $E^S \subseteq E^G$, $s^S = s^G|_{E^S}$, $t^S = t^G|_{E^S}$, $l_v^S = l_v^G|_{V^S}$ and $l_e^S = l_e^G|_{E^S}$. A *partial graph morphism* $f\colon G \rightharpoonup G'$ is a total graph morphism $f\colon dom(f) \to G'$ from a subgraph $dom(f) \subseteq G$ to G'.

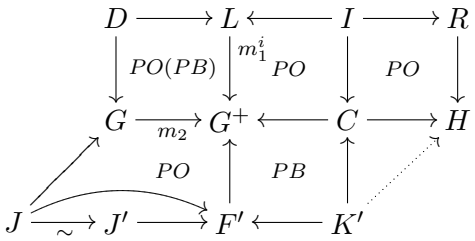
Given L (the left-hand side of a production) and G, we try to find partial matches which lead to a feasible BC step with a dependent label. We describe a procedure in 5 steps for one single production p, but it must be carried out for all productions of $\mathcal{P}$. Step **1** determines a subgraph L^{clean} of L, which is the largest subgraph of L containing only node and edge labels that also occur in G. The graph G^{clean} is defined analogously (with the roles of L and G exchanged). Step **2** creates all possible subgraphs L_i^{sub} $(i \in \mathbb{N})$ of L^{clean}. Step **3** finds all injective partial matches $pm_j\colon L_i^{sub} \rightharpoonup G^{clean}$

($j \in \mathbb{N}$). Step **4** splits each partial match pm_j as a span of total injective morphisms $L \leftarrow L^{clean} \leftarrow D_j \rightarrow G^{clean} \rightarrow G$. Step **5** stores $L \leftarrow D_j \rightarrow G$ as a partial match to derive a label if $L \leftarrow D_j \rightarrow G$ satisfies the conditions of Proposition 1 (for the existence of F and C) and will lead to a dependent label (Definition 5).

5 Matching Labels and Existence of Derivable Labels

The bisimulation game for graphs demands the comparison of labels. More specifically, two labels $\mu_i = J_i \rightarrow F_i \leftarrow K_i$ ($i = 1, 2$) are called isomorphic ($\mu_1 \cong \mu_2$) if they are isomorphic cospans. Remember that a dependent label can be answered by either a dependent or independent label. Since the algorithm in the previous section derives only dependent labels, we propose a way to check whether a dependent label for one graph is also derivable for the other graph. This if more efficient than deriving all independent labels for the other graph, which could be a lot, and checking whether they match.

Definition 7 (**Derivable Label**) Given a graph $J \rightarrow G$, a label $J' \rightarrow F' \leftarrow K'$ and a set $\mathcal{P}$ of productions, we say that $J' \rightarrow F' \leftarrow K'$ is *derivable* from $J \rightarrow G$ and $\mathcal{P}$ if it yields a feasible BC step, as in Definition 3.



This can be checked as follows: if there exists an isomorphism $J \xrightarrow{\sim} J'$ we obtain $J \rightarrow F'$ as the composition $J \xrightarrow{\sim} J' \rightarrow F'$ and in addition $G \rightarrow G^+ \leftarrow F'$ as a pushout of $G \leftarrow J \rightarrow F'$. For all productions $\mathcal{P}$ we find all possible total matches $m_1^i \colon L \rightarrow G^+$ ($i \in \mathbb{N}$). For each m_1^i and $m_2 \colon G \rightarrow G^+$, if m_1^i and m_2 are jointly surjective (i.e., $m_{1,V}^i(L_V) \cup m_{2,V}(G_V) = G_V^+$ and $m_{1,E}^i(L_E) \cup m_{2,E}(G_E) = G_E^+$) we can take $G \leftarrow D \rightarrow L$ as a pullback of $G \rightarrow G^+ \leftarrow L$ and thus obtain a pushout. We compute the pushout complement $G^+ \leftarrow C \leftarrow I$ of $G^+ \leftarrow L \leftarrow I$ and the pushout $C \rightarrow H \leftarrow R$ of $C \leftarrow I \rightarrow R$. We then check if there exists a morphism $K' \rightarrow C$ such that the rightmost square in the second row is a pullback and add the induced morphism $K' \rightarrow H$. If there exists a total match $m_1^i \colon L \rightarrow G^+$, which allows us to complete this diagram, we say that the label $J' \rightarrow F' \leftarrow K'$ is derivable from $J \rightarrow G$ and $\mathcal{P}$. Note that this is easier than partial match finding since we are only looking for total matches.

6 Algorithm for Bisimulation "On the Fly"

Classical methods for bisimulation checking (e.g., see [PT87]) take as input the full state spaces which are derived from the initial processes to be compared. Their drawback is

that the whole state space must first be computed and stored. Fernandez and Mounier defined in [FM91] a method for building the state space on the fly and checking bisimilarity based on depth-first search (DFS). Hirschkoff [Hir01] extended their work to not only allow breadth-first search (BFS), but also to deal with bisimulation up-to.

Hirschkoff's algorithm takes two states P and Q of labeled transition systems (LTSs) and checks their bisimilarity by analyzing their state space product, which consists of pairs of the form (P, Q) as states and transition labels μ between states indicating that both states are able to evolve along the same label μ, i.e., $(P, Q) \xrightarrow{\mu} (P', Q')$. The algorithm initially checks whether P and Q are immediately bisimilar (none of them has further labels leading to successor states) or non-bisimilar (one makes a step which the other is not able to mimic). If P and Q are not found immediately (non-)bisimilar, their state space product is expanded by adding their successors reached by a common label and so the bisimilarity of (P, Q) can be only known after the recursive analysis of all successors in the state space product. With this basic technique the LTSs in question must be finite. But in some cases the algorithm is able to perform finite proofs for states whose state space product is infinite using up-to techniques in order to handle infinite bisimulations as finite bisimulations up-to. Hirschkoff proved that the breadth-first version of the algorithm is computationally complete with respect to a given up-to technique $\mathcal{F}$, which means that the algorithm can check the bisimilarity of two states if and only if a finite bisimulation up to $\mathcal{F}$ relating the two states to be checked exists. Hirschkoff used his algorithm to check bisimilarity of polyadic π-calculus [Mil93] processes.

We extend Hirschkoff's algorithm to check bisimilarity between graphs with interfaces with respect to a given set of graph productions. We also did minor efficiency improvements and added extra details to the algorithm, trying to make clear aspects that were not easy to understand in the original version.

Remember that in order to calculate all dependent labels which originate from a given graph with interface and a set $\mathcal{P}$ of productions we employ the algorithm defined in Section 4 to find partial matches between the left-hand side of the rules of $\mathcal{P}$ and the graph with interface. For every partial match we then use Definition 3 to complete the whole borrowed context diagram, which gives us the dependent label and the resulting graph with interface. The matching of labels is specified in Definition 7.

In the setting of graphs, the bisimulation checking algorithm explores the state space product (defined below) of two graphs to be compared. We use here some shortcuts: graphs with interfaces $J \to G$ are represented as P and Q, and labels $J \to F \leftarrow K$ as μ.

Definition 8 (State Space Product) The *state space product* of two graphs P_0 and Q_0 is the transition system generated from the initial state (P_0, Q_0) using the following inference rules:

$$\text{dep1}: \quad \frac{P \xrightarrow{\mu}_d P' \quad Q \xrightarrow{\mu'} Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \text{dep2}: \quad \frac{P \xrightarrow{\mu} P' \quad Q \xrightarrow{\mu'}_d Q'}{(P,Q) \xrightarrow{\mu} (P',Q')} \qquad \mu \cong \mu'$$

The successors of (P, Q) are all (P', Q') such that P' and Q' respectively correspond to evolutions of P and Q along an isomorphic label μ. The rules *dep1* and *dep2* cover the situation when one dependent label (indicated with $\to_d$ above) is answered (i.e. matched)

by either a dependent or independent isomorphic label. If one graph can not answer, we say that the pair fails to evolve, i.e., we can infer immediately that P and Q are not bisimilar.

Definition 9 (**Failure**) Given two graphs P and Q we say that the pair (P, Q) *fails* to evolve whenever it holds:
$(P \xrightarrow{\mu}_d P' \wedge \nexists Q' \colon Q \xrightarrow{\mu'} Q' \text{ s.t. } \mu \cong \mu') \vee (Q \xrightarrow{\mu}_d Q' \wedge \nexists P' \colon P \xrightarrow{\mu'} P' \text{ s.t. } \mu \cong \mu')$.

The data structures used by the algorithm are: a structure **S** containing pairs of states that still have to be inspected; a *Table* storing information about each pair of graphs under inspection and three sets V, W and R, containing pairs of graphs that are respectively supposed to be bisimilar, known to be non-bisimilar and known to be bisimilar. By accessing **S** as stack (resp. queue) the algorithm performs a depth-first (resp. breadth-first) search on the state space product.

The main procedure is **bisimulation_check**, which calls: **succeeds**, **fails** and **propagate**. The procedure **bisimulation_check** first checks with **succeeds** whether (P, Q) is immediately bisimilar (e.g. none of them is able to derive any further (dependent) label). If (P, Q) is not immediately bisimilar, **fails** checks if the pair fails to evolve or if it is already known as non-bisimilar (i.e., it is in W). If it fails we insert it into W and use **propagate** to update the information about the state space product in *Table* with this new result, which can possibly lead to the discovery of new (non-)bisimilar graphs. When we find that a new pair evolves to other pairs, we assume that it is bisimilar (insert it into V) and only after the analysis of all its successors (where the notion of successor is specified in Definition 8) we are able to decide whether the pair is really bisimilar (as we assumed) or not. If by processing **S** we find a pair that is in V (supposed to be bisimilar) we move it to R and update *Table* using **propagate**. When all pairs have already been analyzed ($\mathbf{S} = \emptyset$) the algorithm can determine the bisimilarity of (P, Q).

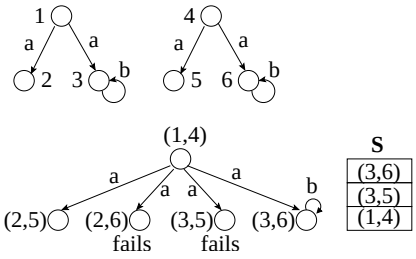


Table			
(P, Q)	*successors*	m	*fails*
$(1, 4)$	(2,5) (2,6) (3,5) (3,6) ● ● ○ ○	2 3 5 6 ☒ ☐ ☒ ☐	*false*
$(3, 5)$			*true*
$(3, 6)$	(3,6) ○	3 6 ☐ ☐	*false*

Above, one can see two small transition systems, their respective state space product, the states under investigation in **S** and their current information in *Table*. The states 1–6 represent graphs and a, b are labels. Consider the pair $(1, 4)$ in *Table*. The entry *successors* shows the successors of $(1, 4)$ in the state space product together with a boolean value *true* (●) or *false* (○), indicating which pairs of successors have already been analyzed. The entry m lists the successor states (e.g. 3 and 6 with *false* [☐], 2 and 5 with *true* [☒]), indicating which state has found a bisimilar partner. Whenever a pair of successors has been analyzed, if it turns out to be bisimilar both m-fields of the pair are set to *true* (☒). Successors $(2, 5)$ and $(2, 6)$ have been explored and only

$(2,5)$ is bisimilar. If all successors of (P,Q) have been analyzed (all are set to $\bullet$) and all fields of m are set to *true* ($\boxtimes$) then (P,Q) is bisimilar. If there is in m at least one graph with *false* ($\square$) then (P,Q) is not bisimilar. The entry *fails* indicates if a pair fails to evolve (according to Definition 9). A non-bisimilar pair has always at least one successor leading to a failure, but it is worth observing that a bisimilar pair might also have successors leading to a failure. In the example even though $(2,6)$ and $(3,5)$ fail, it is clear that $(1,4)$ is bisimilar.

```
bisimulation_check(P,Q) :=
  W := ∅;
  (*) R := ∅; V := ∅;
   insert (P,Q) into S and Table; status := true;
  while S ≠ ∅ do
     take (P0,Q0) from S;
     if succeeds((P0,Q0))
        then insert (P0,Q0) into R;
             propagate((P0,Q0), true);
     else if fails ((P0,Q0))
             then insert (P0,Q0) into W;
                  propagate((P0,Q0), false);
     else if [(P0,Q0) ∈ V]
             then move (P0,Q0) from V to R;
                  propagate((P0,Q0), true);
     else if [(P0,Q0) ∈ R]
             then propagate((P0,Q0), true);
     else {pair (P0,Q0) is new}
             insert (P0,Q0) into V;
             {(P0,Q0) -μ-> successor(P0,Q0)}
             insert successors of (P0,Q0) into S;
             for each successor(P0,Q0) do
               if successor(P0,Q0) ∉ Table
                 ∧ successor(P0,Q0) ∉ W ∪ R ∪ V
                 then insert it into Table;
             end for
  end while
if (P,Q) ∉ R
  then return false
  else if status then return true else loop back to (*)
```

```
succeeds(P,Q) := Table(P,Q).successors = ∅
    ∧ Table(P,Q).fails = false

fails(P,Q):=
  if (P,Q) ∈ Table
     then Table(P,Q).fails ∨ (P,Q) ∈ W
     else (P,Q) ∈ W

propagate((P,Q), success) :=
  if (P,Q) ∈ R ∧ success = false
       then status := false;
  if (P,Q) ∈ Table
     ∧ Table(P,Q).successors is complete[3]
       then remove (P,Q) from Table;
  for each (Pf,Qf) ∈ Table with (Pf,Qf) -μ-> (P,Q) do
       Table(Pf,Qf).successors(P,Q) := true;
       if success
          then Table(Pf,Qf).m(P) := true;
               Table(Pf,Qf).m(Q) := true;
       if Table(Pf,Qf).successors is complete
         then
           if ∃j = false ∈ Table(Pf,Qf).m
            then
               insert (Pf,Qf) into W;
               propagate ((Pf,Qf), false);
            else
               if (Pf,Qf) ∈ V
                  then take it from V to R;
               propagate ((Pf,Qf), true);
  end for
```

A new pair (P,Q) is inserted into *Table* as follows. Using Definition 8 and Definition 9 we can determine if (P,Q) fails to evolve and if it has successors. We insert (P,Q) into *Table* with the following data in case of failure: $successors = \emptyset$, $m = \emptyset$ and $fails = true$. If the pair does not fail we fill *successors* with the successors of (P,Q), m with the states of the successors of (P,Q) and all boolean values are set to *false*.

The procedure **propagate** is in charge of updating the information in *Table* concerning the state space product analysis. Every time we rediscover a new pair we decide that it is bisimilar (i.e., it is moved to R). If during the exploration of the state space we find that this very pair is non-bisimilar we set the variable *status* to *false*, which means that the result of the current run of the algorithm is not reliable. In this case we restart it retaining the information in W about states which are already known to be non-bisimilar.

The procedure **propagate** keeps in *Table* only the pairs under analysis, removing the

[3] This means that either all successors of (P,Q) have already been analyzed ($\bullet$) or $successors = \emptyset$.

ones that have already been analyzed. Observe that when one pair is propagated, the predecessors of this pair should also be informed of the new results. When a pair is propagated with *true* or *false* the algorithm always sets this pair as analyzed (*true* [•]) in the list of *successors* of each of its predecessors that are still under analysis (in *Table*). Only if the pair is propagated with *true* its respective graphs in m of its predecessors are set to *true* (☒). When the last successor of a given pair has been analyzed we can verify whether our initial hypothesis about its bisimilarity is still true. If the pair has at least one graph in m set as *false* it is non-bisimilar (our hypothesis was wrong) and we propagate this result. Otherwise we conclude that our hypothesis was correct and propagate this information.

If the algorithm has to handle bisimulation up-to we have only to replace in **bisimulation_check** $\boxed{(P_0, Q_0) \in V}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(V)}$ and $\boxed{(P_0, Q_0) \in R}$ by $\boxed{(P_0, Q_0) \in \mathcal{F}(R)}$, where $\mathcal{F}$ describes the up-to technique. For the refactoring proposed in this paper (see Section 8) we use the up-to context technique ($\mathcal{F}^{context}$) [EK04] informally described in Section 3.

Our version of the bisimulation checking algorithm is very similar to Hirschkoff's. Our main contribution to the algorithm is the full specification of how *Table* is used to store and process the state space product investigation. A small efficiency improvement can be seen in the **for each** statement of **bisimulation_check**, where we added extra conditions in order to avoid reanalyzing pairs already investigated. Furthermore we checked that the algorithm also works in our setting of borrowed contexts, taking into account especially the issue of independent and dependent labels.

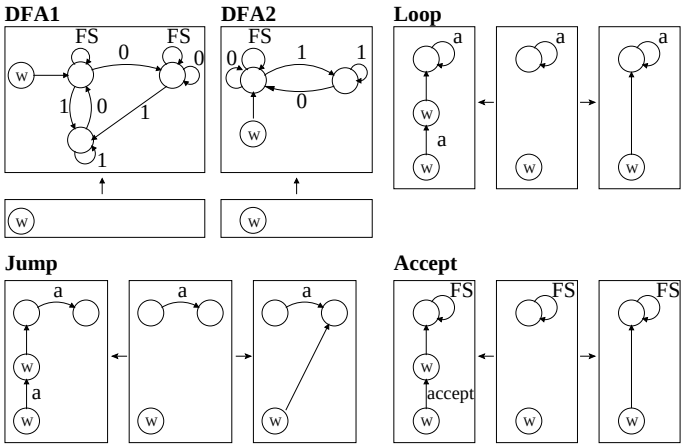
7 Tool Support for DPO with Borrowed Contexts

The derivation of labels and the bisimulation proof demand a great amount of time even for small examples and when done by hand they are particularly susceptible to errors. To overcome this we have implemented a tool in Objective Caml (OCaml), which is a functional language very appropriate for rapid prototyping. The tool uses directed labeled graphs and when we want to check the bisimilarity of two graphs, we specify a set of graph productions and also the graphs with interfaces to be checked. We have already implemented graphs with interfaces, graph productions, and procedures for label derivation and matching. OCaml is mainly textual, but for the sake of visualization, our graphs, rules and derived labels can be visualized with the package Graphviz [gra]. Our next goal is to implement the described algorithm for bisimulation checking.

8 Refactoring Deterministic Finite Automata

We will now come back to our original motivation: showing that refactoring preserves behavior. Let us first explain how the current theory of DPO with borrowed contexts could be used to prove that a refactoring process is behavior-preserving. Consider a model M, whose operational semantics is given by a set $OpSem^M$ of graph productions and a refactoring for this model M as a set $Refactoring^M$ of productions of the form

$L \leftarrow I \rightarrow R$. If we can prove for each rule of $Refactoring^M$ that we have the bisimilarity $(I \rightarrow L) \sim (I \rightarrow R)$ with respect to $OpSem^M$, this means that each rule does not change the behavior of the model under refactoring. Since bisimilarity is a congruence we can compose $(I \rightarrow L)$ and $(I \rightarrow R)$ with identical contexts and the respective compositions remain bisimilar. That is, for all instances of M, $Refactoring^M$ preserves the original behavior. This approach can be currently used only for refactoring rules without features such as negative application conditions and layers, which are often necessary to model refactorings.



Each DFA is described as a graph, where nodes are the states and directed labeled edges are the transitions (see DFA1 and DFA2 above). A loop labeled FS marks a state as final. An interface node labeled W has an edge pointing to the current state and this edge points initially to the start state. Note that the node W is the only possibility to establish interaction with the environment. The node W receives a letter (e.g. 'a') from the environment in form of an a-edge connecting W-nodes. Then, according to the operational semantics (given by the rules **Jump**, **Loop** and **Accept**) the DFA may change its state. Whenever an *accept*-edge between W-nodes is consumed by a DFA, this means that the string previously processed was accepted.

Below we define graph productions to minimize DFAs by eliminating equivalent states. The idea of the algorithm is to identify the distinguishable states, followed by the merging of equivalent states. Note that to the left of each rule we depict the negative application conditions. The algorithm is defined by several graph productions spread over three layers, where each layer applies its rules as long as possible before the rules of the next layer can be used. In practical terms, the transformation begins with rules of layer 0. If no more rule of layer 0 is applicable, the rules of layer 1 come into play. The rules in **layer 0** (see Figure 1) examine the transitions labels for every two states and determine if they are distinguishable. The rule in **layer 1** merges equivalent nodes, i.e., nodes without a *dist* edge between them. Finally, the rules in **layer 2** remove all *dist* edges and redundant transitions between two states.

Observe that the minimization algorithm demands rules spread over layers and negative application conditions and so we are not able to prove that all refactorings via

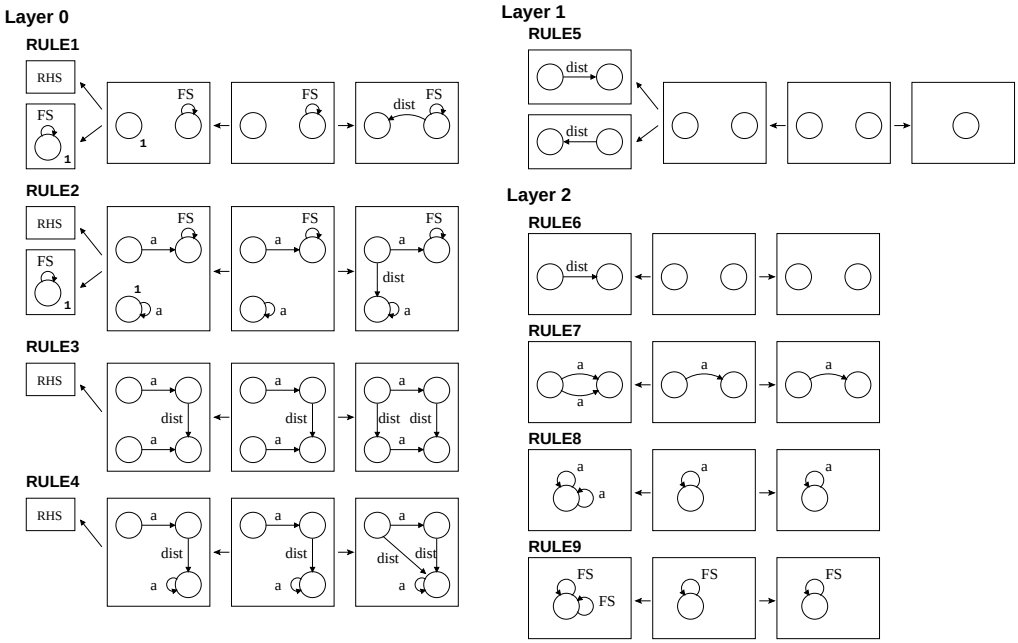


Figure 1: Productions for DFA minimization

these rules are behavior-preserving. For this reason our goal in this paper is to check that a given DFA and its minimal refactored version are bisimilar. Note that for DFAs borrowed context bisimilarity coincides with language equivalence. Furthermore in our setting bisimilarity on automata seen as transition systems corresponds to the bisimilarity that we obtain via the borrowed context technique.

Consider DFA1 and DFA2 previously depicted. By applying the minimization algorithm on DFA1 we obtain DFA2 as its minimal version. We then call the procedure **bisimulation_check** and verify that DFA1 and DFA2 are indeed bisimilar. In Figure 2 we show the state space product for this example, where the omitted interfaces of the graphs in the tuples contain only one node labeled W. Note that the state space product does not contain independent labels (which exist).

9 Conclusions and Future Work

We have shown how to use the DPO approach with borrowed contexts to automatically check the bisimilarity of systems specified in terms of graphs. Furthermore we suggested as a case study for refactoring the minimization of DFAs.

Our plan is to extend this work in such a way that whenever we define a refactoring as graph productions we should also be able to prove that all instances prior to refactoring are bisimilar to their refactored counterparts. The first step to be made to accomplish such an objective is the extension of the current theory of DPO with BC to handle rules with negative application conditions and layers, which are often used in refactorings.

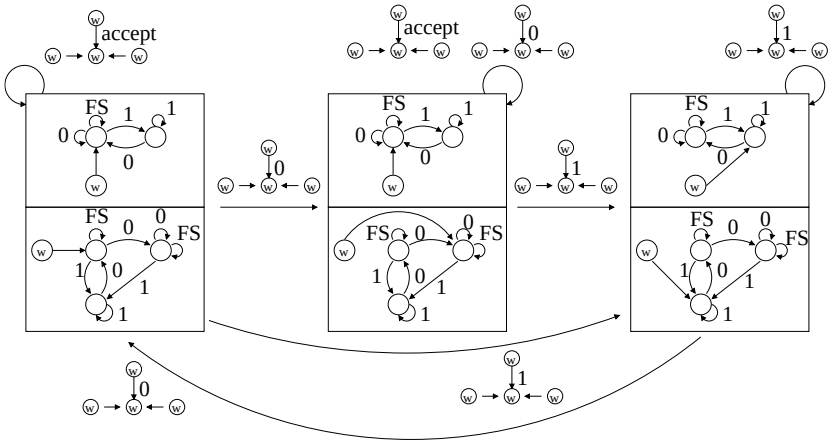


Figure 2: State space product for DFA1 and DFA2

Acknowledgements: We are grateful to Gabriele Taentzer and Paolo Baldan for interesting discussions on using DPO with borrowed contexts in the context of refactoring.

Bibliography

[BGK06] F. Bonchi, F. Gadducci, B. König. Process Bisimulation *via* a Graphical Encoding. In Corradini et al. (eds.), *Proc. of ICGT'06*. LNCS 4178, pp. 168–183. Springer, 2006.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Loewe. Algebraic Approaches to Graph Transformation Part I: Basic Concepts and Double Pushout Approach. In Rozenberg (ed.), *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. Pp. 163–246. World Scientific, 1997.

[EE06] H. Ehrig, K. Ehrig. Overview of Formal Concepts for Model Transformations Based on Typed Attributed Graph Transformation. In *Proc. of GraMoT '05*. ENTCS 152, pp. 3–22. Elsevier Science, 2006.

[EK04] H. Ehrig, B. König. Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting. In Walukiewicz (ed.), *Proc. of FoSSaCS '04*. LNCS 2987, pp. 151–166. Springer, 2004.

[EW06] K. Ehrig, J. Winkelmann. Model Transformation From VisualOCL to OCL Using Graph Transformation. In *Proc. of GraMoT '05*. ENTCS 152, pp. 23–37. 2006.

[FM91] J.-C. Fernandez, L. Mounier. On the Fly Verification of Behavioural Equivalences and Preorders. In *Proc. of CAV'91*. LNCS 757, pp. 181–191. Springer-Verlag, 1991.

[gra] Graphviz - Graph Visualization Software. Internet. http://www.graphviz.org.

[Hir01] D. Hirschkoff. Bisimulation Verification Using the Up to Techniques. *International Journal on Software Tools for Technology Transfer* 3(3):271–285, Aug. 2001.

[LM00] J. J. Leifer, R. Milner. Deriving Bisimulation Congruences for Reactive Systems. In *Proc. of CONCUR '00*. Volume 1877, pp. 243–258. Springer-Verlag, London, UK, 2000.

[Mil93] R. Milner. The polyadic π-Calculus: a tutorial. In Hamer et al. (eds.), *Logic and Algebra of Specification*. Springer-Verlag, Heidelberg, 1993.

[MT04] T. Mens, T. Tourwe. A Survey of Software Refactoring. *IEEE Transactions on Software Engineering* 30(2):126–139, 2004.

[MV05] T. Mens, P. Van Gorp. A Taxonomy of Model Transformation. In *Proc. of GraMoT '05*. Volume 152, pp. 125–142. 2005.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In Bruni and Varró (eds.), *Proc. of GT-VMT '06*. ENTCS, pp. 185–194. Vienna, 2006.

[OCa] Objective Caml. http://caml.inria.fr/ocaml/.

[PT87] R. Paige, R. E. Tarjan. Three Partition Refinement Algorithms. *SIAM Journal on Computing* 16(6):973–989, 1987.

[San95] D. Sangiorgi. On the Proof Method for Bisimulation. In Wiedermann and Hájek (eds.), *Proc. of MFCS '95*. LNCS 969, pp. 479–488. Springer, 1995.

[SS05] V. Sassone, P. Sobociński. Reactive systems over cospans. In *Proc. of LICS '05*. Pp. 311–320. IEEE, 2005.

[VVE+06] D. Varró, S. Varró-Gyapay, H. Ehrig, U. Prange, G. Taentzer. Termination Analysis of Model Transformations by Petri Nets. In Corradini et al. (eds.), *Proc. of ICGT '06*. LNCS 4178, pp. 260–274. Springer, Natal, Brazil, 2006.

Transforming Collaborative Service Specifications into Efficiently Executable State Machines

Frank Alexander Kraemer and Peter Herrmann

Norwegian University of Science and Technology (NTNU),
Department of Telematics, N-7491 Trondheim, Norway

Abstract: We describe an algorithm to transform UML 2.0 activities into state machines. The implementation of this algorithm is an integral part of our tool-supported engineering approach for the design of interactive services, in which we compose services from reusable building blocks. In contrast to traditional approaches, these building blocks are not only components, but also collaborations involving several participants. For the description of their behavior, we use UML 2.0 activities, which are convenient for composition. To generate code running on existing service execution platforms, however, we need a behavioral description for each individual component, for which we use a special form of UML 2.0 state machines. The algorithm presented here transforms the activities directly into state machines, so that the step from collaborative service specifications to efficiently executable code is completely automated. Each activity partition is transformed into a separate state machine that communicates with other state machines by means of signals, so that the system can easily be distributed. The algorithm creates a state machine by reachability analysis on the states modeled by a single activity partition. It is implemented in Java and works directly on an Eclipse UML2 repository.

Keywords: Model Transformation, UML 2.0, Activities, State Machines

1 Introduction

In a highly competitive market for modern networked services, it is important to deliver new services with short development times, in order to react on new customer demands quickly and to keep development costs low. These efforts are hampered by the typically high complexity of such services, which arises mainly from the fact that a service needs the coordinated effort of several participating components (cf. [1]). Hence, if we want to understand what a service does, we have to look at the behavior of all its participating components. Moreover, when services need to be adjusted or composed from other services, we must consider the descriptions of all participating components again and make sure that they interact correctly. Literature (e.g., [2]) as well as experience from our own work [3, 4, 5] stated that there are two dominant perspectives on a system delivering services:

- In the component-oriented perspective, systems are decomposed into physically distributed components, which are modelled separately. Services are specified indirectly by the composed behavior of the components. This perspective is well supported by traditional standards like SDL and its descriptions are easily transformable into executable code.

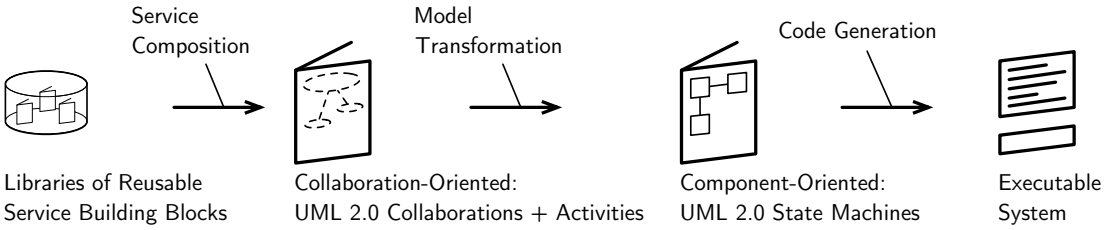


Figure 1: Engineering approach for interactive services

- In the collaboration-oriented perspective, services are modeled by a number of collaborations as the main structuring elements. A collaboration specifies the interactions between the components involved in it, as well as the corresponding local behavior of the components to accomplish the service. Collaborations describe services in a self-contained form and may be composed from other ones. Within an application domain, collaborations contributing to a service are often similar which makes them ideal elements of reuse.

These two perspectives are the shaping forces behind our approach for the rapid engineering of interactive services, outlined in Fig. 1. Services are composed from collaborations that identify the interactions as well as the local behavior of a set of components that are necessary to fulfill a certain task. To express the structural aspects of collaborations as well as their composition (e.g., the participants and which roles they play in a service), we use the conforming concept of UML 2.0 collaborations. For the behavioral aspect (e.g., what a collaboration does as well as how collaborations are coupled together), we use UML 2.0 activities.

As an example, we consider an access control system (ACS) [6, 7]. It controls the opening mechanism of a door and lets pass only authorized people that can prove their identity by presenting a security card and a secret number at an input panel. The opening mechanism and input panel are connected to a local station installed close to the door. Once a user draws the card and enters the pin, the resulting data (called pid) is transferred to a central station that authenticates the user and checks authorization right by querying two servers. If both, the authentication and authorization are successful, *ok* is sent back to the local station that opens the door.

In [4] we introduced how the ACS can be easily composed from reusable collaboration elements expressed by a combination of UML 2.0 collaborations for the structure (Fig. 2) and activities for the behavior (Fig. 3). These diagrams describe the system from a collaboration oriented perspective. To execute the system, however, we need a description of the behavior of the individual components, i.e., a description from a component-oriented perspective, as outlined above. In our approach, we use for this purpose so-called *executable* UML 2.0 state machines and composite structures, that are a suitable input for our code generators. To automate the step from the collaboration-oriented specifications in form of activities to the component-oriented design in form of state machines, we use a model transformation performed by the algorithm described in this article. Evidently, the introduction of such an automated transformation step accelerates the development of services drastically. In addition to the omission of manual labor for constructing the state machines, no new errors are introduced. Whenever a service specification needs to be updated, the state machines are simply generated again to ensure consistency. The algorithm creates the state machines without any intermediate representation and is therefore quite efficient concerning memory usage. Before we describe the principles of the transformation in Sect. 4 and the detailed algorithm in Sect. 5, we outline in the next two sections the two

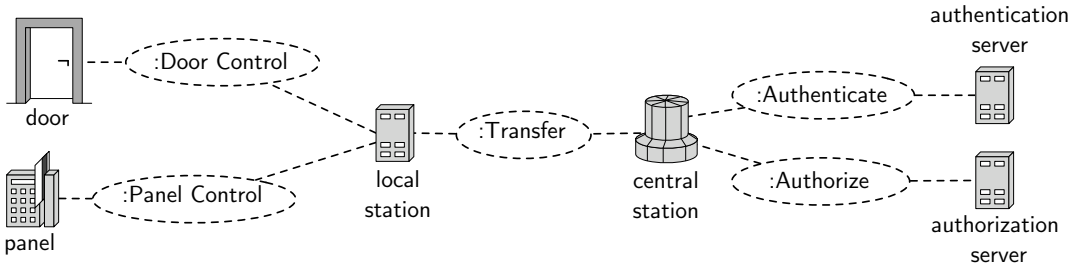


Figure 2: Collaboration to compose the sub-services of the access control system

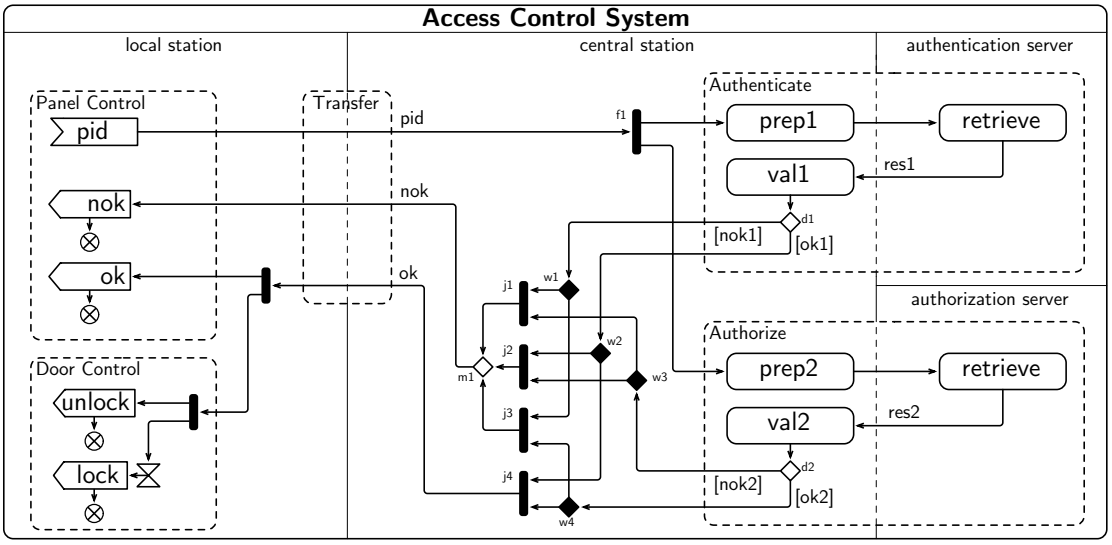


Figure 3: Activity diagram modeling the detailed behavior of the system

development perspectives outlined above. Sect. 6 sketches then a proof of the correctness of the transformation in temporal logic. We close with a discussion of related approaches and some concluding remarks.

2 Collaborations and Activities for Service Composition

While the collaboration in Fig. 2 shows how the system is composed structurally from elementary collaborations that were taken from a library, the activity diagram in Fig. 3 states how their behavior is coordinated. For each collaboration use of Fig. 2 (e.g., *Authenticate*), we find a structured node (in dashed lines) in the activity that specifies the behavior contributed by the collaboration use. Each collaboration role of Fig. 2 (e.g., *central station*) is a location of computation and represented by an activity partition in Fig. 3. The door and the panel are part of the environment, and, hence, do not have their own activity partition. Instead, they communicate with the local station by explicit signal send and receive actions. The local station receives a pid from the panel control and forwards it via the transfer collaboration to the central station. Depending on the result received from the central station (*ok* or *nok*), the local station will either cause the panel to display *nok* and leave the door locked, or it will cause the panel display to show *ok* and unlock the shutter of the door. In this case, a timer will be started which locks the

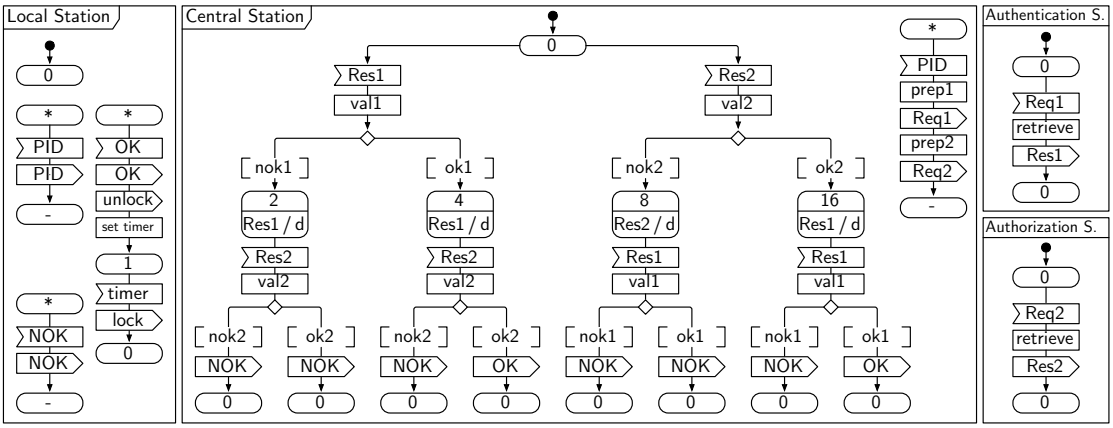


Figure 4: Executable state machines for the system components

door shutter again after a while. As activities have a Petri net like semantics [8], we can use tokens and places to understand the behavior of the diagram in Fig. 3. Once a token representing a pid arrives at the central station, it is prepared (described by the operations *prep1* and *prep2*) and sent to the authentication respective the authorization server. For that, the token is duplicated at the fork node *f1*, so that the subsequent behaviors may happen in parallel. Both servers evaluate the pid and send their results back to the central station.

The results may arrive in any order. For example, if the result of the authorization server arrives first, it is evaluated by the central station (operation *val2*) which branches in decision *d2* depending on the validity of the authorization. If the result was valid, a token is placed in *w4*. This node is an extension of a decision node (cf. [4]) as tokens can rest in it. It is represented by a filled diamond. The central station waits now for the arrival of the authentication result, which is evaluated in *val1*, and a token is placed either on *w1* or *w2*. When the other result arrives, two waiting decisions hold one token, so that exactly one of the join nodes *j1..j4* can fire. Obviously, *j4* fires in the case that both results were ok and causes the central station to send an ok to the local station. In the other three cases (when at least one result is *nok*) one of the other join nodes *j1..j3* fires. These cases are combined by merge node *m1* and a *nok* is sent to the local station. We assume that the panel control only sends a new *pid* after it received an *ok* or a *nok*.

3 State Machines for Service Execution

Fig. 4 presents the executable UML state machines generated by our algorithm from the activity in Fig. 3. The state machines interact with each other by transmitting signals which are buffered in event queues. Similar to SDL, UML allows for the use of send signal actions and signal triggers to describe the transmission and reception of signals. Each state machine has an initial state and a number of transitions that are triggered by either signal receptions or by timeouts. Transitions may include choices guarded by constraints (like *ok1*). As an effect, a transition may execute actions such as the sending of signals, the call of an operation (like *val1*) or the control of a timer. States can declare an event to be deferred by listing it in their body followed by the keyword “/defer” (abridged here to “/d”). This event is left in the queue until a state is entered that does not defer it anymore but declares a transition for its consumption. For compactness, we

presented a transition that can be executed from any control state by referring to a state called "∗". After it executes, the state machine returns into its originating state, denoted by "-".

As these executable state machines are the input for our code generators [9], they must fulfill some constraints to achieve efficient code. In particular, they are event-driven, which means that each transition is only executed as the reaction to either the creation of the state machine itself, the reception of a signal, or the expiration of an internal timer. In consequence, transitions are enabled based purely on their source state and trigger, so that guards may only be declared on branches following choices. Moreover, for each pair of control state and trigger, merely one transition may be declared to prevent fairness conflicts between competing transitions.

These executable state machines have a long tradition in the telecommunication area (see for example [10]) and facilitate the efficient implementation on a range of different platforms and architectures, including J2EE. We defined in [5] their execution semantics in terms of temporal logic and described, how they can be efficiently implemented using a scheduler as virtual machine layer. Of course, the constraints on the executable state machines needed to generate efficient code highly influence the layout of our algorithm which we discuss in the following.

4 Transformation from Activities to State Machines

In our approach, an activity partition corresponds to one physical point of execution. We therefore generate one state machine for each activity partition. This also makes it possible to consider the activity partitions separately and not the entire activity, as discussed later. To separate the partitions from each other, we have to cut those edges which cross partition borders. These edges model the control flows between different system components. As communication between the state machines is done entirely by means of signals, a flow crossing the boundaries of activity partitions must be implemented as a signal transmission. In activities, flows between actions occur instantaneously, i.e., a token leaves an action and enters a subsequent one without resting in the flow. The transmission between state machines, however, is buffered. Introducing signal transmissions in flows between partitions therefore implies virtual places that may hold tokens. We add these places where flows enter a partition, as illustrated in Fig. 5 by the circles with the queue symbols inside. These so-called *queue places*, which are also attached to receive actions, simulate the input queue of the state machines implementing an activity partition. In the model, these input queues are of unlimited capacity[1]. Thus, the virtual places are unbounded (i.e., can hold any number of tokens).

As described above, the state machines execute a transition as a reaction to the arrival of a signal. This event corresponds to the emission of a token from the virtual queue places. Hence, when we construct a transition, we simulate the emission of one token from a queue place. The token passes along the flows and nodes of the activity diagram until it reaches a control node where it has to wait for further events to happen. These are three kinds of nodes: (1) *Join nodes* synchronize different flows, that may arrive in any order. An incoming token may have to wait for the other incoming flows to arrive. (2) *Waiting decisions* synchronize competing join nodes (see Sect. 2). A token has to rest inside a waiting decision if none of the succeeding joins can fire. (3) *Timer nodes* may contain tokens describing that the timers are active. At these nodes

[1] Of course, in an implementation, buffer capacity is limited, which can be addressed the means described in [5].

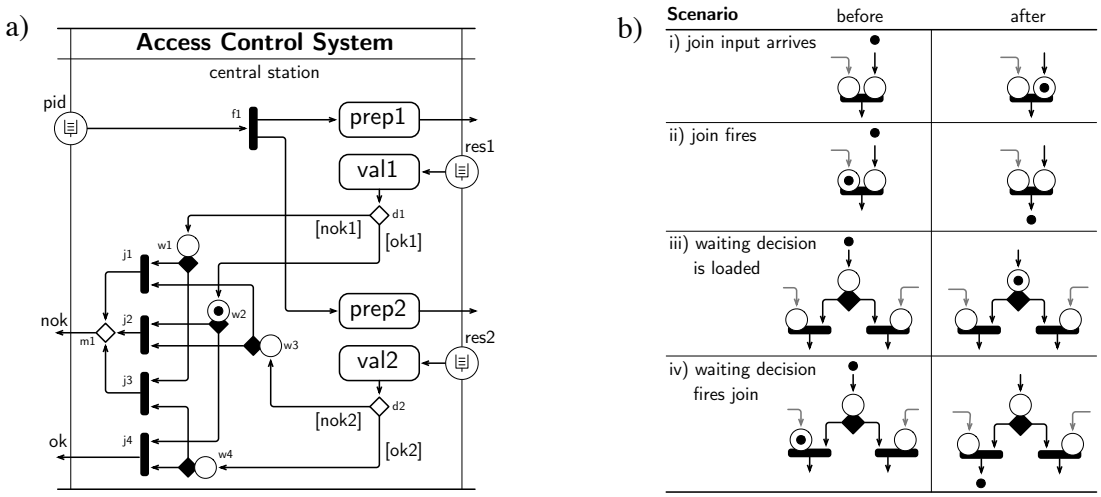


Figure 5: (a) Places for the nodes (b) Rules for token transitions

and also at initial nodes, we append *inner places*. For instance, Fig. 5 (a) illustrates the inner places of the central station in which tokens rest to wait for further input events. In contrast to the queue places, these inner places will constitute the control states of the state machine. For instance, the token in the waiting decision *w2* of Fig. 5 (a) means that a valid authentication result arrived and that the central node waits for the result of the authorization. (The join nodes do not have own places, as all their incoming flow originate from waiting decisions, which hold the token instead.) For the number of control states to be finite, the number of tokens in an inner place must be bounded. Moreover, to keep the state space small, we allow only one token in each inner place. This is not a limitation since tokens that would fill an inner place are stored in the unlimited queue places as discussed later. The set of control states for one state machine is then the powerset of the inner places.

We can construct a state machine transition by following the passing of the tokens between two stable token markings. The marking of the inner places before passing the tokens define the source state of the transition and the next stable marking refer to the target state. The token taken from a queue place models the input signal consumed by the transition. The activity nodes passed by the token are transformed in the following way: Call operation actions and send signal actions are simply copied into the effect of a transition. Decision nodes with guards are added to the transition and lead to different branches. A flow leaving the current activity partition is translated to a send signal action. Fork nodes duplicate tokens, to that the subsequent flows are executed in parallel. In the transition, this is mapped by executing their actions interleaved. For instance, the transition triggered by *PID* in *Central Station* in Fig. 4 simply executes first the action *prep1* and then *prep2*. Initial nodes emit tokens once the activity is started and are treated by the initial transition of a state machine.

A problem is the handling of joins. The passing of a token after all incoming flows arrived would result in a transition without a trigger event, violating constraints of our event-driven state machines. Therefore we use the token passing rules illustrated in Fig. 5 (b). When a token arrives at a join and there are other incoming edges that do not yet offer a token (since their flow did not arrive yet), the token is stored and a new stable control state is reached *(i)*, awaiting the next event. If, however, the arriving token completes the join *(ii)*, the transition continues with

its outgoing edge, and all tokens of the incoming edges are removed. Waiting decisions work similarly, but consider a set of subsequent join nodes. If none of these joins is ready, the decision is filled with a token *(iii)*, and a new stable state is reached. If one of the joins is ready, the transition continues at its outgoing edge, consuming the token from the decision node *(iv)*.

In addition to the events of signal reception resulting from the split control flows, explicit signal receptions contribute to the set of unbounded queue places. Furthermore, timers are sources of events. When a timeout occurs, a token is emitted on the timer's outgoing edge. The transition is then constructed in the same way as for signal receptions. Some events may lead to states describing that an inner place contains more than one token. For example, if the central station is connected to several local stations, a pid could arrive while another pid is under evaluation. In this case it might happen, that, after the central station received a valid *res1* and waits for *res2*, another valid *res1* is coming, which requires node *w2* to hold two tokens. To prevent this, we do not create a transition for flows leading to a marking with several tokens in an inner place, but defer the incoming event, which may proceed after the inner place is emptied.

5 The Transformation Algorithm

To realize the transformation from activities to state machines introduced above, we can proceed in quite different ways. For instance, one could perform a complete reachability analysis over all allowed token allocations in the previously introduced inner and queue places of an activity and create a transition for every step. The disadvantage is that, especially in highly concurrent systems, the number of reachable states is very large, rendering the approach not scalable. Another possibility would be to perform a purely syntactical analysis of an activity. Here, for each edge between places, a set of transitions is generated. Thereby, a separate transition is created for all states in which tokens are contained in the corresponding places. This algorithm is quite efficient since every inner place of the activity is checked only once, but will lead to a large number of transitions leaving unreachable states. To prevent these disadvantages, we follow an intermediate approach. Reflecting that for every activity partition a separate state machine is created, we perform a reachability analysis over the states of an activity partition only, which are constituted by its inner places. Thus, the number of reachable states is kept small. Starting from the initial state, for each reached state and every possible input signal a separate transition is created. As we handle each incoming signal in all control states, some of these transitions may be never fired (if their input signal cannot occur in the state). However, an unnecessary transition would simply result in a code fragment that is never executed. While this is not a real problem, nevertheless, we plan to eliminate these transitions using interface descriptions of the other partitions. These interface descriptions may be offered as part of the collaboration building blocks of a library.

In the following, we explain our algorithm in detail. Fig. 6 depicts the main loop (lines *7* to *27*). As in most reachability analysis algorithms, this loop guarantees that all reachable markings of an activity partition are analyzed. The markings yet to be checked are listed in the variable *reachable* while *visited* contains all markings which were already analyzed. In the initial part of the algorithm *(1..6)*, a new and empty state machine is created. Thereafter, the first marking to be checked, the initial transition of the state machine and the set of events to be received by the state machine are computed. As our algorithm creates a state machine transition for each pair

transform(ActivityPartition a) : StateMachine

```
1  var stm: StateMachine = new StateMachine()
2  var firstState: State = computeFirstState(a)
3  createInitialTransition(firstState, stm)
4  var events: Set of Event = computeEvents(a)
5  var visited: Set of State = ∅
6  var reachable: Set of State = {firstState}
7  while reachable ≠ ∅ do
8    var current: State = reachable.removeFirst();
9    visited = visited ∪ {current}
10   for all e ∈ events do
11     if ¬(e is timeout ∧ timerActive(current)) then
12       if harmsBoundedness(current, e) then
13         current.deferEvent(e)
14       else
15         var t: Transition = new Transition(stm);
16         t.setSource(current)
17         t.setTrigger(e)
18         var marking: long = getMarking(current)
19         if e is timeout then marking = unsetTimer(e,marking)
20         var Edge edge = retrieveEdge(e)
21         var targets: Set of State
22           = buildTransition(edge,marking,t,a,stm)
23         reachable = reachable ∪ (targets / visited)
24       end if
25     end if
26   end for
27 end while
28 return stm.                                                  □
```

Figure 6: Main control

of reachable marking and event (see Sect. 4), the loop contains a nested for-loop *(10..26)* cycling through all events. The for-loop contains two nested if-statements. The first one *(11..25)* is used to ignore events triggered by a timer which is not active in the current state. The second if-statement enables us to handle violations of the desired 1-boundedness property correctly. If the traversal of an edge in the checked activity would lead to two or more tokens in any inner place, the algorithm does not create a transition but defers the event in the current state *(13)*. Otherwise, a new transition is built in the else-statement *(15..23)*.

The transitions of a state machine are created by means of the recursive method *buildTransition* *(20)*, listed in Fig. 7. It considers the traversal of a token from one stable marking to another. For each edge part of the flow triggered by the event it is called recursively and builds the corresponding transition along the way. The method returns the set of stable states reached by the transition. It is a set, as a flow may lead to several distinct reachable states after a decision node. The returned states are used by the main loop to determine the reachable markings of the partition yet to be checked. The method contains an order of nested if-statements describing the behavior for each possible node in the analyzed activity edge. It returns if the edge leaves the partition *(2)*, reaches a join which cannot be fired in the current activity marking *(12)*, starts a timer *(17)*, arrives at a waiting decision in which none of the corresponding joins can be fired in the current marking *(50)*, or reaches a flow final resp. activity final node *(58, 62)*. In all these cases a new stable state is reached and the created transition can be completed. When another edge is reached, the transition is not yet complete and its building process has to be continued by a recursive call of *buildTransition*. These cases are a join which can be executed after being reached by a token on the analyzed edge *(10)*, a send action *(27)*, an operation call *(30)*, a merge *(32)*, a decision *(40)*, a waiting decision from which a corresponding join can be fired after being reached by a token *(48)*, and a fork *(57)*. While most steps in creating a transition follow directly the ideas presented in Sect. 4, we will look now on the decisions and forks which are a little subtle. A decision leads to the addition of a choice pseudo state to the transition behind which more than one continuing transition fragments are added. This is done by the for-loop *(36..42)* which calls *buildTransition* for each of the choice's branches. The parallel emission of tokens at forks is addressed at *(55)*. As one state machine executes only one action at a time, we map parallel executing flows inside one activity partition to an interleaved execution, which is a correct refinement. This execution is computed by the method *collectEffects* which is not listed here for the sake of brevity.

```
buildTransition(Edge edge, long c, Transition t, ActivityPartition a, StateMachine stm) : Set of State

1  var node: Node = edge.getTarget()
2  if leavesPartition(edge,a) then
3    addSendSignalAction(t,edge)
4    var target: State = getState(c)
5    t.setTarget(target)
6    return {target}
7  else if node is join then
8    if canFire(join,c) then
9      var n: long = markingAfterJoinFired(c)
10     return buildTransition(outgoing(edge),n,t,a,stm)
11   else
12     var n: long = markingAfterJoinInputArrived(c)
13     var target: State = getState(n)
14     t.setTarget(target)
15     return {target}
16   end if
17 else if node is timer then
18   addSetTimerAction(t,node)
19   var n : long = markingAfterTimerSet(c)
20   var target: State = getState(n)
21   t.setTarget(target)
22   return {target}
23 else if node is send action then
24   addSendSignalAction(t,node)
25   var target: State = getState(c)
26   t.setTarget(target)
27   return buildTransition(outgoing(node),c,t,a,stm)
28 else if node is call operation action then
29   t.addEffect(node)
30   return buildTransition(outgoing(node),c,t,a,stm)
31 else if node is merge then
32   return buildTransition(outgoing(node),c,t,a,stm)
33 else if node is decision then
34   var p: Pseudostate = new Pseudostate(stm,CHOICE)
35   var reachable: Set of State
36   for all o ∈ node.outgoings() do
37     var t = new Transition(stm)
38     t.setSource(p)
39     t.setGuard(o.getGuard())
40     var r: Set of State = buildTransition(o,c,t,a,stm)
41     reachable = reachable ∪ r
42   end for
43   return reachable
44 else if node is waiting decision then
45   for all o ∈ node.outgoings() do
46     var join: Node = o.target;
47     if canFire(join, marking) then
48       return buildTransition(o,c,t,a,stm)
49     end if
50   end for    //no join could fire
51   var n : long = markingAfterDecisionSet(c)
52   var target: State = getState(n)
53   t.setTarget(target)
54   return {target}
55 else if node is fork then
56   collectEffects(outgoings(node),t)
57   return computeForkedState(outgoings(node))
58 else if node is flow final then
59   var target: State = getState(c)
60   t.setTarget(target)
61   return {target}
62 else if node is activity final then
63   t.setTarget(new FinalState(stm))
64   return {}
65 end if                                                    □
```

Figure 7: Method to build a transition

6 Correctness of the Transformation

To verify that the algorithm carries out transformations in a correctness-preserving manner, we use the linear-time temporal logic cTLA [12] as a formalism which is based on Leslie Lamport's TLA [13]. cTLA enables the description of resources and constraints in a process-like notion and provides a coupling structure based on conjoining actions (i.e., predicates on pairs of states describing sets of transitions). Refinement verifications are carried out as temporal logic implication proofs (cf. [13]). As the semantics of activities is based on Petri-nets [8], UML 2.0 activities can easily be expressed by cTLA processes as pointed out in [14]. An activity, basically, is a cTLA system description consisting of processes each describing a single activity partition. The variables of a process model its inner places while each queue place of a partition is described by a separate input queue.

For the state machines forming the input of our code generators, we defined a special dialect cTLA/e [5] which describes the coupling between components by assigning a single input queue to each component. A state machine transition is specified by a cTLA action which

reflects that the transition depends only on the current state and the first signal in the input queue. Moreover, each component contains an extra queue to handle deferred events. The refinement of specifications modeling activities to cTLA/e-based descriptions is carried out by a sequence of correctness-preserving refinement steps accompanied by cTLA/TLA implication proofs (cf. [13]). For the sake of brevity, we do not give a thorough introduction to cTLA here and sketch the proof steps only briefly.

To verify formally that a state machine S derived from an activity partition A keeps all the functional properties state by A, we must perform by temporal logic deductions that the implication $S \Rightarrow A$ holds. According to Abadi and Lamport [15], this can be achieved by finding a so-called refinement mapping from the states of S to those of A. A refinement mapping takes into account that cTLA models applications as state transition systems. A system formula consists of an initial condition describing the set of initial states, cTLA actions which are predicates on a pair of a current state and a next state and model a set of state transitions each, and liveness properties expressed by fairness assumptions on actions which enforce that actions are eventually executed when they are consistently enabled. A refinement mapping fulfills the following properties:

- An initial state of S is mapped to an initial state of A.
- Each cTLA action of S is either mapped to an action of A or to a so-called stuttering step in which the mapped current and next states of A are identical.
- Each fairness assumption of A is provided by the fairness assumptions of S (i.e., if an action ψ of A is consistently enabled, the fairness assumptions of S enforce a state sequence in which eventually an action is carried out which is mapped to ψ).

In Sec. 4 we stated that the state space of an activity partition A is partly defined by its inner places which are placed before joins, at decision nodes, at initial nodes, and at timers. Moreover, it contains queue places which are situated at points where an incoming flow passes the partition border and on receive actions. The state space of a state machine is defined in [5] and consists of the literal states of the state machine, an input queue, a defer queue, output queues for all connected state machines, and flags for each timer. Furthermore, activities may contain auxiliary variables which our algorithm directly maps to auxiliary variables of the corresponding state machines. Every auxiliary variable must be read and modified in one activity partition only. To outline the correctness of the algorithm, we introduce a mapping of the state space from S to that of A and sketch thereafter that it fulfills the refinement mapping properties:

- To find a mapping from S to the queue places of A, we have to consider the state machines S_n linked with S since the queue places describe the interaction between different system elements. At an activity partition, we have a separate queue place for every signal type st while in the corresponding state machine, we have central queues for all signals. Moreover, in the activity we do not distinguish if a signal is still at the side of the outgoing partition, already in the incoming partition, or deferred. Reflecting these properties, we map all signals s of type st, which are either in the output queue of a neighboring state machine S_n, in the input queue of S, or in its defer queue, to the queue place qp_{st} for st in A:

$$\forall st : qp_{st} = \{s | s.type = st \wedge s \in inputQ_S \cup deferQ_S \cup \bigcup_{S_n \in Neighbors_S} S_n.outputQ_S\}$$

- A mapping of S to the inner places of A located at joins, decision nodes, and initial nodes has to consider that we use 1-boundedness in the inner places ip and that the algorithm creates the states of S as a string of flags fl_{ip} each being set to 0 if the corresponding inner place ip is empty and to 1 if ip contains a token to:

$$\forall ip : ip = \text{IF } fl_{ip} = 1 \text{ THEN } \{to\} \text{ ELSE } \{\}$$

- To find a mapping from S to the inner places of A describing a timer is a little more complex. Indeed, the algorithm adds also a flag fl_t for each timer t in A to the state representation in S. Nevertheless, to find a decent mapping, one has to consider the handling of timers in state machines. When a timer expires, it creates a signal which is attached to the local input queue. Thus, we must map both the states of S in which the flag fl_t of timer t is enabled and in which a signal s_t caused by t is in the input or defer queue to a setting in A where a token to is on the inner place ip_t of t. That is expressed by the mapping listed below:

$$\forall ip_t : ip_t = \text{IF } fl_t = 1 \vee s_t \in inputQ_S \cup deferQ_S \text{ THEN } \{to\} \text{ ELSE } \{\}$$

- The mapping from the auxiliary variables from S to those of A is the identity function.

In the first step of the proof that the function listed above fulfills the refinement mapping properties, we have to verify that the initial state of S is mapped to that of A. Initially, the queue places in A are empty while the input, output, and defer queues of S do not contain elements as well. Thus, the mapping of the queue places fulfills the property. The inner places of A are empty except those located at an initial node and the algorithm maps this token placement to the initial state of S in which just the flags representing the inner places of the initial nodes are set to 1. Since the auxiliary variables of S and A contain the same initial settings, therefore, the initial state of S is mapped to the initial state of A.

Next, we prove that every cTLA action in the model of the state machine S is mapped either to a cTLA action of the activity partition A or to a stuttering step. As introduced in [5], the model of S contains different kinds of actions. One type describes the transitions of S and for each transition tr_f, a cTLA action ϕ_{tr_f} is defined. The algorithm creates tr_f only if a flow f exists modifying the token setting of A. In the following, we state a number of properties preserved by the algorithm in the creation of the corresponding transition tr_f which are used for the refinement proof:

1. A transition tr_f is only created if in its source state all flags fl_{ip} representing the inner places ip of f, which must contain tokens to enable the execution of f, are set to 1.
2. The algorithm creates tr_f only for a flow f if the execution of f does not violate the 1-boundedness property of the inner places in A.
3. If the queue place in f, from which a token is removed, has the type st, tr_f is only triggered if st is at the front of the input queue.
4. By executing a transition tr_f which does not leave an initial state, the signal at the front of the input queue is consumed.
5. A transition tr_f consuming a signal from the input queue, which was created by a timer, is generated if the corresponding flow f starts at an inner place describing a timer node.

6. The target states of tr_f are generated by starting with the source state and resetting the flags representing inner places, from which tokens were removed, to 0 while those with a new token are set to 1.
7. If in the flow f a token crosses the border to a partition A_n or heads to a send action with destination A_n, tr_f puts a send signal into the output queue devoted to the state machine S_n realizing A_n.
8. A call operation action passed in f is reflected by adding its code to tr_f. Here, we demand that an auxiliary variable may be modified only once in f and, in consequence, in tr_f.

Assuming that ϕ_{tr_f} is the cTLA action modeling tr_f and ψ_f those of the flow f, these properties are sufficient to prove the implication $\phi_{tr_f} \Rightarrow \psi_f$. By the first three properties, we can assure that the enabling condition of ϕ_{tr_f} implies that of ψ_f since according to the mapping all necessary tokens are set (1), the 1-boundedness after carrying out f is preserved (2), and the queue place, from which f leaves, contains an element (3).

The other properties are used to verify that the effects of ϕ_{tr_f} are correctly mapped to those of ψ_f. The elimination of a signal of type *st* from the input queue is mapped to the removal of a token from the queue place *st* (4). In addition, if tr_f consumes a signal s_t created by a timer from the input queue, s_t is mapped to a flow f removing a token from the corresponding timer node (5). We can further verify that tr_f is a correct realization of the token flow between the inner places in f (6). The delivery of a signal s to an adjacent state machine S_n does not spoil the corresponding mapping of S_n to a neighboring activity partition A_n since s is added to an incoming queue place of A_n if S puts it to its output queue devoted to S_n (7). Finally, it is guaranteed that the auxiliary variables are correctly mapped (8). It is not difficult to verify that these properties imply that the mapping listed above maps ϕ_{tr_f} to ψ_f which is omitted, however, for brevity.

Other cTLA actions in S specify the execution of timers and the addition of timer signals to the input queue, model the deferral of a signal by transferring it from the input to the defer queue, and describe the transfer of signals from the neighbor's output queue to the own input queue. It can be easily shown that these actions lead to stuttering steps in A.

In the third step, we have to verify that the fairness assumptions of the actions ψ_f describing the flows in A are kept. The algorithm guarantees that for every token placement in the inner nodes of A enabling a flow f, a transition tr_f is generated implementing f. Thus, with respect to the first two properties listed above, an action tr_f is enabled whenever f can fire. The only impeding condition is the third property since tr_f may only be executed if the signal s consumed by it is at the first place of the input queue. According to the mapping, however, the cTLA action ψ_f specifying f can be enabled if s is either in the output queue of the neighboring state machine S_n or in any place on the input or defer queues of S. Thus, we must verify that s is eventually being moved to the front of the input queue where it will remain consistently until an action ϕ_{tr_f} is executed. If s is still in the output buffer of S_n, it will be moved to the end of the input buffer of S by the fair[2] action modeling the transmission from S_n to S. Since signals before s in the input resp. defer queue are either continuously being deferred or eventually being consumed, s will eventually be consistently at the front of the input queue. If f is not enabled, s may be deferred

[2] In [12] we established that liveness can only be guaranteed in a distributed system if transmitted messages are eventually being delivered. This is expressed by the fairness assumption on the action specifying the transmission.

itself but is brought back to the front of the input queue by other transitions. As there is only a finite number of transitions tr_f modeling f, in consequence, one of those will be consistently being enabled if f can be triggered infinitely often as well. Due to the fairness assumption of the corresponding cTLA action ϕ_{tr_f} it will be eventually fired which, because of the mapping, causes also the triggering of ψ_f.

Thus, we could verify that the mapping listed above is a refine mapping. According to [15], we thereby proved that the state machine S together with its neighboring state machines S_n produced by the algorithm is a correct implementation of the activity partition A. Since this proof can be carried out for all partitions of the activity, we established that the algorithm transforms activities to state machines in a correct way.

7 Related Work

To our best knowledge, the algorithm presented here is the first one that directly transforms UML 2.0 activity diagrams into the executable state machines described above. Our work is related to that of Eshuis on model checking of activity diagrams [11], in which activity diagrams are transformed into the input language of NuSMV, a symbolic model verifier [16]. We could not adapt this algorithm for our work, since, as discussed in Sect. 5, syntactical algorithms cause in our field of application a high number of considered unreachable states. To execute activity graphs, Eshuis and Wieringa [17] describe an algorithm for an event router to coordinate the behavior of components. Aiming at workflow systems, their execution differs from ours as it assumes a centralized architecture and the activity is considered as a whole, rather than splitting up the activity into its partitions and creating distributed state machines as we do.

There is a number of approaches that take scenario descriptions based on sequence diagrams (like MSCs or UML sequence diagrams) to synthesize state machines [18, 19, 20, 21]. While the resulting state machines are similarly executable as the ones we described, the input of these synthesizers in form of sequence diagrams differs from activity diagrams. Sequence diagrams often specify only a set of scenarios rather than a complete behavior, which may lead to behaviors that are not expressed explicitly. They focus on the interactions and identify signals. In contrast, activities focus on the operations and decisions that have to be performed by its participants, and our algorithm generates the necessary interactions in form of signal transmissions automatically.

Use case maps (UCM, [22]) offer a notation that is close to that of UML activities, as they also allow the specification of behavior in terms of causal paths that may involve several components. Yong He et al. conducted an experiment [23] in which a specification expressed by use case maps was transformed into message sequence charts. These, in turn, were transformed into executable SDL specifications using the tool MSC2SDL [18]. Similar to that, Castejón [24] outlines an algorithm that takes specifications in UCM and UML 2.0 collaborations to generate state machines from sequence diagram fragments contained in the collaborations.

8 Concluding Remarks

We described an algorithm that transforms UML 2.0 activities into a UML 2.0 state machines, from which we can easily generate efficiently executable code. The algorithm is implemented in

Java and integrated into our Eclipse-based tool suite, so that we now have a complete automated development process from collaborative specifications based on activities to implementations on various platforms. As input and output we use models stored in the Java UML 2.0 repository from the Eclipse UML2 project. The algorithm does not construct an intermediate graph, but only UML model elements that are part of the desired output state machines, so that it is efficient with respect to the memory needed. The time for the transformation of the presented example is negligible; the state machines appear practically instantly. Moreover, we expect the algorithm to scale well also for more complex systems, as the increased complexity of a system leads more to a higher number of partitions than to more complex ones causing only a linear increase.

This work describes a step of a more comprehensive engineering approach for the creation of interactive services by correctness-preserving design steps. Initially, a service specification is composed from various abstract collaborations that, to a large extent, can be obtained from domain-specific libraries. Such abstract collaborations are often quite simple and can also be understood by customers, who are not experts in software technology but want to focus on their actual business. In succeeding steps, such abstract specifications are incrementally refined until the specification has a degree of detail that enables direct translation to software. Due to the algorithm, we are now able to perform these refining design steps entirely in the collaboration-oriented perspective. As pointed out in [4], for this purpose we can use the activities with their convenient properties as reusable building blocks.

Bibliography

[1] Floch, J., Bræk, R.: Towards Dynamic Composition of Hybrid Communication Services. 6th Int. Conf. on Intelligence in Networks (SMARTNET), Deventer, Kluwer, (2000)

[2] Rößler, F., Geppert, B., Gotzhein, R.: Collaboration-Based Design of SDL Systems. 10th Int. SDL Forum on Meeting UML, Springer-Verlag (2001) 72–89

[3] Sanders, R.T., Castejón, H.N., Kraemer, F.A., Bræk, R.: Using UML 2.0 Collaborations for Compositional Service Specification. In: ACM / IEEE 8th Int. Conf. on Model Driven Engineering Languages and Systems. (2005)

[4] Kraemer, F.A., Herrmann, P.: Service Specification by Composition of Collaborations — An Example. 2nd Int. Workshop on Service Composition (Sercomp), Hong Kong (2006)

[5] Kraemer, F.A., Herrmann, P., Bræk, R.: Aligning UML 2.0 State Machines and Temporal Logic for the Efficient Execution of Services. Int. Conf. on Distributed Objects and Applications (DOA), 2006, Montpellier, LNCS 4276, Springer (2006) 1613–1632

[6] Bræk, R., Haugen, Ø.: Engineering Real Time Systems: An Object-Oriented Methodology Using SDL. The BCS Practitioner Series. Prentice Hall (1993)

[7] Broy, M., Stølen, K.: Specification and Development of Interactive Systems: Focus on Streams, Interfaces, and Refinement. Springer (2001)

[8] Object Management Group: Unified Modeling Language: Superstructure (2006)

[9] Kraemer, F.A.: Rapid Service Development for Service Frame. Master's thesis, University of Stuttgart (2003)

[10] Bræk, R.: Unified System Modelling and Implementation. Int. Switching Symposium, Paris, France (1979) 1180–1187

[11] Eshuis, R.: Symbolic Model Checking of UML Activity Diagrams. ACM Transactions on Software Engineering and Methodology **15**(1) (2006) 1–38

[12] Herrmann, P., Krumm, H.: A Framework for Modeling Transfer Protocols. Computer Networks **34**(2) (2000) 317–337

[13] Lamport, L.: Specifying Systems. Addison-Wesley (2002)

[14] Graw, G., Herrmann, P.: Transformation and Verification of Executable UML Models. Electronic Notes on Theoretical Computer Science, Elsevier Science **101** (2004) 3–24

[15] Abadi, M., Lamport L.: The Existence of Refinement Mappings. Theoretical Computer Science **82** (2) (1991) 253–284

[16] Cimatti, A., Clarke, E.M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A.: NuSMV 2: An Opensource Tool for Symbolic Model Checking. 14th Int. Conf. on Computer Aided Verification (CAV), LNCS 2404, Springer (2002)

[17] Eshuis, R., Wieringa, R.: An Execution Algorithm for UML Activity Graphs. 4th Int. Conf. on The Unified Modeling Language, Modeling Languages, Concepts, and Tools (UML), London, Springer (2001) 47–61

[18] Mansurov, N., Zhukov, D.: Automatic Synthesis of SDL Models in Use Case Methodology. In Dssouli, R., von Bochmann, G., Lahav, Y., eds.: SDL Forum, Elsevier (1999) 225–240

[19] Whittle, J., Schumann, J.: Generating Statechart Designs from Scenarios. 22nd Int. Conf. on Software Engineering (ICSE), New York, ACM Press (2000) 314–323

[20] Krüger, I., Grosu, R., Scholz, P., Broy, M.: From MSCs to Statecharts (1999)

[21] Uchitel, S., Kramer, J., Magee, J.: Synthesis of Behavioral Models from Scenarios. IEEE Trans. Softw. Eng. **29**(2) (2003) 99–115

[22] Buhr, R.J.A., Casselman, R.S.: Use Case Maps for Object-Oriented Systems. (1996)

[23] He, Y., Amyot, D., Williams, A.W.: Synthesizing SDL from Use Case Maps: An Experiment. 11th SDL Forum, Stuttgart. LNCS 2708, Springer (2003) 117–136

[24] Castejón, H.N.: Synthesizing State-machine Behaviour from UML Collaborations and Use Case Maps. 12th Int. SDL Forum, Grimstad. LNCS 3530, Springer (2005)

Ensuring Containment Constraints in Graph-based Model Transformation Approaches

Christian Köhler[1], **Holger Lewin**[2], **Gabriele Taentzer**[3]

[1] christian.koehler@cwi.nl
Department of Software Engineering,
CWI Amsterdam, The Netherlands

[2] kinscher@cs.tu-berlin.de
Department of Software Engineering and Theoretical Computer Science
Technical University of Berlin, Germany

[3] taentzer@mathematik.uni-marburg.de
Department of Mathematics and Computer Science
Philipps-University Marburg, Germany

Abstract:

Within model driven software development, model transformation has become a key activity. A number of transformation approaches for metamodel-defined modeling languages have been developed in the past years and are going to be established in research and industry. None of these have made it to a standard yet. There is a demand for correct model transformation in various senses. Formal methods are helpful for showing correctness issues of model transformations. As one approach, graph transformation has been applied to the field of model transformation and is a perspective for achieving provable correct model transformations. We show in this paper, that *containment associations* as proposed by the OMG are an integral part of MOF-based languages and imply a couple of constraints which must be ensured in model transformation approaches. Based on a double-pushout approach to graph transformation, conditions are stated that ensure these containment constraints. This is an important step for achieving formal transformation semantics for modeling languages based on MOF, or specifically EMF.

Keywords: Model Transformation, Graph Transformation, MOF, EMF

1 Introduction

Model driven engineering is an emerging approach to software engineering aiming at fast development of high-quality software. The central entities in the terminology of model-driven software development are of course the *models*. Relations between these models are basically of two kinds: *instantiations* of so-called metamodels by models and *model transformations* between models typed by the same metamodel. A metamodel defines the common structure of all possible instantiations and therefore can be seen as some kind of language definition.

In this paper we consider a special kind of models, which are called *structural data models*. These models do not include any definition of behavior, but structural aspects only. Therefore, they are mainly used to specify domain specific languages. The two most popular modeling languages for structural data models are the *Meta Object Facility* (MOF) [OMG06] and the *Eclipse Modeling Framework* (EMF) [EMF]. While MOF is a slightly richer language, EMF comes with powerful code generation facilities. The key concepts in both languages are classes with inheritance and attribution and associations with multiplicity and containment properties. EMF can generate Java code which supports the creation, modification, storage, and loading of model instances. Moreover, it provides generators to support the editing of instance models.

It is possible to interpret these structures as graphs and define model transformations formally through graph transformations. Attribution of nodes and edges [EEPT06], node inheritance [EEPT05] and multiplicity constraints for nodes and edges [TR05] have already been studied in this area. An important property that has not been considered in this context are *containment* or *composite* associations.

Containment associations define an ownership relation between objects. Thereby, they induce a tree structure in model instantiations. In UML 2, ownership does not only occur in the form of composite associations between classes, but also at various other points, like components and their subcomponents or states and substates in state charts. In MOF and EMF, the tree structure induced by containment associations is further used to implement a mapping to XML, known as XMI (XML Metadata Interchange).

Containment always implies a number of constraints for model instantiations that must be ensured at run-time. The MOF specification [OMG06] states as semantical constraints for containment edges the following:

- *"An object may have at most one container."*
- *"Cyclic containment is invalid."*

As mentioned earlier, EMF provides full implementations of their models. These implementations always ensure these constraints. In fact, the MOF specification also says how such a constraint should be ensured, e.g.:

- *"If an object has an existing container and a new container is to be set, the object is removed from the old container before the new container is set."*

However, there is no suggestion how to avoid a cyclic containment of objects. Being a part of the MOF specification, these two properties are essential for valid models and therefore must be always ensured. This is especially the case when it comes to defining transformations for these models. A model transformation approach for MOF and EMF has to deal with these issues, i.e. must ensure that the result of a model transformation conforms to these constraints. We consider in this paper a graph transformation approach with formal semantics. The way of ensuring the containment constraints as stated in the MOF specification is not applicable in this context, since it would break the formal transformation semantics and therefore make existing results on termination and confluence of graph transformations invalid.

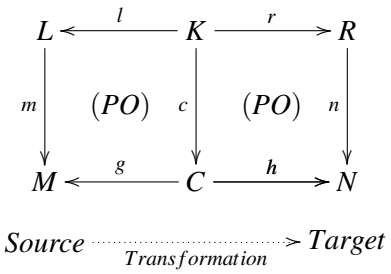
Therefore, we consider containment associations explicitly in the following and apply their properties in the context of a graph-based model transformation approach.

This paper is structured as follows: Section 2 gives a short introduction to model transformation by algebraic graph transformation. In Section 3, we consider graphs with containment relations and define the necessary containment conditions which have to be fulfilled by graph transformations. Section 4 gives a short description of related work in this area. At last, Section 5 contains a conclusion and refers to possible future work in the field of model checking and model transformation.

2 Model Transformation by Graph Transformation

Model transformations define relations between model instances. The most common scenario is a mapping from a source model to a target model. In this situation, an instance of the target model can be either completely constructed from scratch or updated incrementally. Transformations are usually defined through transformation rules. Basing model transformation on graph transformation, rules consist of one positive pattern, an arbitrary number of negative patterns and a description of the in-place modifications that should be performed. To apply a rule, it is necessary to either specify or automatically find a pattern match in the model instance that should be transformed. Thereafter, the existence of negative patterns has to be checked. If they are fulfilled, the in-place modifications can be applied.

In the graph transformation approach, patterns are given through graphs, while transformations of graphs are usually defined through pushout constructions. Here, we use the *double-pushout* approach (DPO) where a transformation rule is given by a span of injective graph homomorphisms $(L \xleftarrow{l} K \xrightarrow{r} R)$. Such a rule can be applied w.r.t. a given match m into a source model instance. See the following figure:



The input should be a graph corresponding to a valid model instance according to the MOF specification / EMF implementation. Therefore it must fulfill the containment constraints stated in the introduction. The aim is now to ensure that the result is also not violating the containment constraints. We achieve this goal by restricting the possible transformations. A rule application w.r.t. a given match is allowed, if the result does not violate the containment constraints. This is crucial for working with MOF and EMF models, especially for ensuring semantical properties like termination and confluence of transformations. For this purpose, we now introduce graphs with distinguished containment edges and use the double-pushout approach for defining transformations between them. Of course, other model constraints like multiplicities, have to be ensured, too, but are outside of the scope of this paper. For more details see [TR05].

3 Graphs with Containment Edges

Classes and associations can be interpreted as nodes and edges in a model graph. Accordingly, objects and links can be seen as nodes and edges in an instance graph. The 'instance of'-relation between these two graphs is achieved through a typing graph homomorphism, assigning to each object (each link) of an instance graph, a class (an association) in the corresponding model graph.

Definition 1 (Graph with containment edges) A graph with containment edges is a tuple $G = (V, E, C, source, target)$ consisting of a set of nodes V, a set of edges E, a distinguished set of containment edges $C \subseteq E$ and two functions $source, target : E \to V$ assigning a source and a target node to each edge. The containment edges induce the following transitive binary relation:

- $contains$[1] $= \{(x,y) \in V \times V \mid \exists e \in C : (source(e) = x \wedge target(e) = y)\} \cup$
  $\{(x,y) \in V \times V \mid \exists z \in V : (x\ contains\ z \wedge z\ contains\ y)\}$

The containment edges must have the following properties:

- $e_1, e_2 \in C : target(e_1) = target(e_2) \Rightarrow e_1 = e_2$ (at most one container).
- $(x,x) \notin contains$ for all $x \in V$ (no cycles).

This definition ensures that there are no containment cycles and that an object has at most one incoming containment edge, i.e. not more than one container. Accordingly, a homomorphism for graphs with containment edges is a usual graph homomorphism that preserves containment edges and their order properties.

Definition 2 (Homomorphism for graphs with containment edges) Given two graphs with containment edges G_1, G_2, a pair of functions (h_V, h_E) with $h_V : V_1 \to V_2$ and $h_E : E_1 \to E_2$ forms a valid homomorphism $h : G_1 \to G_2$ for graphs with containment edges, if it has the following property:

- $e \in C_1 \Rightarrow h_E(e) \in C_2$ (containment edges are preserved).

These two definitions induce the category of graphs with containment edges, which will be denoted as **CGraphs** in the following.

As already mentioned above, typing is accomplished by a graph homomorphism from an instance graph to a model graph. It is important to note that the constraints for containment edges as stated in Definition 1 apply to instance graphs only, not to model graphs. As shown in Figure 1, model graphs can have containment cycles, being the *substates* relation in this example. However, an instance of this model must not violate these constraints. Correspondingly, the constraints in the MOF specification apply to objects and links only, not to classes and associations.

Since we use the DPO approach to transform graphs with containment edges, it is necessary to show that the category of graphs with containment edges has pushouts[2] [EEPT06]. Pushouts

[1] If there is no confusion, we use infix notation for *contains*, e.g. $(x\ contains\ y)$ instead of $(x,y) \in contains$.

[2] Analogously for the category of graphs with containment edges typed over a fixed metamodel graph.

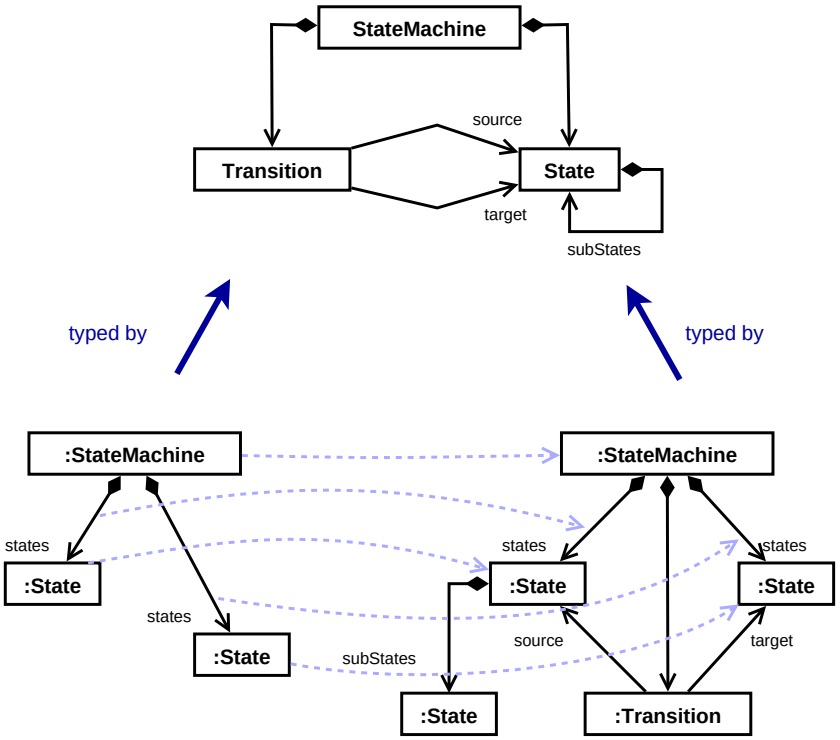


Figure 1: A model graph (top), two instance graphs (bottom) and a typed graph homomorphism.

of plain graphs are constructed as in **Set** (componentwise for nodes and edges). Given a span of graph homomorphisms $(G_1 \leftarrow G_0 \rightarrow G_2)$, the pushout result can be constructed by gluing the graphs G_1, G_2 over the interface graph G_0. Transformation rules in the DPO approach are spans of injective graph homomorphisms. Applying such a transformation rule consists of two steps. At first, a pushout complement graph and then a usual pushout graph has to be constructed. This construction is now considered for graphs with containment edges.

Containment edges can be seen as a conservative extension of plain graphs. A pushout of graphs with containment edges must also be a valid pushout of plain graphs, forgetting the containment properties of the edges. Given a span of valid graphs with containment edges, the question arises whether the pushout result as constructed for plain graphs also is a valid graph with containment edges. The pushout morphisms must be valid homomorphisms for graphs with containment edges respectively (preserve the containment edges).

As shown in Figure 2, the result graph does not necessarily have the designated order properties. Although graphs G_0, G_1 and G_2 are valid graphs with containment edges, the *'at most one container for each object'*-property in the pushout result G_3 is violated. The second example in Figure 3 shows, that there might also appear a containment cycle in the pushout result. This gives rise to state a condition under which a pushout in the category **Graphs** is also a valid graph with containment edges, i.e. has no containment cycles and each node has at most one container. Therefore, we introduce first the notions of so-called *newly contained points* and *cyclic contained points*.

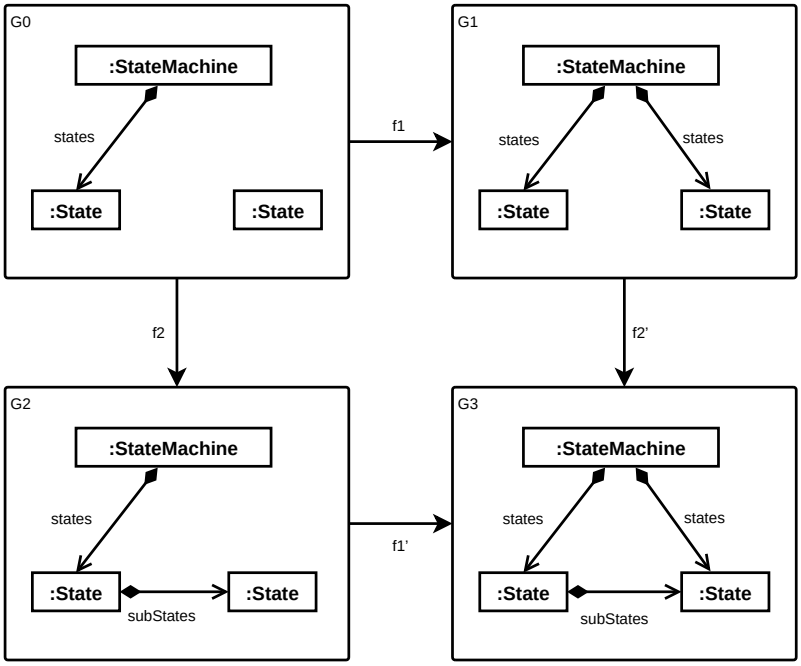


Figure 2: Invalid pushout of graphs with containment edges. A State node in the result graph G_3 has two containers.

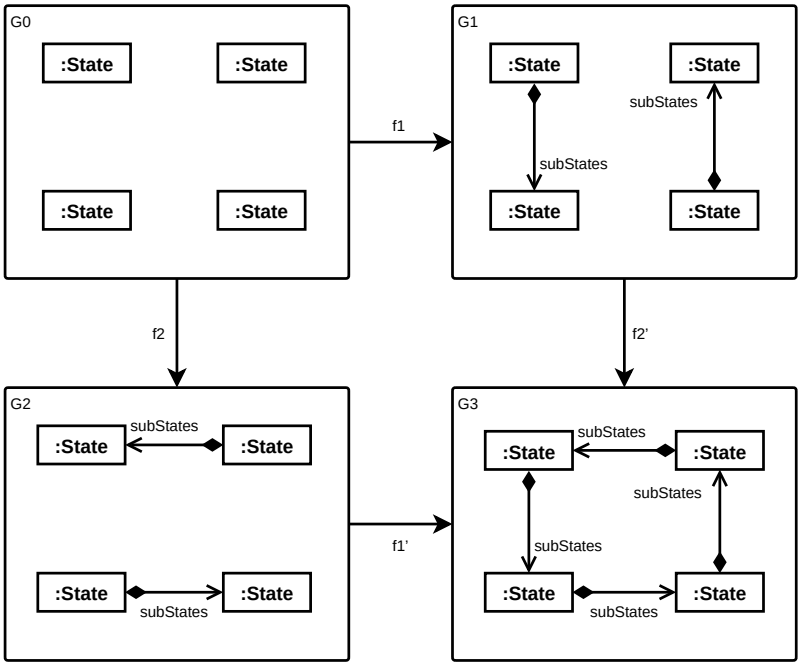


Figure 3: Invalid pushout of graphs with containment edges The pushout result G_3 has a containment cycle.

Definition 3 (Newly contained points) For a homomorphism between graphs with containment edges $h : G_0 \to G_1$, the set of *newly contained points* $NCP_h \subseteq V_0$ is defined as

- $NCP_h = \{x \in V_0 \mid \forall y \in V_0 : \ (y,x) \notin contains_0 \ \wedge \ \exists z \in V_1 : \ (z, h_V(x)) \in contains_1\}$[3]

Definition 4 (Cyclic contained points) For a span of homomorphisms between graphs with containment edges $(G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$, the set of *cyclic contained points* $CCP_{f_1,f_2} \subseteq V_1 \cup V_2$ is defined as

- $CCP_{f_1,f_2} = \{x \in V_1 \cup V_2 \mid ([x]_{\equiv}, [x]_{\equiv}) \in \ t(R) \ \}$

where t(R) is the transitive completion of $R = \{ \ ([x_1]_{\equiv}, [x_2]_{\equiv}) \mid (x_1, x_2) \in \ contains_1 \cup contains_2 \ \}$ and $[x]_{\equiv}$ denotes the equivalence class of an $x \ \in V_1 \cup V_2$ w.r.t. the equivalence relation $\equiv = t(s(r(\sim)))$, with $\sim = \ \{(f_1(x_0), f_2(x_0)) \mid x_0 \ \in V_0\}$.

Given a homomorphism $h : G_0 \to G_1$ of graphs with containments, NCP_h is the set of nodes in G_0 that do not have a container in G_0, but do have one in G_1. The definition of the set of cyclic contained points CCP_{f_1,f_2} is more complex. It is based on the equivalence relation $\equiv$ that states which nodes of G_1 and G_2 have a common source in the interface graph G_0 and which are therefore glued together in the pushout result G_3. The relation $t(R)$ is the transitive closure of the containment relations of G_1 and G_2, based on the equivalence classes induced by $\equiv$. Cyclic contained points are those nodes for which this relation is reflexive, i.e. $([x]_{\equiv}, [x]_{\equiv}) \in \ t(R)$. These nodes must form a containment cycle in the pushout result G_3 by construction.

These two definitions of *newly contained points* and *cyclic contained points* induce the condition under which a pushout of graphs with containment edges exists.

Definition 5 (Containment condition) Given a span of graphs with containment edges $(G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$, where f_1 and f_2 are valid homomorphisms for graphs with containment edges, the containment condition is stated as:

- $NCP_{f_1} \cap NCP_{f_2} = \emptyset$ and
- $CCP_{f_1,f_2} = \emptyset$

If the newly contained points of f_1 and f_2 are disjoint, as stated in the first condition, each node in the result graph has at most one container. For ensuring that there is no containment cyclic in G_3 the set of cyclic contained points must be empty.

Theorem 1 (Pushouts of graphs with containment edges) *Given a span of homomorphisms between graphs with containment edges* $(G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$*, the pushout result in the category of graphs* $(G_1 \xrightarrow{f_2'} G_3 \xleftarrow{f_1'} G_2)$ *forms also a valid pushout in* ***CGraphs****, if and only if the containment condition holds for the span* $(G_1 \xleftarrow{f_1} G_0 \xrightarrow{f_2} G_2)$.

[3] $contains_0$ and $contains_1$ are the induced containment relations of graphs G_0 and G_1 .

Proof. **At most one container.** Let $a \in NCP_{f_1} \cap NCP_{f_2}$, then:

- $f_1(a), f_2(a)$ have incoming containment edges $e_1 \in C_1, e_2 \in C_2$ that have no origin in G_0.
- $b := f_2' \circ f_1(a) = f_1' \circ f_2(a) \in V_3$. The images of a are glued together by construction.
- b has two incoming containment edges $f_1'(e_2) \neq f_2'(e_1)$.

Conversely, consider $b \in V_3$ with two incoming containment edges:

- Without loss of generality, the containment edges can be written as $f_2'(e_1)$ and $f_1'(e_2)$ with $e_1 \in C_1$ and $e_2 \in C_2$. If both had an origin in G_1 for instance, G_1 would already violate the containment properties.
- There is $a \in V_0$ with $f_2' \circ f_1(a) = b = f_1' \circ f_2(a)$. The node b must have origins in V_1, V_2 and finally $a \in V_0$ because there must be target nodes of e_1 and e_2 and these must have been glued together to the node $b \in V_3$.
- Node a has no incoming containment edge. Otherwise this edge would have images in G_1 and G_2 and there would be only one incoming containment edge for b in G_3. Therefore, a is a is a newly contained point, both of f_1 and f_2.

No cycles. The idea of the proof is the construction of a sequence of nodes $a_0, ..., a_n$ in $G_1 \cup G_2$ that corresponds to a containment cycle in the pushout graph G_3. Every cycle in G_3 consists of edges from G_1 and G_2. Since G_1 and G_2 are cycle free, the cycle in G_3 emerges from gluing of edges by identifying certain nodes. Nodes from $G_1 \cup G_2$ are identified, iff they are in the same equivalence class w.r.t. equivalence relation $\equiv$. So every containment cycle in G_3 corresponds to a sequence $a_0, .., a_n$ s.t. there is either a containment edge from a_i to a_{i+1} or a_i and a_{i+1} are identified in G_3. Furthermore a_0 and a_n are identified in G_3.

If there is a $x_3 \in V_3 : x_3 \; contains_3 \; x_3$, then exist $a_0, ..., a_n \in V_1 \cup V_2$ with:

- $i \in [0,n] : \exists \; (a_i, a_{i+1}) \in contains_1 \cup contains_2$. If there is a path of containment edges in G_3, the path consists of containment edges that have preimages in $E_1 \cup E_2$. We write this path as list of $a_i \in V_1 \cup V_2$. At least one of the a_i is container for a_{i+1}, since every containment path consists of at least one edge.
- $(a_i, a_{i+1}) \notin contains_1 \cup contains_2 \Rightarrow [a_i]_\equiv = [a_{i+1}]_\equiv$. If there is no containment edge from a_i to a_{i+1}, both are in the same equivalence class w.r.t. the pushout construction, i.e. they will be identified in G_3. These are the points where the subpaths from G_1 and G_2 are glued to the containment path in G_3.
- $[a_0]_\equiv = [a_n]_\equiv$. Without loss of generality, the path begins and ends at nodes which are glued.
- $f_{1/2}'(a_0) = f_{1/2}'(a_n) = x_3$, with $f_{1/2}'(a_i) = f_1'(a_i)$ if $a_i \in V_1$ and $f_{1/2}'(a_i) = f_2'(a_i)$ if $a_i \notin V_1$.

So in the transitive completion t(R) exists a tuple (r_1, r_2), $r_1 = [a_0]$, $r_2 = [a_n]$ with $r_1 = r_2$.

Conversely, if there is a pair $(r_1, r_2) \in t(R)$ with $r_1 = r_2$, there exist $a_0, ..., a_n \in V_1 \cup V_2$ with

- $i \in [0,n] : \exists\, (a_i, a_{i+1}) \in contains_1 \cup contains_2.\ (r_1, r_2) \in t(R)$ implies a containment path consisting of at least one containment edge of $E_1 \cup E_2$.
- $(a_i, a_{i+1}) \notin contains_1 \cup contains_2 \Rightarrow [a_i]_{\equiv} = [a_{i+1}]_{\equiv}$. If a_i does not contain a_{i+1}, they denote the points where the subpaths from G_1 and G_2 are glued in G_3.
- $[a_0]_{\equiv} = r_1 = r_2 = [a_n]_{\equiv}$.

So there is at least one containment edge in G_3 s.t.:

- $(f'_{1/2}(a_i), f'_{1/2}(a_{i+1})) \in contains_3$
- $(f'_{1/2}(a_i), f'_{1/2}(a_{i+1})) \notin contains_3 \Rightarrow f'_{1/2}(a_i) = f'_{1/2}(a_{i+1})$
- $f'_{1/2}(a_0) = f'_{1/2}(a_n)$

So there is a containment cycle in G_3: $(f'_{1/2}(a_0), f'_{1/2}(a_n)) \in contains_3$. □

Using the DPO approach to graph transformation, we can state the conditions under which a match is valid for a given rule now. That means the result has a valid graph structure (no dangling edges) and all containment constraints are fulfilled. The first constraint is ensured by checking the *gluing condition* [EEPT06] before constructing the pushout complement. When constructing the pushout complement, edges and nodes may be deleted from the input graph. Containment constraints cannot get violated in this phase, because containment cycles or multiple containers for an object can only appear if containment edges are *added* to a valid model instance. After computing the pushout complement, the containment condition must be verified to construct the actual transformation result through a pushout of graphs with containment edges.

4 Related Work

Graphs with additional order structure are also discussed in [DSLO04], where the nodes and edges are labelled using partially ordered sets. However, this partial order is an additional feature and not induced by the graph structure. Therefore, one injective morphism in a span is already enough for ensuring the existence of a pushout in the category of these so called *Poset labelled graphs*. Moreover, these graphs are applied in the area of architectural design of software components.

Further, graphs with additional containment relations are also considered in the context of hierarchical graph transformation (see e.g. [BKK05]). Similarly, each graph item may have at most one container and the containment relation has to be acyclic. But in contrast to our approach, each node and edge (except of the root) must belong to a container in most kinds of hierarchical graphs. Thus, a typical hierarchical graph forms a special case of our graphs with containment edges which does not require that all graph items are contained and that a unique root does exist. For graphs with containment relations as well as for hierarchical graphs, graph transformation has to be defined accordingly to the kind of graphs considered. That means the

pushout construction being the basic building block of algebraic graph transformation, is defined such that the result graph is an object of the corresponding category.

So called *ownership types* are an object oriented model for containment. An extensive introduction to ownership types can be found in [Cla01]. This approach is based on an extension of Abadi and Cardelli's object calculus with subtyping. A so called *containment invariant* is introduced to prove the soundness of ownership types systems.

5 Conclusion and Future Work

The extension of the graph-based approach to model transformation by graphs with containment edges better reflects transformations of MOF, EMF and UML 2 models. The containment condition stated above ensures that a transformation rule can only be applied to a model instance, if the result does not violate any containment constraints.

In the case of EMF, the implementations generated from a model ensure at run-time that these containment constraints are always satisfied. If EMF detects a violation of these constraints at a certain point, it deletes containment edges that produce the problem. This behavior breaks the formal semantics that was achieved through the graph transformation approach. Therefore, it is important to first check whether the result of a transformation step would be valid. Only if this is ensured, the rule can actually be applied. By that, we can avoid the problems of invalid model instances, ensuring formal transformation semantics.

Even though it can be argued, that this approach is conservative (since it restricts the possible applications of a transformation rule), it has the advantage of avoiding situations where an invalid model instance has to be repaired, by deleting edges for instance. Especially, the naive strategy of applying a transformation first, without any restrictions and repairing the model in the end might lead to unexpected results. Moreover, it is not obvious how to resolve problems like containment cycles in a canonical way.

The containment condition can be used in model transformation frameworks to ensure well-defined transformation semantics. The gluing condition together with the containment condition are the basis for semantical analysis of MOF/EMF model transformation, e.g. termination and confluence (critical pair analysis) [EEPT06].

Further work should lead to notions of consistency of model transformations which might not only be limited to basic model constraints, but also high-level properties of models and model transformations formulated in OCL for instance. A translation of OCL constraints into the language of graph transformation has been started in [WTEK06]. Consistency checks are probably going to play an important role in the field of model driven engineering in the future. A comprehensive theoretical foundation can lead to an improved tool support to check the quality of models and model transformations.

Bibliography

[BKK05] G. Busatto, H.-J. Kreowski, S. Kuske. Abstract Hierarchical Graph Transformation. *Mathematical Structures in Computer Science 15(4), pp. 773-819.*, 2005.

[BRST05] J. Bezivin, B. Rumpe, A. Schürr, L. Tratt. Model Transformation in Practice Workshop Announcement. 2005.
http://sosym.dcs.kcl.ac.uk/events/mtip/

[CH03] K. Czarnecki, S. Helsen. Classification of Model Transformation Approaches. *OOPSLA 03 Workshop on Generative Techniques in the Context of Model-Driven Architecture*, 2003.

[Cla01] D. Clarke. Object Ownership and Containment. *PhD thesis, School of Computer Science and Engineering University of New South Wales*, 2001.
http://citeseer.ist.psu.edu/468103.html

[DSLO04] M. Denford, A. Solomon, J. Leaney, T. O'Neill. Architectural Abstraction as Transformation of Poset Labelled Graphs. *Journal of Universal Computer Science, vol. 10, no. 10, 1408-1428*, 2004.
http://www.jucs.org/jucs_10_10/architectural_abstraction_as_transformation/Denford_M.pdf

[EEL+05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, S. Varró-Gyapay. Termination Criteria for Model Transformation. *Proc. Fundamental Approaches to Software Engineering (FASE), Lecture Notes in Computer Science, ISSN 0302-9743, Springer Verlag*, 2005.
http://www.springerlink.com/content/dkpvlnfgrn3k8xp7/

[EEPT05] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Formal Integration of Inheritance with Typed Attributed Graph Transformation for Efficient VL Definition and Model Manipulation. *Proc. of IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'05), IEEE Computer Society*, 2005.

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

[EMF] Eclipse Modeling Framework (EMF) Homepage.
http://www.eclipse.of/emf

[EMT] Eclipse Model Transformation (EMT) Project Homepage at TU-Berlin.
http://tfs.cs.tu-berlin.de/emftrans

[Koe06] C. Koehler. A Visual Model Transformation Environment for the Eclipse Modeling Framework. Diploma thesis, Technical University of Berlin. 2006.
http://tfs.cs.tu-berlin.de/emftrans/papers/06-ChristianKoehler.pdf

[KS06] A. Königs, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science 148, 113-150*, 2006.

[KSW04] J. M. Kuester, S. Sendall, M. Wahler. Comparing Two Model Transformation Approaches. *Proc. of OCL MDE*, 2004.

http://www.cs.kent.ac.uk/projects/ocl/oclmdewsuml04/papers/6-kuster_sendall_wahler.pdf

[OMG06] Object Management Group (OMG). Meta Object Facility (MOF) Core Specification Version 2.0. 2006.
http://www.omg.org/technology/documents/modeling_spec_catalog.htm#MOF

[TEB+06a] G. Taentzer, K. Ehrig, E. Biermann, G. Kuhns, E. Weiss, C. Koehler. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. *Model Driven Engineering Languages and Systems, Lecture Notes in Computer Science 4199, Springer Verlag*, 2006.
http://www.springerlink.com/content/t681013811w30537/

[TEB+06b] G. Taentzer, K. Ehrig, E. Biermann, G. Kuhns, E. Weiss, C. Koehler. EMF Model Refactoring based of Graph Transformation Concepts. *To appear in the Electronic Communications of the EASST*, 2006.

[TEG+06] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, D. Varró, S. Varró-Gyapay. Model Transformation by Graph Transformation: A Comapartive Study. *Model Transformations in Practice Workshop, MoDELS 2005*, 2006.
http://sosym.dcs.kcl.ac.uk/events/mtip05/submissions/

[TR05] G. Taentzer, A. Rensink. Ensuring Structural Constraints in Graph-Based Models with Type Inheritance. *Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science 0302-9743, Springer Verlag*, 2005.
http://www.springerlink.com/content/98tey0jerhy7pryn

[WTEK06] J. Winkelmann, G. Taentzer, K. Ehrig, J. M. Küster. Translation of Restricted OCL Constraints into Graph Constraints for Generating Meta Model Instances by Graph Grammars. *To appear in GT-VMT 2006, Electronic Notes in Theoretical Computer Science*, 2006.

Generic Search Plans for Matching Advanced Graph Patterns

Ákos Horváth[1], **Gergely Varró**[2] **and Dániel Varró**[3]

[1] ahorvath@mit.bme.hu
[3] varro@mit.bme.hu
Department of Measurement and Information Systems
Budapest University of Technology and Economics,
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

[2] gervarro@cs.bme.hu
Department of Computer Science and Information Theory
Budapest University of Technology and Economics,
H-1521 Budapest, Magyar tudósok körútja 2., Hungary

Abstract: In the current paper, we present search plans which can guide pattern matching for advanced graph patterns with edge identities, containment constraints, type variables, negative application conditions, attribute conditions, and injectivity constraints. Based on a generic search graph representation, all search plan operations (e.g. checking the existence of an edge, or extending a matching candidate by navigating along an edge) are uniformly represented as special predicates with heuristically assigned costs. Finally, an executable search plan is defined as an appropriate ordering of these predicates. As a main consequence, attribute, injectivity, and negative application conditions can be checked early (but not unnecessarily early) in the pattern matching process to cut off infeasible matching candidates at the right time.

Keywords: graph pattern matching, search plan

1 Introduction

While nowadays model-driven system development is being supported by a wide range of conceptually different *model transformation tools*, nearly all of these tools have to solve a common problem: the efficient query and manipulation of complex graph-like model structures. Tools based on the rule and pattern-based formal paradigm of *graph transformation (GT)* [Roz97, EEKR99] already integrate research results of several decades. In these tools, a matching of the left-hand side (LHS) of a graph transformation rule is being sought by some graph pattern matching algorithm, which might be invalidated by valid matchings of negative application conditions (NAC) [HHT96]. Finally, the engine performs some local modifications to add or remove graph elements to the matching pattern.

Graph pattern matching leads to the subgraph isomorphism problem that is known to be NP-complete in general [Ata99], which means that highly time-consuming computations are expected for the worst-case scenario from theoretical aspects. However, practical model transformation problems rather have a regular and sparse graph structure, which drastically reduces the

execution time of graph pattern matching. In order to provide acceptable performance in real-world application scenarios, graph transformation tools apply sophisticated pattern matching algorithms, which are mostly based on either constraint satisfaction (like AGG [ERT99]) or local searches driven by search plans (like PROGRES [Zün96], Dörr's approach [Dör95], FUJABA [FNTZ98] or GReAT [AKN+06]). As a commonality, all such algorithms have to appropriately order elementary operations (such as navigations and edge existence checks) in advance by using tool-specific heuristics, which later guide the pattern matching process itself.

Research [Zün96, GSR05, GBG+06, VVF05] has been focusing so far on the performance optimal ordering of elementary pattern matching operations like (i) the enumeration of objects and links of a certain type, (ii) the navigation along links of a given type, and (iii) the existence checks for links. On the other hand, the ordering of (iv) attribute, (v) injectivity and (vi) NAC constraint checking operations has been hard wired into the graph transformation engines by using some simple heuristics.

For instance, in case of NAC checking operations, two wiring strategies are known in GT tools. The "*as soon as possible*" (ASAP) style positioning (used by Fujaba) places the NAC checking operation to the first possible location where all its arguments are bound. Intuitively, when the size of the NAC pattern is small compared to the unexplored part of the LHS pattern, a quickly retrieved matching for the NAC may significantly reduce the search space by avoiding the unnecessary traversal of the remaining part of the LHS. On the other hand, when the NAC is large, the corresponding check operation can be time consuming, so a delayed execution may provide better overall performance for the pattern matching of the LHS. This idea is implemented by the "*as late as possible*" (ALAP) strategy (used by PROGRES), which executes NAC checking only when a complete matching for the LHS has been found.

These best engineering practices are acceptable for performing cheap checks like checking attribute and injectivity constraints, but when a single search plan operation represents a complete pattern matching process like in case of checking a NAC or calling native external libraries (as in AGG or VIATRA2), hard-wired positioning may cause performance degradation as (a) it lacks flexibility and extensibility and (b) it ignores the complexity of the actual search plan operation. However, checking NAC is critical to model transformation problems in order to forbid multiple application of a rule on the same matching.

This is a common situation in case of model-specific search plans [VVF05, GBG+06] where the cost of search plan operations depends on the actual graph being transformed. However, even in the case of (traditional) metamodel-specific search plans (like in FUJABA, GReAT or PROGRES), the bindings of input parameters of rules may have a huge impact on the optimal ordering of complex search plan operations. Intuitively, if many input parameters are passed to a rule, the ASAP strategy can be too expensive for complex NACs.

In the current paper, we propose a general framework for uniformly representing a large variety of search plan operations by expressing them as cost-weighted predicates. As an appropriate ordering of these predicates defines an executable search plan, this approach is able to uniformly guide the pattern matching process for advanced graph patterns regardless of how we assign the actual costs to different search plan operations. As a result, better performance is expected, especially, for checking negative application conditions, which avoids the previous problems.

The main practical advantages of our approach are modularity, flexibility, and extensibility. The different phases of pattern matching (e.g. cost assignment, generation of search plans, exe-

cution of search plans etc.) are fully separated and independent, thus they can be adapted to very different graph transformation engines and strategies (metamodel-based vs. model-based search plans). Furthermore, new types of predicates can be introduced easily by assigning appropriate costs without altering the algorithms for search plan generation.

The rest of the paper is structured as follows. First, Section 2 briefly introduces a combined graph-based representation for models and metamodels used in the paper (and in the VIATRA2 framework). Then Section 3 proposes our unified predicate based framework for driving pattern matching processes. Related work is discussed in Section 4, while Section 5 concludes our paper.

2 Background

2.1 Models and Metamodels

Metamodeling is a fundamental part of model transformation design as it allows the structural definition (i.e., abstract syntax) of modeling languages.

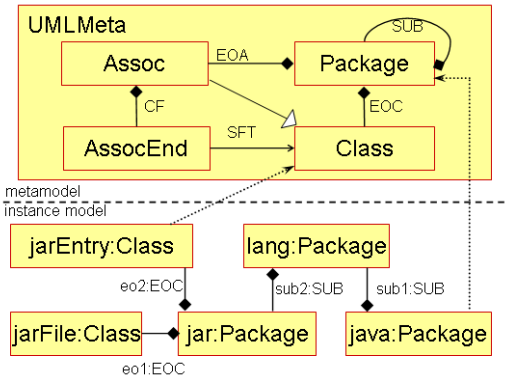
In the paper, we use a unified hierarchical and directed graph representation which stores metamodels and models in a combined model space. Intuitively, the morphisms from instance nodes (and edges) to their respective node (edge) types are stored explicitly in our graph model. This unified graph representation serves as the underlying model of the VIATRA2 framework.

This way, graph nodes (called entities in VIATRA2) uniformly represent MOF packages, classes, or objects on different metalevels, while graph edges with identities (called relations in VIATRA2) denote MOF association ends, attributes, link ends, and slots in a uniform way. As a summary, nodes represent basic concepts of a (modeling) domain, while edges represent the relationships between other model elements. Nodes are arranged into a strict containment hierarchy (to denote model element containment either on the metamodel or model-level).

There are two special relationships between graph elements: the **supertypeOf** (inheritance, generalization) relation represents binary superclass-subclass relationships between nodes or edges, while the **instanceOf** relation represents type-instance relationships (to explicitly represent the meta-levels).

By using explicit **instanceOf** relationship, metamodels and models can be stored in the same model space in a compact way. Furthermore, this allows the use of so-called *generic patterns* in transformation rules (see later in Figure 2), which capture common graph algorithms (e.g. transitive closure, graph searches, etc.) independently of a certain metamodel.

Example 1 *Figure 1 presents the joint representation of a simplified* UML *metamodel and an instance model. The metamodel is contained in the* ***UMLMeta*** *element. Both the classes of the metamodel (such as* ***Package***, ***Assoc***, *etc.) and the objects of the instance model (such as* ***jar***, ***jarEntry***, *etc.) uniformly appear as nodes (entities). Instance-of relation between nodes is also represented by dotted edges (for easier readability not all edges are illustrated). This example illustrates the joint graph representation and also the* VIATRA2 *representation.*




```
//metamodel
entity(UMLMeta){
  entity(Package){
    relation(EOA,Package,Assoc);
    relation(EOC,Package,Class);
    relation(SUB,Package,Package);}
  entity(Class);
  entity(Assoc){
    relation(CF,Assoc,AssocEnd);
    supertypeOf(Assoc,Class);}
  entity(AssocEnd){
   relation(STF,AssocEnd,Class);}
}
//instance model
 Package(java);
 Package.SUB(sub1,java,lang);
 Package(lang);
 Package.SUB(sub2,lang,jar);
 Package(jar);
 Package.EOC(eo1,jar,jarEntry);
 Package.EOC(eo2,jar,jarFile);
 class(jarEntry); class(jarFile);
```

Figure 1: VPM example

2.2 Graph Patterns

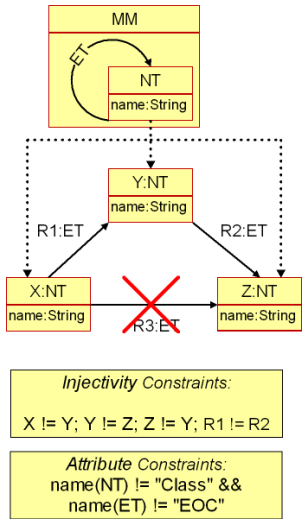
Graph transformation (GT) is a rule and pattern-based paradigm frequently used for describing model transformation. A graph transformation rule contains a left-hand side graph LHS, a right-hand side graph RHS, and (one or more) negative application condition graphs NAC connected to LHS. A negative application condition [HHT96] is a graph morphism, which maps the LHS pattern to a *NAC pattern*. The *application* of a rule to a *host (instance) model M* replaces a matching of the LHS in M by an image of the RHS.

Graph patterns (precondition pattern) consist of the LHS pattern, the NAC pattern, and the mapping between them. They represent conditions (or constraints) that have to be fulfilled by a model in order to execute transformation steps on the model. The most critical step of graph transformation is graph pattern matching, i.e., to find such a matching of the LHS pattern in the model space that is not invalidated by a matching of the negative application condition graph NAC, which prohibits the presence of certain combinations of nodes and edges. So we restrict our current investigations only to graph patterns and graph pattern matching.

Example 2 *An example graph pattern is depicted in Figure 2, where the left side illustrates the graph representation and the right side shows the* VIATRA2 *representation, which is detailed in [BV06]. This pattern is our running example through the paper, as it contains all advance graph pattern elements (e.g., attribute check, nac, etc.), and requires a generic graph representation.*

TransitiveC *pattern is a generic implementation of the transitive closure, which can be used with any metamodel* ***MM*** *passed as a parameter, where* ***NT*** *is matched to the* Node Type *and* ***ET***

to the Edge Type, *which runs between* ***NT*** *type nodes (with a restriction on the name attributes).* ***X*** *represents the inspected element for the transitive closure and* ***Z*** *expresses the closure nodes. The injectivity constraints define that all variables are matched to different elements and the* ***negative*** *application condition expresses that there is no edge* ***R3*** *between the* ***X*** *and* ***Z*** *nodes. For easier readability the explicit "instance of" edges (dotted lines) are only depicted between the* ***NT*** *type element and the* ***X, Y*** *and* ***Z*** *nodes.*




```
pattern transitiveC(X,Z,MM) ={ //meta model
  entity(NT) in MM;
  relation(ET,NT,NT);
  check(name(NT)!="CLASS" && name(ET)!="EOC");
  //instance model nodes
  entity(X);  instanceOf(X,NT);
  entity(Z);  instanceOf(Z,NT);
  entity(Y);  instanceOf(Y,NT);
  X!=Y; Y!=Z; X!=Z;
  //instance model edges
  relation(R1,X,Y); instanceOf(R1,ET);
  relation(R2,Y,Z); instanceOf(R2,ET);
  R1!=R2;
  //nac check
    neg nac1(X,Z,ET)={
    entity(X); entity(Z);
    relation(R3,X,Z); instanceOf(R3,ET);}
}
```

Figure 2: The pattern graph of transitiveC(X,Z,MM)

2.3 Graph Pattern Matching

Each variable of a graph pattern is bound to a constant node in the model such that this *matching* (binding) is consistent with edge labels, and source and target nodes of the model. A *matching for a precondition pattern* is a matching for its LHS pattern, provided that no matching should exist for its NAC pattern.

To drive the pattern matching process, the generation of search plans is a frequently used concept. Informally, a search plan defines the order of traversal (a search sequence) for the nodes of the instance model to check whether the pattern can be matched. The model is traversed according to a specific search plan.

Example 3 *For instance, a matching of the pattern transitiveC of Figure 2 in model Figure 1 with UMLMeta as the input of* ***MM*** *is the following:* ***X*** *= java,* ***Y*** *= lang,* ***Z*** *= jar,* ***R1*** *= sub1,* ***R2*** *= sub2,* ***ET*** *= SUB and* ***NT*** *= Package (****MM*** *= UMLMeta). Where a possible traversal order for the pattern is: (0) MM (it is an input parameter),(1) NT, (2) ET, (3) X, (4) Y, (5) Z, and for the* ***nac1*** *negative application condition is: (1) ET, (2) X, (3) Z.*

3 Unified Search Plan Representation

This section introduces our approach on handling advanced graph patterns. First Subsection 3.1 presents the concept of *search graphs*, which can handle complex constraints on the graph patterns (e.g., containment, generic type parameters etc.). Then Subsection 3.2 introduces the adornment constraints, followed by Subsection 3.3 which proposes our complex constraint cost approximation, and finally, Subsection 3.4 introduces the revised representation of search plans based on the elementary steps of a pattern matching process (search operations).

3.1 Search Graph

A *search graph* is a joint representation of pattern graph elements and operation constraints that drives the pattern matching process. In our interpretation a search graph is a *hypergraph* representing a *constraint net*, where graph nodes reflect variables, and hyperedges express constraints (predicates) between the variables. A search graph is directly derived from the pattern graph as follows:

1. Pattern variable: Each element (node or edge) of the pattern graph is mapped to a *pattern variable*. These elements (depicted by grey ovals, e.g., **X** in Figure 3) represent the arguments of the constraints.

2. Operation Constraint: Each constraint on the pattern graph (containment, connectivity etc.) is mapped to an n-ary (usually binary) operation predicate, (illustrated by rectangles in our figures, e.g., **trg** in Figure 3) that has to be fulfilled during the matching process. The constraints used in our approach are the following (however, the actual set of constraints can be extended easily):

1. **Simple predicates** represent core constraints between two pattern variables.
   - A *source constraint* **src**(Src,E) expresses that node **Src** is the source of edge **E** in the pattern graph.
   - A *target constraint* **trg**(Trg,E) expresses that node **Trg** is the target of edge **E** in the pattern graph.
   - An *'instance of' constraint* **inst**(A,Type) means that **A** is an instance of **Type**, where both elements are represented in the model space.
   - A *containment constraint* **in**(A,Container) expresses that **Container** contains **A**.
2. **Complex predicates** are defined between an arbitrary number of pattern graph elements.
   - The *injectivity constraint* $\mathbf{inj}(A_1,...,A_n)$ means that the A_i must be matched to different graph elements (injective matching).
   - The *negative application condition* $\mathbf{nac}_j(A_1,...,A_n)$ expresses that the check of $\mathbf{nac}_j$ should be initiated with the given input parameters.

- The *attribute check constraint* **attr**$(A_1,...,A_n)$ evaluates a Boolean expression checking, based on the attributes of the pattern nodes, which are accessed by variables $A_1,...,A_n$.

Example 4 *The search graph of Figure 2 is illustrated in Figure 3 (including some parts of the pattern graph itself to improve readability). The search graph contains eight pattern variables;* ***MM, ET*** *and* ***NT*** *represent the metamodel part of the pattern graph, while variables* ***X, Y, Z, R1*** *and* ***R2*** *represent the nodes and edges of the instance model. The operation predicates directly define the constraints of the pattern graph:* ***src, trg*** *define the source and target node of an edge. For example,* ***X*** *is the source of edge* ***R1****,* ***in*** *defines that* ***ET*** *and* ***NT*** *should be contained directly by node* ***MM****,* ***nac1*** *represents the negative application condition with input parameters* ***ET, X*** *and* ***Z****,* ***inj*** *defines the injectivity check between its input variables like between* ***X, Y****, and finally* ***inst*** *represents the direct instance of relations (e.g., between* ***R1*** *and* ***ET****).*

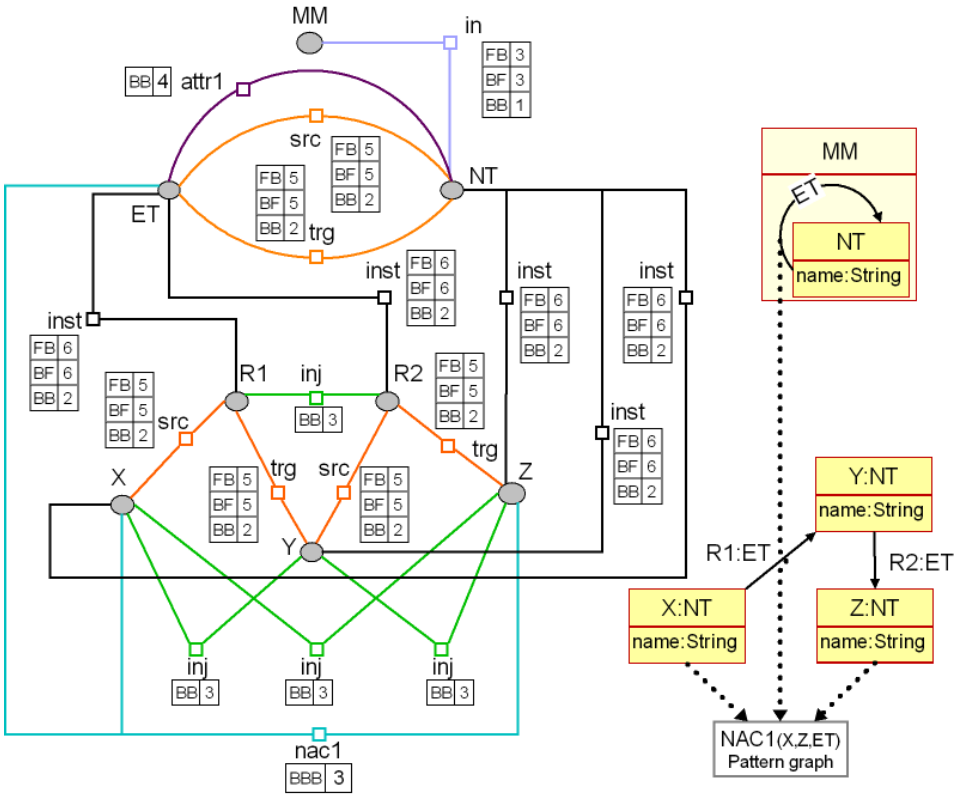


Figure 3: Search graph of GT pattern transitiveC(X,Z,MM)

3.2 Adornment

An operation constraint may represent different concrete search operations depending on the binding of its arguments.

An *adornment* (see in [Ull89]) consists of a string, composed of letters B (bound) and F (free). The meaning of letter B in an adornment is that the variable must be bound to a value in that position. The meaning of the letter F in an adornment is that the variable is not bound in that position.

A *search operation* consists of a constraint and an adornment, they are the atomic units of pattern matching and represents a single step in the matching process. A search operation is either an *extend* type operation which extends the matching by a new element (e.g match the target node along an edge), or a *check* type operation used for checking constraints between pattern elements (e.g., whether an edge runs between two nodes). For example, if adornments FB or BF are attached to a simple constraint, then they represent an extend type search operation, and in case of BB it is a check type operation.

Though 2^n different adornments can be assigned to each n-ary operation constraint in theory, only a subset of these adornments are used, which respect elementary complexity consideration.

Usually for the simple constraints the permitted adornments are FB; BF; BB, while FF represents a far too expensive operation, as we need to cumulate all pairs of elements in the model.

For complex constraints only the B...B adornments are permitted, as all variables must be bound to an element in case of injectivity checking, NAC checking and Boolean term evaluation.

In Figure 3, permitted adornment values are illustrated with small tables near the constraints. The table has two columns, which show the adornment and the cost of the operation (which is discussed in Subsection 3.3), respectively.

3.3 Cost of Search Operations

At this point, a joint representation of search constraints is available. In order to generate efficient search plans cost functions (weights) have to be defined for the operations. Due to space limitations in the paper we are using the more common meta-model based (compile time) weighting and the number values are only used to illustrate the order of magnitude of operation costs, but it is important to mention that the approach is also capable of handling model (runtime) based weighting. The only differences are in case of runtime weighting are (i) the usage of runtime statistics collected from the instance model, (ii) and the more precise weighting of *simple extend* type predicates (e.g., the weight of an *'instance of' constraint* is based on *actual number* of the instance elements in the model), and *nac check* predicates, where the cost can be directly derived from the pattern graph of the *nac*.

Weighting the *simple operation* follows the guidelines of edge multiplicity based cost functions (e.g., if an edge multiplicity is one-to-many, then its cost is higher then if it is one-to-one) with the following restriction: the lowest cost is assigned to the BB adornments (*check* type operation), and there is no difference between the cost of FB and the BF (*extend* type operation). Among the constraints we use the cost ordering based on our earlier transformation experiments and [VSV06]: $in << trg = src << inst$

In case of complex constraints, assigning costs to operations is easier on one hand as they have only one permitted adornment B...B, but on the other hand better cost prediction is possible using a priori knowledge. In case of *inj* and *attr* constraints the number of input parameters provides a good prediction for complexity, while in case of a *nac* constraint the whole *nac* pattern

graph matching cost can be evaluated at compile time. The cost functions are the following:

- For *inj* and *attr* constraint the cost function is linear in the number of parameters.
- For *nac* constraint the cost function is proportional to the number of constraints in the search graph of the *nac* pattern. The idea behind this selection is that a *nac* check may cut the search space significantly when the *nac* pattern is small.

3.4 Search Plans

A *search plan* is a totally ordered list of search operations (one possible traversal of the search graph). As the atomic operations already have cost values attached, we can evaluate the cost of a whole search plan.

The *cost of a search plan* (denoted by $w(P)$) is defined by the formula $w(P) = \sum_{i=1}^{n} \prod_{j=1}^{i} w_j$, where w_j is the weight of the jth operation according to the order defined by the search plan, and n is the number of operation constraints in the original search graph. As described in [VVF05], this formula is an estimation for the size of the search space that has to be traversed during pattern matching, if model-specific search graphs are used and the weight w_j expresses the expected number of iterations performed during the execution of the jth operation. As a consequence, the search plan with the minimum cost $w(P)$ will have the best expected run-time performance. If the weights are fixed and determined at compile-time, this cost function is still an acceptable choice as the search plan that is optimal wrt. $w(P)$ prefers the early execution of low cost operations. For generating the actual search plans, only minor modifications to the techniques described in [VVF05] are needed.

Example 5 *The following example shows two search plans of the running example pattern graph: Figure 4(a) represents the search plan where* ***X,Z*** *and* ***MM*** *parameters are bound, while Figure 4(b) shows the search plan where only* ***X*** *and* ***MM*** *are the fixed input parameters.*

In case of Figure 4(a), the nac_1 *is in the first quarter of the search plan, which means it is an ASAP like positioning, and checks the negative application condition before the extend operation towards the* ***R1,R1*** *and* ***Y*** *variables. While in case of the search plan depicted in Figure 4(b) the* nac_1 *check is in the end as in case of an ALAP like ordering.*

This simple example shows, that the joint search plan representation is capable of handling different positioning for advance constraints, which (like in case of checking a NAC) could result better search plans, compared to hard-wired positioning.

4 Related work

All graph transformation tools use some clever strategies for pattern matching. Since an intensive research has been focused to graph transformation for a couple of decades, several powerful methods have already been developed. First we focus on the three most advanced compiled approaches that use search plan guided and local search based algorithms.

Fujaba [KNNZ00] performs local search starting from the node selected by the system designer and extending the matching step-by-step to neighbouring nodes and edges. Fujaba fixes

constraint	type	comment
$inj^{BB}(Z,X)$	check	Z not equals to X
$in^{FB}(NT,MM)$	extend	NT under MM
$inst^{BB}(X,NT)$	check	X is instance of NT
$inst^{BB}(Z,NT)$	check	Z is instance of NT
$trg^{FB}(ET,NT)$	extend	target of ET is NT
$src^{BB}(ET,NT)$	check	source of ET is NT
$nac_1^{BBB}(X,Z,ET)$	check	nac_1 check
$attr_1^{BB}(ET,NT)$	check	$attr_1$ is evaluated
$trg^{FB}(R2,Z)$	extend	target of R2 is Z
$inst^{BB}(R2,ET)$	check	R2 is instance of ET
$src^{BF}(R2,Y)$	extend	source of R2 is Y
$inst^{BB}(Y,NT)$	check	Y is instance of NT
$inj^{BB}(Y,Z)$	check	Y not equals to Z
$inj^{BB}(X,Y)$	check	X not equals to Y
$trg^{FB}(R1,Y)$	extend	target of R1 is Y
$src^{BB}(R1,X)$	check	source of R1 is X
$inst^{BB}(R1,ET)$	check	R1 is instance of ET
$inj^{BB}(R1,R2)$	check	R1 not equals to R2

(a) X,Z and MM are fixed input parameters

constraint	type	comment
$in^{FB}(NT,MM)$	extend	NT under MM
$inst^{BB}(X,NT)$	check	X is instance of NT
$trg^{FB}(ET,NT)$	extend	target of ET is NT
$src^{BB}(ET,NT)$	check	source of ET is NT
$attr_1^{BB}(ET,NT)$	check	$attr_1$ is evaluated
$src^{FB}(R1,X)$	extend	source of R1 is X
$inst^{BB}(R1,ET)$	check	R1 is instance of ET
$trg^{BF}(R1,Y)$	extend	target of R1 is Y
$inst^{BB}(Y,NT)$	check	Y is instance of NT
$inj^{BB}(X,Y)$	check	X not equals to Y
$src^{FB}(R2,Y)$	extend	source of R2 is Y
$inst^{BB}(R2,ET)$	check	R2 is instance of ET
$inj^{BB}(R1,R2)$	check	R1 not equals to R2
$trg^{BF}(R2,Z)$	extend	target of R2 is Z
$inst^{BB}(Z,NT)$	check	Z is instance of NT
$inj^{BB}(Y,Z)$	check	Y not equals to Z
$inj^{BB}(Z,X)$	check	Z not equals to X
$nac_1^{BBB}(X,Z,ET)$	check	nac_1 check

(b) X and MM are fixed input parameters

Figure 4: Search plans of transitiveC

a single traversal strategy at compile-time for each rule by automatically generating Java code for the pattern matching process. During code generation Fujaba places attribute, injectivity and NAC checks to the earliest allowed location. As a consequence, the corresponding run-time operations are executed immediately after all the necessary variables have been fixed, which means that this engine implements a hard-wired as soon as possible strategy.

PROGRES [Zün96] uses the advanced concept of operation graphs for representing structural constraints on the ordering of basic operations, which are similar to search graphs in the current paper. Costs of search plan operations are defined by using a very sophisticated application domain independent cost model. PROGRES can assign weights to attribute checking operations, which enables their proper scheduling in search plans. On the other hand, injectivity and NAC checks are excluded from the cost model, which results in their hard-wired positioning at runtime. Injectivity constraints are tested as soon as all its arguments are known, while negative application conditions are checked late, i.e., when a complete matching for the pattern has been found. Compared to our approach, PROGRES supports navigation along indexed attributes in addition to attribute checking.

The pattern matching engine of GReAT [AKN+06] only allows injective matchings whose corresponding constraints are checked in an 'as soon as possible' style just like attributes, which are tested immediately whenever a new partial matching has been calculated as a result of an extension of a smaller matching. In GReAT, negative application conditions can only be expressed by zero cardinality edges, which normally restricts the size and complexity of NACs, but reduces expressiveness obviously.

Algorithms that handle pattern matching as a constraint satisfaction problem (CSP) like [LV02]

in AGG [ERT99] do not directly involve the concept of search plans as stated in [VSV06]. However, the underlying constraint solver engine has to define a variable binding order, which can be considered as a search plan derived dynamically at run-time. As a consequence, CSP-based graph transformation engines by their nature support that dynamicity that has been achieved by our approach for local search based algorithms. However, as constraint solver implementations typically use the first-fail principle for determining the variable binding order, this technique still schedules the attribute, injectivity and NAC checking operations to the earliest possible location.

Altogether, we believe that our current contribution is in the increased generality and modularity of the search graphs, and in the more flexible handling of negative application conditions. This way, the current paper is complementary to (and can be integrated with) the recent advances in model-specific search plans (as in GrGen [GBG+06] or [VVF05]).

5 Conclusion

In the current paper, we have presented a general framework for uniformly representing search plan operations. The essence of the approach is to express the operations as cost weighted predicates and assign the weights based on the binding of their input parameters. We used adornments to capture binding constraints on the predicates and introduced a compile-time weighting for a variety of advance pattern graph elements. The implementation based on these techniques will be available in the new VIATRA2 release.

In the future, it will be interesting to apply our solution on recursive graph patterns, where recursive call represents a new predicate in the search graph, which does not have any restrictions on its adornment, making cost assignment complicated.

Acknowledgements: The paper is partially supported by the SENSORIA European IP (IST-3-016004).

Bibliography

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *Software and Systems Modeling* 5(3):261–288, September 2006.

[Ata99] M. J. Atallah (ed.). *Algorithms and Theory of Computation Handbook*. CRC Press, 1999.

[BV06] A. Balogh, D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proc. of the 21st ACM Symposium on Applied Computing*. Pp. 1280–1287. ACM Press, Dijon, France, April 2006.

[Dör95] H. Dörr. *Efficient Graph Rewriting and Its Implementation*. LNCS 922. Springer-Verlag, 1995.

[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.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. *In [EEKR99]*. Chapter The AGG-Approach: Language and Tool Environment, pp. 551–603. World Scientific, 1999.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Engels and Rozenberg (eds.), *Proc. of the 6th International Workshop on Theory and Application of Graph Transformation*. LNCS 1764, pp. 296–309. Springer Verlag, 1998.

[GBG+06] R. Geiß, V. Batz, D. Grund, S. Hack, A. M. Szalkowski. GrGen: A Fast SPO-Based Graph Rewriting Tool. In *Proc. of the 3rd International Conference on Graph Transformation*. 2006. Accepted paper.

[GSR05] L. Geiger, C. Schneider, C. Reckord. Template- and Modelbased Code Generation for MDA-Tools. In Giese and Zündorf (eds.), *Proc. of the 3rd International Fujaba Days*. Pp. 57–62. Paderborn, Germany, September 2005. ftp://ftp.upb.de/doc/techreports/Informatik/tr-ri-05-259.pdf.

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

[KNNZ00] T. Klein, U. Nickel, J. Niere, A. Zündorf. From UML to Java And Back Again. Technical report, University of Paderborn, 2000.

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

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

[Ull89] J. D. Ullman. *Principles of Database and Knowledge-Base Systems*. Volume II: The New Technologies. Computer Science Press, 1989.

[VSV06] G. Varró, A. Schürr, D. Varró. Experimental Evaluation of Optimization Techniques in Graph Transformation Tools by Benchmarking. *Software and Systems Modeling*, 2006. Submitted paper.

[VVF05] G. Varró, D. Varró, K. Friedl. Adaptive Graph Pattern Matching for Model Transformations using Model-sensitive Search Plans. In Karsai and Taentzer (eds.), *Proc. of Int. Workshop on Graph and Model Transformation (GraMoT'05)*. ENTCS 152, pp. 191–205. Tallinn, Estonia, September 2005.

[Zün96] A. Zündorf. Graph pattern-matching in PROGRES. In *Proc. 5th Int. Workshop on Graph Grammars and their Application to Computer Science*. LNCS 1073, pp. 454–468. Springer-Verlag, 1996.

A Query Language With the Star Operator

Johan Lindqvist and Torbjörn Lundkvist and Ivan Porres

{johan.lindqvist,torbjorn.lundkvist,ivan.porres}@abo.fi
TUCS Turku Centre for Computer Science
SoSE Graduate School on Software Systems and Engineering
Department of Information Technologies, Åbo Akademi University
Joukahaisenkatu 3-5A, FIN-20520 Turku, Finland

Abstract: Model pattern matching is an important operation in model transformation and therefore in model-driven development tools. In this paper we present a pattern based approach that includes a star operator that can be used to represent recursive or hierarchical structures in models. We also present a matching algorithm, motivating examples and we discuss its implementation in a modeling tool.

Keywords: Visual languages, Model transformation, Graph query, Graph subgraph matching

1 Introduction

In the context of model-driven software development, a query language is used to find parts of a model that fulfill some given constraints. A query language is a fundamental element in rule-based model transformation languages. Query languages are also used to define model constraints, where a model is invalid if it does not satisfy the query. Finally, a query language combined with different aggregation operators can be used to compute metrics.

We consider that query languages should be declarative, in the sense that they should state what to search for in a model, but not how to perform the actual search. Also, we are interested in expressive query languages that can define complex patterns in a succinct way. The Object Management Group (OMG) proposes a standard for a model transformation language called Query-View-Transform (QVT) [OMG05a], that contains a query language. The OMG Object Constraint Language [OMG03] can also be used to query models.

In this article we explore the idea of a query language based on graph matching. Our approach can benefit from the fact that modeling languages and models are considered as graphs, since the application of graph theory to computer science provides a solid foundation to model-driven development tools, specially in the area of model transformations [Roz97]. Successful approaches to graph transformation in the context of software development are presented for example in [BH02, VVP02, ARS05].

Probably, the simplest graph matching approach is one based on subgraph isomorphism. A software model and a query are represented as graphs and a match of the query is any subgraph of the target model that is isomorphic to the query. However, this approach is not sufficient to express many queries succinctly. Therefore, it has been extended to include negative application conditions [HHT96] and multi-objects [SWZ99]. Still, these extended forms of graph pattern matching may not be able to express many interesting queries. Many computer languages con-

tain hierarchical and recursive structures. Examples of these structures in UML [OMG05b] are package containment hierarchies in class diagrams or state hierarchies in statecharts. As a consequence, we often need to specify queries to match recursive structures where the number of elements to match is not known a priori.

In this article, we propose a new query language that supports what we call the star operator. This operator conceptually resembles the Kleene star operation over sets of strings. Used in our query language, it can match against a subgraph that appears repeatedly zero or more times in a graph representing a model. In our opinion, when we combine the star operator with the isomorphism operator that denotes isomorphic matches and the negation operator, that denotes the absence of a match, we can express complex queries by rather using short and intuitive pattern.

We proceed as follows: In Section 2 we describe the basics of our query language and provide some examples of queries for the UML language. Section 3 presents an overview of a matching algorithm for this query language. The next section discusses the practical implementation of the approach in an experimental modeling tool. Finally, we conclude in Section 5 with a description of future work.

2 Regions in a Pattern

In this section we will describe the concept of regions in a pattern, and introduce three operators that can be applied to regions: the isomorphic, star and negation operator.

A pattern consists of a of a typed and directed graph annotated with information necessary to perform a query. The graphs in the patterns are constructed according to a metamodel. The pattern graph can be compared against a target graph. A match occurs if all nodes and edges of the pattern graph can be mapped to a subgraph of the target graph, with respect to the annotations of the pattern graph. The result is a mapping of the pattern and target graphs, which allows the nodes and edges of a pattern graph to be bound to the target graph.

In order to increase the expressiveness of a pattern based query, we have introduced the concept of operators and regions in a pattern. A *region* is defined over a connected subgraph of a pattern, such that a node belongs only to one region. In our approach we have defined a region as the scope of a matching operator. As a consequence, a pattern consists of several non-overlapping regions, where each region is associated with an operator. Edges can still connect nodes in separate regions. Edges that cross the boundaries of a region are called *connection points* since they connect two regions. These connection points can be computed from the pattern graph and are used, depending on the operator associated with the region, to validate whether the region fulfills the specific requirements of the operator associated with the region.

Next, we will describe the definition of the isomorphic, negation and star operator applied over a region.

2.1 Isomorphic Regions

The semantics of an isomorphic region as part of a pattern graph, is that it is possible to find a subgraph in the target model that is isomorphic to the region. A pattern graph can have several

isomorphic regions. However, if a pattern consists only of isomorphic regions, the regions could be merged without affecting the result of the pattern matching process.

2.2 Negative Regions

The semantics of a negative region as part of a pattern graph, is that an occurrence of all nodes and edges of a negative region in the target results in a failed match. Since the negation operator is always defined over a region in the pattern, it is possible to model complex negative conditions that involve several nodes and edges. A similar approach where the negation operator is defined over regions can be found in [HHT96].

2.3 Star Regions

In order to be able to describe patterns with recursive or hierarchical structures, we have introduced the concept of a star operator. The star operator in a pattern is conceptually similar to the star operator in Kleene algebra [Koz91]. A pattern with a star region can be used to generate a set of patterns where the contents of the star region is inserted an arbitrary number of times and replaced by an isomorphic region. Analogous to the Kleene star operator, the generation of patterns begins with a pattern where the subgraph is not inserted. We will discuss the constraints that apply to valid star regions later in this section.

The structure of the subgraph represented in a star region must follow some specific requirements. This is necessary, as the patterns used for defining a query as well as new patterns that are generated by expanding the star regions into several subgraphs must preserve the structure defined by the metamodel. To ensure that the star region can be expanded, it needs to have at least two connection points to other regions. This limitation only applies to star regions. These connection points are called the ends of the star region, where one end is incoming and the other end is outgoing with respect to the nodes in the star region. The connection points define the position in the pattern graph where subgraphs generated from the star region are inserted. When the star region is expanded, the outgoing end of each generated subgraph is connected to the incoming end of the next.

Beside the two required ends, the star region may contain other connection points, which are required to connect to the same nodes inside the region as the ends do. These additional connection points are associated with either one of the ends and connect the last subgraph generated in the direction of the associated end with another region. If zero subgraphs are generated, all the connection points of the star region thus in effect connect the two regions on either side of the ends of the star region.The star region can contain any number of nodes and edges which are instantiated in each generated subgraph.

Figure 1 shows the generation of patterns based on a pattern with a star region in detail. In the top part of the figure an example pattern G with two isomorphic regions R_1 and R_3 and a star region R_2 is shown. R_2 consists of two interconnected nodes, *2'* and *3'*. There are also two directed edges with label m, one incoming edge from node *1'* in R_1 to *2'* in R_2 and one outgoing edge from node *3'* in R_2 to *4'* in R_3. The bottom part of the figure shows three different patterns that can be generated based on G. The generation of G_1 is done by applying a production that replaces R_2 with the empty graph, and creates a new edge m from *1'* to *4'*. The pattern G_2 is

retrieved by replacing the previously rewritten edge m in G_1 with an instance of the star region R_2 and the edges in the connection points are rewritten. Similarly, the pattern G_3 is retrieved by again replacing one of the rewritten edges with a new instance of the star region. To make the figures clearer, all rewritten edges are drawn with a wider stroke. These patterns can now be used to find a mapping to a target graph.

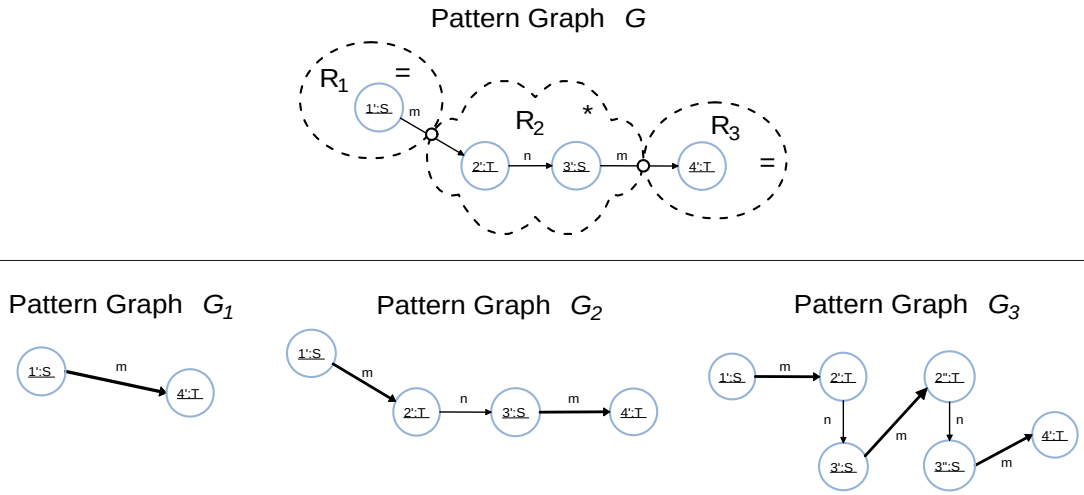


Figure 1: (Top) A pattern graph formed by two isomorphic regions, R_1 and R_3, and one star region R_2. (Bottom) Three possible patterns that could be generated from pattern G in the top of the figure.

Using this approach, it is possible to use a single pattern to describe recursive and hierarchical structures by generating a set of patterns that can be compared to a target graph using subgraph isomorphism.

A star region can be seen as an extension of the concept of *multi-objects*, or *set nodes* as defined in PROGRES [SWZ99]. While a multi-object can express multiple instances of a single node, the star region can express multiple instances of a subgraph. We have extended the concept of multi-objects by defining star regions in the query graph, where all connections of the nodes within or at the border of the region are explicit. This extension is also partly due to the fact that a multi-object can have an edge to another multi-object, but it is unclear whether the edge represents a single edge or multiple edges. Other related approaches are the works of graph transformations with variables presented in [MHar, HJE06, Hof05]. Karsai and Agrawal present in [KA03] an approach that allows cardinalities in individual nodes, but it is unclear whether this approach supports whole regions.

2.4 Examples

In this section we present some examples that illustrate how the CQuery language can be used to define queries to match common model structures in UML. We have chosen to display both the patterns and matching model fragments using the abstract syntax, which is an object graph

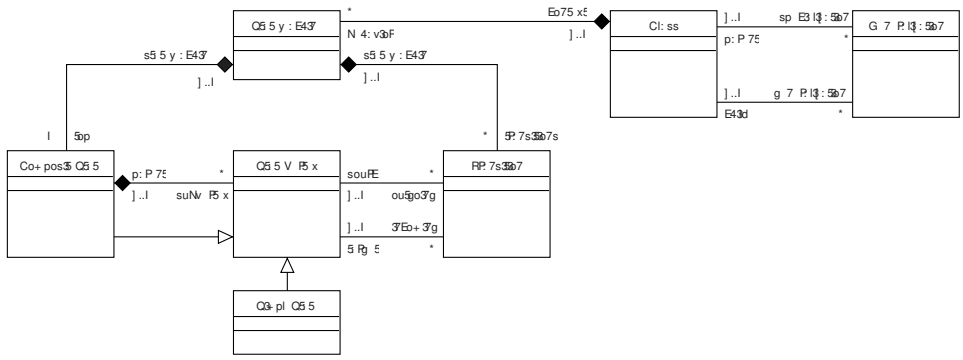


Figure 2: A simplified fragment of the UML 1.4 metamodel.

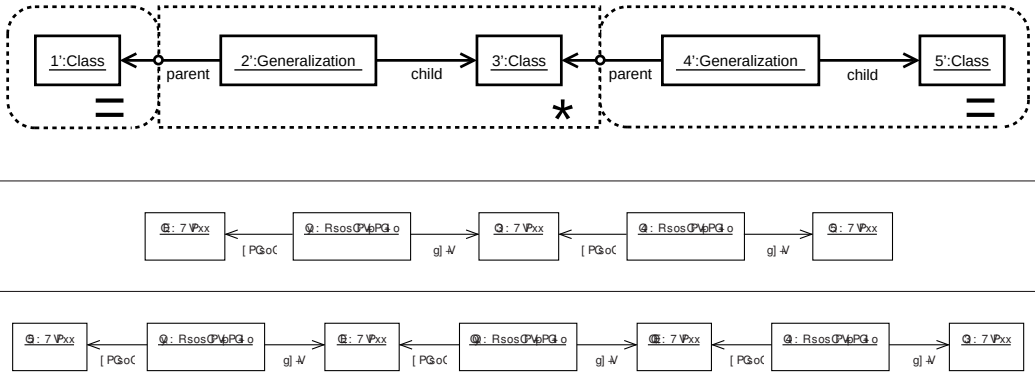


Figure 3: (Top) An example of a query with a star region defined over a class and a generalization and the *parent* relation. (Center, Bottom) Two model fragments that matches the query defined on the top, shown as object diagrams. The mapping is indicated with the corresponding numbers.

syntax similar to UML object diagrams, rather than the concrete syntax of the target modeling language, since the concrete syntax hides information about relations between the objects. In a tool environment, however, creating the queries using the concrete syntax of the modeling language can be beneficial.

In the examples we will use a slightly simplified version of the UML 1.4 metamodel, which is shown in Figure 2.

2.4.1 UML Generalizations

A sample query with two isomorphic regions and a star region is illustrated in the left part of Figure 3. The star region is marked with a dashed rectangle with the '*' symbol, and the isomorphic regions with rounded rectangles and a '=' symbol. The connection points for the star region are marked with circles at the border of region. The star region contains a UML Generalization *2'* and a Class *3'*. The Generalization is linked to the superclass *1'* and the subclass *3'* via a *parent* and a *child* relationship, respectively. These relations connect the star region to

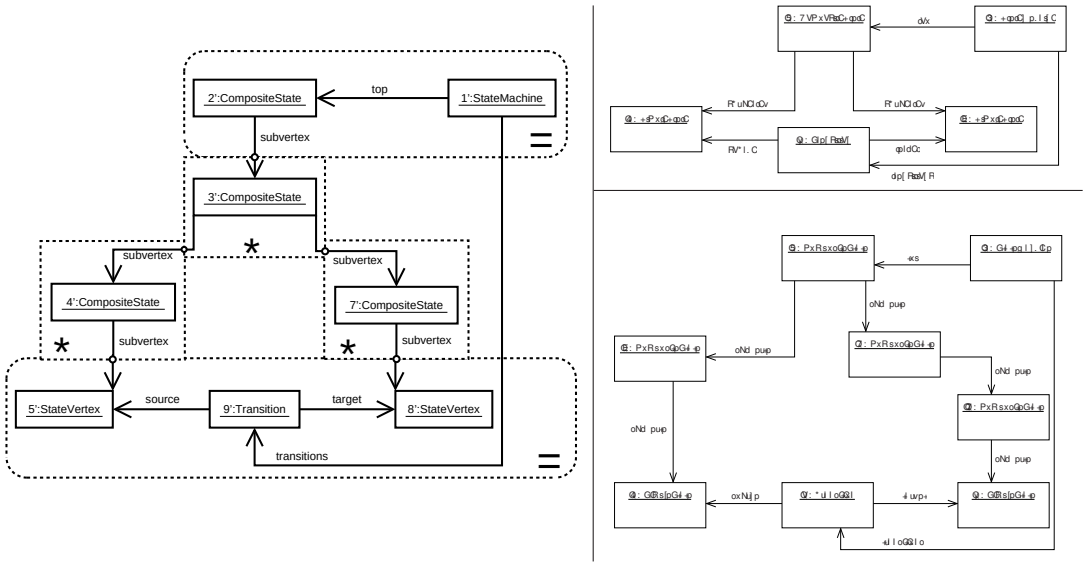


Figure 4: (Left) An example of a pattern that illustrates how transitions are connected to states and state machines. (Right) Two model fragments that matches the query.

the adjacent isomorphic regions. On the right hand side of the figure two different UML model fragments in object diagram syntax are shown that are matches of the query on the left hand side. Here, the mappings between the pattern and the target models are shown using corresponding object names, i.e., *1'* in the pattern corresponds to *'1* in the target. The Generalization *2'* and the Class *3'* in the star region were matched once in the first model fragment, and twice in the second model fragment. The pattern can be matched to targets where the star region is mapped to the empty set. This case is not illustrated in the figure. However, that particular case would imply that Class *'1* has exactly one subclass *'5*.

2.4.2 UML StateMachines

The second example in Figure 4 shows a pattern that can be used to query a UML model for a state machine with a transition between two states, where the states are transitively owned by any number of composite states. The state machine owns a composite state in *StateMachine.top* that can transitively own other states in the *CompositeState.subvertex* slot. Transitions, however, are always owned by the state machine, and have associations to two states in *Transition.source* and *Transition.target*.

The pattern described here is rather complex, as we can identify three different star regions. Each of the three star regions consists of one composite state and is connected to the other regions using the *CompositeState.subvertex* relations to the other regions. This pattern describes that the two states (*5'* and *8'*) that connect to the transition (*9'* in the figure), can be nested in an arbitrary number of common container composite states (star region with composite state *3'*). Additionally, each of the states can independently be contained by any number of composite

states (*4'* and *7'*). It must be noted, however, that there are three connection points in the star region with composite state *3'* (one incoming and one outgoing edge). This is possible, since a composite state can have any number of subvertices. When this star region is expanded to two or more isomorphic regions, only one of these connection points is used to connect the next occurence, as described in Subsection 2.3.

The right side of Figure 4 shows two model fragments that could be matched with patterns generated from the pattern on the left. Due to the fact that each star region can individually be expanded, it is possible to model all these different compositions for a state machine with a transition in one single pattern. This query can be seen as a validation that a transition has been inserted correctly in a statechart. Although the structure of state machines have changed remarkably in UML 2.0, a relatively similar pattern with a larger amount of elements are required for the UML 2.0 counterpart.

3 Matching Algorithm

In this section we will present a matching algorithm for patterns with isomorphic, star and negative regions.

We have discussed an intuitive interpretation of the query language where star regions are expanded into regular graphs. In practice, the actual patterns are not expanded prior to matching since an arbitrary number of possible patterns should be generated. Instead, the star regions are expanded during pattern matching, and only as far as valid mappings against the target graph are found.

The algorithm presented below is used to match the pattern against the target graph and expand the star regions. To match individual regions, any traditional graph matching algorithm may be used; we have used an algorithm based on CSP [Tsa93] and VF2 [CFSV01, CFSV04], as presented in [Lil06].

The result of the matching algorithm is a set where each element is a mapping from the pattern graph to the target graph. In every such mapping, each node in an isomorphic region in the pattern is mapped exactly once, each node in a negative region exactly 0 times and each node in a star region 0..n times. A node in the target graph can be mapped only once in each mapping.

The algorithm is split into two functions—*query* and *matchRegion*. *Query* initializes the matching by selecting the region to start from, invokes the recursive *matchRegion* and lastly discards any results where negative regions are successfully matched. Generally, the fewer mappings we find for the first isomorphic region matched, the faster the algorithm will work. Therefore, we generally start from the largest isomorphic region in the pattern as we are likely to find relatively few mappings for that region.

```
1 query (pattern, target):
2       r ← choose one isomorphic region in pattern
3       mappings ← matchRegion (r, {}, target, {})
4       for each negative region in pattern:
5           c ← a connection from a non-negative region to negative region
6           discard each mapping in mappings for which matchRegion (negative region, mapping, target, c) returns results
7       return mappings
```

The function *matchRegion* recursively traverses the regions in the pattern, attempting to expand the mappings found until all regions have been matched. When this function is called, we either have the situation where no mappings have been passed, or where one or more neighbors of the passed region have been matched in the inputMapping. In the first case (lines 2–3), the function starts by finding all valid mappings for the passed region, in the second case (lines 5–18), it identifies a set of candidate mappings for the partial pattern consisting of all previously matched regions and the passed region, i.e. a set of mappings where the most recently matched connection point of the passed region is satisfied (lines 5–11). The matching is done recursively for star regions, implementing the pattern generation described in Subsection 2.3 (lines 12-16).

The function then checks that all other connection points to previously matched regions are satisfied, thereby ensuring that the mapping is valid, i.e. that the topology of the candidate mapping is consistent with that of the pattern (lines 17–18). At this stage we have identified all valid mappings for the partial pattern matched so far and continue with the next region in lines 19–20.

A note on connection points: In this algorithm, we assume that each connection point consists of two nodes in separate regions that are connected through an edge. A connection point is satisfied by a mapping where the two nodes are mapped to nodes in the target graph that are likewise connected. There is an implicit connection point between the two ends of a star region which is dealt with on line 13 below. The connection point between the region to match and the previously matched region is passed on to *matchRegion* as a parameter in order to identify a starting node for matching.

```
 1 matchRegion (region, inputMapping, target, connection):
 2     if inputMapping is empty:
 3         Mappings ← all valid mappings region → target
 4     else:
 5         Mappings ← {}
 6         starting node ← the node in region connected through connection
 7         find all mappings starting node → target node satisfying connection
 8         for each target node in these mappings:
 9             start with inputMapping plus a mapping starting node → target node
10             from there, find all valid mappings region → target
11             add these mappings to Mappings
12         if region is a star region:
13             c ← connection to next instance of star region to be mapped
14             for each Mapping in Mappings:
15                 replace Mapping with M ← matchRegion (region, Mapping, target, c)
16             add inputMapping to Mappings
17         for each matched neighbor of region:
18             discard all Mappings where a connection between region and neighbor is not satisfied
19     for each connection c to a non-negative, unmatched neighbor of region:
20         replace each Mapping in Mappings with M ← matchRegion (neighbor, Mapping, target, c)
21     return Mappings
```

4 Validation and Applications

We have built an experimental modeling tool called Coral [AP04]. In this tool we have implemented CQuery and a matching engine that supports the concepts we have discussed in this paper.

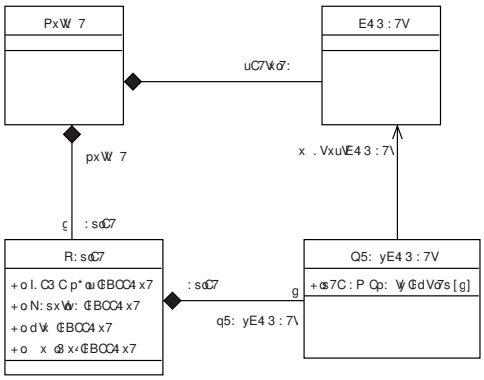


Figure 5: The CQuery Metamodel

4.1 Validation

The main idea in the design of the CQuery language is that the base of the pattern is a model in the target language. The CQuery language consists of elements that extend a modeling language to include information to control a query. That is, the pattern consists of a model fragment in the target language, annotated with query configuration in the CQuery language. Since the CQuery language itself is separated from the modeling language of the target, the target language does not have to be modified to support CQuery.

We have chosen this approach for two reasons: First, there is no need to have a separate component that verifies that the patterns are possible to construct using the target metamodel, since adherence to the metamodel is implicit. Second, we believe that the creation of patterns is easier, since a significant part can be constructed as any other model in the target modeling language. However, this approach does not prevent the queries being presented in any particular syntax, including the concrete syntax of the target modeling language or a more general object diagram syntax. A discussion on using the concrete syntax of a modeling language in model transformation rules can be found in [BW06].

The CQuery metamodel is shown in Figure 5. The metamodel is rather small, containing only 3 metaclasses, where the *Element* can point to any abstract model element, and hence is not directly a part of the CQuery language. The base element is Pattern, which acts as the starting point of a query. Each *Pattern* consists of a set of *Regions* and an abstract *container* element which is an element in any modeling language. This element owns all model elements in the pattern which are not annotations of CQuery. A Region is either an isomorphic, a negative or a star region. This is indicated by the corresponding flags. However, only one of these flags can be set for a particular region in a pattern. A *Region* consists of an arbitrary number of *QueryElements*. The *QueryElement* contains information to control which attributes and outgoing edges should be ignored when matching a single element. In a well-formed pattern, all abstract elements have a corresponding *QueryElement*, and all *QueryElements* are owned by a *Region*.

The version of CQuery implemented in this tool is slightly different, but shares the same features that have been discussed in this paper. In our tool it is possible to create a query using the concrete syntax of the target modeling language. If the target language does not have a con-

crete syntax, it is still possible to create queries, but without the benefit of having diagrams. The matching engine in CQuery is based on the algorithm described in Section 3 and [Lil06]. The algorithm is based on the VF2 and CSP algorithms and facilitates search planning and backtracking.

All star regions can optionally set an *isMaximal* flag. This flag can be used to indicate whether the matching engine should attempt to expand a star region a maximal number of times, instead of attempting to match an adjacent region to a subgraph that could actually be seen as a match to the star region. This feature can be very useful since an application that uses CQuery does not need to evaluate all possible matches if the point of interest is only the maximal possible matches of the star region. It must, however, be noted that although the *isMaximal* flag is set, this does not rule out the possibility that a star region could match a target graph where the star region had no occurrences.

4.2 Applications

The CQuery implementation is used by a variety of components and add-ons in the Coral tool. The most straightforward application of CQuery is a model search facility. In this component it is possible to load a set of query patterns defined in the Coral tool and search for occurrences of a pattern in open modeling projects. The results of the CQuery based search are reported as a set of mappings between elements in the query pattern and the target model.

We have also implemented a constraint evaluation component based on CQuery. This component is an integral part of the Coral tool, and uses a set of CQuery patterns to detect if modeling language constraints or *well-formedness rules* have been violated. This component is based on an approach where user models are continuously checked for errors. If an error is detected, the offending elements are reported along with an explanation, or a suggestion for correcting the problem. An example of how this constraint evaluation component has been used in a domain-specific language for System-on-Chip design called MICAS, can be found in [LLL+05].

Another application is a generic model to text transformation engine [Nym06], which uses the CQuery language as the query facility. This application can e.g. be used for generating source code or documentation based on UML models.

Perhaps the most ambitious use of CQuery is a transformation engine based on the *double pushout approach* [Roz97]. The transformation rules are given as a pair of a left-hand side (LHS) and a right-hand side (RHS), and an explicit mapping between the LHS and RHS. This transformation engine uses CQuery for matching the LHS to an occurrence in a model, and to specify the RHS. The transformation engine has support for negative, isomorphic regions and star regions, and provides in-place transformation of models. The transformation engine is extensively used in the Coral tool for defining the rules for editing models, e.g. inserting states or transitions in a statechart, or classes and associations in class diagrams. We have found that especially the star regions are necessary when defining model editing transformations in UML, where complex hierarchies of model elements occur frequently. Using the star region, we have been able to reduce the number of model transformation rules to define the editor.

The Coral tool, including CQuery and all components mentioned in this section are open source and are available for download from `http://mde.abo.fi/`.

5 Conclusions and Future Work

We have presented a query language for model-driven development applications that introduces the concept of star regions to represent hierarchical and repetitive structures. This query language has been implemented in a modeling tool and used successfully in different applications based on UML and other domain-specific modeling languages.

There are two clear future directions. First, introduce new region operators, such as cardinality or disjunction. However, the need for these new operators should arise from actual modeling tools. Also, we are studying the application of our query language to model transformations. In fact, a model transformation tool component based on CQuery has already been implemented and we plan to present these results in the near future.

Bibliography

[AP04] M. Alanen, I. Porres. The Coral Modelling Framework. In Kai Koskimies and Porres (eds.), *Proceedings of the 2nd Nordic Workshop on the Unified Modeling Language NWUML 2004*. TUCS General Publications 35. TUCS Turku Centre for Computer Science, Jul 2004.

[ARS05] C. Amelunxen, T. Rötschke, A. Schürr. Graph Transformations with MOF 2.0. In Giese and Zündorf (eds.), *Fujaba Days 2005*. September 2005.

[BH02] L. Baresi, R. Heckel. Tutorial Introduction to Graph Transformation: A Software Engineering Perspective. In Corradini et al. (eds.), *Proc. Graph Transformation - First International Conf., ICGT 2002, Barcelona, Spain*. LNCS 2505. Springer, 2002.

[BW06] T. Baar, J. Whittle. On the Usage of Concrete Syntax in Model Transformation Rules. Technical report LGL-REPORT-2006-002, 2006.

[CFSV01] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. An improved algorithm for matching large graphs. In *Proceedings of the 3rd IAPR-TC-15 International Workshop on Graph-based Representations. Italy*. Pp. 149–159. 2001.

[CFSV04] L. P. Cordella, P. Foggia, C. Sansone, M. Vento. A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs. *IEEE Trans. Pattern Anal. Mach. Intell.* 26(10):1367–1372, 2004.

[HHT96] A. Habel, R. Heckel, G. Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae* 26(3-4):287–313, 1996.

[HJE06] B. Hoffmann, D. Janssens, N. V. Eetvelde. Cloning and Expanding Graph Transformation Rules for Refactoring. *Electr. Notes Theor. Comput. Sci.* 152:53–67, 2006.

[Hof05] B. Hoffmann. Graph Transformation with Variables. In *Formal Methods in Software and Systems Modeling*. Pp. 101–115. 2005.

[KA03] G. Karsai, A. Agrawal. Graph Transformations in OMG's Model-Driven Architecture: (Invited Talk). In Pfaltz et al. (eds.), *AGTIVE*. Lecture Notes in Computer Science 3062, pp. 243–259. Springer, 2003.

[Koz91] D. Kozen. A Completeness Theorem for Kleene Algebras and the Algebra of Regular Events. In *Logic in Computer Science*. Pp. 214–225. 1991.

[Lil06] T. Lillqvist. Subgraph Matching in Model Driven Engineering. Master's Thesis in Computer Science, Department of Information Technologies, Åbo Akademi University, Turku, Finland, March 2006.

[LLL+05] J. Lilius, T. Lillqvist, T. Lundkvist, I. Oliver, I. Porres, K. Sandström, G. Sveholm, A. Pervez Zaka. An Architecture Exploration Environment for System on Chip Design. *Nordic Journal of Computing* 12(4):361–378, 2005.

[MHar] M. Minas, B. Hoffmann. An Example of Cloning Graph Transformation Rules for Programming. *Electronic Notes in Theoretical Computer Science*, To appear.

[Nym06] M. Nyman. A Model-Based Approach to Text Generation from Software Models. Master's Thesis in Computer Science, Department of Information Technologies, Åbo Akademi University, Turku, Finland, May 2006.

[OMG03] OMG. UML 2.0 OCL Specification. October 2003. Document ptc/03-10-14, available at http://www.omg.org/.

[OMG05a] OMG. MOF 2.0 Query / View / Transformation Final Adopted Specification. November 2005. OMG Document ptc/05-11-01, available at http://www.omg.org/.

[OMG05b] OMG. UML 2.0 Superstructure Specification. August 2005. Document formal/05-07-04. Available at http://www.omg.org/.

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

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. *Handbook of Graph Grammars and Computing by Graph Transformation: Vol. 2: Applications, Languages, and Tools*, pp. 487–550, 1999.

[Tsa93] E. Tsang. *Foundations of Constraint Satisfaction*. Academic Press, London and San Diego, 1993.

[VVP02] D. Varró, G. Varró, A. Pataricza. Designing the Automatic Transformation of Visual Languages. *Science of Computer Programming* 44(2):205–227, August 2002. http://www.inf.mit.bme.hu/FTSRG/Publications/varro/2002/scp2002_vvp.pdf

Triple Patterns: Compact Specifications for the Generation of Operational Triple Graph Grammar Rules

Juan de Lara[1], **Esther Guerra**[2], **Paolo Bottoni**[3]

[1] jdelara@uam.es
Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)
[2] eguerra@inf.uc3m.es
Dep. Ingeniería Informática
Universidad Carlos III de Madrid (Spain)
[3] bottoni@di.uniroma1.it
Dip. Informatica
Università di Roma La Sapienza (Italy)

Abstract: Triple Graph Grammars (TGGs) allow the specification of high-level rules modelling the synchronized creation of elements in two graphs related through a correspondence graph. Low-level *operational* rules are then derived to manipulate concrete graphs. However, TGG rules may become unnecessarily verbose when elements have to be replicated from one graph to the other, and their actual derivation cannot exploit the presence of reoccurring patterns. Moreover they do not take advantage from situations in which a normal creation grammar for one of the graphs exists, from which TGG operational rules can be derived to build the other graph.

We present an approach to generating TGG operational rules from normal ones, reducing the information needed to derive them, through the definition of *Triple Patterns*, a high-level, compact, declarative, and visual notation for the description of admissible structures in a triple graph. Patterns can be expressed with respect to classes defined in a meta-model, and instantiated with derived classes at the model level, thus exploiting the inheritance hierarchies. The application of the generated rules results into the (synchronized or batch) creation of the structures specified in the patterns. We illustrate these concepts by showing their application to the synchronized incremental construction of visual models and of their semantics.

Keywords: Graph Transformation, Triple Graph Grammars, Visual Languages.

1 Introduction

Model transformation is becoming increasingly popular with the advent of model-driven development technologies, such as MDA [MSUW04], where model-to-model transformation plays a central role. In such transformations, an input model M_A conforming to a meta-model MM_A is transformed into an output model M_B conforming to a (possibly different) meta-model MM_B. Several scenarios are of interest here. For example, in a syntax directed visual modelling tool with separate models for concrete syntax and for semantic interpretation (which contains the

relevant semantic roles, see [BDD+04]), one would like to model the synchronized evolution of both models (although a batch update of the semantic model could also make sense). For tool integration applications, a (bi-)directional – batch or incremental – transformation is desirable [Sch94]. Finally, one could be interested in checking the consistency of two given models.

Triple Graph Grammars (TGGs) [Sch94] were proposed by Schürr as a means to model the transformation of two graphs (source and target) related through a correspondence graph (whose nodes have morphisms to elements in the other two graphs). The main idea is to model the synchronized evolution of the two graphs, as well as the correspondence graph relating both, by means of triple rules. From these *creation* triple rules, algorithms were given to produce *operational* rules to perform a translation in either direction (from source to target or vice versa), to create the correspondence graph given two already existing source and target graphs, or to check the validity of the correspondence graph.

In the aforementioned scenario of a syntax directed visual modelling tool, the use of TGGs may be too cumbersome. In these environments, one models by means of rules the possible user editing actions. This brings advantages in cases when one has to model complex editing actions, where many elements are created in the concrete syntax, but requires the designer to define complex TGG creation rules as well. It is however possible to identify patterns for such situations, whereby certain elements in the concrete syntax always play the same role in the semantic model. Therefore, we propose an approach in which the designer has to provide the creation grammar for the concrete syntax only, and some triple patterns specifying admissible relations between concrete syntax elements and semantic roles. We have defined a collection of algorithms which exploit these patterns to produce sets of TGG operational rules that either do a batch translation from concrete syntax to the semantic model, or produce the synchronized evolution of both. This reduces the amount of information that the designer has to input, since many triple patterns can be "applied" to a normal graph transformation rule.

The approach we present is suitable for integration in meta-modelling environments, as the algorithms explicitly take into account the inheritance hierarchies of the meta-models. Although the presented examples are taken from the Visual Languages area, these ideas are readily applicable to general model-to-model transformations.

Paper Organization. Section 2 introduces TGGs, and some of the extensions we have provided to the underlying graph model [GL07]. Section 3 presents *triple patterns* and the algorithms for generating operational triple rules. In Section 4, we take into account the inheritance hierarchy of the meta-model, presenting the concept of *abstract triple patterns*. Section 5 compares with related research, and Section 6 discusses conclusions and future work.

2 Triple Graph Grammars

Triple graphs are made of three graphs: source, target and correspondence ones. Correspondence graph nodes are used to relate elements in source and target graphs. Triple graphs are depicted as $p : p_{src} \xleftarrow{ps} p_{corr} \xrightarrow{pt} p_{tar}$, where ps and pt are morphisms from the nodes in graph p_{corr} to nodes in the source and target graphs. The structure of each graph p_X (for $X \in \{src, corr, tar\}$) is given by $p_X = (V_X, E_X, src_X : E_X \to V_X, tar_X : E_X \to V_X)$, where V_X is the set of vertices, E_X is the set of edges, and src_X and tar_X are functions defining the source and target nodes of every

edge $e \in E_X$. Labels for nodes and edges can also be given.

In [GL07], we extended the underlying triple graph structure originally proposed in [Sch94] with attributes for nodes and edges, and a typing by a type triple graph (or meta-model triple) which may contain inheritance relations between nodes or edges. Moreover, we made the relation between the source and target graphs more flexible, by allowing *partial* morphisms from nodes in the correspondence graph to nodes and edges in the other two graphs.

Figure 1(a) shows an example meta-model triple taken from the area of visual modelling languages. The lower part (source graph) contains a simplified meta-model with the base classes for the concrete syntax of a visual language [BG04]. Briefly, in a diagrammatic language, significant *spatial relations* exist among *identifiable elements*. The latter are recognizable entities in the language, to which a semantic role can be associated, and which are univocally materialized by means of a complex graphic element. Each such element is composed in turn of simpler graphic elements, each possessing one or more attach zones, which define its availability to participating in different spatial relations, such as containment or touching.

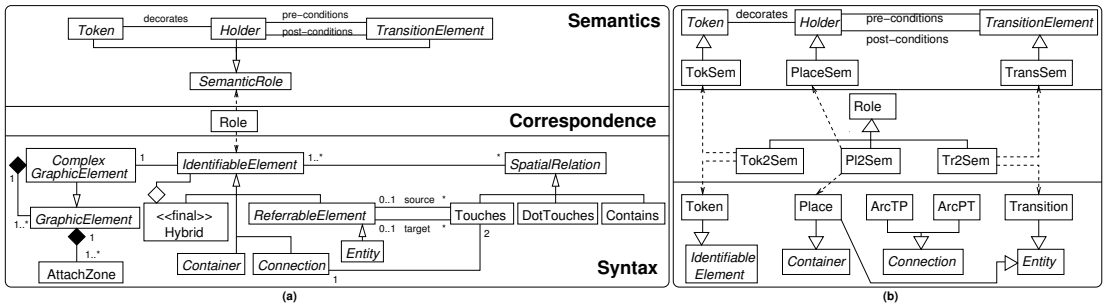


Figure 1: (a) Meta-Model Triple for the Syntax and Semantics of Visual Languages (b) Specialized Meta-Model for Petri Nets.

The upper part (target graph) contains a meta-model that describes the possible abstract roles for a transition-based (i.e. token-holder) semantics (i.e. semantics in the style of Petri nets, UML 2.0 activity diagrams and automata). The correspondence graph assigns semantic roles to syntactic elements. When a meta-model for a new visual language is defined, the newly defined concrete syntax concepts inherit from the classes in the syntax meta-model. If the language has a transition-based semantics, then the designer can create concrete roles by subclassing the classes in the semantics meta-model. Thus, predefined, customizable model transformation libraries implementing the operational semantics can be reused for the new language. Figure 1(b) shows the definition of the syntactic and semantic roles for Petri nets (we have omitted arc weights for simplicity of presentation). The significant spatial relations are refined (by means of a creation graph grammar, with some rules shown in Figures 2(a) and 3) to be the *Touches* relation between instances of *ArcPT* (*ArcTP*) and a source *Place* (*Transition*) or a target *Transition* (*Place*), and the *Contains* relation, between instances of *Place* and *Token*. Note that a *Place* can play both the role of an *Entity*, in relation to the arcs which refer to it, and that of a *Container*, in relation to the *Tokens* it holds.

TGG rules model the transformation of triple graphs. In [GL07] we adapted TGG rules to the Double Pushout approach (DPO) [EEPT06], in which rules are modelled using three com-

ponents, L, K and R, where: L (the left hand side, LHS) contains the elements to be found in the host graph where the rule is applied; K contains the elements preserved by the rule application; and R (the right hand side, RHS) contains the elements that should replace the part identified by L in the host graph. The DPO approach has been lifted to work with any (weak) adhesive HLR category [EEPT06] (such as those for graphs, Petri nets, etc.). In [GL07] we showed that the category $\mathbf{TriAGraph}_{\mathbf{TriATG}}$ of attributed typed triple graphs (short triple graphs) and morphisms is an adhesive HLR category. Therefore, in our case, L, K and R are triple graphs.

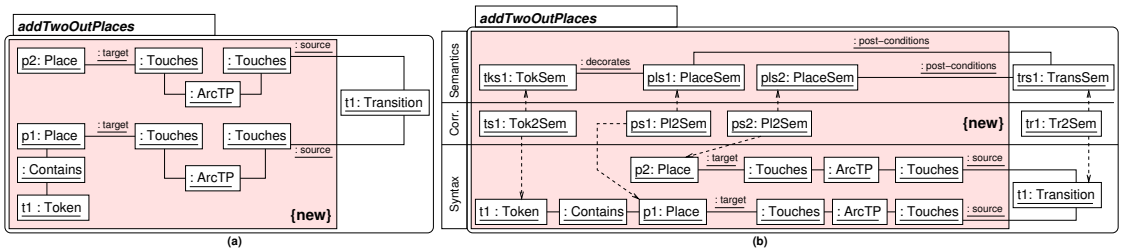


Figure 2: (a) Rule Modelling an Editing Action. (b) Triple Rule Modelling the Synchronized Creation of Semantic Roles.

The motivation for this work is the following. Given a normal graph grammar modelling the possible editing actions in a modelling environment (i.e. working in the concrete syntax only), how can we obtain triple rules that update or build synchronously the semantic model? As an example, Figure 2(a) shows a rule modelling a complex editing action by which, given an existing transition, two places are created, connected with arcs from the transition to the places, and a token is inserted into one of the places. Figure 2(b) illustrates the desired corresponding triple rule that synchronously creates the semantic elements together with the syntactic ones, designating the created places as post-conditions for the transition. We could build by hand a TGG rule for each single syntax editing rule. However, this task is repetitive, as elements in the concrete syntax are related in the same way to elements in the semantic model (as specified in the meta-model triple), i.e. a reoccurring pattern can be identified in the triple rules.

3 Triple Patterns

In this section we present the concept of *Triple Pattern*, together with an algorithm that, given a rule (like the one in Figure 2(a)) and a set of patterns, generates an operational TGG rule (like the one in Figure 2(b)). For simplicity of presentation, we assume the simple graph structure mentioned in the first paragraph of Section 2 (i.e. untyped graphs, with nodes in the correspondence graph having two morphisms: one to a node in the target and one to a node in the source graph, like in [Sch94]). The adaptation of the algorithm to more complex graph models is straightforward. Assuming that the input rule acts on the source graph only, the algorithm generates a TGG rule that synchronously creates the necessary elements in the target graph. Symmetrically, the input rule could act on the target graph, and the generated TGG rule would complete the source graph. Moreover, as in [Sch94], it is also easy to generate slightly different TGG rules: batch rules (i.e. rules assuming that the source elements are already created, and which then create

the target graph elements), rules for creating the correspondence graph, assuming that the source and target graphs are created, and rules for checking the validity of the correspondence graph.

A triple pattern $p : p_{src} \xleftarrow{ps} p_{corr} \xrightarrow{pt} p_{tar}$ is a triple graph conformant to a meta-model triple. Formally, given a triple pattern p and a triple graph G, we say that G satisfies p (written $G \models p$) if an injective triple graph morphism $m : p \to G$ exists.

Example. We first start by giving an intuition of the algorithm through an example. In this paper, we use triple patterns in order to specify in a visual, high level, acausal notation the kind of configurations we want to find in our semantic models when certain syntactic configurations are met (or the other way round). Thus, our triple patterns are triple graphs conforming to the meta-model of Figure 1. Figure 3 shows a triple pattern depicting the needed structure in the syntactic model for a holder to have a token in the semantic model. In this case, a *Place* in the syntactic model has an associated *PlaceSem* role (a subclass of *Holder*) in the semantic model. Similarly, a *Token* in the syntactic model has a *TokSem* role in the semantic model (a subclass of class *Token* of the semantic meta-model). In the semantic model, a token *decorates* a holder, while at the syntactic level the place *contains* the token.

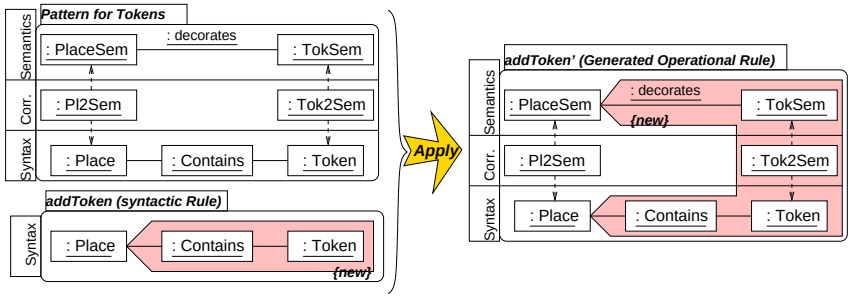


Figure 3: Applying a Pattern to a Rule.

Figure 3 also shows a syntactic editing rule ("addToken") modelling the creation of a token inside a place in the syntactic model. The objective of the algorithm ("Apply") is to obtain the triple rule shown in the figure, where information about the actions to be done at the semantic level has been incorporated, together with the mapping between the syntactic and semantic models. Roughly, we first try to find a match from the pattern to the rule's RHS. Then we glue the pattern and the RHS of the syntactic rule through the matching, to obtain the triple rule's RHS. Finally, we construct the triple rule's LHS by taking the elements in the correspondence and semantic graphs (of the RHS) which are related to elements which were already present in the syntactic rule's LHS.

The following algorithm describes the application of a set of patterns to a non-deleting normal rule, resulting in one triple rule. Later, we show how the algorithm can be easily modified for its application to deleting (and non-creating) rules.

`Apply(P: SetOfTriplePatterns, rl: Rule): TripleRule`

Let $P = \{p^i\}_{i \in I}$ be a set of triple patterns of the form $p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar}$ and rl a non-deleting normal rule $rl : L \xleftarrow{l} K \xrightarrow{r} R$ with $L = K$, and which therefore can be written as $rl : L \xrightarrow{r} R$. The application of P to rule rl results in a triple rule rl' as follows:

1. Initialize the triple rule rl', copying rule rl in the source part of rl'. The resulting triple rule is written as $rl' : L_{rl'} \xrightarrow{r'} R_{rl'}$, where r' is a triple graph morphism (see Figure 4).

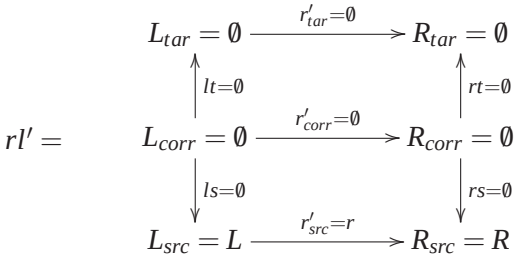


Figure 4: Initialization of Triple Rule rl'.

2. $\forall p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar} \in P$:

   (a) $\forall p^i_{src} \xrightarrow{m^i_j} R_{src}$, with m^i_j an injective match from the source part of p^i (i.e. p^i_{src}) to the source part of $R_{rl'}$ (i.e. R_{src}):

      i. **if** $\exists p^i_{src} \xrightarrow{m'^i_j} L_{src}$ with $m^i_j = r'_{src} \circ m'^i_j$ **then** do nothing (as no elements in the source part of the rule have been newly created for this match) **else**

      ii. $\forall O : (O_{src} = p^i_{src}) \xleftarrow{os} O_{corr} \xrightarrow{ot} O_{tar}$ such that the diagram of Figure 5(a) commutes, and that $\nexists O'_x | O_x \subset O'_x \subseteq p^i_x, x \in \{corr, tar\}$:

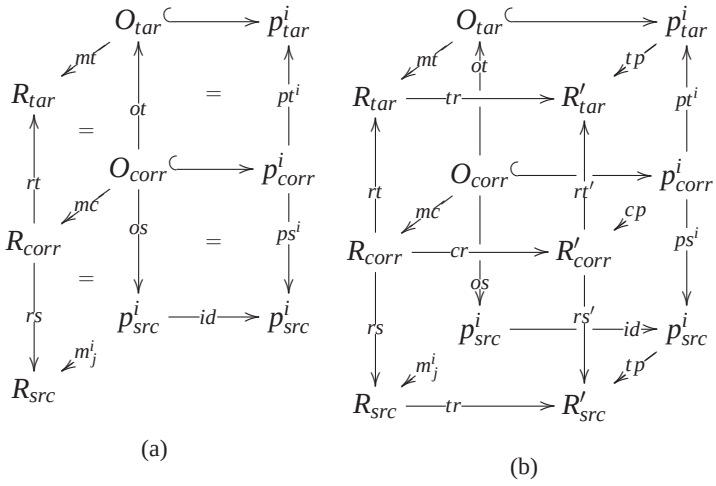


Figure 5: (a) Glueing p^i with the Right Hand Side of rl'. (b) Building the Pushout.

A. replace $R_{rl'} : R_{src} \xleftarrow{rs} R_{corr} \xrightarrow{rt} R_{tar}$ with the pushout object of the previ-

ous diagram, i.e. $R_{rl'} = PushOut(R_{rl'}, O, p^i)$ [1]. The pushout is shown in Figure 5(b).

B. **Add appropriate nodes to L**

[1] $\forall n \in V_{R_{src}}$ // Check if we have to copy the correspondence node to L
[2] $New_{tar} = New_{corr} = New_{unconn} = \emptyset$ // Sets with newly added nodes to L
[3] **if** ($\exists n' \in V_{L_{src}}$ s.t. $r'_{src}(n') = n$) **then** // n is also in L
[4] // seek correspondence nodes which are not in L
[5] $\forall c \in V_{R_{corr}}$ s.t. $rs(c) = n \wedge \nexists c'' \in V_{L_{corr}}$ s.t. $r'_{corr}(c'') = c$
[6] // add correspondence node to L
[7] $V_{L_{corr}} = V_{L_{corr}} \uplus \{c'\}$ and set $ls(c') = n'$, $r'_{corr}(c') = c$
[8] $New_{corr} = New_{corr} \uplus \{c'\}$ // add it to the set
[9] // check for nodes in the target graph of R which are not in L
[10] **if** ($\exists n'' \in V_{R_{tar}} | rt(c) = n'' \wedge \nexists o \in V_{L_{tar}} | r'_{tar}(o) = n''$) **then**
[11] // add node to target graph of L and to New_{tar} set
[12] $V_{L_{tar}} = V_{L_{tar}} \uplus \{n'''\}$ and set $lt(c') = n'''$, $r'_{tar}(n''') = n''$
[13] $New_{tar} = New_{tar} \uplus \{n'''\}$
[14] $\forall n \in New_{tar}$// Add nodes to target graph for which no correspondence exists
[15] $\forall m \in V_{R_{tar}}$ s.t. ($\nexists m' \in V_{L_{tar}}$ s.t. $r'_{tar}(m') = m \wedge \exists path_U(r'_{tar}(n), m)$)[2]
[16] $V_{L_{tar}} = V_{L_{tar}} \uplus \{m'\}$, set $r'_{tar}(m') = m$, $New_{unconn} = New_{unconn} \uplus \{m'\}$
[17] $New_{tar} = New_{tar} \uplus New_{unconn}$

C. **Add appropriate edges to L_X (for $X \in \{corr, tar\}$):**

[1] $\forall n \in New_X$ // visit all new nodes...
[2] // check if some edge stems from $r'_X(n)$ and ends in a node $\in L$
[3] **if** ($\exists e \in E_{R_X}, m' \in V_{L_X}$ s.t. $source_{R_X}(e) = r'_X(n) \wedge target_{R_X}(e) = r'_X(m') \wedge$
[4] $\nexists e' \in E_{L_X}$ s.t. $r'_X(e') = e$) **then**
[5] // Add the edge to L
[6] $E_{L_X} = E_{L_X} \uplus \{e'\}$, $r'_X(e') = e$, $source_{L_X}(e') = n$, $target_{L_X}(e') = m'$

Note that in step ii, we look for a total match from p^i_{src} to R_{src}, and partial matches from p^i_{corr} and p^i_{tar}. Thus, the triple graph O models the domain of such partial matches. With the pushout, we add the part of p^i which was not matched to the rule. In step $ii.B$ we copy the necessary correspondence nodes to the LHS, if they were added to the RHS by the pushout and the RHS node they refer to also belongs to the LHS. More than one correspondence node can be connected to a source or target graph node (line [5]). We also allow nodes in the target graph which are not connected with any correspondence graph node (added to the RHS by the pushout, and appropriately copied to LHS by lines [14-17]). These are useful if we want to model a source graph node related with many elements in the target graph, or "composite" connections in the target graph.

The application of a set of patterns to a rule acting on the target graph simply requires substi-

[1] The pushout of triple graphs [GL07, Sch94] is built component-wise, where in addition all the faces of the two cubes commute. Examples are shown in Figures 5(b) and 6

[2] Predicate $path_U(a,b)$ is true if a path from a to b exists (without taking into account the edge direction) where no node in the path receives a morphism from correspondence graph nodes, except a. Moreover, b should not be connected (directly or indirectly) with a newly created node, i.e. a node p s.t. $\nexists p' \in V_{L_{tar}}$ with $r'_{tar}(p') = p$.

tuting *src* by *tar* in the previous algorithm. It is also easy to apply patterns to deleting rules (i.e. rules which delete elements but do not create anything), by substituting *L* by *R* in the algorithm.

Example (continued). Figure 6 shows some details about the execution of steps *2.a.ii.B* and *2.a.ii.C* of the algorithm in the case of the rule and the patterns shown in Figure 3. The upper part shows how the pushout is performed, and the lower part also shows how the new elements *c'* and *n"'* are added to *L* in the generated TGG rule. Note that the *PlaceSem* object associated with the *Place* belongs to *L*, as the *Place* object also belongs to *L* (step B in the algorithm).

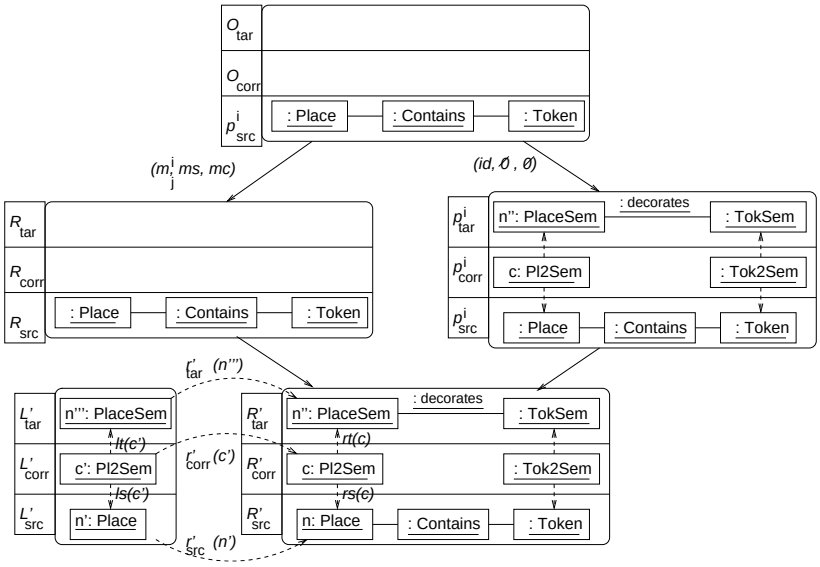


Figure 6: Steps in the Application of the Pattern in Figure 3.

Figure 7 shows additional patterns for the Petri nets example. According to the left pattern, output places of a transition in the syntactic graph are post-condition *PlaceSem* objects for the *TransSem* object associated with the transition. The pattern to the right models the correspondence for input places. By applying the three patterns to the rule in Figure 2(a) (twice the pattern for post-conditions, and once that for tokens), we obtain the operational triple rule in Figure 2(b).

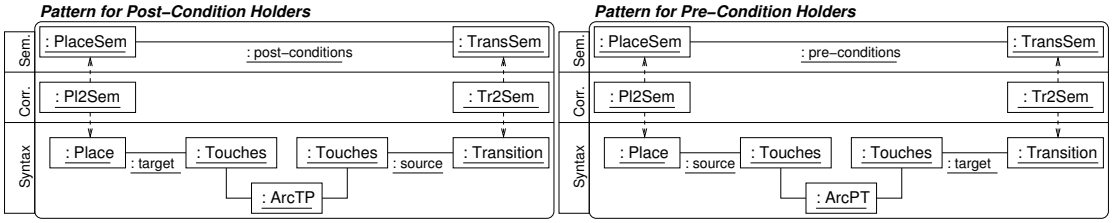


Figure 7: Additional Patterns for the Example.

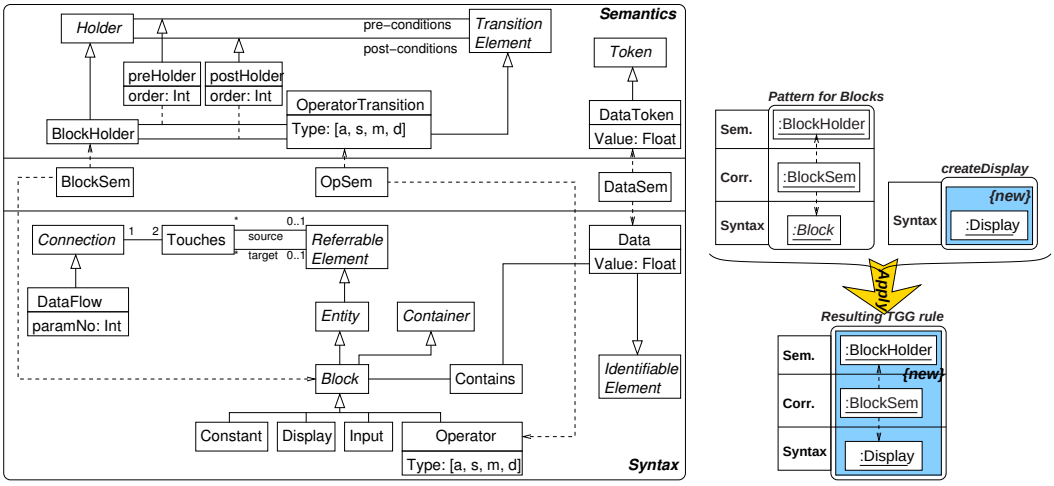
The advantage of these patterns is that they are specified once, and can then be applied to complex syntactic rules. The visual language designer does not have to modify by hand each syntactic rule to add the semantic information, but only has to specify the patterns once. More-

over, the patterns do not have to take into account which elements are created and which are already existing, as this is specified in the normal rules to which patterns are applied. Thus, the pattern may be used in several ways (i.e. in parts of the rule which are newly created or in existing ones).

4 Abstract Triple Patterns

We consider now patterns with "abstract objects" and their application to rules also containing "abstract objects" (i.e. abstract rules). When looking for a match from an abstract pattern to a rule, abstract objects in the pattern can be matched with objects of more concrete classes in the rules. An abstract rule is equivalent to the set of concrete rules resulting from the valid substitutions of the abstract objects by instances of the subclasses of the abstract object class [BELT04].

In order to illustrate these concepts, we introduce a new example, which models the concrete syntax and the semantic roles for a visual language of arithmetic expressions. The meta-model triple is shown in Figure 8(a). The language is made of blocks (abstract *Block* class), which can be interconnected through data flows (*DataFlow* class). Blocks contain data values (*Data* class), which are propagated through the data flows and processed by the blocks. There are four types of blocks: constants (i.e. blocks which store non-modifiable data values), displays (blocks that output a value), inputs (blocks which capture a value from outside the system), and operators. The latter manipulate data values, and can be adders, subtractors, multipliers and dividers. Data flows contain the argument position, which is needed for non-commutative operations.



(a) (b)

Figure 8: (a) Meta-Model Triple for an Arithmetic Expressions Language. (b) Application of an Abstract Triple Pattern.

In the semantic level, blocks are considered holders, data is considered a token (with value),

and operators are transitions. Differently from the Petri net case, operators have both the roles of transitions and holders (for the value resulting from the operation).

Before presenting the algorithm, we show the intuition using simple examples. Figure 8(b) shows an abstract pattern to the left. The pattern shows the desired relation between blocks (any kind of block, as *Block* is an abstract class) at the syntactic level and *BlockHolder* objects in the semantic graph. The syntactic rule to the right models the creation of a display object. The application of the pattern to the rule results in a triple rule where the block object in the triple pattern has been matched to a display object, as display has a more concrete type.

Consider now the situation depicted in Figure 9. The left part shows two patterns. The first one describes that a *BlockHolder* is a post-condition for an *OperatorTransition* object at the semantic level when an *Operator* object is connected through a *DataFlow* to a *Block* abstract object. This pattern is abstract, and would be equivalent to four patterns, resulting from the substitutions of the *Block* object by objects of each one of its subclasses. The second pattern associates an *OperatorTransition* object with an *Operator* object.

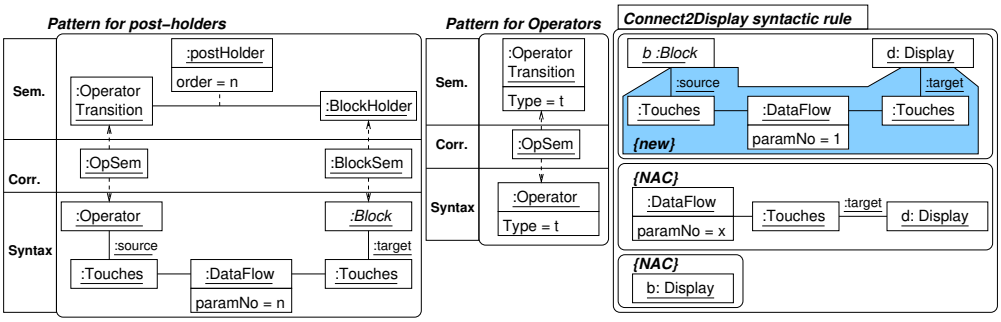


Figure 9: Abstract Triple Patterns and Abstract Rule.

The syntactic rule shown to the right models the connection of a block to a display and forbids the connection of two displays. In principle, the first pattern cannot be applied to the rule because, although the *Block* object in the pattern can get instantiated to the *Display* object in the rule, class *Operator* is more concrete than class *Block*. However, there are cases when a *Block* is an operator. Therefore what we have to do is to consider all concrete rules equivalent to the abstract one, and then apply the patterns. Here, we want to distinguish the case when an object is both a *Block* and an *Operator*, and the case where the object is a *Block* and not an *Operator*.

In order to define the construction of concrete rules from abstract patterns, we rely on the partial order $\preceq$ induced by the inheritance relationship on the set of classes in the meta-model, so that $A \preceq B$ if A inherits, even indirectly, from B, or if A is equal to B.

The following algorithm describes the previous processes.

`AbstractApply(P: SetOfTriplePatterns, rl: Rule): SetOfTripleRules`

Let $P = \{p^i\}_{i \in I}$ be a set of triple patterns of the form $p^i : p^i_{src} \xleftarrow{ps^i} p^i_{corr} \xrightarrow{pt^i} p^i_{tar}$ and $rl : L \xleftarrow{l} K \xrightarrow{r} R$ a non-deleting rule with $L = K$, and therefore $rl : L \xrightarrow{r} R$. The application of P to rl results in a set of triple rules $R'_{rl} = \{rl'_j\}$ as follows:

1. Set $R'_{rl} = \emptyset$.

2. Let $R_{rl} = \{rl_k\} \cup \{rl\}$ be the set of concrete rules equivalent to rl (see [BELT04]) and rl itself.

3. $\forall rl_k \in R_{rl}$, $R'_{rl} = R'_{rl} \cup Apply(P, rl_k)$. That is, we apply the patterns to each rule. Note that we allow a match from a pattern to a rule if a structural match is found, and if all types in the rule are more concrete or equal to the corresponding types in the abstract pattern.

4. $\forall rl' \in R'_{rl}$: if $\exists rl'' \in R'_{rl}$ s.t. rl' is more concrete than rl'' ($rl' \preceq rl''$) then $R'_{rl} = R'_{rl} \setminus \{rl'\}$. That is, we eliminate rules "subsumed" by others (same structure, equal or more concrete types).

5. $\forall rl' \in R'_{rl}$: if $\exists rl'' \in R'_{rl}$ s.t. $rl''_{src} \preceq rl'_{src}$ then add a NAC to rl' with all the nodes in rl'' that are refinements of nodes of rl', where rl'_{src} is the normal rule resulting by taking the source graphs of triple rule rl'.

Figure 10 shows the result of applying the patterns in Figures 8(b) and 9 to the abstract rule in Figure 9. The first rule considers the case when *Block* objects are not *Operators*. The second considers the case when objects are *Operators*. Note how, due to the NACs, the application of these rules is mutually exclusive

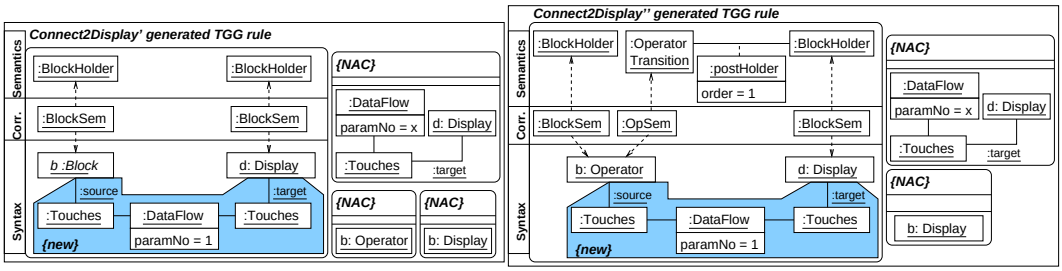


Figure 10: Generated Operational TGG rules.

5 Related Work

Our approach is inspired by the seminal work in [Sch94] and aims at providing an efficient way to obtain TGG operational rules, whenever a grammar for one of the graphs already exists. In this scenario, patterns do not need to specify which elements should be created and which should already exist in one of the graphs, as this is expressed in the normal rule to which the pattern is applied. When specifying a declarative TGG rule, one has still to indicate which elements should be present, and which ones are new. Thus, patterns may be used in several ways, which makes them more flexible and declarative than normal TGG rules as defined in [Sch94].

In addition, we have taken advantage of meta-models with the concept of abstract patterns. These are more compact than normal patterns, as they are equivalent to a number of concrete patterns resulting from the substitution of each object by instances of subclasses of the former object class. The concept of inheritance in TGG rules is of course not new, as existing TGG approaches based on meta-models such as [KS06] and [BGN+04] already consider inheritance.

Our contribution in this aspect is the observation that abstract patterns can be applied to rules with more abstract typing, and that a set of operational triple rules is generated which discriminates the right types by using negative application conditions. In addition, the concepts of triple rules with inheritance is fully formalized in the DPO approach in [GL07].

Due to lack of space, we have only presented the algorithms for translating in both directions; however generating operational rules for the scenarios described in [KS06] is also possible. In particular, it is possible to produce rules for creating the correspondence graph (assuming the source and target graphs already exist), to check the validity of the correspondence graph and for incremental updates. Further developments of TGGs can also be taken into account. For example, in [GW06], an efficient algorithm for incremental transformation was suggested, by relating the created nodes in the correspondence graph. Our patterns can also be used to create such relations.

A precedent to this work can be found in [G$\ddot{\text{o}}$79], where Göttler describes a programming language as a triple with the syntax, the semantics and a function ϕ describing how the semantic model is built from the syntactic one. In addition, he proposes meta-rules that modify either syntactic or semantic standard rules. In our case, we use triple patterns instead of meta-rules. Our triple patterns generate triple rules that are used to build the semantic model. Thus, they play the role of the ϕ function in Göttler's approach.

We believe this work is also relevant for the QVT community [QVT], as some efforts have been made to formalize QVT using TGGs. The most immediate similarities are found in the QVT relations, however attempts to formalize also the QVT Core have also been made[Gre06].

With respect to the application area of visual languages, Baar has proposed the use of TGGs to connect concrete and abstract syntax, so as to make it possible the static verification of the compliance between both [Baa06]. His proposal is however related to the structure of the visual sentence, and not to its operational interpretation. Moreover, it does not exploit inheritance, and requires the presence of display managers, relating the abstract and the concrete syntaxes.

6 Conclusions and Future Work

In this paper we have presented *Triple Patterns* as a compact means to obtaining operational TGG rules starting from normal graph grammar rules. We have shown that the approach is suitable for its combination with meta-modelling by considering the inheritance hierarchy in the meta-models. We have applied these ideas to the synchronized evolution of syntax and semantic models, improving previous work in [BDD+04].

There are several open issues. The first one is to study to which degree patterns can be automatically derived from the meta-model triple. In general this process cannot be fully automated. However, for our particular application case, it could be possible, as we just model three kinds of structures at the semantic level: *Tokens* decorating *Holders*, and *Holders* being pre- and post-conditions for *TransitionElements*. We could take the information of which classes in the semantic meta-model inherit from the base classes, and to which classes they are related in the syntax graph. However, for general applications, only an approximation can be derived.

Although not explicitly mentioned, our abstract patterns can only have abstract objects in the source graph, as, typically, concrete elements in the target graph are created. One can however

extend the notion of abstract rule [BELT04] to allow abstract nodes also in R and in the target graph. For our application, this would allow the designer to include predefined patterns in the semantic level (using only predefined classes *Token*, *Holder* and *TransitionElement*), thus helping towards the automatic generation of the triple patterns from the meta-model triple.

Up to now we have not considered problems related to the manipulation of attributes, which is up to future work. In addition we have only considered positive patterns, but we are currently working to extending this approach with patterns containing also negative conditions. The algorithms we have presented assume that the syntactic rules are "bigger" than the patterns. The study of the opposite case is still an open problem. In principle, by considering partial matches from patterns to rules, one could devise ways to generate additional triple rules with extended context. It is also worth studying whether we can extend the application of a pattern to general rules (i.e. not only deleting or non-deleting). A first line of attack might consider the splitting of a general rule into a sequence of one deleting and one non-deleting rule.

Finally, we are considering the application of the notion of triple graph grammars and meta-rules to the generation of operational semantics, for example the token game in the case of Petri nets.

Acknowledgements: This work has been partially sponsored by the Spanish Ministry of Education and Science with projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN 2006-09678), and the EC's Human Potential Programme under contract HPRN-CT-2002-00275, SegraVis. The authors gratefully thank the referees for their useful suggestions.

References

[Baa06] T. Baar. Correctly defined concrete syntax for visual models. In *Proc. MoDELS/UML 2006*. LNCS 4199, pp. 111–125. Springer, 2006.

[BDD+04] P. Bottoni, M. De Marsico, P. Di Tommaso, S. Levialdi, D. Ventriglia. Definition of visual processes in a language for expressing transitions. *J. Vis. Lang. Comput.* 15(3-4):211–242, 2004.

[BELT04] R. Bardohl, H. Ehrig, J. de Lara, G. Taentzer. Integrating Meta-modelling Aspects with Graph Transformation for Efficient Visual Language Definition and Model Manipulation. In *Proc. FASE*. LNCS, pp. 214–228. Springer, 2004.

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

[BGN+04] S. Burmester, H. Giese, J. Niere, M. Tichy, J. Wadsack, R. Wagner, L. Wendehals, A. Zündorf. Tool integration at the meta-model level: the Fujaba approach. *J. Softw. Tools Technol. Transfer* 6:203–218, 2004.

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

[G̈79] H. Göttler. Semantical Description by Two-Level Graph-Grammars for Quasihierarchical Graphs. In *Proc. Workshop Graph Theoretic Concepts in Computer Science – Graphs, Data Structures and Algorithms*. Applied Computer Science 13. Carl Hansen Verlag, 1979.

[GL07] E. Guerra, J. de Lara. Event-Driven Grammars: Relating Abstract and Concrete Levels of Visual Languages. *To appear in Software and Systems Modeling (SoSyM)*, 2007.

[Gre06] J. Greenyer. A Study of Model Transformation Technologies: Reconciling TGGs with QVT. Master/diploma thesis, University of Paderborn, July 2006.

[GW06] H. Giese, R. Wagner. Incremental Model Synchronization with Triple Graph Grammars. In *Proc. MoDELS/UML 2006*. LNCS 4199, pp. 543–557. Springer, 2006.

[KS06] A. Konigs, A. Schürr. Tool Integration with Triple Graph Grammars - A Survey. *Electronic Notes in Theoretical Computer Science* 148:113–150, 2006.

[MSUW04] S. J. Mellor, K. Scott, A. Uhl, D. Weise. *MDA Distilled*. Addison-Wesley, 2004.

[QVT] QVT specification, at http://www.omg.org/docs/ptc/05-11-01.pdf.

[Sch94] A. Schürr. Specification of Graph Translators with Triple Graph Grammars. In *Proc. WG'94*. LNCS, pp. 151–163. Springer, 1994.

A Subgraph Operator for Graph Transformation Languages

Daniel Balasubramanian, Anantha Narayanan, Sandeep Neema, Feng Shi, Ryan Thibodeaux, and Gabor Karsai

Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Abstract: In practical applications of graph transformation techniques to model transformations one often has the need for copying, deleting, or moving entire subgraphs that match a certain graph pattern. While this can be done using elementary node and edge operations, the transformation is rather cumbersome to write. To simplify the transformation, we have recently developed a novel approach that allows selecting subgraphs from the matched portion of the host graph, applying a filter condition to the selection, and performing a delete, move, or copy operation on the filtered result in the context of a transformation rule. The approach has been implemented in the GReAT language and tested on examples that show the practical efficacy of the technique. The paper describes the technique in detail and illustrates its use on a real-life example.

Keywords: model transformations, graph transformations

1 Introduction

Practical model-driven software development necessitates software tools that transform models, and these tools are often implemented using graph transformation principles. Graph transformation formalisms (e.g. single and double pushout [Roz97]) are based on node and edge matching, followed by creation of new nodes and edges, and/or removal of matched nodes and edges. In real-life applications, where the model transformations were implemented using graph transformation rules, we have observed the need for a more sophisticated operation that can move, delete, or copy entire subgraphs after the match has been computed.

The specific problem can be described as follows. We have a graph transformation language that supports graph rewriting rules; the left hand side of the rule is matched against a host graph, and if the match is successful, then the actions specified by the right hand side are executed. These actions can include deleting and inserting new edges and nodes. We argue that these elementary graph modification operations are often very low level, and practical users may want to use more complex, subgraph-oriented operations. The problem is especially apparent in graph transformation languages where the pattern nodes and edges could have cardinalities (like GReAT [AKN+06]), resulting in multiple matches from one activation of a rule. In this case, one has to create many simple rules to iterate over the results of the matching, as well as relating elements of different matches: clearly a cumbersome and error prone task.

For example, consider a model that contains a graph of nodes that are connected in a simple linear fashion, and some of the nodes have their ‘color’ attribute set to ‘red’ and some set to

'green'. If we want to find all the 'red' nodes and the edges that connect them (but only them), this can be done with a simple graph pattern. However, moving these 'red' nodes plus the edges between them into a different model container while discarding any 'red-green' edges requires a rather non-trivial sequence of operations (consisting of simple node and edge addition and deletions).

Motivated by this and similar examples, we have developed an approach for specifying ('selecting') subgraphs from the result of the 'match' phase of the rule execution. These resulting subgraphs can then be deleted, copied, or moved to a different part of the graph. We have implemented this feature in the context of the GReAT language. The paper revisits the GReAT rule execution behavior, introduces the new 'group' concept and how it has extended the rule execution mechanism, presents a detailed example, describes related results, and summarizes the results and discusses potential research directions.

2 Background

The GReAT toolkit is an integrated environment for the specification and execution of model transformations using graph transformation rules. The GReAT language is a graphical language for specifying the transformation rules in terms of the meta-models of the source and target languages. In this section, we will review the basics of the GReAT language.

2.1 The GReAT Language

The meta-models of the source and target languages of the transformation are specified using UML class diagrams and added as packages in GReAT. In addition to this, GReAT allows users to add additional packages to specify cross-domain links and temporary links using the UML notation. Along with rule construction ingredients such as *Blocks* and *Rules*, these form the core concepts of the GReAT language, which can be subdivided into the following three sub-languages:

- Pattern Specification Language
- Transformation Rule Language
- Sequencing and Control-Flow Language

Pattern Specification Language

The Pattern Specification Language is used to specify the graph pattern to be matched in the host graph. The graph pattern consists of nodes and edges, which must correspond to classes and associations from one of the existing packages. A pattern is a sub-graph in the host graph, such as an association *conn* between two classes *ClassA* and *ClassB* contained in a class *ClassC*. Figure 1 shows this pattern as seen in a GReAT rule. Such a pattern will match all associations of type 'conn' between instances of classes *ClassA* and *ClassB* that are contained in a class of type *ClassC*. A match binds the pattern variables classA, classB, and classC to three instances (of the respective type) that satisfy the topological constraint expressed by the pattern. A condition

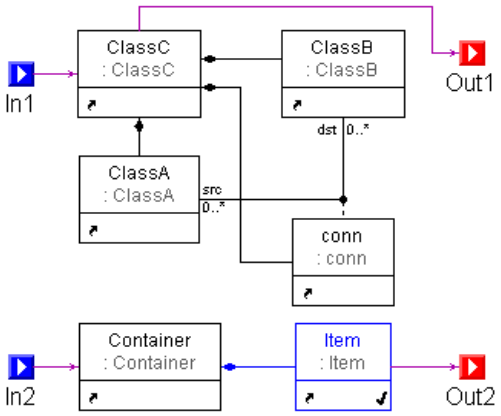


Figure 1: Example GReAT Rule

requiring the absence of a node or edge is called a *negative application condition*, and it is specified by using a pattern cardinality of zero for the appropriate elements.

Transformation Rule Language

The basic transformation entity of a GReAT transformation is called a *transformation rule*. A rule consists of a graph pattern specified using the pattern specification language and an *action* on the pattern elements or newly created elements. Three types of actions are allowed: *Bind*, specifying that the element is to be matched in the host graph; *CreateNew*, specifying that the element should be newly created; and *Delete*, specifying that the element must be matched and then deleted from the graph. The rule interacts with other rules in the transformation using an *Input Interface* for receiving nodes matched in a previous rule and an *Output Interface* for passing elements to the next rule. A boolean expression can be inserted as a *Guard* to enforce the execution of a rule only under certain conditions. In addition, a special element called *Attribute Mapping* can be added to supplement the rule with C++ code. The Attribute Mapping code can be used to perform additional functions such as performing computations and setting the values of attributes for the existing or newly created objects. Note that a rule may generate multiple matches. For instance, if classC contains two instances of classA and two instances of classB, with the appropriate pairs connected via separate instances of the 'conn' association, the pattern matching yields two matches and the rule actions will be executed for each match. Each rule can also be configured to be executed for one, randomly selected match or for all the possible matches. Figure 1 shows an example GReAT rule with the input ports on the left and the output ports on the right. The node *Item* is marked with the *CreateNew* action (indicated by the tick mark on the bottom right), and the rest of the nodes are marked with *Bind*.

A transformation rule in GReAT is a composite structure that includes both the LHS and the RHS of the transformation. In other words, elements that are marked with the *Bind* action can be considered as the LHS of the rule, and other elements in the same pattern that are marked with the *CreateNew* or *Delete* actions can be considered as the RHS. (Note that nodes to be deleted

are matched first.)

The basic unit on which a rule operates is called a *packet*, which consists of nodes of the host graph to be bound to the input ports of the rule, one-to-one. When a rule fires, the pattern matching algorithm finds the matching subgraphs in the host graph, depending on the pattern specified in the rule. As mentioned above, a single rule execution may find multiple matches. From our example, given a classC, there could be multiple classB and classA objects associated via a conn, all being the children of the classC. Once these matches are found, the rule actions are executed for each match. For each match and action execution output packets are generated and placed at the output ports of the rule. The nodes bound to the output ports of the rule will provide the output packet(s) of the rule, and that will subsequently function as the input packet(s) of the next rule in the sequence of rules. In the case of the first rule, the nodes forming the input packet are specified by the user.

Sequencing and Control Flow Language

The sequencing and control flow language allows users to compose a series of rules specified using the transformation rule language. The sequencing allows users to connect multiple rules in a sequence such that each rule is executed in the order determined by the direction of the connections between the ports of the rules.

For the sake of convenience and manageability, rules can be composed in a hierarchy by placing a set of rules in a *Block* or a *ForBlock*. When a sequence of rules is placed in a *Block*, each rule processes all its input packets and then passes the produced set of output packets to the next rule. In the case of a *ForBlock*, each input packet of the parent *ForBlock* passes through all the rules before the next packet is processed. Blocks also allow recursion by connecting the output of the last rule in the block to the input interface of the block.

Conditional execution of rules can be specified by using the *Test* and *Case* blocks. A *Test* block contains multiple *Case* blocks, each specifying a desired match and a set of connected output ports, but no *CreateNew* and *Delete* elements. The output(s) of the *Cases* are connected to the output(s) of the parent *Test*. Each *Case* is tried, and the output produced by the successful one is place on the output interface of the *Test*.

3 The Group Operator

As it was discussed in section 2, our current implementation for a graph re-writing rule's execution works in two sequential steps:

- Step 1: Find all the valid matches in the host graph for the given pattern (i.e., match the LHS of the rule in the host graph) and the given binding for the input ports.
- Step 2: For each match, independently, apply the rule action(s). This can include deleting elements, creating new elements, and changing attributes.

The major limitation with this algorithm is the inability to apply a single rule action across multiple matches. After all matches are computed, the rule's action is executed individually

on each match; furthermore, there exists no mechanism by which one can access any information about other matches while processing a specific match. This can sometimes pose a severe limitation to the types of transformations one can write.

For instance, the user may need the ability to operate on an entire subgraph (composed from multiple matches) as a whole rather than on individual elements. If this subgraph may contain an arbitrary number of elements, then the graph pattern cannot be specified as a simple rule. What is needed is a way to find smaller pieces of the subgraph, and then combine these pieces to form the entire subgraph. In terms of GReAT this means we need a way to combine multiple matches of a rule, and then perform the rule's action on these combined matches.

To achieve this capability, we had to extend the existing GReAT language. The solution was the introduction of a new syntactic element known as a '*Group*' into the language. A single *Group* element can be inserted into any rule, which allows the user to select an arbitrary set of pattern elements from that rule. A *Group* element is associated with a set of rule elements (pattern nodes and edges), but the only action allowed on these elements is *Bind*. The elements in this set are then used to form the subgraphs mentioned in the previous paragraph. It is important to note that the introduction of a *Group* element into a rule does not affect the pre-existing semantics of pattern matching in any way; that is, the set of matches found by a rule is the same regardless of whether or not that rule contains a *Group* element. A *Group* affects the semantics of a rule only after the pattern matching phase of that rule's execution, at which point the *Group* is used to specify how to form the set of subgraphs (also called subgroups to emphasize the relationship with the *Group* element). The user has the ability to control the way in which matched elements are added to the subgroups by specifying 'filtering criteria' for the *Group* that determine whether a matched element should be added to a subgroup or not. After all subgroups have been formed, then normal rule execution proceeds, with the exception that actions are performed for each subgroup (instead of for each match per the default behavior).

The precise semantics of a transformation rule that contains a *Group* element are informally described below:

- Step 1: Find all the valid matches in the host graph for the given pattern and given bindings. A match contains one matched element from the host graph for each pattern element.
- Step 2: After all the matches are found, insert a subset of the matched elements into a single subgroup based on the user specified criteria. These criteria are boolean expressions that compare matched elements of the current match to elements already inserted into subgroups; if the expressions evaluate to true, then the matched elements are inserted into the subgroup against which the expressions were evaluated. If the expressions evaluate to false for all existing subgroups, then the matched elements are inserted into a newly created subgroup.
- Step 3: Iterating over the subgroups, for each subgroup, execute the regular rule actions. Note that these actions apply to rule elements that are not part of the *Group*, and they include actions like creating new elements, and changing attributes. Matched (non-grouped) elements that are slated for deletion are removed at this time as well.
- Step 4: For each subgroup, apply the group action. This can be any of the following:

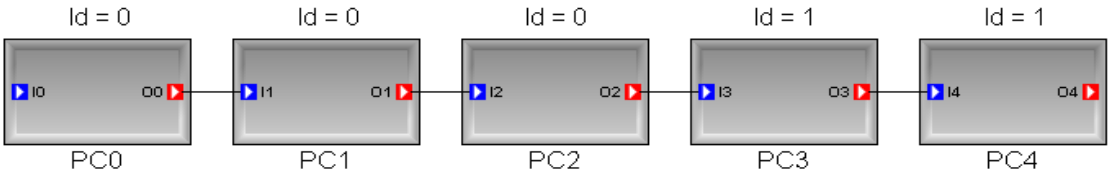


Figure 2: Signal Flow Model

- *Bind*: No action will be taken on any of the objects in the subgroup.
- *Move*: All of the objects in the subgroup will be moved into a single container: a node that has a 'containment' relationship with the *Group* expressed in the rule. If this container is found via the matching process, then it must be a unique node. If this container is newly created, then it is unique (per Step 3). Edges whose endpoints are in different subgroups (or one of them is outside of all subgroups) are removed.
- *Copy*: All objects of the subgroup will be copied into all containers that have a 'containment' relationship with the *Group* expressed in the rule. Edges whose endpoints are in different subgroups (or one of them is outside of all subgroups) are removed.
- *Delete*: All objects in the subgroup will be deleted.

Steps 3 and 4 describe how the introduction of the group operator gives the ability to apply a single rule action across multiple matches. Further, step 4 describes two abilities that were not present in GReAT before the introduction of the *Group* operator: the ability to copy or move entire subgraphs. While this moving and copying ability is trivial in the case that only a single object is moved or copied, moving an entire subgraph, which includes a number of nodes and edges, is a non-trivial task and requires the use of a group-like operator. This is especially true if one does not know a priori the number of elements that will have to be moved.

The next section describes an example of the *Group* operator in a transformation.

4 Example of the Group Operator

This example, drawn from the domain of signal processing, demonstrates how to use the *Group* operator in a transformation. Suppose we have a signal flow chain with five primitive components, as shown in Figure 2.

Each primitive component has a numeric Id associated with it. We assume that the components PC0, PC1, and PC2 have an Id of 0, while components PC3 and PC4 have an Id of 1. The user wishes to write a transformation that will disconnect the components according to their Id number and put each group of connected components in its own newly created compound component; components with the same Id should be moved to the same compound component, and the existing connections between them should be preserved. In our case, the result of the transformation applied to the example model shown in Figure 2 should be composed of two new compound components that contain the signal flow components as shown in Figure 3.

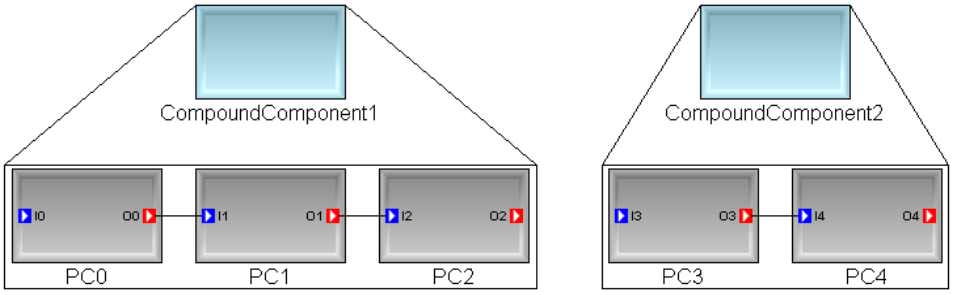


Figure 3: Resulting Model

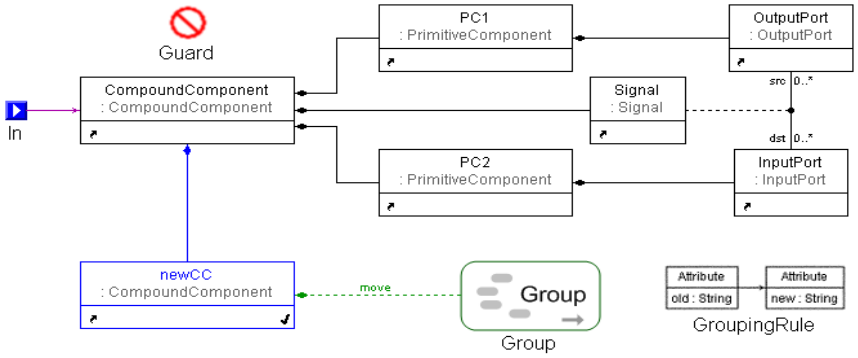


Figure 4: Group Rule

Figure 4 shows a GReAT transformation rule with a *Group* element that will accomplish this transformation. The incoming context for this rule is the compound component in which the signal flow chain is found. Figure 5 shows this transformation rule using a special visualization, highlighting only those elements that belong to the *Group* (PC1, PC2 and Signal). As can be seen in the rule shown in Figure 5, the Group element contains three elements: two primitive components and the association class Signal. The action of the Group is "move", meaning that objects bound to pattern variables contained in the Group (i.e., PC1, PC2, and Signal) are moved to the Compound Component "newCC". The execution order of the rule is:

- Step 1: Pattern matching. Find all matches for the pattern described in the rule and apply the Guard to discard matches in which two primitive components with different Id-s are bound to PC1 and PC2. The code in the Guard is:

  ```
  return (PC1.Id()==PC2.Id());
  ```

  This ensures that only matches consisting of two connected primitive components with the same Id will be considered. For the input model shown in figure 2, the matches are:

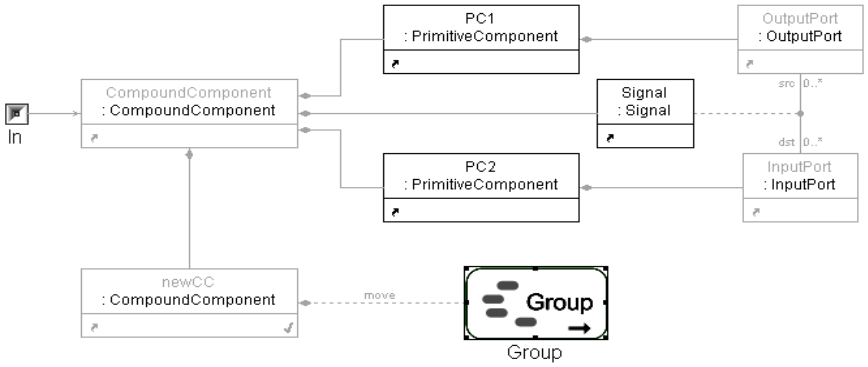


Figure 5: Group Rule visualized in Set Mode

```
{(PC0, PC1), (PC1, PC2), (PC3, PC4)}
```

- Step 2: Form the subgroups. Host graph elements that are part of the same match will all be inserted into the same subgroup, and the subgroups are formed using the user specified grouping criteria. The grouping criteria code for our example above is the following:

  ```
  return (the_PC1.Id()==other_PC2.Id());
  ```

  This code is an attribute of the *Group* and works as follows. PC1 and PC2 are prefixed with "the_" and "other_", respectively. These special prefixes give the ability to compare model elements that are part of different matches. The "the_" prefix refers to a graph element (bound to a pattern element) found in the current match that is being considered for insertion into a subgroup, while the "other_" prefix refers to graph elements that are already contained in a subgroup. In our example, the code returns true (and results in the insertion of the graph elements from *Groups* of two different matches into the same subgroup) if the Id of primitive PC1 in one match is the same as the Id of the primitive PC2 in a different match. The two subgroups formed in our example will be:

  ```
  Group 1: PC0, PC1, PC2 (and the connections between them)
  Group 2: PC3, PC4 (and the connections between them)
  ```

- Step 3: Apply the regular rule action, which in our example is the creation of new objects. After the subgroups are formed in step 2, a new CompoundComponent object is created for each subgroup.

- Step 4: Perform the *Group* action on each subgroup, which in our example is to move the subgroups. The constituent parts of the subgroups, assigned in Step 2, are moved

into the newly created CompoundComponent from Step 3. Note that the edge that did exist between PC2 and PC3 is deleted due to the fact that those two pattern classes were inserted into different subgroups.

5 Group Implementation

Using a *Group* in a transformation rule requires that the rule

1. specifies the *Group* and the elements it contains,
2. specifies the grouping criteria for the matches, and
3. specifies the action to be performed on the Group.

We have implemented *Groups* using the *Set* concept of GME [GME]. A *Set* is a universal container that can hold any number of arbitrary elements. In this manner, a *Group* can contain any type and number of pattern nodes and edges normally found in a transformation rule. The visualization interface of GME allows the user to specify which pattern classes are part of the *Group* without drawing any explicit edges or associations (see previous figure).

In addition to specifying which elements of a rule belong to the Group, one must also specify the grouping criteria, which determine the subgroup into which each match should be placed. These criteria are written as boolean expressions that evaluate to either true or false, and are added as an attribute to the *Group* element. The expressions are used to compare each match to matches that have already been placed into a subgroup; if the expressions evaluate to true, then the current match is added to the subgroup against which the criteria were evaluated. If the expressions evaluate to false for all existing subgroups, then the current match is inserted into a newly created subgroup. The elements of two distinct matches are identified in the grouping criteria by prefixing "the_" and "other_" to the class names of the elements. For instance, to place all instances of *ClassA* that have the same *InvoiceNumber* into one *Group*, the expression would be: the_ClassA.InvoiceNumber() = other_ClassA.InvoiceNumber().

It should be noted that the distinct pattern matches may share common elements, and it could happen that the grouping criteria allow the insertion of the same element into two different subgroups. As this could lead to erroneous rule execution, we believe this is an error that should result in raising exception. We are currently investigating how exception handling could be incorporated into the language.

Once the *Group* has been completely specified, the action for the *Group* can be set to *Bind*, *Move*, *Copy*, or *Delete* by selecting the desired value for the *GroupAction* attribute. The effect of these actions was described earlier in section 3.

Currently, *Groups* have been fully implemented in the GR-Engine [AKN+06], our interpreter for performing model transformations, and we have a prototype implementation in our code generator. The previous version of the GR-Engine had an architecture as shown in Figure 6, except without the Group Manager module. Because of the loose coupling between the Pattern Matcher and the Effector, implementing *Groups* primarily consisted of inserting a 'Group Manager' module between the two and defining the appropriate interfaces between the modules.

In the case that a rule contains a Group, the GR-Engine functions in the following manner:

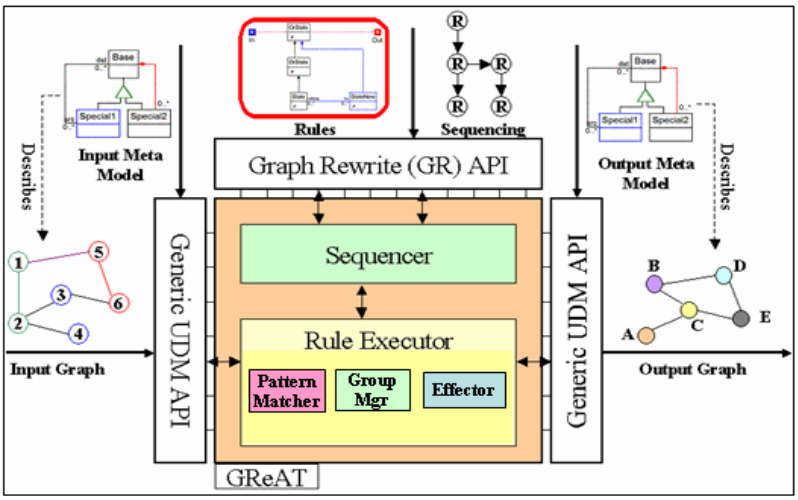


Figure 6: Modified GR-Engine Architecture

- The Pattern Matcher functions exactly as before, finding all valid matches. After all matches are found, they are passed to the Group Manager.
- Using the grouping criteria given by the user, the Group Manager forms the set of subgroups. After all subgroups are formed, they are passed to the Effector using a newly defined interface created to handle subgroups.
- For each subgroup, the Effector first performs the regular rule action and then performs the *Group* action (as previously described).

6 Related Work

Transformations on hierarchical subgraphs have been approached previously by Hoffman and then Janssens. Hoffman [DHP02] introduced transformations on subgraphs by delimiting the subgraphs with frames in which edges do not cross between frame boundaries. The copying of subgraphs within this context permits copying only the nodes and edges contained within a frame. The *Group* operator uses a similar idea for specifying membership; however, it allows for restricting node membership in a *Group* by having the user select specific nodes and associations within the subgraph using the set utility of GME. Only those selected nodes and associations (between the selected nodes) will have the *Group* action (bind, move, copy, or delete) performed on them.

Janssens [GSJ06] further generalized the copying of subgraphs by introducing two concepts known as “copy” and “oncopy” within the MotMot project [SG05]. MotMot generates code for templatized graph transformation rules created in the UML profile for Story Driven Modeling (SDM) [SGJ05]. The “copy” and “onCopy” constructs allow for copying specific nodes and

edges in the same way the *Group* operator does; however, we believe, the *Group* operator's membership criteria allow for more selectivity in the GReAT implementation than in MotMot.

It is also worthwhile to mention the concept of amalgamated graph transformations presented by Ehrig et al. in [BFH87] and utilized in the Algebraic Graph Grammar system (AGG) [TB93]. Amalgamated graph transformations decompose transformations into simpler graph productions that can be performed in parallel and the results combined synchronously. The synchronous combination restriction follows from the sharing of graph objects between the productions, i.e., actions performed on the results must be done such that conflicting or duplicate actions are not performed on the same object. Currently, our *Group* operator does not automatically detect whether or not a single element is part of two different subgroups. In the case that an element does belong to two distinct subgroups, then the transformation is non-deterministic if the subgroups are moved to different containers. Otherwise, in the case that the subgroups are simply copied, deleted, or bound, the result is deterministic. Ensuring that the subgroups do not contain common elements is an area in which we are currently working.

Concerning the capabilities of other transformation tools, it seems those that use textual frameworks as part of their back-end have more ease and expressive power for implementing complex transformations in their pattern/rule specifications. Most likely, tools such as ATL [ATL] or VIATRA2 [VIA] could implement a feature similar to our *Group* operator more easily with their textual transformation languages instead of implementing a graphical artifact interpreted by transformation engines.

7 Summary and Future Work

We have introduced a technique for specifying arbitrary subgraphs in graph transformation rules that allows the handling of such subgraphs as a unit to be moved, copied, and deleted. We have implemented this new extension in the GReAT language and have tested it on several examples. The current implementation is available only in the interpreted mode, but we started working on the code generator that compiles such rules into executable code. We believe the technique is fairly efficient and allows the compact description of complex graph operations.

There are several potential interesting research topics that arise from the initial idea. One is related to code generation: how to generate efficient code that computes the *Groups* and performs the necessary graph operations on the subgraph? Another direction is related to treating the *Groups* as "the" result of pattern matching and passing it to subsequent rules (note that *Groups* essentially combine results from multiple matches). A third direction could be a more compact specification of the *Groups* with node and edge types that are 'generic' (i.e. not of a specific node or edge type). We plan to investigate these directions in the near future.

Acknowledgements: The research described in this paper has been supported by a grant from NSF/CSR-EHS, titled "Software Composition for Embedded Systems using Graph Transformations", award number CNS-0509098, and by NSF/ITR, titled "Foundations of Hybrid and Embedded Software Systems", award number CCR-0225610.

Bibliography

[AKN+06] A. Agrawal, G. Karsai, S. Neema, F. Shi, A. Vizhanyo. The Design of a Language for Model Transformations. *Journal on Software and System Modeling* 5(3), Sept. 2006.

[ATL] ATL Project. An ECLIPSE GMT Subproject. http://www.eclipse.org/m2m/atl/

[BFH87] P. Boehm, H.-R. Fonio, A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *J. Comput. Syst. Sci.* 34(2-3):377–408, 1987.

[DHP02] F. Drewes, B. Hoffmann, D. Plump. Hierarchical Graph Transformation. *Journal of Computer and System Sciences* 64:249–283, 2002.

[GME] GME 6 User's Manual. http://www.isis.vanderbilt.edu/Projects/gme/

[GSJ06] P. V. Gorp, H. Schippers, D. Jannsens. Copying Subgraphs within Model Repositories. In *5th International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT).* Vienna, Austria., 2006.

[Roz97] G. Rozenburg. *Handbook of Graph Grammars and Computing by Graph Transformation*. World Scientific, 1997.

[SG05] O. M. H. Schippers, P. V. Gorp. Model Driven, Template Based, Model Transformer (MoTMoT). 2005. http://motmot.sourceforge.net/

[SGJ05] H. Schippers, P. V. Gorp, D. Janssens. Leveraging UML Profiles to Generate Plugins From Visual Model Transformations. *Electr. Notes Theor. Comput. Sci.* 127(3):5–16, 2005.

[TB93] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In *Dagstuhl Seminar on Graph Transformations in Computer Science*. Pp. 380–394. 1993.

[VIA] VIATRA2 Framework. An ECLIPSE GMT Subproject. http://www.eclsipse.org/gmt

Adding Recursion to Graph Transformation.

Esther Guerra[1], **Juan de Lara**[2]

[1] eguerra@inf.uc3m.es
Dep. Ingeniería Informática
Universidad Carlos III de Madrid (Spain)
[2] jdelara@uam.es
Escuela Politécnica Superior
Universidad Autónoma de Madrid (Spain)

Abstract: In this paper we define recursive rules in the double pushout approach (DPO) to graph transformation. Classical DPO rules are extended with a *base case condition* and a *recursion condition*. Mechanisms are provided to pass the match from both conditions to the rule's left hand side, and also between two consecutive steps in the recursion. The approach is useful when recursive structures (such as inheritance hierarchies, nested component hierarchies, networks of functional blocks, etc.) have to be processed. Although we present the recursion for DPO, it can also be adapted to other approaches to graph and model transformation. We present examples for model transformation, model simulation and model optimization in different application domains.

Keywords: Graph Transformation, Double Pushout, Recursion.

1 Introduction

Graph transformation [Roz97] is becoming increasingly popular to express computations on graphs due to its formal, declarative and graphical nature. One of the most popular formalizations of graph transformation is the double pushout approach (DPO) [EEPT06], which uses category theory to model rules and derivations. Graph transformation has been used in many application areas, such as modelling with visual languages [Min02], visual simulation [LV04], model transformation [EGL+05] and refactoring [EJ04]. The manipulation of structures with nested, iterated or recursive elements is common in many of these areas.

The formal nature of DPO graph transformation allows interesting analysis techniques, for example to investigate confluence, termination and rule independence [EEPT06]. Moreover, the categorical framework has lifted the results from graphs to any (weak) adhesive HLR category [LS04, EEPT06] (short AHLR category). However, compared to other approaches [BV06, KASS03, SWZ99, NNZ00], it lacks expressivity when handling complex structures involving recursion, iteration or nesting. Processing such structures usually implies performing a certain action in their different parts (e.g. copying all the attributes of a class to its children). Moreover, sometimes the structure has to be traversed in a certain order (e.g. when propagating a change in a network of logical gates). Having high-level constructs to process these kinds of structures is interesting for model transformation, but as we show, it is also useful for other manipulations such as optimization and simulation.

Usually, there are two options when processing a recursive structure with DPO rules. As each element has to be consecutively processed, a solution is encoding helper control elements in the graph, which guide the application of the rules. However, this is sometimes undesirable or impossible, as it implies modifying the meta-model (or type graph) of the model to be transformed. Another possibility is to *flatten* the structure. For example, performing the transitive closure (by adding *ancestor* edges) can flatten an inheritance hierarchy. Again, this solution implies a modification of the type graph as well as pre- and post- processing phases.

In this paper we propose *recursive rules*, which enhance the expressive power of DPO rules in order to make the recursive processing of structures easier. We extend classical DPO rules with mechanisms for passing matching information between consecutive rule applications, and to guide rule execution by traversing structures recursively, where the rule's action is executed at each step in the recursion. We present our approach in the AHLR framework, in such a way that it becomes valid for any AHLR category. In particular, we show an instantiation for attributed typed graphs. Our approach has several benefits. On the one hand, there is no need to add extra elements to process the recursive structure. This usually leads to simpler and higher-level rules as we abstract from "accidental details", concentrating on the essence of the problem. On the other hand, the execution of a recursive rule can be more efficient than several DPO simple rules, as the matches are guided through the structure. Moreover, the framework can be adapted to other graph and model-transformation approaches.

Paper organization. Section 2 gives an overview of transformation formalisms that consider the processing of recursive structures. Section 3 introduces the DPO approach. Section 4 presents our proposal for recursive rules. Section 5 shows additional examples for model simulation and transformation. Finally, section 6 ends with the conclusions and future work.

2 Related Work

Some approaches to model transformation are found in the literature to handle recursive applications of a rule. Usually, they are based on the use of some control flow language that guides the transformation execution. For example, GReAT [KASS03] provides hierarchical data flow diagrams for control execution, where rules can be placed on blocks and participate in recursive calls. Control flow in VIATRA [BV06] is specified by means of Abstract State Machine (ASM) programs, while in Fujaba [NNZ00] it is done by means of story diagrams. In the OMG's QVT [OMG] specification, complex transformations can be implemented by using the Operational Mapping Language either in an imperative or in an hybrid approach.

Other model transformation languages embed control mechanisms in the rules. One example is UMLX [Wil03], which provides rule encapsulation, thus allowing composition and some degree of recursion. However, traversing a structure can be incomplete if a match of the rule is not found in some recursion step. MOLA [KBC05] incorporates mainly loopings in the rules as graphical control structure, and allows the transitive closure and the traversal of the recursive structure in a single rule. However, the recursion semantics is not formally defined.

In addition to control languages, some approaches offer recursive mechanisms in the left hand side (LHS) of the rules. These are conditions that can be recursively evaluated by traversing certain structures in the graph. For example, PROGRES [SWZ99] and Fujaba provide path

expressions. Thus, given an initial match, edges can be matched by testing their existence or by traversing them. Another example is VIATRA, where queries on graphs can be expressed by generalized (recursive) graph patterns, which may contain a nesting of positive and negative patterns of arbitrary depth. However, neither path expressions nor recursive patterns can guide the rule execution. That is, they are expanded by a recursive evaluation, but the rule that contains them is applied once. Consecutive rule applications require the use of a control language and a mechanism to pass the elements matched in previous executions to the next execution step.

With respect to approaches based on DPO, there are some attempts to increase the expressiveness of DPO rules for refactoring [EJ04]. However, these do not consider recursive application of rules. In parallel graph transformation [Tae96], standard DPO rules are extended by allowing certain parts of the rules to be instantiated an arbitrary number of times, and providing synchronization mechanisms controlling the way the matching should occur. Thus, the approach is a way to describe in a concise way a (possibly infinite) number of rules in which a certain part is replicated. Again, this is a mechanism for rule expansion, where the rule is applied once.

3 The Double Pushout Approach

In this section we give a brief overview of the double pushout approach (DPO) to graph transformation. See [EEPT06, Roz97] for a more extensive presentation.

Graph grammars are made of rules with a left and right hand side (LHS and RHS). When a rule is applied to a graph G (the *host graph*), an occurrence of the LHS (a matching morphism) has to be found in G, which can be then substituted by the rule's RHS. DPO uses category theory to model rules and derivations, and its theory has been lifted from graphs to (weak) AHLR categories [LS04, EEPT06]. These categories are based on a distinguished class $\mathscr{M}$ of monomorphisms. Examples of AHLR categories are graphs, typed graphs, P/T nets (indeed a weak AHLR category) and attributed typed graphs. Thus, not only graphs, but also objects in any (weak) AHLR category $(\mathbf{C}, \mathscr{M})$ can be rewritten using DPO rules.

DPO rules are modelled using three components: L, K and R. L contains the necessary elements to be found in the object to which the rule is applied. K (the gluing object) contains the elements that are preserved and R those that should replace the identified part in the object being rewritten. Roughly, $L-K$ are the elements that should be deleted by the rule application, while $R-K$ are the elements that should be added. We present these two concepts in the following definitions, taken from [EEPT06].

Definition 1 *(DPO rule) Given a (weak) AHLR category* $(\mathbf{C}, \mathscr{M})$*, a DPO* rule $p = (L \xleftarrow{l} K \xrightarrow{r} R)$ *consists of three objects L, K and R called left hand side, gluing object and right hand side respectively, and morphisms* $l : K \to L$, $r : K \to R$ *with* $l, r \in \mathscr{M}$.

Definition 2 *(DPO derivation) Given a DPO rule* $p = (L \xleftarrow{l} K \xrightarrow{r} R)$*, an object G, and a morphism* $m : L \to G$ *called match. A* direct derivation $G \overset{p,m}{\Longrightarrow} H$ *from G to an object H is given by the diagram to the left of Fig. 1, where (1) and (2) are pushouts. A sequence* $G_0 \Rightarrow G_1 \Rightarrow \ldots \Rightarrow G_n$ *of direct derivations is called a* derivation *and is denoted as* $G_0 \overset{*}{\Rightarrow} G_n$.

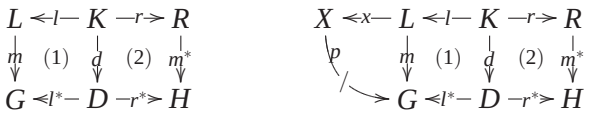


Figure 1: DPO Direct Derivation (left). Direct Derivation by DPO Rule with NAC (right).

Sometimes, rules are equipped with application conditions [EEPT06] constraining their applicability. For simplicity, we only deal with negative application conditions (NACs). These have the form $NAC(x)$, where $x\colon L \to X$ is a morphism. Morphism $m\colon L \to G$ satisfies $NAC(x)$ if there is no morphism $p\colon X \to G$ in $\mathscr{M}'$ with $p \circ x = m$ (see right of Fig. 1). $\mathscr{M}'$ is an additional class of distinguished morphisms than can be $\mathscr{M}$ if the latter is the class of all monomorphisms [EEPT06]. Next, we define the concepts of AHLR system, grammar and language.

Definition 3 *(AHLR system, grammar and language) An* AHLR system $AHS = (\mathbf{C}, \mathscr{M}, P)$ *consists of a (weak) AHLR category* $(\mathbf{C}, \mathscr{M})$ *and a set of productions* P*. An* AHLR grammar $AHG = (AHS, S)$ *is an AHLR system together with a distinguished start object* S*. The* language L *of an AHLR grammar is defined by* $L = \{G \mid \exists S \overset{*}{\Rightarrow} G\}$.

Example and Motivation. Fig. 2 shows some DPO rules in the category of typed graphs $\mathbf{Graph_{TG}}$, where objects are tuples $(G, type_G)$. The first element is a graph, and the second one a typing function $type_G\colon G \to TG$ from G to a distinguished graph called the type graph. The type graph in the example is taken from a Role Based Access Control system. It models hierarchies of roles (through relation *parent*), which can be granted permission (relation *perm*) to execute certain functions. The rules present together in a single graph the L, K, R and X components. Elements in $L - K$ are marked as "del", elements in $R - K$ as "new" and elements in $X - L$ as "NAC". We follow a UML-like notation, where the types are shown after a colon.

Rules in the example are used to eliminate redundant permissions in role hierarchies. A permission is redundant for a role if an ancestor already defines it. The rules first calculate the transitive closure of relation parent by adding helper edges of type *anc*. Rule *createDirectAncestor* creates such edge to a direct parent. The iterated execution of rule *createAncestor* performs the transitive closure. Rule *removeRedundantPermission* removes relation *perm* from a role if an ancestor already has the same permission. We use an execution control structure for rules based on layers. The three previous rules are assigned the layer one and are applied as long as possible. Once no rule in this layer can be applied, the next layer is executed. The second layer contains rule *deleteAncestor*, which deletes the helper *anc* edges (i.e. a post-processing step).

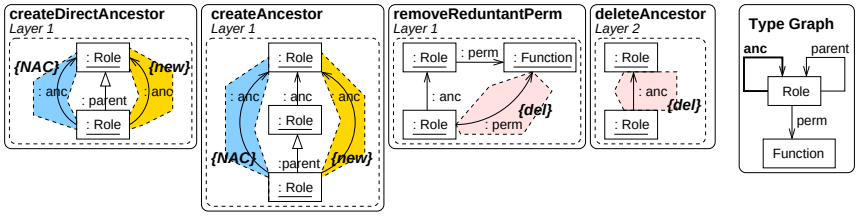


Figure 2: Example DPO Rules.

Note how, in order to detect redundant permissions, we need to add *anc* helper edges from each role to all its ancestors. This is done because we do not know how long is the path of parent edges starting from a given role. Without the helper edges one could build different rules (similar to *removeRedundantPermission*) to eliminate the redundant permission when there is a direct connection between two roles, when they are separated by a path of two, of three and so forth. However, for arbitrary structures, this does not work as we may need arbitrarily many rules. In addition, DPO simple rules only have information of nodes and edges matched by its LHS. There is no control mechanism that allows moving through a given structure and pass the matching information between consecutive derivations. Hence, control information has to be encoded in the data or, as in this case, the recursive structure has to be flattened.

The solution of adding helper edges (*anc* in the example) is not optimal. First, we have to modify the type graph (see Fig. 2). This is undesirable or even impossible in real applications, if the type graph is a standard meta-model of some modelling language (such as UML), and is being used by other users and tools. Moreover, if several computations have to be performed, it is not feasible that each one of them adds different helper structures to the type graph. Second, there is a pre-processing phase in which helper edges are explicitly added, and a post-processing phase in which the edges have to be removed. These two phases produce a computation overload. In the next section we propose a solution to alleviate these problems.

4 DPO Recursive Rules

In this section we extend DPO rules with several artefacts to model *conditions* for the base and recursive cases, as well as a mechanism to pass part of the match between the base and recursive cases, and between two consecutive steps in the recursion.

Definition 4 *(DPO recursive rule) A DPO recursive rule* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *is made of:*

- *A DPO rule* $L \xleftarrow{l} K \xrightarrow{r} R$*, a base condition* I^b *and a recursion condition* I^r*.*
- *The relations between the base* $(j = b)$ *and recursion* $(j = r)$ *conditions and* L *by means of their common elements (object* P^j*),* $(I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}$*, with* i^j*,* $p^j \in \mathscr{M}$*.*
- *The relation between the base and the recursion conditions by means of their common elements (object* P^{br}*),* $I^b \xleftarrow{i^{br}} P^{br} \xrightarrow{p^{br}} I^r$*, with* i^{br}*,* $p^{br} \in \mathscr{M}$*.*
- *The relation between the recursion condition for two consecutive steps in the recursion by means of their common elements (object* P^{rr}*),* $I^r \xleftarrow{i^{rr}} P^{rr} \xrightarrow{p^{rr}} I^r$*, with* i^{rr}*,* $p^{rr} \in \mathscr{M}$*.*

with the constraint that P^b *and* P^r *have to be preserved by the DPO rule application, that is, there are morphisms* $a\colon P^b \to K$*,* $b\colon P^r \to K$ *s.t.* $l \circ a = p^b$ *and* $l \circ b = p^r$*, i.e. triangles (1) and (2) in Fig. 3 commute.*

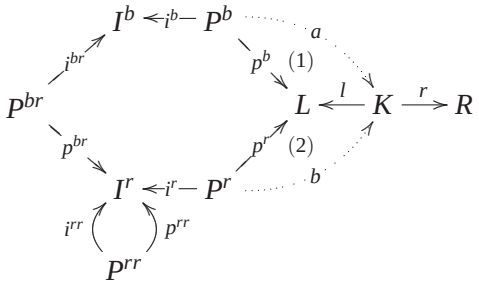
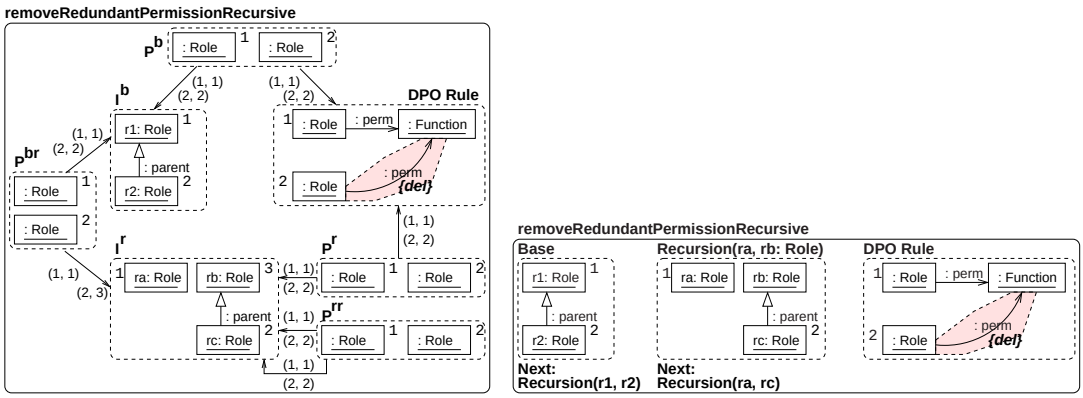


Figure 3: Formalization of a DPO Recursive Rule.

Fig. 4 shows a recursive rule that eliminates redundant permissions in role hierarchies, equivalent to the set of standard DPO rules in Fig. 2. To the left, the rule is shown according to the theory, to the right using a more compact and intuitive notation that will be used throughout the paper.



(a) (b)

Figure 4: A DPO Recursive Rule Example (a) Theoretical Notation. (b) Compact Notation.

The base condition I^b (labelled "Base" in the compact notation) identifies two roles related through a parent relation. The DPO rule specifies that if both roles have permission to the same function, then the permission of the child is deleted. The recursion condition I^r (labelled "Recursion(...)" in the compact notation) goes down the role hierarchy, maintaining the match of the highest role in the hierarchy identified by the base case. As we will see later, the DPO rule will be applied for each step in the hierarchy identified by consecutive matchings of I^r. The following shortcut is used in the compact notation for recursive rules: elements P^b and P^r are hidden, but can be calculated from the numeric labels. In this way, P^b (resp. P^r) is the intersection of the numeric labels in I^b (resp. I^r) and the rule. Morphisms p^b and p^r identify elements with the same numbers. On the other hand, the relation between I^b and I^r (and between two recursive cases) in the compact notation is given by means of the call after "Next:" in the base case. Thus, P^{br} (and

therefore i^{br}) is given by the actual parameters of the recursion call from the base case (that we depict using the identities of nodes and edges, shown before the colon). Note also that morphism p^{br} is given by the assignment of the formal parameters in the recursive condition (i.e. *ra* and *rb*). In a similar way, P^{rr} (and i^{rr}) is given by the actual parameters of the recursion call from the recursive case and p^{rr} by the assignment of the formal parameters in the recursive condition (i.e. *ra* and *rb*). Note how some formal parameters of the recursion may become unused by one of the two recursive calls, and in this case they are just ignored.

4.1 Derivation by DPO Recursive Rule

A DPO recursive rule derivation is made of three steps, each composed by several sub-steps.

First step. The rule is executed for the base case (see Fig. 5). A match $e^b : I^b \to G$ (called *base match*) has to be found for the *base condition* I^b, identifying the starting point in the recursive structure to be processed. Then, a *rule match* $m_1^b : L \to G$ is sought for L, such that $e^b \circ i^b = m_1^b \circ p^b$, and which has to make $(P^b, i^b : P^b \to I^b, p^b : P^b \to L)$ the pullback of $(G, e^b : I^b \to G, m_1^b : L \to G)$, as square (1) in Fig. 5 shows. Then, rule $L \xleftarrow{l} K \xrightarrow{r} R$ is applied once in G, yielding graph H_1. As Proposition 1 shows, match e^b still exists in H_1. If a match $m_2^b : L \to H_1$ is found such that square (2) in Fig. 5 is pullback, we apply again the rule at that match. The operation is repeated until no match from L is found making P^b a pullback object. The output of this step is the base match e^b, together with graph H_n, obtained as a result of the repeated applications of the DPO rule. The execution of the recursive rule finishes if $\nexists e^b$ such that (1) is pullback. The execution continues at step 2 even if the DPO rule is not applied.

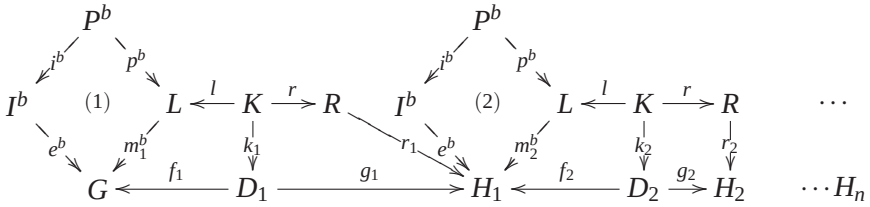


Figure 5: Application of Recursive Rule. Step 1: Base Case.

Second step. The rule is executed for the first step in the recursion. Thus, a match $e_1^r : I^r \to H_n$ (called *recursive match*) is sought such that square (1) in Fig. 6 commutes (i.e. $e^b \circ i^{br} = e_1^r \circ p^{br}$) and makes P^{br} a pullback object. As in the base case, a rule match $m_{1,1}^r : L \to H_n$ is sought for L such that $e_1^r \circ i^r = m_{1,1}^r \circ p^r$ and makes P^r a pullback object, as square (2) shows. If match $m_{1,1}^r$ satisfies the constraints given by Proposition 2 (which guarantee the preservation of e^b), the DPO rule is executed at that match yielding H_{n+1}. As Proposition 1 shows, match e_1^r still exists in H_{n+1}. If an additional match $m_{1,2}^r : L \to H_{n+1}$ is found such that square (3) in Fig. 6 is pullback (and that satisfies Proposition 2), we apply the rule at that match. The operation is repeated until no such rule match is found. In addition, the process is repeated for additional recursive matches $e_1'^r$ commuting with e^b and making P^{br} a pullback object. This can be done as e^b is preserved by the DPO rule applications. The output of this step is a graph H_m resulting from the DPO rule executions, together with the set of recursive matches: $\{e_1^r, e_1'^r, \ldots\}$. The execution

of the recursive rule stops if $\nexists e_1^r$ such that (1) is pullback. Even if the DPO rule is not applied, the recursive rule continues in step 3.

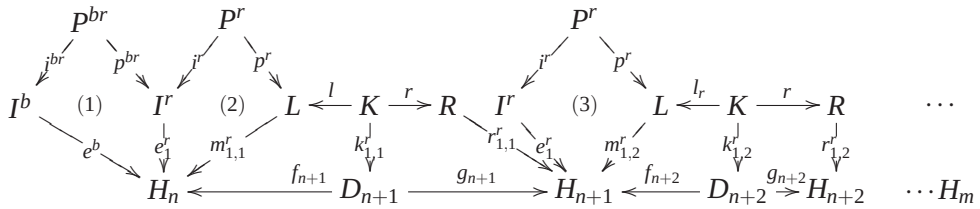


Figure 6: Application of Recursive Rule. Step 2: First Recursive Call.

Third step. We execute the following steps in the recursion. The idea is similar to step 2, but starting from $I^r \xleftarrow{i^{rr}} P^{rr} \xrightarrow{p^{rr}} I^r$ instead of $I^b \xleftarrow{i^{br}} P^{br} \xrightarrow{p^{br}} I^r$. Recursive step $i+1$ is applied for each recursive match provided by previous recursive step, $\{e_i^r, e_i'^r, ...\}$. Each DPO rule application must preserve all these matches, thus each rule match $m_{i+1,j}^r$ must satisfy the conditions of Proposition 2 for each match provided by previous recursive step. The recursive steps are executed as long as a match e_{j+1}^r is found for the next step in the recursion such that square (1) in Fig. 7 is pullback.

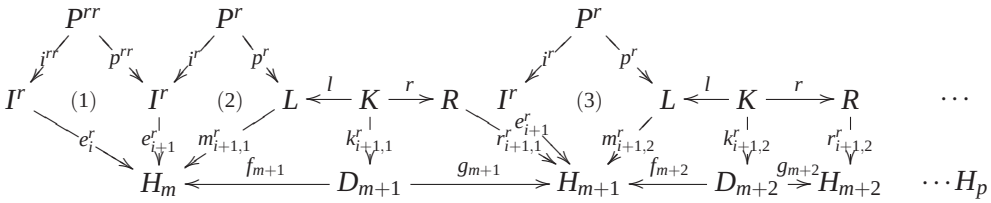


Figure 7: Application of Recursive Rule. Step 3: Successive Recursive Calls.

Definition 5 *(DPO recursive rule derivation) Given a DPO recursive rule* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$*, an object* G*, and a morphism* $e^b : I^b \to G$*. A DPO recursive rule derivation* $G \xRightarrow{p_r, e^b} H_p$ *is built as follows:*

1. *The first step is given by the diagram in Fig. 5, yielding graph* H_n *and base match* e^b*. (written* $G \xRightarrow{(e^b, m_1^b)} H_1 \xRightarrow{*} H_n$ *).*
2. *The second step is given by the diagram in Fig. 6, yielding graph* H_m *and (a possibly empty) set of recursive matches* $\{e_1^r, e_1'^r, ...\}$ *(written* $H_n \xRightarrow{(e_1^r, m_{1,1}^r)} H_{n+1} \xRightarrow{*} \cdots H_k \xRightarrow{(e_1'^r, m_{1,1}'^r)} \cdots H_m$ *)*
3. *The third step is given by the diagram in Fig. 7, for each recursive match coming from the previous application* $\{e_j^r, e_j'^r, ...\}$*, yielding graph* H_p *and a (possibly empty) set of recursive matches* $\{e_{j+1}^r, e_{j+1}'^r, ...\}$ *(written* $H_m \xRightarrow{(e_j^r, m_{j,1}^r)} H_{m+1} \xRightarrow{*} \cdots H_s \xRightarrow{(e_j'^r, m_{j,1}'^r)} \cdots H_p$ *)*

This third step is repeated until the set of newly found recursive matches is empty.

Remarks: The recursive rule execution starts at a unique match e^b. However, in each recursive step, the execution considers all morphisms e''^r_n. In a recursive step, the match is passed between I^b and I^r, and between I^r and I^r. This is because we want to continue the traversal of the structure, even if no rule match for L is found at some step. As the DPO rule is executed as long as possible in every step, and as we seek e^r_{i+1} morphisms as long as possible[1], it is possible to have non-terminating derivations, written $G \overset{p_r, e^b}{\Longrightarrow} \infty$. The DPO recursive rule is applied if the DPO rule is applied at least once.

Definition 3 is modified in a straightforward way, by allowing productions in set P to be either standard DPO rules, or DPO recursive rules. In a derivation, each step can be given by a standard derivation or a recursive one.

Next, we show that morphism e^b can be extended to the resulting graph of applying the DPO rule. The discussion for morphism e^r is similar, so we only present the case for e^b.

Proposition 1 *(Morphism Extension for Recursive Rule) Given the diagram in Fig. 8(a) (with (1) pullback, (3) and (4) pushouts, i^b, p^b, l and $r \in \mathscr{M}$), morphism e^b can be extended to H_1.*

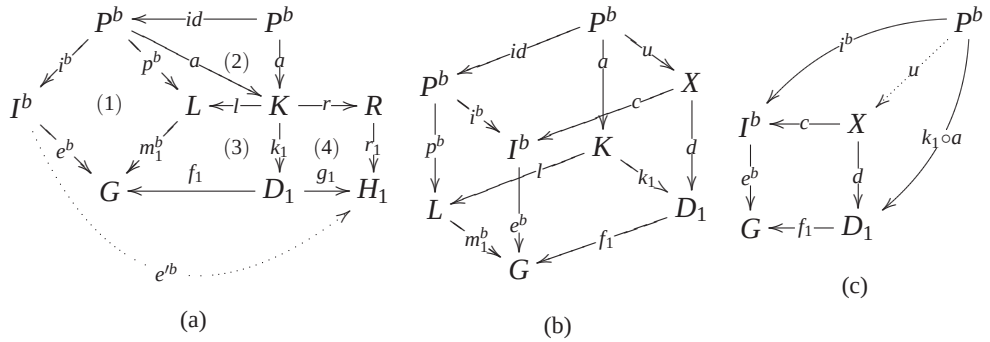


Figure 8: (a) Extension of e^b Morphism. (b) Showing that (3) is a van Kampen Square. (c) Universal Pullback Property.

Proof: First, note that (2) is a pullback. In order to show the existence of a morphism $e'^b\colon I^b \to H_1$, we use the fact that pushout (3) is a van Kampen square. For that purpose, we build a cube taking (3) at the bottom, and pullbacks (2) and (1) as back-left and front-left faces (see Fig. 8(b)). We close the cube by calculating the pullback of $e^b\colon I^b \to G$ and $f_1\colon D_1 \to G$, given by $(X, c\colon X \to I^b, d\colon X \to D_1)$. By the universal property of pullbacks (see Fig. 8(c)), there is a unique arrow $u\colon P^b \to X$ (as $e^b \circ i^b = f_1 \circ k_1 \circ a$). P^b is the pullback object of k_1 and d by the pullback composition and decomposition lemma. The four lateral faces are pullbacks, and (3) is a pushout along $\mathscr{M}$ ($l, f_1 \in \mathscr{M}$), and therefore a van Kampen square by definition of AHLR

[1] Due to the possible presence of cycles in the graph to be traversed. This can be controlled with NACs in the recursive condition, see the end of the section, and the last example in the paper.

category. Thus the top square is a pushout as well. In particular $c\colon X \to I^b$ is an isomorphism as the oposite arrow is also an isomorphism. Therefore, we obtain $e'^b = g_1 \circ d \circ c^{-1}$ □

Note that in the first and successive recursive steps, the DPO rule can be applied as long as possible. Moreover, after such rule applications, an additional match $e_i'^r$ is sought in order to repeat the rule applications. Therefore, a DPO rule application in the first recursive step has to preserve e^b (see Figure 6). In a similar way, in recursive step $i+1$ a DPO rule application has also to preserve match e_i^r (see Figure 6) as well as the set of matches output by previous recursive step. The conditions for this preservation are stated in next proposition. We only show the preservation of the base morphism e^b in the first recursion step, as the other cases are similar.

Proposition 2 *(Preservation of Base Match in First Recursive Step) Match $e^b\colon I^b \to H_n$ is preserved after a standard rule application through match $m^r_{1,1}\colon L \to H_n$ (as Figure 9(a) shows) if $\exists b^r\colon P \to P^r$ where $(P, b^b\colon P \to I^b, b^l\colon P \to L)$ is the pullback of $e^b\colon I^b \to H_n$ and $m^r_{1,1}\colon L \to H_n$ (as Figure 9(b) shows).*

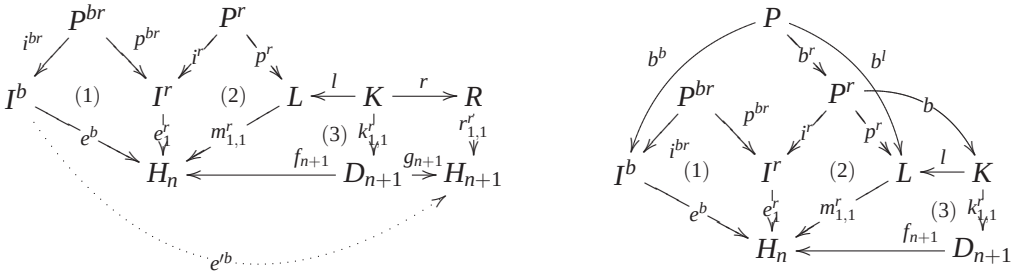


Figure 9: (a) Extension of e^b in Recursive Step One. (b) Condition for Standard Rule Derivation.

Proof: Using the fact that (3) is a van Kampen square, by using the pullback square spawned by P as front left face, taking $b \circ b^r\colon P \to K$, and then following the construction shown in Proposition 1. □

Remarks: In the general case P is not isomorphic to the pullback object of $p^{br}\colon P^{br} \to I^r$ and $i^r\colon P^r \to I^r$. If the conditions stated in this proposition are not satisfied, then the DPO rule cannot be applied.

Example. Fig. 10 shows a derivation of the DPO recursive rule shown in Fig. 4. First, a match of the base condition I^b is identified in G. The match is given by the elements labelled with the same numbers as in I^b, which in addition are coloured. Next, a match of the LHS is sought in G through the elements identified by I^b (i.e. the roles matched by the base condition are the ones used in the math of the LHS). One match is found for which the rule is applied, yielding graph H1. The rule deletes a permission for a role if its parent already defines it (as the base condition identified). Since no other match of the LHS is found in the graph for the base condition, the first recursive step starts. A match of I^r in the graph is found that maintains the match for the role labelled as "1", and identifies the role labelled as "2" with its direct child. Next, a match

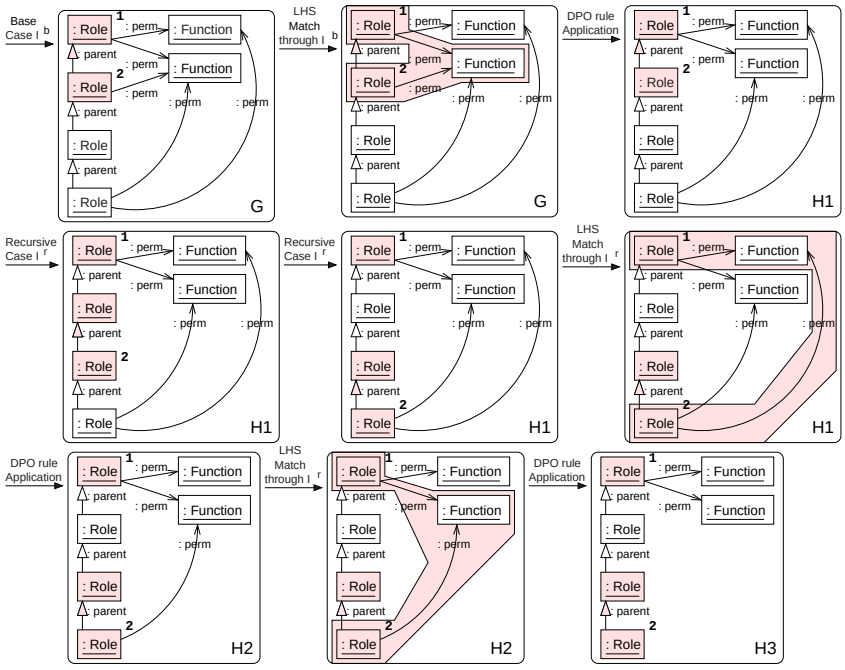


Figure 10: A DPO Recursive Derivation.

of the LHS is sought on the graph through the roles newly labelled "1" and "2", but none is found. Thus, a new step in the recursion starts. A new match of I^r is searched in the graph that maintains the match for role "1" and identifies role "2" with its direct child. Now, two matches of the LHS are found (i.e. the parent role has two permissions which role "2" also defines). In a first application of the DPO rule, one of the permissions is deleted yielding graph H2. A second application of the DPO rule at the other match deletes the second redundant permission yielding graph H3. Since neither matches of the LHS nor matches of the recursive condition are found, the derivation concludes.

Application Conditions. Previous derivation has started in the top-most role in the hierarchy. However, the initial match e^b could also have identified other roles as base condition. In order to identify the top-most role, we need a NAC associated to I^b, which forbids the match if the role has some parent. We extend recursive rules with NACs for I^b, I^r and the DPO rule.

Definition 6 *(DPO recursive rule with NACs) A recursive rule with NACs* $p_r = (L \xleftarrow{l} K \xrightarrow{r} R, NAC_L, I^b, NAC_{I^b}, I^r, NAC_{I^r}, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *is made of a a recursive rule* $(L \xleftarrow{l} K \xrightarrow{r} R, I^b, I^r, (I^j \xleftarrow{i^j} P^j \xrightarrow{p^j} L)_{j\in\{b,r\}}, (I^j \xleftarrow{i^{jr}} P^{jr} \xrightarrow{p^{jr}} I^r)_{j\in\{b,r\}})$ *and three sets* $NAC_J = \{(X^J, x_i^J : J \to X_i^J)\}$ *(for* $J \in \{L, I^b, I^r\}$*) of NACs for* L*,* I^b *and* I^r*.*

Previous definition modifies the concept of derivation in the following way. A valid match $e^b : I^b \to G$ has to satisfy $NAC(x_j^{I^b})$, $\forall (X_j^{I^b}, x_j^{I^b}) \in NAC_{I^b}$, and similar for I^r and L.

5 Additional Examples

This section shows further examples in the category of attributed typed graphs [EEPT06] (also an AHLR category). In this category, objects are typed graphs, with edge and node attribution.

Model Transformation. Transforming class diagrams into relational data base models is a common case study when studying the expressivity of model transformation languages [EGL+05]. It requires handling complex structures such as inheritance hierarchies. This transformation implies mapping each persistent class to a table, and all its attributes and associations to columns in this table. However, only top-most classes in the inheritance hierarchy have to be mapped into tables; additional attributes and associations of subclasses result in additional columns of the top-most classes. For the present example, we only consider this excerpt of the problem and restrict to transformation of attributes with a primitive data type.

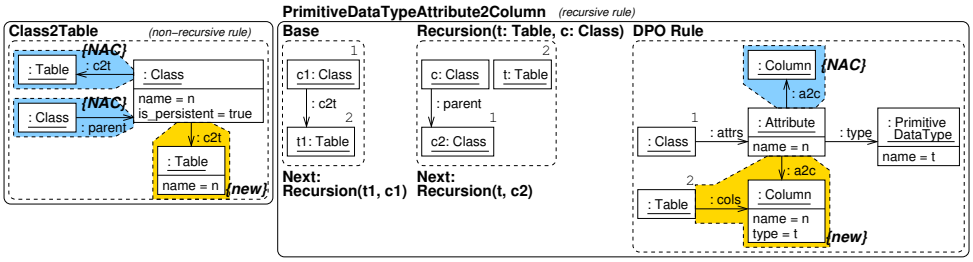


Figure 11: DPO Recursive Rules for Model Transformation.

Using standard DPO rules to perform the transformation would possibly imply the flattening of the hierarchy (see the solutions proposed in [EGL+05]). This is not optimal as it requires modifying the type graph (the meta-model), together with pre- and post-processing rules. We can avoid this flattening by using a DPO recursive rule as the one shown in Fig. 11. The first rule in this figure is a DPO standard rule that creates a table for each top-most class in the hierarchy. The second rule is recursive. The base condition identifies a top-most class in the hierarchy (the one with an associated table). The DPO rule maps a column to each attribute of this class, if such column has not been created before. The column is added to the table mapped to the class. The recursive condition goes down the class hierarchy, maintaining the match of the table. In this way, the application of the rule in the recursive steps creates a column in the table for each attribute of the descendant classes of the base class.

Model Simulation. Fig. 12 shows a recursive rule that updates a network of two-input logical gates when one of the inputs changes. The network of gates can be seen as a recursive structure, which must be traversed to update the output of the gates in each step. The type graph contains a node *Bit*, connected to a *Gate* by relations *in* and *out*. Bits have a self-loop to indicate that their value has been changed externally, and an attribute *value* of type *bit*. Gates have attribute *operation*, an enumerate type with values *or* and *and*, indicating the operation it performs.

In the DPO recursive rule, the base condition looks for a bit that has been changed (i.e. it has a self-loop). The DPO rule updates the output of the gate[2]. An application condition makes the DPO rule applicable only if the value of the output bit has to be updated. We add this attribute

[2] "value=[c, op(a,b)]" denotes that "value" changes from "c" to "op(a,b)" due to the DPO rule application.

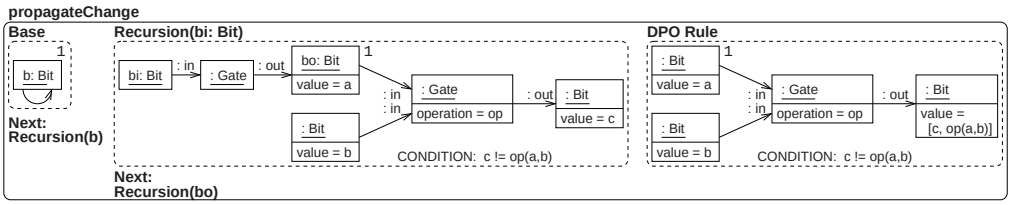


Figure 12: DPO Recursive Rule for Model Simulation.

condition as we may have loops in the network, so the rule must not be applicable if a bit has already the correct value. The recursive condition advances through the network of gates and has a condition which prohibits finding a match if the output gate already has a correct value. Thus, rule application ends when all the bits dependent on the changed bit are updated.

Solving this problem using DPO simple rules would probably imply encoding control elements in the graph to guide rule application. These control elements have to mark the bits to be recalculated, and take care that this calculation occurs in the right order.

6 Conclusions and Future Work

We have presented a novel approach for modelling recursive rules in the DPO approach. The main idea is to provide DPO rules with base and recursive conditions, together with mechanisms to pass the matching between successive recursion steps. The execution mechanism performs a width-first traversal of the recursive structure, while guaranteeing its preservation by the DPO rule applications. We have shown the utility of the approach for structures that can be recursively dealt with, such as inheritance hierarchies, networks of components, etc. We have presented several examples in the areas of model simulation, model optimization and model transformation. The solution of the presented problems using DPO simple rules would imply either the *flattening* of the structure by adding helper edges, or encoding control elements in the host graph to guide rule execution. As we showed, both solutions are not optimal as they imply modifying the type graph and defining pre- and post-processing rules. We believe our DPO recursive rules are a solution to this problem. Moreover, the presented techniques may serve as the basis for a formal rule execution control language with parameter passing.

There are several useful extensions to this approach. The first one is having more than one recursive condition (i.e. more than one I^r). This is useful if the structure to be traversed is not uniform, but it is made of different edge types. The second one is having more than one DPO rule in a recursive rule. This is useful if slightly different actions have to be performed at each step in the recursion, and allows for indirect recursion. It is also worth studying termination and conflicts of DPO recursive rules, as well as different execution policies. Moreover, given a recursive rule, we are investigating the construction of a (possibly infinite) set of standard DPO rules, such that the execution of one of them is equivalent to the execution of the recursive rule.

Acknowledgements: This work has been partially sponsored by the Spanish Ministry of Education and Science with projects MOSAIC (TSI2005-08225-C07-06) and MODUWEB (TIN

2006-09678). The authors gratefully thank the referees for their useful suggestions.

References

[BV06] A. Balogh, D. Varró. Advanced Model Transformation Language Constructs in the VIATRA2 Framework. In *Proc. ACM SAC'06*. Pp. 1280–1287. 2006.

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

[EGL+05] K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovszky, U. Prange, G. Taentzer, D. Varró, S. Varró-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *MTiP 2005, (Satellite Event of MoDELS 2005)*. 2005.

[EJ04] N. V. Eetvelde, D. Janssens. Extending Graph Rewriting for Refactoring. In *Proc. ICGT'04*. LNCS 3256, pp. 399–415. Springer, 2004.

[KASS03] G. Karsai, A. Agrawal, F. Shi, J. Sprinkle. On the Use of Graph Transformation in the Formal Specification of Model Interpreters. *JUCS* 9(11):1296–1321, 2003.

[KBC05] A. Kalnins, J. Barzdins, E. Celms. Model Transformation Language MOLA: Extended Patterns. In *Proc. DB&IS'2004*. Volume 118, pp. 169–184. IOS Press, 2005.

[LS04] S. Lack, P. Sobocinski. Adhesive Categories. In Walukiewicz (ed.), *FoSSaCS*. LNCS 2987, pp. 273–288. Springer, 2004.

[LV04] J. de Lara, H. Vangheluwe. Defining visual notations and their manipulation through meta-modelling and graph transformation. *JVLC* 15(3-4):309–330, 2004.

[Min02] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Sci. Comput. Program.* 44(2):157–180, 2002.

[NNZ00] U. Nickel, J. Niere, A. Zündorf. The FUJABA environment. In *Proc. ICSE '00*. Pp. 742–745. ACM Press, 2000.

[OMG] OMG. QVT Specification at:http://www.omg.org/docs/ptc/05-11-01.pdf.

[Roz97] G. Rozenberg (ed.). *Handbook of graph grammars and computing by graph transformation: volume I. foundations*. World Scientific Publishing Co., Inc., 1997.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. *The PROGRES approach: language and environment*. World Scientific Publishing Co., Inc., 1999.

[Tae96] G. Taentzer. Parallel and Distributed Graph Transformation. Formal Description and Application to Communication-Based Systems. Shaker Verlag, 1996.

[Wil03] E. D. Willink. A concrete UML-based graphical transformation syntax: The UML to RDBMS example in UMLX. Metamodelling for MDA, York, England, 2003.

Visual Programming with Recursion Patterns in Interaction Nets

Ian Mackie[1,2], **Jorge Sousa Pinto**[3] **and Miguel Vilaça**[3]

[1] LIX, CNRS UMR 7161, École Polytechnique, 91128 Palaiseau Cedex, France
[2] King's College London, Department of Computer Science, Strand, London WC2R 2LS, U.K.
[3] Departamento de Informática, Universidade do Minho , 4710 Braga, Portugal

Abstract: In this paper we propose to use Interaction Nets as a formalism for Visual Functional Programming. We consider the use of recursion patterns as a programming idiom, and introduce a suitable archetype/instantiation mechanism for interaction agents, which allows one to define agents whose behaviour is based on recursion patterns.

Keywords: Interaction nets, recursion patterns

1 Introduction

This paper is about visual programming with Interaction Nets, a graph-rewriting formalism introduced by Lafont [Laf90], inspired by Proof-nets for Multiplicative Linear Logic. In Interaction Nets, a program consists of a number of interaction rules and an initial net that will be reduced by repeated application of the rules. The formalism combines the usual advantages of visual programming languages, but with the following additional features:

- Programs and data structures are represented in the same framework, which is useful for tracing and debugging purposes;
- All aspects of computations, such as duplication and garbage-collection, are explicit.

Interaction Nets have been extensively used for functional programming as an efficient intermediate (or implementation) language. In particular, functional programs can be *encoded* in Interaction Nets, using one of the many encodings of the λ-calculus. Section 3 reviews how a functional language can be encoded in Interaction Nets following this approach, without entering the details of any particular encoding of β-reduction. The focus of this paper will be the adequate treatment of inductive datatypes, pattern-matching, and recursive function definitions.

The remaining sections of the paper introduce and systematise the use of a functional style for programming with Interaction Nets with *recursion patterns*, and introduce a new construct (the *archetype*, Section 4) which captures the behaviour of recursion patterns. We claim that this style is a good choice for defining and executing visual functional programs.

The style of programming we refer to is widely used by functional programmers: it is based on programs that perform iteration on their arguments, usually known in the field as *folds*, and (the dual notion) programs that construct results by co-iteration, *unfolds*. Among other advantages of using folds and unfolds for programming, they can be composed to construct complex recursive programs, and they are particularly adequate for equational reasoning: proofs of equality can be done using a *fusion law* instead of recursion.

The body of theoretical work on recursion patterns comes from the field of *datatype-generic programming* (see [Gib02] for an introduction), which studies these patterns in a datatype-

parameterized way. The examples in the paper use lists, but it is straightforward to generalize the ideas to arbitrary regular datatypes.

Section 5 introduces interaction net programming with recursion patterns; Section 6 and Section 7 then present archetypes for folds and unfolds respectively. Section 8 concludes and discusses future work.

2 Background

Recursion Patterns. The ideas developed in this paper for Interaction Nets are very much inspired by Functional Programming. One fundamental aspect that we will use extensively is the ability to use a set of *recursion patterns* for each datatype. For instance few Haskell programmers would write a list sum program with explicit recursion as

```
sum [] = 0
sum (x:xs) = x + (sum xs)
```

Most would define `sum = foldr (+) 0`, where `foldr` is a recursion pattern encoded as the following higher-order function:

```
foldr f z []     = z
foldr f z (x:xs) = f x (foldr f z xs)
```

A function like `sum` is often called a *fold*. The use of recursion patterns has the advantage of being appropriate for program transformation and reasoning using the so-called *calculation-based* style [Bir84]. To see how less obvious folds can be defined, consider the list *append* function:

```
app :: [a] -> [a] -> [a]
app     []  l = l
app (x:xs)  l = x:(app xs l)
```

This is a higher-order fold that iterates over its first argument to produce a function (`id` is the identity function): `app = foldr (\x r l -> x:(r l)) id`.

The dual notion of fold is the co-recursive *unfold* that allows one to produce lists whose tails are constructed recursively by the function being defined. For instance the Haskell function

```
downfrom 0     = []
downfrom (n+1) = (n+1):(downfrom n)
```

can be written alternatively as `downfrom = unfold (==0) id pred` where `pred` is the predecessor function, and `unfold` is defined as follows

```
unfold :: (t -> Bool) -> (t -> a) -> (t -> t) -> t -> [a]
unfold p f g x = if p x then [] else f x : unfold p f g (g x)
```

One of the reasons why unfolds are important [GJ99] is that together with folds they give us back the power of arbitrary recursion: the composition of a fold with an unfold is a function (known as a *hylomorphism* [MFP91]) whose recursion tree is the intermediate structure constructed by the unfold. In a language with a sufficiently rich type system, most useful recursive functions can be defined in this way.

Interaction Nets. An interaction net system [Laf90] is specified by giving a set Σ of symbols, and a set $\mathscr{R}$ of interaction rules. Each symbol $\alpha \in \Sigma$ has an associated (fixed) *arity*. An occurrence of a symbol $\alpha \in \Sigma$ will be called an *agent*. If the arity of α is n, then the agent has $n+1$ *ports*: a distinguished one called the *principal port*, and n *auxiliary ports* labelled $x_1, \ldots, x_n$.

A net built on Σ is a graph (not necessarily connected) where the nodes are agents. The edges between nodes of the graph are connected to *ports* in the agents, such that there is at most one edge connected to every port in the net. Edges may be connected to two ports of the same agent. Principal ports of agents are depicted by an arrow. The ports of agents that have no edge connected are called the *free ports* of the net. The set of free ports define the *interface* of the net.

The dynamics of Interaction Nets are based on the notion of *active pair*: any pair of agents (α, β) in a net, with an edge connecting together their principal ports. An *interaction rule* $((\alpha, \beta) \Longrightarrow N) \in \mathscr{R}$ replaces an occurrence of the active pair (α, β) by the net N. Rules must satisfy two conditions: the interfaces of the left-hand side and right-hand side are equal (this implies that the free ports are preserved during reduction), and there is at most one rule for each pair of agents, so there is no ambiguity regarding which rule to apply.

If a net does not contain any active pairs then we say that it is in normal form. We use the notation $\Longrightarrow$ for one-step reduction and $\Longrightarrow^*$ for its transitive reflexive closure. Additionally, we write $N \Downarrow N'$ if there is a sequence of interaction steps $N \Longrightarrow^* N'$, such that N' is a net in normal form. The strong constraints on the definition of interaction rules imply that reduction is strongly commutative (the one-step diamond property holds), and thus confluence is easily obtained. Consequently, any normalizing interaction net is strongly normalizing.

As an example, we show a system for representing lists of numbers. Lists are inductively defined by an agent Nil of arity 0 representing the empty list, and an agent Cons of arity 2 representing a cell in the list, containing an element and a tail list. Lists are constructed such that the principal port of each Cons agent corresponds to the root of the list.

To implement, for instance, list concatenation, we need an additional binary agent app. Concatenation is defined recursively on one of the argument lists, as expected. As such, the principal port of the agent must be used for interacting with this argument. The necessary interaction rules are given in Figure 1, together with an example net, representing the concatenation of lists [1,2] and [3,4].

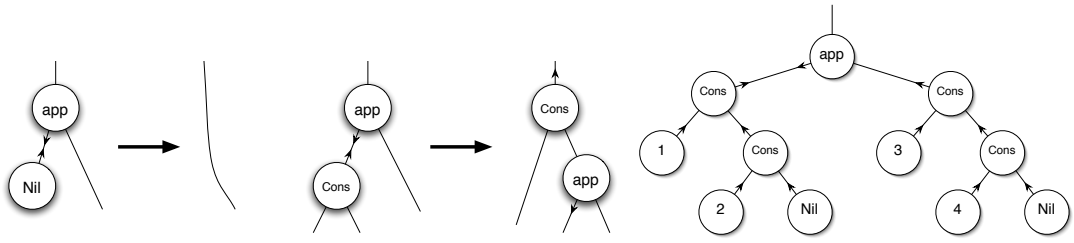


Figure 1: Interaction rules of agent app and an example net

Thus, an implementation of list concatenation can be obtained by Σ containing $\{\mathsf{Nil}, \mathsf{Cons}, \mathsf{app}\}$, with arity 0, 2, 2 respectively, and $\mathscr{R}$ consisting of the rules in Figure 1.

Related Work: Visual Functional Programming. Work in this area has addressed different aspects of visual programming. The Pivotal project [Han02] offers a visual notation (and Haskell programming environment) for data structures, not programs. Reekie's Visual Haskell [Ree95] more or less stands at the opposite side of the spectrum of possibilities: this is a dataflow-style visual *notation* for Haskell programs, which allows programmers to define their programs visually (with the assistance of a tool) and then have them translated automatically to Haskell code. Kelso's VFP system [Kel02] is a complete environment that allows one to define functional programs visually and then reduce them step by step. Finally, VisualLambda [DV96] is a formalism based on graph-rewriting: programs are defined as graphs whose reduction mimics the execution of a functional program. As far as we know none of these systems is widely used.

Visual Haskell and VisualLambda have in common that functions are represented as boxes with input ports for the arguments and an output port for the result; the contents of the box correspond to the body of the function. They differ in that Visual Haskell uses variables to refer to function arguments, while VisualLambda uses a purely graphical notation based on arrows.

Kelso's VFP uses a notation without boxes, more inspired by the traditional representations of functional programs used in implementation-oriented abstract machines (see Section 5). In particular, it allows for named functions but also for λ-abstractions, and an explicit application node exists. Variables are used for arguments, as in Visual Haskell.

Higher-order programming is a fundamental feature of functional programming. A function f can take function g as an argument and g can then applied within the body of f. Expressing this feature is easy if variables are used as in Visual Haskell and VFP; in VisualLambda a special box would be used as a placeholder for g (in the body of f) to be instantiated later, and an arrow would link an input port in the box of f to the box of g.

The work presented in the present paper uses a pure visual representation of programs, without variables. In this aspect it resembles VisualLambda, however our work differs significantly from this in that no boxes are used, and all the graph-rewriting operations are *local* in the sense that only two nodes of the graph are involved in each step.

A second difficulty arising from the higher-order nature of programs is that a (curried) function of two arguments may receive only its first argument and return as result a function. In a box-based representation this means that it must be possible for a box to lose its input ports one by one—a complicated process. Interaction nets treat this problem naturally as will become clear. Moreover in this paper we introduce a new notion (the archetype), which captures precisely the behaviour of many typical curried functions.

3 Visual Functional Programming with Interaction Nets Using Explicit Abstraction and Application Nodes

In this section we explain how a very simple functional programming language can be *encoded* in Interaction Nets. The language has inductive types, pattern-matching on these types, and explicit recursion.

We first review the basic principle shared by most well-known encodings [Mac04, Mac98, Sin06] of the λ-calculus into Interaction Nets, and show how this basic language can easily be extended to cover other features of functional languages.

The usual way of representing functional programs with interaction nets is based on a pair of symbols λ, @ of arity 2, such that a β-reduction step corresponds to an interaction between an agent λ and an agent @. These representations are based on an explicit depiction of the λ-abstractions in the program, as well as applications of functions to arguments.

While this may be visually more complicated than the boxes used by some of the systems reviewed in Section 2, it certainly solves the "higher-order" problem in a natural way, since function arguments are treated like any other arguments. The definition of the application function `ap f x = f x` in Figure 2 (left) illustrates this point.

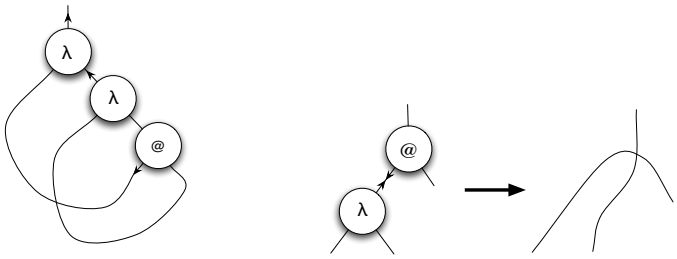


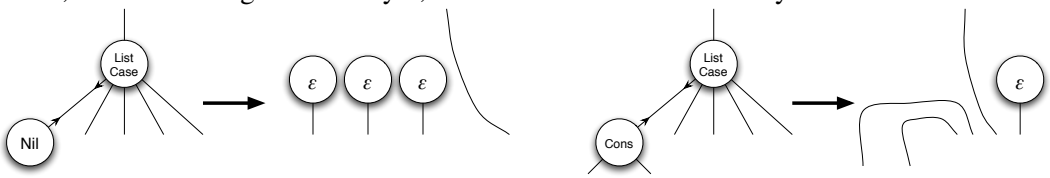
Figure 2: `App` definition (left) and β rule (right)

We only discuss the features of the linear λ-calculus here, which is shared by all encodings. It is beyond the scope of this paper to include the details of the non-linear aspects, but refer the reader to [Mac98, Mac04] for the encodings of the full λ-calculus.

Consider an interaction system containing the symbols $\{@, \lambda\}$ as explained above, as well as the β interaction rule of Figure 2 (right). This system defines the visual programming language for the linear λ-calculus. A visual functional program consists of this interaction system, together with a closed functional expression to be evaluated, represented by a net with a single free port. We now outline how other features can be introduced in the interaction system to enrich this core language.

The next feature is Inductive Types and Pattern-matching. Consider a datatype T with constructors $C_1 \ldots C_n$, with arities $a_1 \ldots a_n$. This can be modelled in a straightforward way by an interaction system containing n agents labelled C_i with arity a_i, $i = 1 \ldots n$; values of type T correspond to closed trees built exclusively from these agents (in a tree all principal ports are oriented in the same direction). In a constructor agent, auxiliary ports are input ports, and the principal port is an output port. An example of this is the datatype of lists with constructors Nil and Cons, as in Figure 1.

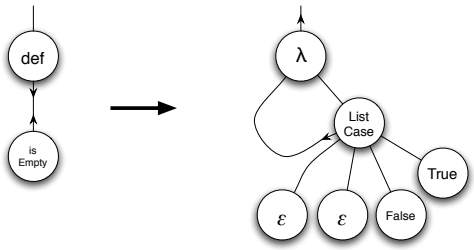
Pattern-matching over an inductive type T can be implemented by a special agent Tcase. For instance, the ListCase agent has arity 5, and its behaviour is defined by the two rules:



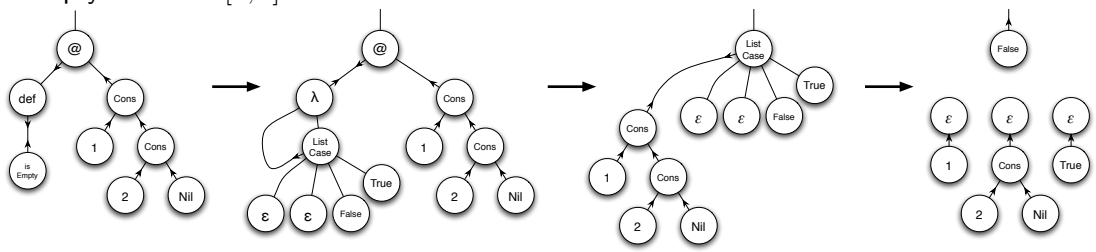
Here two different nets are connected to the ListCase agent. One is a net to be returned when the argument list is empty, and the other is a net with three ports, used to combine the head and

tail of a non-empty list. Observe that one of these nets is not used in each rule, and must be erased with ε agents. The interaction of an ε agent with any other agent erases the latter and propagates new ε agents along its auxiliary ports.

The approach outlined above allows for unnamed functions only. In a visual language one would like to have the possibility of defining named functions, which would most naturally correspond to agents in the interaction system. A special agent def can be used for unfolding named function definitions. For instance a function isEmpty can be defined by the following interaction rule for the agent def:



The following figure shows the reduction of the visual program corresponding to the application of isEmpty to the list $[1,2]$.



This agent also allows for a visually appealing treatment of recursion: it suffices that the right-hand side of interaction rules involving def reintroduces an active pair (the left-hand side of the rule) as a sub-net.

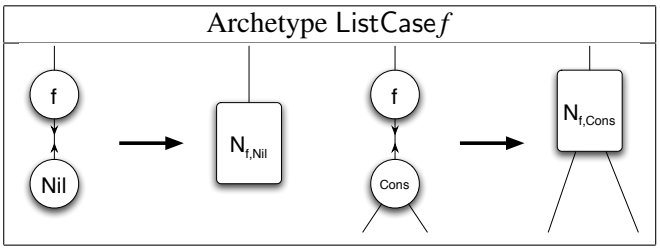
To sum up, Interaction Nets allow to visually represent functional programs and data structures in the same very simple formalism; moreover higher-order features, which are a typical difficulty in the visual setting, are treated in a natural way, and the execution of the program can be efficiently implemented within the formalism.

4 Agent Archetypes

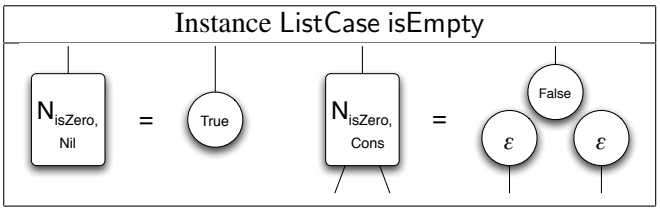
Although no standard programming language exists as a front-end for programming with Interaction Nets, it is generally well accepted that any such language should contain some form of support for modularity and reusability. In particular, a mechanism should exist to facilitate the definition of interaction rules that follow identical patterns.

To illustrate, consider again the app agent of Figure 1. It is defined by case analysis on the structure of the argument, and in fact any other agent defined in this way must have two interaction rules with a similar structure to those in Figure 1. We now introduce a concept designed precisely to isolate this structure, which we designate *archetype*. The ListCase archetype given below should be interpreted as follows: any agent f that fits this archetype interacts with both

Nil and Cons, and the right hand sides of the corresponding rules are nets to be instantiated, that will be called respectively $N_{f,\mathsf{Nil}}$ and $N_{f,\mathsf{Cons}}$.



To define a new agent following the archetype, an *instance* is created, by simply providing the nets in the right-hand side of the interaction rules. This implicitly includes the agent declaration, as well as the instantiated interaction rules for this agent, in the interaction system being defined. As an example, the isEmpty agent (and its behaviour) is defined as an instance of the ListCase archetype. ε agents are used to explicitly *erase* the head and tail of the list, which are not used in the result.



For a second example, take the agent def used in Section 5 for function definitions. We create an archetype for *defined functions*, whose only mandatory rule is for interaction with def, with the right-hand side to be instantiated.

Recursive archetypes are most interesting, and will be very useful in the rest of the paper (in Section 6 a recursive archetype will be given for the app agent). Although archetypes can be useful for programming with interaction nets in general, our examples of using them here concern features of functional programming languages.

The ListCase example shows that archetypes allow for a natural treatment of higher-order concepts: ListCase can be seen as a function that takes certain arguments (the instantiated nets) and returns another function (an agent with its own rules) as result.

5 Interaction Net Programming with Recursion Patterns

Section 3 was about using Interaction Nets for *encoding* functional programs, with the practical goal of producing efficient functional compilers. From the point of view of visual programming, the drawbacks of this approach are that the introduction of explicit abstraction and application nodes complicates the visual representation of programs (the same is true of explicit case constructs), and also raises the matching duplicators problem. Solving this problem implies introducing in the system machinery that destroys the clean visual representation of terms.

Our goal in this paper is to propose a number of principles and extensions for direct visual programming with Interaction Nets, in a *functional style*. To see what we mean by *direct*, consider

again the interaction rules given in Figure 1 . It is easy to see that both define a behaviour for the agent app similar to the standard list concatenation function, which can be written in Haskell as shown in Section 2.

In both cases, a program consists of a collection of function definitions encoded directly as interaction rules in a particular interaction system, together with a closed functional expression to be evaluated in the context of those definitions, represented by a net with a single free port. While in the second approach a function definition corresponds to interaction rules for a special agent def, in Figure 1 there is a direct correspondence between the clauses in the definition of a function f and the interaction rules defining the behaviour of the agent f. A comparison of both definitions reveals that the first approach is visually simpler, and thus more appropriate for representing programs, than the second, standard approach.

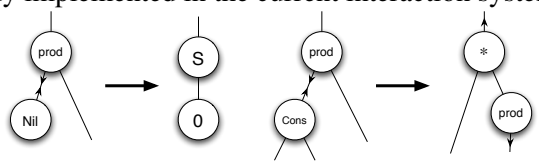
Naturally, there are important limitations to the class of programs with which the simplified representation can be used. The example above takes advantage of a fundamental aspect of Interaction Nets, which is that pattern-matching on the outermost constructor is *built-in* through the rule selection mechanism (in the definition of app in Figure 1, only the outermost constructor is matched). Matching deeper constructors, or matching more than one argument in the same clause, would force us to use an explicit case agent.

For a certain class of patterns (match-sequential systems [Tha87]) there are known transformations which result in a system which examines arguments one at a time. We can use this transformation to obtain an explicit system. We refer the reader to [Ken90] for a detailed presentation of one such transformation.

It will now be shown that the use of a programming style based on recursion patterns allows precisely for the direct representation to be used, since these operators perform pattern-matching on a single top-level constructor.

Iteration: Fold Agents. The simplest form of recursion is iteration, which substitutes the datatype constructors by some given functions. Taking lists as an example, a fold is a function that combines the head of the list with the result of recursively applying the function to the tail of the list, to produce the result (a given value is returned at the end of the list).

Since this requires matching on the outermost constructor only, iteration neatly fits the interaction paradigm. The following is the definition of the *product* fold, which computes the product of all the numbers in a list (it uses an agent $*$ for multiplication of numbers; we assume the arithmetics to be correctly implemented in the current interaction system).

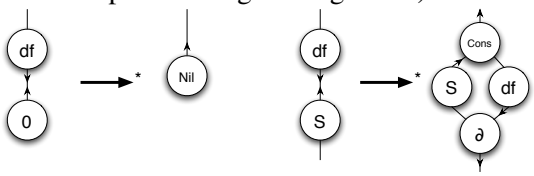


It is easy to see that other, more powerful recursion patterns on an inductive type can be captured in the same way. For instance, *primitive recursion* on a list would allow for the *tail* of the list itself to be used as well.

Co-recursion Patterns: Unfold Agents. *Co-recursive* functions provide structured ways to construct values of recursive types. Taking lists as an example again, the *unfold* co-recursion

pattern, which is the dual of fold, corresponds to functions that construct lists by giving an element to be placed at the head position, together with a seed used as an argument to recursively construct the tail of the list. This is the simplest form of co-recursion.

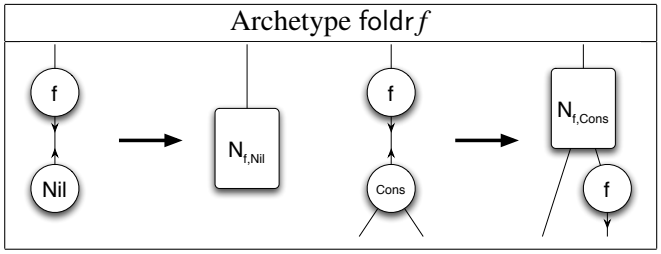
Let us consider an unfold agent *downfrom* (df) which interacts with a natural number n to construct the list containing all the numbers from n down to 1, in this order. Its rules are the following (where ∂ returns two copies of the given argument):



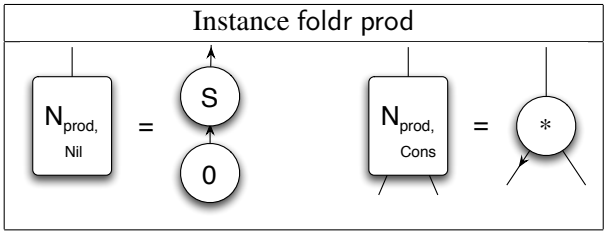
6 Fold Archetypes

The rules that characterize the behaviour of a fold agent (on a particular inductive type) can be described by a *recursive* archetype, in the sense that the parameterized agent occurs in the right-hand side of one of the rules.

Taking the case of lists, interaction rules must be defined for f to interact with both Nil and Cons. The archetype is:



where interaction with Nil results in an arbitrary net, and interaction with Cons sends an f agent along the tail of the argument list, and a net $N_{f,\mathsf{Cons}}$ then combines the head of the list with the recursive result. As an example, the agent prod can be alternatively defined as the following instance:

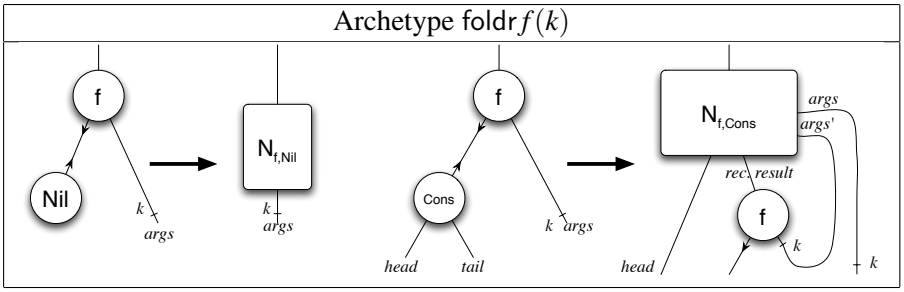


The principles developed above can be applied to folds over any regular inductive type.

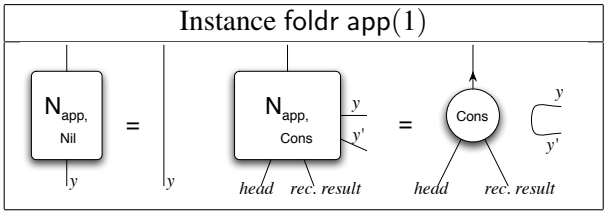
Higher-order Folds. Our current definition of a fold agent is still not satisfactory, and will now be generalized. Consider again the list *append* function. As seen in Section 2, this is a higher-order function of two arguments, defined by recursion on its first argument. This fold can

be defined with Interaction Nets as a binary agent (see Figure 1), which clearly does not match our current definition of the fold archetype.

Functions of more than one argument defined as folds over one of the arguments lead us to the generalization of the foldr archetype, as shown in the following figure. This is parameterized by the number of extra arguments of the fold agent; our previous definition is of course a particular case of this where $k = 0$.

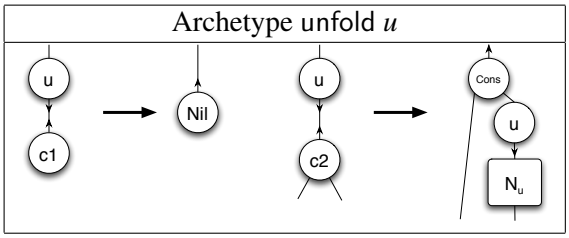


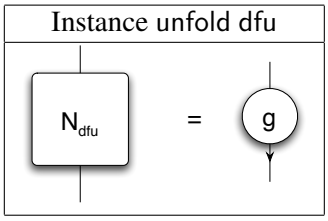
The definition of *append* as an instance of this archetype, for $k = 1$, can be seen below. The open wire in the net $N_{\mathsf{app,Cons}}$ corresponds to the fact that the second argument of the fold is preserved in the recursive call.



7 Unfold Archetypes

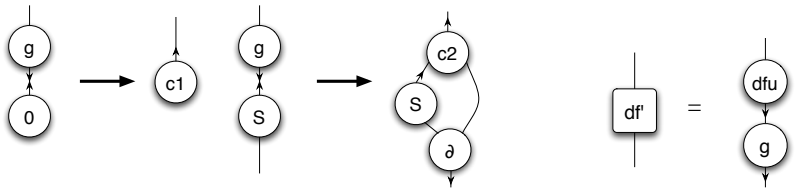
It is less obvious to define an archetype for unfolds. Consider again the case of lists; it is clear that there must be two rules (to produce empty and non-empty lists respectively); the contents of the right-hand side of each interaction rule is also clear (with a parameter net appearing in the second rule). However, since the arguments of an unfold are arbitrary, the two rules cannot correspond to interactions between the unfold and its arguments. Instead, we will consider two special agents that interact with the unfold.





Now observe that an instance of this archetype does not immediately behave as an unfold; it must be connected to the net $\mathsf{N_u}$. To take *downfrom* as an example again, this net is in this case

simply an agent (see right of previous figure) whose behaviour is given by the rules on the left in the following figure. Then the following macro (on the right) can be defined:



8 Conclusions and Future Work

One of our long-term goals is to develop a full environment for interaction net programming—a tool is currently being developed. We are currently working on the archetype definition and instantiation mechanism. Subsequently, we plan to incorporate in the programming environment a mechanism that generates the appropriate archetype for a given user-defined inductive type.

This paper opens a number of other research questions at the theoretical level. One of the main reasons for using recursion patterns and datatype-generic programming is that this style of programming is good for *reasoning* about programs equationally. The work in this paper allows us to reason about functional programs *visually*. In [MPV05] we derive a *fusion law* for the fold archetype, that already makes it possible to transpose to the visual setting classic program transformation techniques such as the introduction of accumulators or tupling.

A different approach that we intend to explore is to establish a formal correspondence between a core functional language with recursion patterns and an interaction system for that language. This will allows us to be more precise in the study of the calculation laws for visual programs.

At the level of the programming environment, the graphical notation then becomes an alternative to writing functional programs textually. The environment should be able to translate between visual programs and textual programs, and all operations performed on programs at the visual level correspond closely to the same operations at the expression level.

Acknowledgements: The work of the third author is financed by FCT (SFRH/BD/18874/2004). This work is partially supported by the British Council Treaty of Windsor Grant: "Visual Programming".

Bibliography

[Bir84] R. S. Bird. The Promotion and Accumulation Strategies in Transformational Programming. *ACM Transactions on Programming Languages and Systems* 6(4):487–504, Oct. 1984.
http://dx.doi.org/10.1145/1780.1781

[DV96] L. Dami, D. Vallet. Higher-Order Functional Composition in Visual Form. Technical report, University of Geneva, 1996.
citeseer.ist.psu.edu/dami96higherorder.html

[Fer98] M. Fernández. Type Assignment and Termination of Interaction Nets. *Mathematical Structures in Computer Science* 8(6):593–636, 1998.

[FM98] M. Fernández, I. Mackie. Coinductive Techniques for Operational Equivalence of Interaction Nets. In *LICS*. Pp. 321–332. 1998.

[Gib02] J. Gibbons. Calculating Functional Programs. In Backhouse et al. (eds.), *Algebraic and Coalgebraic Methods in the Mathematics of Program Construction*. LNCS 2297, chapter 5, pp. 148–203. Springer-Verlag, 2002.

[GJ99] J. Gibbons, G. Jones. The Under-Appreciated Unfold. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming (ICFP '98)*. SIGPLAN 34(1), pp. 273–279. June 1999.

[Han02] K. Hanna. Interactive Visual Functional Programming. In Jones (ed.), *Proc. Intnl Conf. on Functional Programming*. Pp. 100–112. ACM, October 2002. http://www.cs.ukc.ac.uk/pubs/2002/1516

[Kel02] J. Kelso. *A Visual Programming Environment for Functional Languages*. PhD thesis, Murdoch University, 2002.

[Ken90] J. Kennaway. Implementing term rewrite languages in DACTL. *Theoretical Computer Science* 72:225–249, 1990.

[Laf90] Y. Lafont. Interaction Nets. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*. Pp. 95–108. ACM, Jan. 1990.

[Mac98] I. Mackie. YALE: Yet Another Lambda Evaluator Based on Interaction Nets. In *ICFP*. Pp. 117–128. 1998.

[Mac04] I. Mackie. Efficient λ-evaluation with Interaction Nets. In Oostrom (ed.), *Rewriting Techniques and Applications, 15th International Conference (RTA 2004)*. Lecture Notes in Computer Science 3091. Springer, June 2004.

[MFP91] E. Meijer, M. Fokkinga, R. Paterson. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. In Hughes (ed.), *Proceedings of the 5th ACM Conference on Functional Programming Languages and Computer Architecture (FPCA'91)*. LNCS 523. Springer-Verlag, 1991.

[MPV05] I. Mackie, J. S. Pinto, M. Vilaça. Functional Programming and Program Transformation with Interaction Nets. Technical report 05.05.02, Universidade do Minho, 2005.

[Ree95] H. J. Reekie. *Realtime Signal Processing – Dataflow, Visual, and Functional Programming*. PhD thesis, University of Technology at Sydney, 1995.

[Sin06] F.-R. Sinot. Token-Passing Nets: Call-by-Need for Free. *Electr. Notes Theor. Comput. Sci.* 135(3):129–139, 2006.

[Tha87] S. Thatte. A Refinement of Strong Sequentiality for Term Rewriting systems with Constructors. *Information and Computation* 72:46–65, 1987.

Simulating Multigraph Transformations Using Simple Graphs

Iovka Boneva[1], Frank Hermann[2], Harmen Kastenberg[1], and Arend Rensink[1]

[1] bonevai, h.kastenberg, rensink [at] cs.utwente.nl
Department of Computer Science, University of Twente
P.O. Box 217, NL-7500 AE Enschede, The Netherlands

[2]frank [at] cs.tu-berlin.de
Department of Electrical Engineering and Computer Science
Technical University of Berlin, D-10587 Berlin, Germany

Abstract: Application of graph transformations for software verification and model transformation is an emergent field of research. In particular, graph transformation approaches provide a natural way of modelling object oriented systems and semantics of object-oriented languages.

There exist a number of tools for graph transformations that are often specialised in a particular kind of graphs and/or graph transformation approaches, depending on the desired application domain. The main drawback of this diversity is the lack of interoperability.

In this paper we show how (typed) multigraph production systems can be translated into (typed) simple-graph production systems. The presented construction enables the use of multigraphs with DPO transformation approach in tools that only support simple graphs with SPO transformation approach, e.g. the GROOVE tool.

Keywords: graph transformations, graph transformation tools, tool interoperability, multigraphs, simple graphs

1 Introduction

Application of graph transformations for software verification and model transformation is an emergent field of research. In particular, graph transformation approaches provide a natural way of modelling object oriented systems and semantics of object-oriented languages [KKR06] or graphical modelling languages such as the UML [OMG05], see for instance [Hau06].

For performing the actual graph transformations, different approaches are around ranging from hyperedge replacement approach (see e.g. [DKH97]), logic based approach (see e.g. [Cou97]) to different algebraic approaches such as Single Pushout (SPO) [EHK+97] and Double Pushout (DPO) [CMR+97] approach. These different approaches all have specific application areas in which their features are used in an optimal fashion.

Another difference is the use of either multigraphs or simple graphs for modelling the application domain. Whereas the former is more general, the latter suites better when using graphs for representing relations between object in order to reason about these objects using (first-order) logical formulae [Ren04b]. While SPO can be applied for both multigraphs and simple graphs, DPO is not defined for simple graphs in general.

For most tools performing graph transformations the graph representation formalism and the transformation approach are determined by the targeted application domain. For instance, the GROOVE tool [Ren04a] is designed for modelling dynamic systems by generating all possible system configurations and verifying properties about their behaviour. GROOVE uses simple graphs and performs SPO based graph transformations. Another example is the AGG tool [TER99] which handles multigraphs with SPO and is used e.g. for independence and termination analysis on graph grammars.

The main drawback of this diversity in tools is their poor interoperability. One attempt to bridge this gap is the introduction of a common language used for exchanging models among tools, called the Graph eXchange Language (or GXL for short) [SSHW]. In order to extend this work for also exchanging the transformation specifications, GTXL [Tae01] has been proposed. However, since every implementation of a specific approach is not aware of details of other approaches it is very difficult to include all the features in one common standard and thereby enable tools to perform semantically equivalent transformations.

In a previous work [HKM06] we have proposed translations of graph production systems between GROOVE and AGG, but these translations were too specific to be applicable in a more general context. Moreover, these translations were not invertible.

In the current paper, we generalise this translation to a context that is tool independent. We show how one can encode typed multigraph production systems into simple-graph production systems and simulate DPO transformations of multigraphs with DPO transformations on simple graphs. Then we shortly discuss how DPO transformations for simple graphs can be handled by a tool supporting only SPO on simple graphs. These results should allow, for instance, to use the GROOVE tool (or any other tool using simple graphs) with multigraphs. As a further extension, we believe that it would be possible to apply the theory of Subobject Transformation Systems [CHS06] in GROOVE.

Running Example. Throughout this paper we will clarify our ideas and results using a simple example. In the example we model the dynamic behaviour of Lists and Objects that can be elements of some specific Lists. One Object may occur in a List several times. We assume that Objects can be created instantly by the environment (which we do not model in this example). Once Objects are around, different actions can be performed on Lists and Objects like adding Objects to Lists and moving, removing or copying Objects.

Fig. 1 depicts a possible configuration with two Lists: one containing a single Object and another having two entries referring to the same Object. In each configuration we assume that all List- and Object-instances have their own identity, although we do not show these identities.

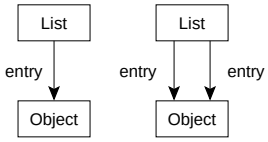


Figure 1: Example configuration of Lists and Objects.

Organisation of the Paper. The remaining of the paper is structured as follows. In Section 2 we provide a formal basis for the rest of the paper. In Section 3 we define our translation of multigraphs to simple graphs and prove the equivalence of DPO transformations on multigraphs on the one hand, and SPO transformations on (special) simple graphs on the other hand. In Section 4 we describe how this equivalence can be extended to typed/labelled graphs. Then, in Section 5 we describe how DPO transformations on (special) simple graphs can be handled by tools implementing the SPO transformation approach, such as the GROOVE tool. Finally, in Section 6 concludes and gives some hints on the way we would like to use the results of this work for improving state space exploration in GROOVE.

2 Background

2.1 Graphs and Graph Morphisms

Graphs are a very powerful means of modelling systems and their behaviour. As will become clear in this paper, in some cases it is very important which notion of graphs are used, since the theory applied may depend on this choice quite heavily.

The *graph* concept is differently interpreted by people working in different domains or even in the same domain. Graphs can e.g. be said to be *deterministic*, *directed* or *labelled*. In this paper we will explicitly distinguish between what we call *multigraphs* and *simple graphs*.

Definition 1 (multigraph, multigraph morphism) A *multigraph* is a tuple $G = \langle V_G, E_G, \mathsf{src}_G, \mathsf{tgt}_G \rangle$ where:

- V_G is a set of *nodes* (or vertices);
- E_G is a set of *edges*;
- $\mathsf{src}_G, \mathsf{tgt}_G : E_G \to V_G$ are *source* and *target* functions.

A *multigraph morphism* $f : G \to H$ is a pair $\langle f_V, f_E \rangle$, where $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ are functions compatible with src and tgt functions, i.e.

- $f_V \circ \mathsf{src}_G = \mathsf{src}_H \circ f_E$;
- $f_V \circ \mathsf{tgt}_G = \mathsf{tgt}_H \circ f_E$.

Definition 2 (simple graph, simple graph morphism) Let Lab be a finite set of labels. A *simple graph* labelled over Lab is a tuple $G = \langle V_G, E_G \rangle$ where

- V_G is a set of *nodes* (or vertices);
- $E_G \subseteq V_G \times \mathsf{Lab} \times V_G$ is a set of *edges*.

The source and target functions $\mathsf{src}_G, \mathsf{tgt}_G : E_G \to V_G$ are defined for any edge $e = (v, l, v') \in E_G$ by $\mathsf{src}_G(e) = v$ and $\mathsf{tgt}_G(e) = v'$.

A *simple graph morphism* $f : G \to H$ is a pair $\langle f_V, f_E \rangle$, where $f_V : V_G \to V_H$ and $f_E : E_G \to E_H$ are functions compatible with src and tgt functions and with labelling, i.e. for any edge $(v, l, v') \in E_G$, $f_E((v, l, v')) = (f_V(v), l, f_V(v'))$.

In the sequel we will call a graph morphism $f\colon G \to H$ *total* if its components f_V and f_E are total functions, and *partial* if its components are total functions from G' to H, where G' is some subgraph of G. An *injective* morphism is a morphism induced by injective functions. We will denote the set of multigraphs as $\mathscr{MG}$ and the set of simple graphs over Lab as $\mathscr{SG}(\mathsf{Lab})$. Hereafter, we will write *graph* when something holds for both multigraphs and simple graphs.

In our formal definitions we use unlabelled multigraphs and labelled simple graphs. We start with unlabelled multigraphs in order to keep proofs simple. However, all results of the paper can be extended to labelled graphs, as it will be discussed in Section 4. Therefore, our examples will already freely use labels on both nodes and edges.

2.2 Graph Transformations

When modelling system states as graphs, the dynamics of the system can be specified by graph transformations. The changes of states are then described by *graph productions*, also called *graph transformation rules*.

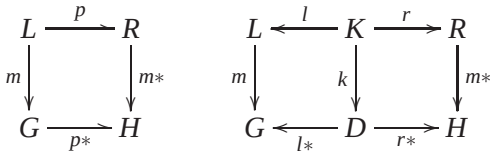
Definition 3 (graph production) A *graph production* p consists of two graphs L and R, being its *left-hand-side* and *right-hand-side*, respectively, together with a partial graph morphism from L to R, called the *rule morphism*.

We often denote a graph production p as $p\colon L \to R$, also using p when referring to the rule morphism. When combining a graph G with a set $\mathscr{P}$ of graph productions, we get a *graph production system* $GPS = \langle G, \mathscr{P} \rangle$. In a graph production system, G is called the *start graph*. By *applying* graph productions to G we can *derive* other graphs. The applications of graph productions are defined on categories in which the objects are a suitable class of graphs and the arrows are the corresponding graph morphisms. For an introduction to category theory, see e.g. [BW95]. Whether a rule is applicable and to what resulting graph a derivation leads, depends on the particular graph transformation approach being applied. In this paper we distinguish between the Single Pushout (SPO) [EHK+97] and Double Pushout (DPO) [CMR+97] approach. For applying a production in the SPO approach, we only need an occurrence of the left-hand-side of the graph production. When the application of a graph production would delete a node but not all of its adjacent edges, those *dangling edges* will also be removed. Furthermore, if the application prescribes one node (or edge) to be both deleted and preserved, this conflict is solved in favour of deletion. These conflicts are resolved in the DPO approach by forbidding such applications of productions, i.e. the DPO approach requires additional conditions on the applications which are called the *dangling edge condition* and the *identification condition* (together referred to as the *gluing condition*).

In the DPO approach, a graph production $p\colon L \to R$ is depicted as a span $L \xleftarrow{l} K \xrightarrow{r} R$ of total graph morphisms, such that $K = L \cap R$, $l\colon dom(p) \to L$, and $r\colon dom(p) \to R$. To be deterministic, it is necessary that either rule morphisms or matchings are injective. We will now define how applications of graph productions and the corresponding derivations for both SPO and DPO.

Definition 4 (application, derivation) The *application* of a graph production $p\colon L \to R$ to a graph G is given by a total graph morphism $m\colon L \to G$, also called a *matching*. The *direct*

derivation from a graph G to a graph H through an application of production p via a matching m, denoted $G \overset{p,m}{\Longrightarrow} H$, is constructed
(SPO) as the pushout of p and m in the considered category of graphs (see Fig. 2(a));
(DPO) by first taking the pushout complement D (with $k\colon K \to D$ and $l*\colon D \to G$) of l and m, if it exists (ensured by the gluing condition), and then taking the pushout of r and k (see Fig. 2(b)).



(a) SPO (b) DPO

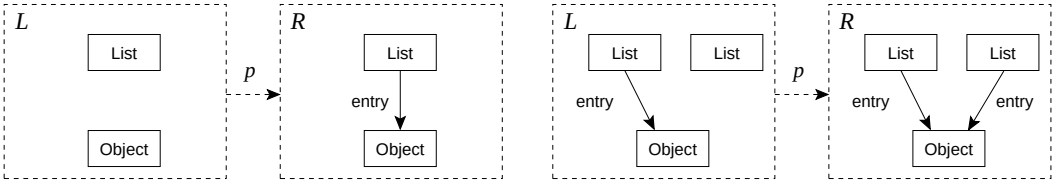
Figure 2: Graph H as the result of an SPO and a DPO derivation.

Intuitively, applying a graph production p to a graph G can be seen as a sequence of two actions: *find* an occurrence of L in G and then *replace* that occurrence by R. This then results in the graph H. An example direct derivation is shown in Fig. 4.

Another important difference between SPO and DPO is the fact that DPO does not work on simple graphs with arbitrary matchings, because in some cases the required pushout construction is not unique or does not exist. In this paper we do apply DPO on simple graphs, but then ensure that we restrict to a special class of matchings and/or morphisms. This issue will discussed in Section 3.

2.3 Back to the Example

Now that we have introduced the notion of graphs and the graph transformation technique, we can recall the example and give a formal description of the actions. In Fig. 3 we specify some of the actions from the example as graph transformation rules by showing their left-hand-side and right-hand-side graph. The rule morphisms in Fig. 3 are defined by the placing of the elements.



(a) add (b) copy

Figure 3: Graph transformation rules for some of the actions in the example.

In Fig. 4 we show a single rule application in which we apply the copy-rule (Fig. 3(b)) on a graph G consisting of two Lists each containing one Object, also showing the resulting graph H.

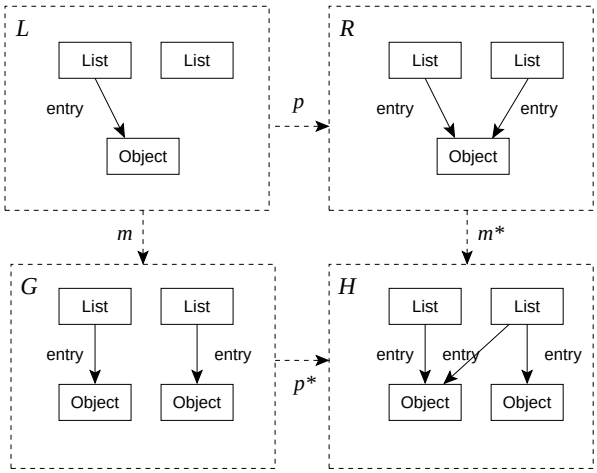


Figure 4: An example direct derivation.

3 From Multigraphs to Simple Graphs and back again

In this section we describe our translation between multigraphs and simple graphs. At a categorical level we will show that these translations are functors which are isomorphisms, moreover being each others inverse.

3.1 From Multigraphs to Simple Graphs

Consider the set of labels $L_{MG} = \{\mathtt{s},\mathtt{t}\}$. The function **Sim** maps multigraphs from $\mathscr{MG}$ into simple graphs in $\mathscr{SG}(L_{MG})$ as follows: every edge e in the multigraph with source node v_s and target node v_t is replaced by a special node v_e (this we call the *proxy* node) and two edges $(v_\mathrm{e},\mathtt{s},v_\mathrm{s})$ and $(v_\mathrm{e},\mathtt{t},v_\mathrm{t})$. Fig. 5 shows an example applying the **Sim** function.

Let $G = \langle V_G, E_G, \mathtt{src}_G, \mathtt{tgt}_G\rangle$. Then $\mathbf{Sim}(G)$ is the graph $H = \langle V_H, E_H\rangle$ with

- $V_H = V_G \cup E_G$;
- $E_H = \bigcup_{v_\mathrm{e}\in E_G}\{(v_\mathrm{e},\mathtt{s},\mathtt{src}_G(v_\mathrm{e})),(v_\mathrm{e},\mathtt{t},\mathtt{tgt}_G(v_\mathrm{e}))\}$.

The **Sim** function can be extended on graph morphisms. That is, if G and H are multigraphs and $m\colon G\to H$ is a morphism, then $\mathbf{Sim}(m)\colon \mathbf{Sim}(G)\to\mathbf{Sim}(H)$ is the morphism defined by:

- for any v in $V_G\cup E_G$ (i.e. $v\in V_{\mathbf{Sim}(G)}$), $(\mathbf{Sim}(m))(v) = m(v)$;
- for any (v,l,v') in $E_{\mathbf{Sim}(G)}$, $(\mathbf{Sim}(m))((v,l,v')) = (m(v),l,m(v'))$.

Note that the definition of $\mathbf{Sim}(m)$ on edges of $\mathbf{Sim}(G)$ ensures that $\mathbf{Sim}(m)$ is indeed a simple graph morphism.

3.2 From Simple Graphs to Multigraphs

Let $\mathscr{SG}_{\mathscr{MG}}$ be the set of bipartite simple graphs over L_{MG} satisfying the following conditions: $G = (V,E)\in\mathscr{SG}_{\mathscr{MG}}$ if

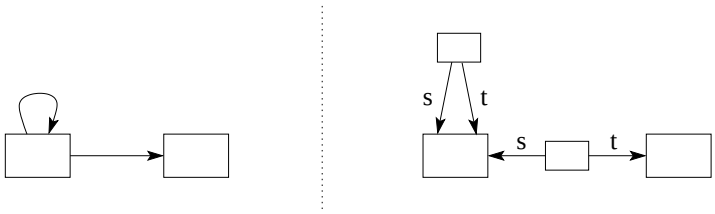


Figure 5: Encoding of a multigraph (on the left) into simple graphs with proxy nodes (on the right) by the **Sim** function.

1. $V = V_{\mathrm{n}} \cup V_{\mathrm{e}}$ where V_{n} and V_{e} are two disjoint sets;
2. $E = E_{\mathrm{s}} \cup E_{\mathrm{t}}$ where E_{s} and E_{t} are disjoint sets and $E_{\mathrm{s}} \subseteq V_{\mathrm{e}} \times \{\mathsf{s}\} \times V_{\mathrm{n}}$, and $E_{\mathrm{t}} \subseteq V_{\mathrm{e}} \times \{\mathsf{t}\} \times V_{\mathrm{n}}$.
3. any node v_{e} in V_{e} has exactly two adjacent edges $(v_{\mathrm{e}}, \mathsf{s}, v'_{\mathrm{n}}) \in E_{\mathrm{s}}$ and $(v, \mathsf{t}, v''_{\mathrm{n}})$ for some $v'_{\mathrm{n}}, v''_{\mathrm{n}} \in V_{\mathrm{n}}$.

We now define the function $\mathbf{Sim}^{-1}\colon \mathscr{SG}_{\mathscr{MG}} \to \mathscr{MG}$ as follows: if $G = \langle V_{\mathrm{n}} \cup V_{\mathrm{e}}, E_G \rangle$ where V_{n} and V_{e} are as in the description of $\mathscr{SG}_{\mathscr{MG}}$ stated above, then $H = \mathbf{Sim}^{-1}(G)$ is the graph $\langle V, E, \mathsf{src}, \mathsf{tgt} \rangle$ such that $V = V_{\mathrm{n}}$, $E = V_{\mathrm{e}}$, and for any $e \in E_H$, $\mathsf{src}(e) = v_{\mathrm{s}}$ and $\mathsf{tgt}(e) = v_{\mathrm{t}}$, where $v_{\mathrm{s}}, v_{\mathrm{t}} \in V_{\mathrm{n}}$ are the nodes such that $(e, \mathsf{s}, v_{\mathrm{s}}), (e, \mathsf{t}, v_{\mathrm{t}}) \in E_G$. We know by condition 3 of the definition of the set of graphs $\mathscr{SG}_{\mathscr{MG}}$ that the nodes v_{s} and v_{t} exist and are unique.

The $\mathbf{Sim}^{-1}$ function can also be extended on graph morphisms. If $m\colon G \to H$ is a simple graph morphism, then $\mathbf{Sim}^{-1}(m)\colon \mathbf{Sim}^{-1}(G) \to \mathbf{Sim}^{-1}(H)$ is the multigraph morphism such that for any x in $V_G \cup E_G$, $(\mathbf{Sim}^{-1}(m))(x) = m(x)$. We now show that $\mathbf{Sim}^{-1}(m)$ defined this way is indeed a multigraph morphism.

Let $G' = \mathbf{Sim}^{-1}(G)$, $H' = \mathbf{Sim}^{-1}(H)$ and $m' = \mathbf{Sim}^{-1}(m)$. Then for any edge $e \in E_{G'}$, $(m' \circ \mathsf{src}_{G'})(e) = m(v_{\mathrm{s}})$ where v_{s} is the unique node in G such that $(e, \mathsf{s}, v_{\mathrm{s}})$ is an edge of G. As m is a simple graph morphism, $(m(e), \mathsf{s}, m(v_{\mathrm{s}}))$ is an edge in H. On the other hand, $(\mathsf{src}_{H'} \circ m')(e) = \mathsf{src}_{H'}(m(e))$ is the unique node v'_{s} in H such that $(m(e), \mathsf{s}, v'_{\mathrm{s}})$ is a edge in H. We deduce then that both $(m(e), \mathsf{s}, v'_{\mathrm{s}})$ and $(m(e), \mathsf{s}, m(v_{\mathrm{s}}))$ are edges in H. By uniqueness of v'_{s}, necessarily $v'_{\mathrm{s}} = m(v_{\mathrm{s}})$, so $m' \circ \mathsf{src}_{G'} = \mathsf{src}_{H'} \circ m'$. On a similar way we can see that $m' \circ \mathsf{tgt}_{G'} = \mathsf{tgt}_{H'} \circ m'$.

It is not very hard to see that $\mathscr{SG}_{\mathscr{MG}}$ is exactly the set of simple graphs that are images of multigraphs by the **Sim** function, and that the function $\mathbf{Sim}^{-1}$ is the inverse of the function **Sim**. This will be formally stated in the following section.

3.3 Categories for Multigraphs and Simple Graphs

In this section we define the categories **MG** and $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$ on which DPO transformation is defined for multigraphs and for simple graphs that are encodings of multigraphs. We show also that the functions **Sim** and $\mathbf{Sim}^{-1}$ define free functors from **MG** to $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$ and from $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$ to **MG** respectively. This will guarantee that performing DPO transformations on multigraphs can be simulated by DPO transformations on simple graphs that belong to $\mathscr{SG}_{\mathscr{MG}}$, as stated in Theorem 1. The reader who is not familiar with category theory will probably only be interested in the result of this theorem.

Definition 5 (categories **MG**, **SG**(L), and **SG**$_{\mathbf{MG}}(L_{MG})$) **MG** is the category whose objects are elements of $\mathscr{MG}$ and whose arrows are multigraph morphisms. **SG**(L) is the category whose objects are simple graphs over the set of labels L and whose arrows are simple graph morphisms. Finally, **SG**$_{\mathbf{MG}}(L_{MG})$ is the category whose objects are elements of $\mathscr{SG}_{\mathscr{MG}}$ and whose arrows are simple graph morphisms.

Note that **SG**$_{\mathbf{MG}}(L_{MG})$ can be equivalently defined as the full subcategory of **SG**(L_{MG}) induced by $\mathscr{SG}_{\mathscr{MG}}$.

Recall that a functor $f = \langle f_{\mathrm{o}}, f_{\mathrm{m}} \rangle$ from a category **C** to a category **D** is a function with f_{o} (resp. f_{m}) associating objects (resp. morphisms) of **D** with objects (resp. morphisms) of **C** and such that f preserves morphisms, identities and composition.

The following lemma easily follows from the definitions.

Lemma 1 *It holds that*

1. **Sim** *is a functor from* **MG** *to* **SG**$_{\mathbf{MG}}(L_{MG})$ *and*
2. **Sim**$^{-1}$ *is a functor from* **SG**$_{\mathbf{MG}}(L_{MG})$ *to* **MG***;*
3. *the functors* **Sim** *and* **Sim**$^{-1}$ *are isomorphisms:*

$$\mathbf{Sim} \circ \mathbf{Sim}^{-1} = ID_{\mathbf{SG}_{\mathbf{MG}}(L_{MG})} \quad \text{and} \quad \mathbf{Sim}^{-1} \circ \mathbf{Sim} = ID_{\mathbf{MG}}.$$

Graph morphisms are called edge reflecting, if edges are reflected along their boundary, i.e. such a morphism f must not map two nodes, if the image nodes are connected by an edge, which is not reached by f.

Lemma 2 *All morphisms* $f : G \to H$ *in* **SG**$_{\mathbf{MG}}(L_{MG})$ *are edge reflecting, i.e.*

$$\textbf{if} \quad (f(x), l, f(y)) \in E_H \quad \textbf{then} \quad (x, l, y) \in E_G.$$

Proof. It is enough to show that **Sim** translates to edge reflecting morphisms, because the categories are isomorphic. By definition, **Sim** translates edges to special nodes with two outgoing edges to other nodes. Nodes in **MG** are connected via structured edges in **SG**$_{\mathbf{MG}}(L_{MG})$, thus edges connect an original node with a proxy node. Let f be a graph morphism in **MG**. If **Sim**(f) reaches a proxy node, f has to map to the original edge. Therefore, also the adjacent edges are reached by **Sim**(f) and thus, **Sim**(f) is edge reflecting. □

3.4 Multigraph versus Simple Graph transformations

In the sequel we combine the graph categories **MG**, **SG**$_{\mathbf{MG}}(L_{MG})$ and **SG**(L_{MG}) with the transformation approaches SPO and DPO. We will denote such combinations with **MG**+DPO etc. The aim of this paper is to translate **MG**+DPO into **SG**(L_{MG})+SPO. This is achieved in two steps:

$$\mathbf{MG}\text{+DPO} \;\to\; \mathbf{SG}_{\mathbf{MG}}(L_{MG})\text{+DPO} \;\to\; \mathbf{SG}(L_{MG})\text{+SPO}$$

The first step consists in encoding multigraphs and production rules using the **Sim** function, thus obtaining simple graphs in $\mathscr{SG}_{\mathscr{MG}}$ and simple graph morphisms. The second step consists in

encoding the DPO rules into SPO rules. In [HHT96] (Proposition 3.5) it has been shown that it is possible to translate the application conditions of a DPO derivation (i.e. dangling edge and identification condition) in **MG** to equivalent negative application conditions (NACs) for performing SPO derivations in **MG**. In Theorem 1 we show that the initial DPO transformations in **MG** can be simulated by the translated SPO transformation in $\mathbf{SG}(L_{MG})$.

Remark 1 *(Uniqueness of derivations)* DPO derivations need the uniqueness of pushout complements to be deterministic to a given rule and match. In adhesive categories this is the case if the rule morphisms are or the match is monomorphic (see Lemma 15 in [LS04]), meaning injective in the category **Graph**. In our setting the category **MG** is adhesive and therefore also $\mathbf{SG_{MG}}(L_{MG})$ is, because it is isomorphic. The monomorphisms in the latter one are also euqalizers by their property of being edge reflecting and thus, they are regular monomorphisms.

Given a DPO rule $p = L \xleftarrow{l} K \xrightarrow{r} R$, we use $\mathbf{Sim}(p)$ to denote $\mathbf{Sim}(L) \xleftarrow{\mathbf{Sim}(l)} \mathbf{Sim}(K) \xrightarrow{\mathbf{Sim}(r)} \mathbf{Sim}(R)$, and we denote by $\mathbf{Sim}^*(p)$ the translated rule equipped with additional NACs, as described in [HHT96]. For the following lemma we interpret graphs of $\mathbf{SG_{MG}}(L_{MG})$ as graphs in **MG** by forgetting all labels. This allows us to show that pushouts are not only translated to those in a different category, but also remain pushouts in the original category of multigraphs, after applying **Sim**. An extension of **MG** with lables is direct and only adds information, which does not interfere with the pushout construction.

Lemma 3

$$\begin{array}{ccc} A & \longrightarrow & B \\ \downarrow & (PO) & \downarrow \\ C & \longrightarrow & D \end{array}$$ *in* **MG** *implies* $$\begin{array}{ccc} \mathbf{Sim}(A) & \longrightarrow & \mathbf{Sim}(B) \\ \downarrow & (PO) & \downarrow \\ \mathbf{Sim}(C) & \longrightarrow & \mathbf{Sim}(D) \end{array}$$ *in* **MG** *up to label information.*

Proof. (sketch) Pushouts in **MG** are constructed componentwise for the sets of edges and nodes by building the disjoint union and factorizing along the equivalence generated by the span of morphisms. The definition of **Sim** is compatible with the standard pushout construction, i.e. $\mathbf{Sim}(D) = \mathbf{Sim}(B +_A C) \cong \mathbf{Sim}(B) +_{\mathbf{Sim}(A)} \mathbf{Sim}(C)$. □

Theorem 1 (simulation) *Given a rule* $p = L \xleftarrow{l} K \xrightarrow{r} R$ *and a match* $m : L \to G$ *in* **MG**, *where* l *is injective, the following three are equivalent:*

1. $G \xRightarrow{p,m}_{\text{DPO}} G'$ *in* **MG**;
2. $\mathbf{Sim}(G) \xRightarrow{\mathbf{Sim}(p),\mathbf{Sim}(m)}_{\text{DPO}} \mathbf{Sim}(G')$ *in* $\mathbf{SG_{MG}}(L_{MG})$;
3. $\mathbf{Sim}(G) \xRightarrow{\mathbf{Sim}^*(p),\mathbf{Sim}(m)}_{\text{SPO}} H$ *in* $\mathbf{SG}(L_{MG})$.

Furthermore, if the derivation $\mathbf{Sim}(G) \xRightarrow{\mathbf{Sim}^*(p),\mathbf{Sim}(m)}_{\text{SPO}} H$ *exists in* $\mathbf{SG}(L_{MG})$, *then* $H \cong \mathbf{Sim}(G')$.

Proof. $1 \Leftrightarrow 2$ **Sim** and $\mathbf{Sim}^{-1}$ are isomorphisms by Lemma 1 and hence, they preserve all Limits and Colimits. Since l or m, respectively, is injective the DPO-derivations are unique up to isomorphism.

$2 \Rightarrow 3$ The derivation in 2 can be considered as a derivation in **MG** up to labels, according to Lemma 3. Then using [HHT96], it is equivalent to an SPO derivation with NACs in **MG** with result $\mathbf{Sim}(G')$, that is, $\mathbf{Sim}(G')$ is the pushout of p and m in **MG**. But, as $\mathbf{Sim}(G')$ is a simple graph, it is also the pushout of p and m in $\mathbf{SG}(L_{MG})$, up to labels. Because of the strict relation between the labels in graphs in $\mathscr{SG}_{\mathscr{MG}}$ and their structure, it is not difficult to see that $\mathbf{Sim}(G')$ is also the pushout of p and m in $\mathbf{SG}(L_{MG})$ without ignoring the labels.

$3 \Rightarrow 2$ Let H' be the result of the derivation (a) $\mathbf{Sim}(G) \xRightarrow{\mathbf{Sim}^*(p),\mathbf{Sim}(m)}_{\text{SPO}} H'$ in **MG**. By [HHT96] we know that then (b) $\mathbf{Sim}(G) \xRightarrow{\mathbf{Sim}(p),\mathbf{Sim}(m)}_{\text{DPO}} H'$ is a derivation in **MG**. Since $\mathbf{Sim}(p)$, $\mathbf{Sim}(m)$ are morphisms in $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$, by Lemma 2 we know that they are edge reflecting, and this allows to deduce that the graph H' is a simple graph, that is, an object of $\mathbf{SG}(L_{MG})$. Now, as $\mathbf{SG}(L_{MG})$ is a full subcategory of **MG** and by (a), we have that H' is the pushout of $\mathbf{Sim}^*(p)$ and $\mathbf{Sim}(m)$ in $\mathbf{SG}(L_{MG})$. By uniqueness of this pushout and the derivation in point 3 we deduce that $H' = H$, thus (b) is a derivation in $\mathbf{SG}(L_{MG})$. Finally, one can see that H' and the context graph in (b) are also objects of $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$ because the translated rule will only produce and delete complete structured edges by definition of **Sim**. Hence, no garbage (i.e. proxy nodes with either an outgoing s-edge or a t-edge, but not both) will occur. Thus, (b) is also a derivation in $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$.

Result $H \cong \mathbf{Sim}(G')$ is a direct consequence of the last part of the proof for the previous item. □

4 Extensions

Theorem 1 immediately extends to rules with negative application conditions, because they contain just additional graphs and morphisms of the same kind. Thus, we will not describe this aspect in more detail.

We are also confident that the results from this paper can be extended in a straightforward manner to hypergraphs [Kön02], which differ from multigraphs in not having source and target functions, but rather a single function $\mathsf{ends}\colon E_G \to V_G^*$ that associates with every edge a *string* of nodes. Hypergraphs can be translated to simple graphs using precisely the same technique of encoding edges as proxy nodes, with in this case as many auxiliary edges (to nodes) as there are elements in $\mathsf{ends}(e)$.

Up to now we have only considered unlabelled and untyped multigraphs, but all the results that we have shown can be easily extended to typed multigraphs, and hence to labelled ones, since labelling can be insured by typing; see, e.g., [EEPT06]. Fig. 6 shows how one of our example labelled multigraphs would be encoded into a simple graph.

A typed graph $\langle G, m\rangle$ is a graph G together with a morphism $m : G \to TG$ to some graph TG called the type graph. A typed graph morphism $f : \langle G, m\rangle \to \langle G', m'\rangle$ is a morphism for which $m = m' \circ f$. Transformations of typed graphs should involve only typed graph morphisms. It is equivalent to consider transformations in a *slice category*. That is, typed transformations in $\mathbf{C}$ w.r.t. the type graph TG are equivalent to transformations in the slice category $\mathbf{C} \downarrow TG$, where $\mathbf{C}$ is either **MG** or $\mathbf{SG}_{\mathbf{MG}}(L_{MG})$ and TG is a multigraph or simple graph, respectively. Now, as **MG**

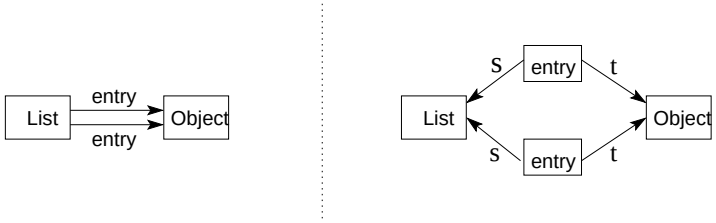


Figure 6: Encoding of a labelled multigraph.

and $\mathbf{SG_{MG}}(L_{MG})$ are isomorphic with **Sim** as isomorphism functor, it is trivial to see that the slice categories are also isomorphic. Thus, there is a pushout in $\mathbf{MG} \downarrow TG$ if and only if there is a pushout in $\mathbf{SG_{MG}}(L_{MG}) \downarrow \mathbf{Sim}(TG)$. Then the simulation result stated in Theorem 1 also holds for a typed transformation.

However, in this case an additional translation step is still required to translate to untyped simple graphs. In this case we have to extend the labels to encode the typing; hence, the translation is from $[\mathbf{SG_{MG}}(L_{MG}) \downarrow \mathbf{Sim}(TG)]+\text{SPO}$ to $[\mathbf{SG_{MG}}(L_{MG} \times (V_{TG} \cup E_{TG}))]+\text{SPO}$. We are convinced that this translation is straightforward, but we have not given the proof.

5 Simulation in SPO Tools

Tools performing graph transformations often implement SPO since this requires only one pushout construction where for DPO an additional pushout complement construction is needed. Problems arise when performing rule applications using SPO that do not satisfy the gluing conditions. In the running example such a situation would occur when applying the delete rule on an Object that is e.g. contained in more than one List.

In order still to be able to perform DPO transformation, there are basically two alternatives:

1. restrict rule applications by checking the gluing conditions after searching for matchings;
2. encode the gluing conditions using additional negative application conditions in the transformation rules.

Choosing the first alternative requires that the tool performs an additional gluing check on the found matches. This gluing check means that for all identifications in the matching and for all node deletions we need to ensure that there is no preserve-delete conflict (identification condition) and that the node-deletions do not cause dangling edges (dangling condition), respectively. The AGG tool's kernel implements SPO and uses a similar mechanism for handling DPO transformations.

The second alternative is based on Theorem 1, in which we show that it is possible to simulate DPO on our special simple graphs by adding additional negative application conditions as described in [HHT96].

Let us now briefly describe how one can use the GROOVE tool (or some other tool supporting simple graph transformations with SPO) for performing DPO transformations on multigraphs. Given a (multigraph) graph production system (GPS) $T = \langle G, \mathscr{P} \rangle$, one first has to create a production system $\mathbf{Sim}(T)$ by encoding the graph G and all graphs and morphisms that are parts

of the productions in $\mathscr{P}$ in the manner described in Section 3. Note that if some productions include negative application conditions, these conditions together with the morphisms that relate them to the corresponding production are encoded just as normal graphs and morphisms. Now, if the tool offers the possibility to check for the gluing condition (choice 1 above), then the GPS $\mathbf{Sim}(T)$ can be submitted to the tool, specifying that the check for the gluing condition has to be performed. Otherwise (choice 2 above), one has to construct the production system $\mathbf{Sim}^*(T)$ by augmenting $\mathbf{Sim}(T)$ with additional NACs for encoding the gluing condition in $\mathbf{Sim}(T)$. The GPS $\mathbf{Sim}^*(T)$ is then submitted to the tool as a normal (simple) graph production system. Any derivation results obtained by the tool (e.g. graphs that can be derived from the start graph or the actual rule applications) can be transformed back to multigraphs using the $\mathbf{Sim}^{-1}$ mapping. This forth and back translation can be used, for instance, for exchanging results between different graph transformation tools.

6 Conclusion and Future Work

We have proposed a method for performing DPO multigraph transformations using tools handling SPO simple graph transformations. Compared to previous work [HKM06], this method is generic, i.e. has been proved correct on categorical level and does not depend on the tools to be used.

Pushing theory to work in practice. Tool interoperability is one major motivating point to translate graph transformation systems using multigraphs and DPO to equivalent systems with simple graphs and SPO derivations. On the more fundamental level it is even more interesting to have the possibilities of applying a wide range of theoretical results and implementing them in the tool of favour. During the last three decades a lot of theory was developed using DPO and multigraphs. One special new technique is the analysis of derivations using Subobject Transformation Systems (STS) presented in [CHS06]. Since the GROOVE tool performs graph derivations to verify systems, there could be the possibility of combining the power of both. And indeed, this idea already has a concrete structure: basically one can exploit the possible results of dependencies using a translation to STSs and furthermore, the branching derivations of the state space can be folded into one summary object. Thus, only a small number of derivation steps will have to be performed to construct an abstraction of a much bigger state space. The idea is then to use the abstraction equipped with an STS to deliver only effective states and perform model checking on these states and their concrete successors.

Acknowledgements. The first and third authors are employed in the GROOVE project funded by the Dutch NWO (project number 612.000.314).

Bibliography

[BW95] M. Barr, C. Wells. *Category Theroy for Computing Science*. Prentice Hall, 1995.

[CHS06] A. Corradini, F. Hermann, P. Sobociński. Subobject Transformation Systems. *Applied Categorical Structures*, 2006. To appear.

[CMR+97] A. Corradini, U. Montanari, F. Rossi, H. Ehrig, R. Heckel, M. Löwe. Algebraic Approaches to Graph Transformation, Part I: Basic Concepts and Double Pushout Approach. Pp. 163–246 in [Roz97].

[Cou97] B. Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. Pp. 313–400 in [Roz97].

[DKH97] F. Drewes, H.-J. Kreowski, A. Habel. Hyperedge Replacement Graph Grammars. Pp. 95–162 in [Roz97].

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

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. Algebraic Approaches to Graph Transformation, Part II: Single Pushout Approach and Comparison with Double Pushout Approach. Pp. 247–312 in [Roz97].

[Hau06] J. H. Hausmann. *Dynamic Meta Modeling: A Semantics Description Technique for Visual Modeling Techniques*. PhD thesis, Universität Paderborn, 2006.

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

[HKM06] F. Hermann, H. Kastenberg, T. Modica. Towards Translating Graph Transformation Approaches by Model Transformation. In *Proc. of the Int. Workshop on Graph and Model Transformation (GraMoT'06)*. 2006.

[KKR06] H. Kastenberg, A. Kleppe, A. Rensink. Defining Object-Oriented Execution Semantics Using Graph Transformations. In Gorrieri and Wehrheim (eds.), *Proc. of the 8th IFIP Int. Conf. on Formal Methods for Open Object-Based Distributed Systems (FMOODS'06)*. LNCS 4037, pp. 186–201. Springer Verlag, 2006.

[Kön02] B. König. Hypergraph Construction and its Application to the Static Analysis of Concurrent Systems. *Mathematical Structures in Computer Science* 12(2):149–175, 2002.

[LS04] S. Lack, P. Sobociński. Adhesive Categories. In Walukiewicz (ed.), *Proc. of the 7th Int. Conf. on Foundations of Software Science and Computation Structures (FOSSACS'04)*. LNCS 2987, pp. 273–288. Springer Verlag, 2004.

[LS05] S. Lack, P. Sobociński. Adhesive and Quasiadhesive Categories. *Theoretical Informatics and Applications* 39(2):511–546, 2005.

[OMG05] OMG. Unified Modeling Language Specification. 2005. http://www.omg.org/.

[Ren04a] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE'03)*. LNCS 3062, pp. 479–485. Springer Verlag, 2004.

[Ren04b] A. Rensink. Representing First-Order Logic using Graphs. In Ehrig et al. (eds.), *Proc. of the 2nd Int. Conf. on Graph Transformations (ICGT'04)*. LNCS 3256, pp. 319–335. Springer Verlag, 2004.

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

[SSHW] A. Schürr, S. E. Sim, R. Holt, A. Winter. The GXL Graph eXchange Language. `http://www.gupro.de/GXL`.

[Tae01] G. Taentzer. Towards Common Exchange Formats for Graphs and Graph Transformation Systems. In Padberg (ed.), *Proc. of the Workshop on Uniform Approaches to Graphical Process Specification Techniques (UNIGRA'01)*. ENTCS 44. 2001.

[TER99] G. Taentzer, C. Ermel, M. Rudolf. The AGG Approach: Language and Tool Environment. In Ehrig et al. (eds.), *Handbook of Graph Grammars and Computing by Graph Transformations*. Volume II: Applications, Languages and Tools, pp. 163–246. World Scientific, 1999.

Evaluating Workflow Definition Language Revisions with Graph-Based Tools

René Wörzberger, Markus Heller, Frank Häßler

(rwoerz|heller|haessler)@i3.informatik.rwth-aachen.de
Lehrstuhl für Informatik 3 (Softwaretechnik)
RWTH Aachen, Germany

Abstract: In industry, there are many workflow management systems (wfms) which have been incrementally developed over a long time. The workflow definition languages that come along with these wfms are mostly graph oriented. These incrementally developed definition languages sometimes lack important modeling constructs as well as a clear conceptual foundation. Any workflow definition language revision has to be evaluated against end user's acceptance before implementation in the respective wfms.

In this paper, we report about an industrial application of graph-based techniques in the workflow domain. We present an evaluation environment which has been developed in a graph-based rapid prototyping approach. The evaluation environment comprises editing support for workflow definitions conforming to the language's revision. Furthermore, it provides a translator that maps definitions from the revised to the original language. Thus, the wfms can be used to enact definitions of the revised language without modifying its implementation.

Keywords: workflow, graph transformation

1 Introduction

Workflow Management Systems (wfms) are software systems that target at the support for collaborative processes. They usually provide a graphical language for workflow definitions wherein nodes represent the activities that constitute a process type and edges determine the possible activity execution sequences. A workflow definition can be instantiated in a wfms in order to support one single process of a certain type. In a workflow instance, activities are executed in one of the possible sequences by humans or software services which thereby produce instance specific data.

In this paper, we report about an industrial *application* of graph-based methods in the workflow modeling domain. Our industrial partner develops a commercial workflow management system (WfMS) which is embedded in a large enterprise resource planning system (ERP). The ERP and WfMS are continuously developed. It is particularly difficult to completely anticipate all requirements for the workflow definition language of the WfMS. Our industrial partner wanted to improve their workflow modeling language to reflect new requirements gathered by their modeling consultants in recent years while adapting the WfMS to customers' needs.

Despite its current shortcomings, longtime customers are used to the workflow definition language to some degree. Hence, not only replacing the language by some standard language (e.g. UML, BMPN) is not an option for our partner. Furthermore, changes in the definition language have to be introduced with extreme care. Together with our industrial partner, we have revised the workflow definition language of the WfMS and developed an Evaluation Environment (EE) with the benefit, that modeling experts can use the EE to extend their modeling language in a try-and-test manner. They can model workflow definitions in the revised workflow modeling language with the WDE and can discuss their suggestions with consultants or customers by using realistic examples. Using this approach, cost and time consuming modifications of the WfMS' implementation for evaluation purposes can be reduced. The Evaluation Environment described in this paper is based on the graph-based rapid prototyping approach (PROGRES, UPGRADE, GRAS) developed at our department.

The paper is structured as follows: Section 2 describes the original workflow definition language of the WfMS and points out some of its current deficiencies and possible improvements in a revised language. Section 3 describes the design of the evaluation environment which provides editing support for workflow definition in the revised language and a translator that converts workflow definitions in the revised language to definitions in the original language. Section 4 sketches similar approaches of other research groups. The paper concludes with Section 5.

2 Problem Description

This section briefly describes the workflow definition language of the WfMS (original language) thereby pointing out some language features whose modification in the revised language is also depicted.

2.1 Original Workflow Definition Language

The WfMS provides a language that allows for modeling of workflow definitions as typed and directed graphs. The language is separated in two modeling-layers with two different types of workflow definitions.

People to application (P2A) workflow definitions. A P2A workflow definition normally models a rather short-running interaction between a single human workflow participant and technical resources, i.e. other software modules of the ERP. Usually, these definitions are used to model possible sequences of GUI-dialogs which constitute the interface between a workflow participant and technical resources.

Figure 1 depicts an example of a telephone marketing process modeled as a P2A workflow definition. It contains all existing model elements that can be used in P2A workflow definitions.

- A *P2A start activity* (big octagon) marks the unique start of a P2A workflow and has no incoming transitions. Phone marketing in Figure 1 is an example for a P2A start activity. At runtime, this activity is invoked at first when the P2A workflow definition is instantiated. It has no other effect but to invoke its successor activities.

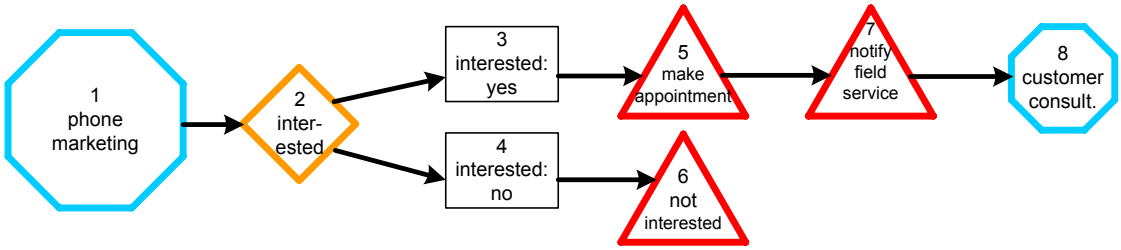


Figure 1: Example of a P2A workflow definition

- A *P2A call activity* (triangle) represents the runtime invocation of a hard coded subprogram in a software module of the ERP. A subprogram can run either with or without workflow participant interaction. In the former case the workflow participant can enter data via activity specific GUI forms.
- A *P2A XOR-decision* (rhomb) models a branching between two or more alternative paths of P2A activities. The example in Figure 1 shows the *P2A XOR-decision* interested which reflects the decision whether the phoned customer is interested in further information or not. Every P2A workflow instance can only follow exactly one of these alternative paths. The decision is made at runtime by a workflow participant with a standard GUI that lists the alternatives. In the workflow definition, each selectable alternative is represented by a *P2A alternative* (box) which is also the start of each alternative path. Therefore, P2A XOR-decisions can only be succeeded by P2A alternatives. In the example, there are just two P2A alternatives: yes and no.
- A *P2A reference* (small octagon) represents the instantiation of another P2A workflow definition. The new P2A workflow instance is automatically assigned to the same workflow participant that interacts with the referencing P2A workflow instance. Paths of P2A activities can either end in a P2A call activity like not_interested in the example or in a P2A reference. In the former case the P2A workflow instance terminates after termination of the last P2P call activity. In the latter case the P2A workflow instance with the reference is suspended but does not terminate.

The listed model elements can be connected with directed *transitions* (arrows) which specify the order of invocation and associate a P2A XOR-decision with its P2A alternatives, respectively.

People to people (P2P) workflow definitions. P2P workflow definitions model interactions among human workflow participants. The only model elements on this layer are the transition and the *P2P activity* (octagon).

Every P2P activity refers to one P2A start activity. Invoking a P2P activity in a P2P workflow instance creates a new P2A workflow instance. The P2A workflow definition of this instance is uniquely identified by the referenced P2A start activity. Termination of the P2A workflow instance conversely terminates the referencing P2P activity.

P2P activities are assigned to one or more workflow participants. At runtime, workflow participants receive a notification when one of their assigned P2P activity is invoked. P2P activities are

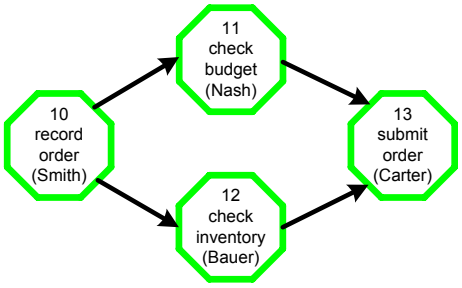


Figure 2: Example of a P2P workflow definition

connected with directed transitions. A P2P activity is invoked when all preceding P2P activities have terminated. A P2P activity signals its termination to *all* succeeding P2P activities.

Figure 2 shows an example of a P2P workflow definition. All P2P activities refer to P2A workflow definitions which are not shown for brevity. Assigned workflow participants are written in brackets under the name of each P2P activity. The termination of record order invokes both check budget and check inventory. After termination of the latter two submit order is invoked.

2.2 Shortcomings of the Original Language

The WfMS has been repeatedly adapted to new requirements in recent years according to newly risen customer needs. Therefore, its workflow definition language now has a less clear conceptual design compared to other workflow definition languages which have been developed from scratch. In the following, we describe some of these shortcomings.

2.2.1 No Real Decomposition Hierarchy

The P2P layer and the P2A layer beneath it have a decomposition relationship inasmuch that a P2A workflow definition can be considered as a refinement of the corresponding P2P activity. Furthermore, P2A workflow definitions can be refined by using P2A references to some degree. However, there is no way to model decompositions within the P2P layer, i.e. to refine a complex P2P activity by another P2P workflow definition.

2.2.2 XOR in the P2P Workflow Definitions

Alternative paths in P2A workflow definitions can be modeled via P2A XOR-decisions. Currently, there is no equivalent model element for P2P workflow definitions. Instead, decisions between alternative P2P activity paths have to be modeled by a pattern of several P2P and P2A modeling elements.

2.2.3 No Explicit Start and End Elements

In contrast to other workflow definition languages, workflow definitions of the WfMS have no dedicated model elements that mark the origin and end of the control flow. This is particularly disadvantageous in P2P workflow definitions where those P2P activities are determined to be

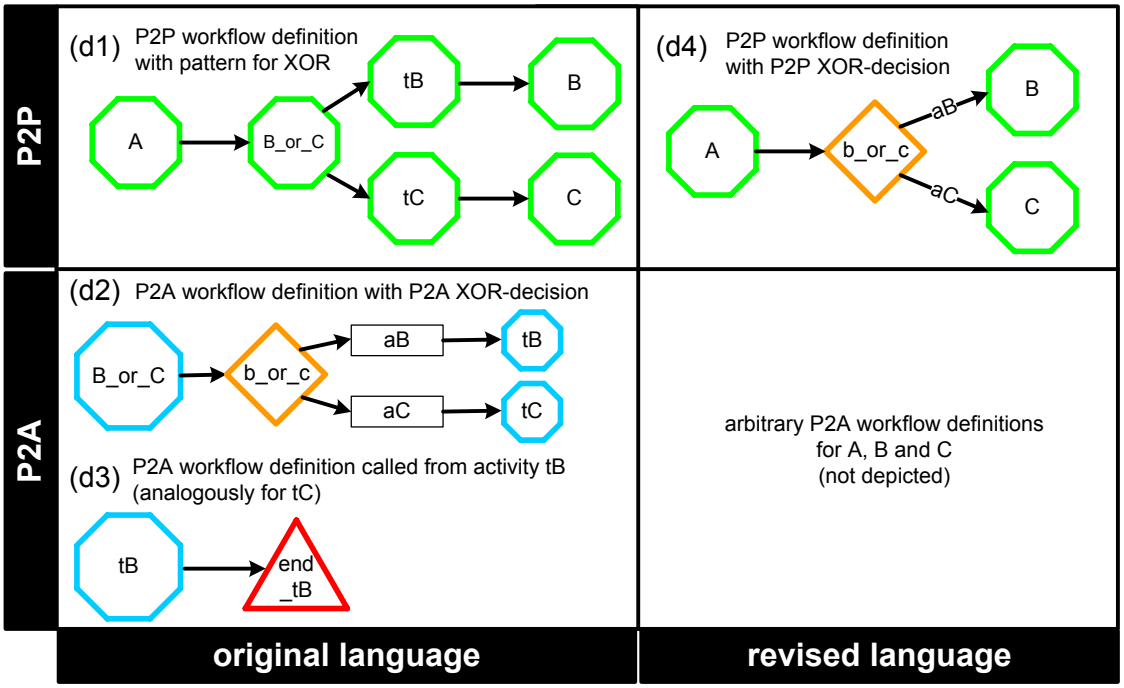


Figure 3: Modeling XOR-decisions in P2P workflow definitions

start (end) activities that have only outgoing (incoming) edges but do not further differ from other P2P activities.

Figure 3 shows how XOR-decisions in P2P workflow definitions are modeled in the original (left hand side) and the revised workflow definition language (right hand side). The P2P workflow definition (d1) contains a P2P activity B_or_C with two outgoing transitions to alternative paths. The P2P activity B_or_C invokes a P2A workflow definition (d2) which contains a P2A XOR-decision b_or_c. Each alternative path ends in a P2A reference (e.g. tB) which instantiates a P2A workflow definition exemplified by (d3). This P2A workflow instance is assigned to the same workflow participant as the instance of (d2). It just prompts the workflow participant for termination via P2A activity end_tB.

Termination of the instance of (d3) leaves the instance of (d2) in a suspended state. Consequently, the corresponding P2P activity B_or_C in the instance of (d1) is left suspended and does not invoke any successor P2P activities. Furthermore, the P2P activity tB in (d1) terminates due to the termination of the instance of (d3) and invokes its successors, B in this case, which can be assigned to a different workflow participant.

2.3 Revised Workflow Definition Language

In Subsection 2.2 we described some of the shortcomings of the original workflow definition language. These shortcomings were subject of a revision of the workflow definition language [Häß05] with two major goals: (1) to eliminate the deficiencies described above and (2) to keep

most model elements and language properties, particularly the graph oriented paradigm and the distinction between P2P and P2A workflow definitions. The second goal is due to the requirement that users which are familiar to the original language should be able to quickly adapt to the revised one. Furthermore, we have to assure that workflow definitions in the revised language can be automatically translated into the original language (s. Subsection 3.4). The first goal is addressed as follows:

- Decompositions can be modeled in the revised language by means of a new model element named *P2P subflow call*.
- In addition, the revised language provides dedicated model elements for both language layers that are used to explicitly indicate the start and end of the control flow within a workflow instance.
- Furthermore, the revised language allows for expressing decisions in P2P workflow definitions, too.

The right hand side of Figure 3 depicts a P2P workflow definition (d4) in the revised language which is equivalent to the pattern in the original language on the left hand side. The workflow definition (d4) has a *P2P XOR-decision* b_or_c which is incident to two *P2P XOR-transitions*. Each of the P2P XOR-transition carries an attribute (aB, aC) that represents the respective alternative.

A comparison of both sides in Figure 3 clearly shows that the new model element types help reducing the complexity of workflow definitions if a decision on the P2P layer has to be modeled.

The revised workflow definition language represents just one among many possible language derivatives. To evaluate each alternative by implementing a dedicated variant of the WfMS cannot be afforded. In the sequel, we present an alternative approach that leaves the WfMS as it is but provides support for the revised workflow definition language.

3 Evaluation Environment (EE)

In order to evaluate the revised workflow definition language without changing the implementation of the WfMS, we have developed an Evaluation Environment (EE) which provides (a) a workflow definition editor (WDE) to model workflow definitions according to the revised language and (b) a workflow definition translator (WDT) to translate workflow definitions from the revised to the original language.

3.1 Graph-Based Rapid Prototyping Support

Both the WDT and the WDE have been developed using a *rapid prototyping approach* which is developed at our department for several years now. The approach facilitates the specification of graph-grammar specifications in PROGRES [SWZ99] as well as the rapid construction of prototypical applications based on the specification. Briefly, a PROGRES specification comprises the following parts: (1) A *graph schema* that defines a set of valid graphs. This is done by the declaration of node and edge types and the definition of valid combinations of their respective

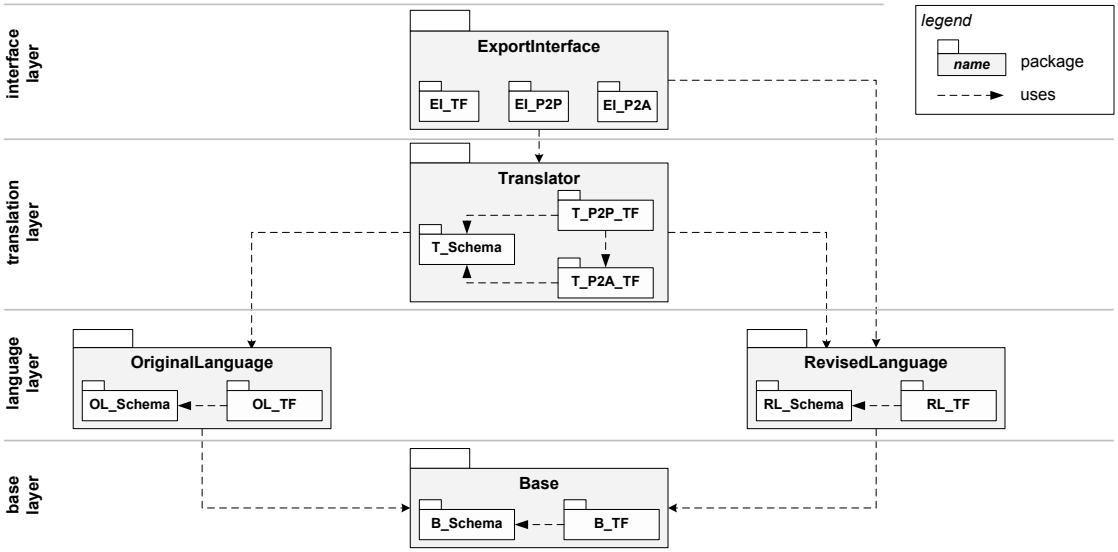


Figure 4: Package structure of the PROGRES specification

instances, (2) a set of *graph transformations* that transform a valid graph into another, and (3) a set of *graph queries* that test a graph for certain properties.

PROGRES can generate executable C-code from a PROGRES specification. The C-code is embedded into the rapid prototyping framework UPGRADE [BJSW02] resulting in an executable UPGRADE prototype. The prototype allows for displaying multiple views of a graph instance (host graph) stored in a graph database and manipulating this host graph by invoking transformations of the application logic resp. PROGRES specification.

The described graph-based prototyping framework has been successfully applied in a number of research projects at our department, for example, to develop an advanced workflow management system for dynamically evolving development processes for different application domains (e.g.software engineering, chemical engineering) ([HJS+04]).

The application logic of both the WDT and the WDE is fully specified in a single PROGRES specification. Based on this specification, the Evaluation Environment is realized as an UPGRADE prototype.

3.2 Architecture

Figure 4 presents the coarse architecture of the PROGRES specification for the Evaluation Environment (EE). The parts of the PROGRES specification are organized in packages modeled as sections and subpackages which are associated with architectural layers. The use-relationship indicates which parts of a package are used in which other package.

The package Base contains two subpackages. B_Schema just comprises the declaration of the node class `ELEMENT` (a node class is a type that is not instantiatable). The class `ELEMENT` serves as the root of a class hierarchy for all model elements of workflow definitions in the original as well as the revised language and declares common attributes like `name` storing the name of a

model element. B_TF contains very basic graph transformations, like `SetElementName` which sets the name of a given element.

Package RevisedLanguage includes the schema part for model elements in the revised languages (RL_Schema) and transformations that a can be applied to workflow definitions in the host graph (RL_TF). The package OriginalLanguage is build up analogously.

Translator is the package that encompasses all transformations which constitute the translation algorithm. Only one edge type `transformed_to` is declared in T_Schema which is used for embedding transformed edges in the workflow definitions of the original language. Transformations for translating P2P and P2A workflow definitions are located in dedicated subpackages T_P2P_TF and T_P2A_TF, respectively.

The package ExportInterface covers all transformations that are accessible from the UPGRADE prototype. Edit operations concerning P2P workflow definitions in the revised language are located in the subpackage EI_P2P whereas transformations for P2A workflow definitions are in subpackage EI_P2A. EI_TF only contains a single transformation that starts a translation run.

It is the main goal of the architecture to accommodate future changes of the revised languages in the PROGRES specification with local modifications only. Such changes ideally imply modifications in RevisedLanguage and Translator only (whereas OriginalLanguage and Base can remain unchanged). In the same manner, the package ExportInterface needs to be modified when extending the revised language with new workflow edit operations.

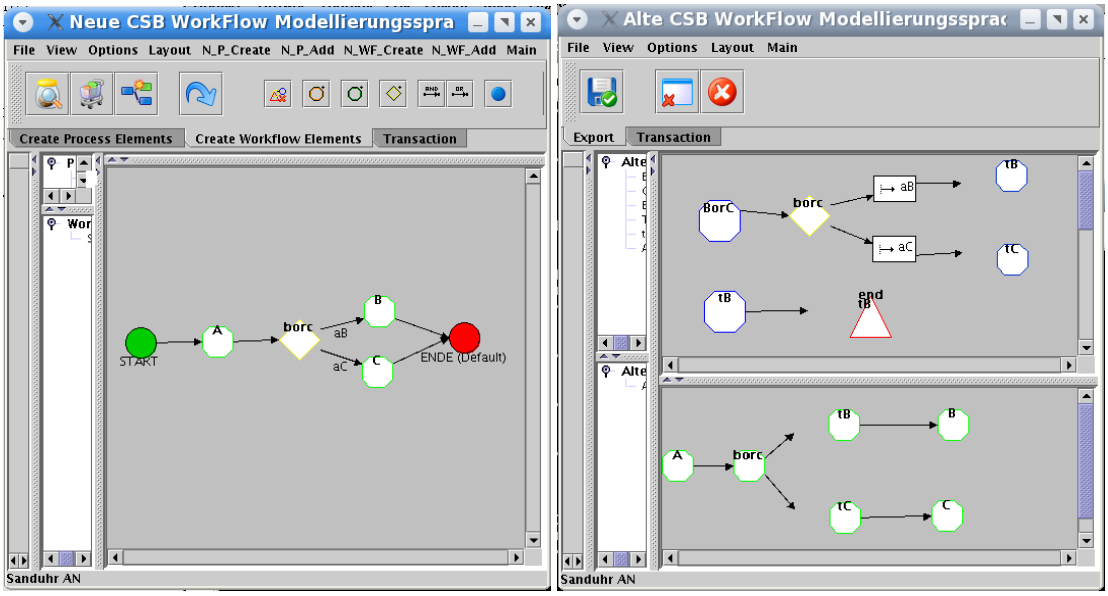
3.3 Workflow Definition Editor (WDE)

The PROGRES specification contains several graph transformations which constitute the interface of the workflow definition editor (WDE). Each graph transformation represents an edit operation on the host graph that contains all workflow definitions in the revised language. Figure 5(a) is a screenshot of the WDE captured during modeling of the P2P workflow definition (d1) of Figure 3. The WDE provides toolbars with buttons each of which executes a transformation, e.g. insertion of a P2P activity in a P2P workflow definition.

At every point in time, a workflow definition modeled with the WDE complies with certain conformance rules which can be subdivided in two sets:

- *Strong conformance rules (SCR)* are rules which every workflow definition of the revised language has to comply with at any time. For instance, this rule set includes a rather common rule “Every transition has a source and a target”, but also a rule “The outgoing transitions of a P2P XOR-decision can be only P2P XOR-transitions” which is specific to the revised language. The violation of SCR is immediately signaled to the WDE’s user by means of a message box.

- *Weak conformance rules (WCR)* are rules which every workflow definition has to comply with in order to be a valid expression of the revised language, but which can be violated *temporarily*. The rule “A P2P XOR-decision has at least two outgoing P2P XOR-transitions” is a weak conformance rule. Even though supporting the user to model valid workflow definitions in the revised language is clearly a goal of the WDE, the permission of temporary violations of WCR eases the modeling task. In the mentioned example the user would be forced to create the context of a P2P XOR-decision before creation of



(a) Workflow Definition Editor (b) Result of a Workflow Definition Translator run

Figure 5: UPGRADE Prototype

the P2P XOR-decision could take place. Such restrictions might unnecessarily contradict user's modeling preferences. Hence, violations of WCR are not signaled to the user before the workflow definitions are translated (s. Subsection 3.4).

The distinction between SRC and WCR is not specific to our EE but likely to be applicable for similar cases, i.e. WDEs for revisions of other workflow definitions languages.

3.4 Workflow Definition Translator (WDT)

The host graph of the Evaluation Environment maintains graph structures for workflow definitions according to the revised modeling language and graph structures for workflow definitions in the old modeling language. The workflow definition translator (WDT) translates workflow definitions form the revised language into workflow definitions in the old language. These newly generated workflow definitions can then be imported and executed in the CBS-WFMS. The translation algorithm is specified as a set of graph transformations (in package *Translator* in Figure 4). A translation run can be initiated from within the WDE prototype as an automated batch run without any user interaction. The translation algorithm of the WDT comprises several steps:

1. Check conformance of all workflow definitions against WCR. Stop if there is a non-conforming workflow definition.

2. For each P2A workflow do

   (a) Transform all P2A call activities

(b) Transform all P2A references

(c) Transform all P2A XOR-decisions

(d) Transform all P2A transitions

3. For each P2P workflow do

   (a) Transform all P2P activities

   (b) Transform all P2P XOR-decisions

   (c) Transform all P2P transitions

Some characteristics of the WDT concerning the sequence of translation steps are described in the following:

1. *Language layers and coarse grained relationships.* The WDT works bottom up inasmuch it first transforms the P2A and afterwards the P2P workflow definitions (step 2 and 3). This sequence is crucial since P2A workflow definitions are referenced by P2P workflow definitions. Generally speaking, the sequence is aligned to coarse grained reference relationships between language layers.

2. *Nodes and edges.* Both the revised and the original workflow definition language are graph oriented. Therefore, model elements can be specified as node elements and edge elements. During translation a directed edge has to be embedded between a source and a target node at any time whereas a node might exists at least temporarily without any incident edges. Thus, it is necessary to transform model elements classified as nodes (e.g. P2A call activities in step 2a) before those model elements classified as edges (e.g. P2A transition in step 2d).

3. *Model element types.* Each coarse grained translation step is decomposed into fine grained steps (2a to 2d and 3a to 3c). Every fine grained translation step transforms model elements of only one type. This is a consequence of the fact that mainly the type of a model element (e.g. P2P XOR-decision) determines the static specification of its translation while other model element properties (e.g. number of incident activities) just affect the execution dynamics (e.g. actual loop iterations).

The described criteria can also be applied in the specification of the EE for revisions of other workflow definition languages. Therefore, the coarse design of the algorithm can be reused in other cases.

The result of a WDT run is a set of workflow definitions in the original language which are also constructed as graph structures in the host graph during the transformator run. The UPGRADE prototype provides an export function that serializes these generated workflow definitions into a file according to a proprietary file format readable by the WfMS. This export function has been implemented in Java and is independent of the specification of the revised language. Thus, future revisions of the revised language will surely lead to changes in the PROGRES specification but not in the export function.

4 Related Work

The WDE presented in this paper allows for editing workflow definitions whose validity is partly checked during editing and fully checked before translation against certain conformance rules specified within the PROGRES specification. We shortly discuss important related work with the focus on the workflow domain.

In the workflow area, there are various groups working on translators between *different* workflow definitions languages. The following approaches do not use graph-based techniques. Van d. Aalst et al. [ODHA06] describe an algorithm that translates workflow definitions from the graph oriented Business Process Modeling Notation (BPMN) used for notation of workflow definitions to the block-structured Business Process Execution Language (BPEL) which specifies workflow implementations on top of web services. Guelfi and Mammar [GM06] provide a set of formal transformation rules for mapping UML activity diagrams to workflow definitions in the workflow definition exchange format XPDL. [Sch02] describes an UPGRADE prototype specified in PROGRES for editing dynamic task nets together with the transformation of workflow definitions modeled as UML class diagrams and collaboration diagrams into PROGRES code.

The proposed approach with the WDT unidirectionally translates between different versions of *one* graph oriented workflow definition language without user interaction based on a graph-based approach.

The focus of this paper was on reporting about an *application* of graph-based techniques in industrial application domains and not on the development of new *general* transformation approaches. Therefore, the workflow definition translator (WDT) of the presented approach works in a rather straightforward manner in a automated batch mode without user interaction. For more advanced incremental bidirectional and interactive integration frameworks see [BHW05, KS06].

5 Conclusion and Impact

In this paper we have reported about an industrial *application* of graph-based methods in the domain of workflow modeling. Our industrial partner wanted to revise the workflow definition language of their commercial workflow management system (WfMS) according to additional requirements. Using our graph-based tools for rapid prototyping (PROGRES, UPGRADE and GRAS), we have developed an Evaluation Environment which offers an editor (WDE) for workflow definitions in the revised language. Furthermore, it provides a translator (WDT) that translates these definitions back to definitions in the original language for enactment within the unmodified WfMS. This approachs helps reducing costly experimental reimplementation cycles for the WfMS.

The modeling experts and development teams for the WFMS have used the Evaluation Environment to model sample workflows in the described manner. Within the EE, the improvements contained in the revised language were evaluated and, by the time of writing, some of them have already been adapted in the implementation of the commercial WfMS.

Using PROGRES and UPGRADE as technical foundation for the Evaluation Environment has proven to be a good decision due to the facts that the WDE as well as the WDT could be rapidly

developed in a relatively short time-span. In the future, the investigation of workflow definition language translation as described in this paper serves as preliminary work for the development of a dynamic workflow management systems based on existing static wfms [HNW07].

Bibliography

[BHW05] S. M. Becker, T. Haase, B. Westfechtel. Model-based a-posteriori integration of engineering tools for incremental development processes. *Software and Systems Modeling* 4:2, 2005.

[BJSW02] B. Böhlen, D. Jäger, A. Schleicher, B. Westfechtel. UPGRADE: A Framework for Building Graph-Based Interactive Tools. *Electronic Notes in Theoretical Computer Science* 72:2, 2002.

[GM06] N. Guelfi, A. Mammar. A formal framework to generate XPDL specifications from UML activity diagrams. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*. 2006.

[Häß05] F. W. Häßler. Analyse und Erweiterung einer bestehenden Workflow-Modellierungssprache. Master's thesis, RWTH Aachen University, 2005.

[HJS+04] M. Heller, D. Jäger, M. Schlüter, R. Schneider, B. Westfechtel. A Management System for Dynamic and Interorganizational Design Processes in Chemical Engineering. *Computers & Chemical Engineering* 29:1, 2004.

[HNW07] M. Heller, M. Nagl, R. Wörzberger. Dynamic Process Management Based Upon Existing Systems. In *Collaborative and Distributed Chemical Engineering Design Processes: From Understanding to Substantial Support*. LNCS. Springer, 2007. (to appear).

[KS06] A. Königs, A. Schürr. MDI - a Rule-Based Multi-Document and Tool Integration Approach. *Special Section on Model-based Tool Integration in Journal of Software and System Modeling*, 2006.

[ODHA06] C. Ouyang, M. Dumas, A. H. M. ter Hofstede, W. M. P. van der Aalst. From BPMN Process Models to BPEL Web Services. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services (ICWS'06)*. 2006.

[Sch02] A. Schleicher. *Management of Software Development Processes: An Evolutionary Approach*. PhD thesis, RWTH Aachen University, 2002.

[SWZ99] A. Schürr, A. Winter, A. Zündorf. The PROGRES Approach: Language and Environment. In *Handbook on Graph Grammars and Computing by Graph Transformation – Volume 2: Applications, Languages, and Tools*. World Scientific, 1999.

Graph-Based Engineering Systems
A Family of Software Applications and their Underlying Framework

Gregor Wrobel, Ralf-Erik Ebert, Matthias Pleßow

{wrobel,ebert,plessow}@gfai.de
R&D Department of Graph-Based Engineering Systems
Society for the Promotion of Applied Computer Science
Berlin, Germany

Abstract: In various engineering disciplines visual modeling techniques are used for the definition as well as representation of complex systems. Besides the pictorial illustration, the included structural information is often used for application-specific procedures. This paper presents a few engineering systems for quite different application fields, but they use a common graph-based model. This model is part of a framework that underlies these applications. Various kinds of applications can be developed on the basis of this framework by means of configurations and extensions. The development of new applications is supported by convenient assemblies of suitable system functions and layout methods as well as by integration of application functionalities. The introduced framework is the basis for a product line of graph-based engineering systems.

Keywords: visual modeling techniques in engineering, software architectures and frameworks, software product line, tool support

1 Introduction

In the early 1980s, the use of computers for the design and documentation of schematic diagrams increased in response to both the new possibilities of computers and the growing use of IT in engineering. In many engineering disciplines, apart from computer-aided design, computers are important for the production of logic models. In line with human thought and decision processes, these models are complex, frequently have a net-like structure, and represent hierarchical phenomena [PP98].
Independently of that, extensive research was conducted in a completely different field: the generation of graph layouts (graph drawing [DETT99]). In graph drawing a key role is played by the automatic generation of such layouts and for that purpose, numerous software tools have been developed [JM03]. However, the modeling of engineering systems requires graphic editors.
This paper presents a framework that combines computer aided modeling with schematic drawing, based on a structural model. It includes graph drawing techniques. As the result it constitutes the basis of a product line. The domain of this product line is referred to as graph-based engineering systems and its main elements are presented here. Finally, a number of applications are introduced.

2 System Architecture

Based on engineering applications developed [PP98][BPPS01] and former research activities in drawing schematics [PS89], we performed a domain engineering process to detect common features of what we call graph-based engineering systems. In order to create such systems in an efficient way, a system architecture has been developed whose main element is a domain framework so-called CASTool (Computer Aided Schematics ToolBox). A core software component of this framework is a meta data model. Based on this model, several framework software components for domain functionalities are implemented. Hence, by using this framework, the development of a concrete system should be restricted to implement specific application knowledge. In order to achieve this goal the following implementation strategies are used: *configuration management*, *component orientation*, *common and extendable meta model* and *generic programming*.
Figure 1 shows the system architecture. The main elements and some implementation techniques to create CASTool-based applications are outlined below.

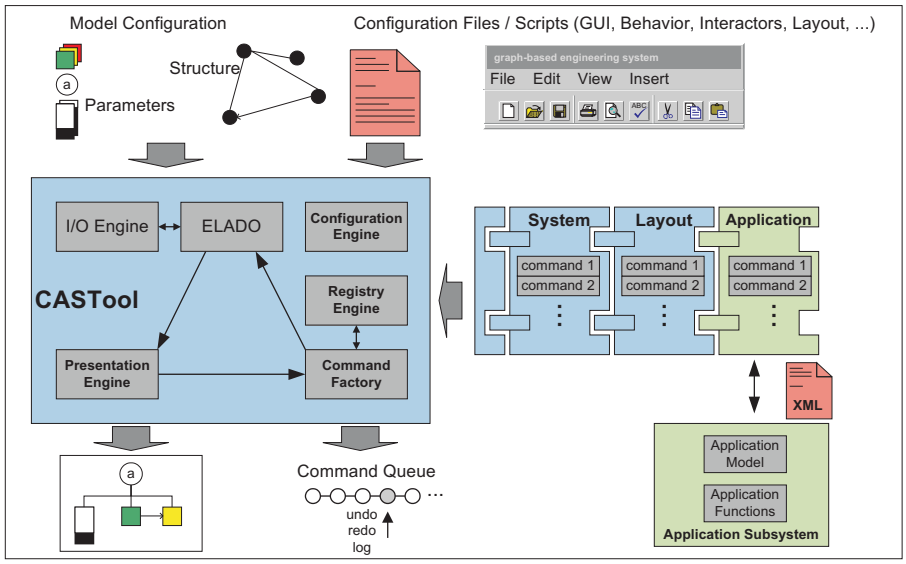


Figure 1: System architecture

2.1 Data Model

One common characteristic of graph based engineering systems is the usage of an extended graph model. ELADO (Extended Layout Data Model) represents such structure and is comprised of three main sections: a *structure model*, an *application model* and a *visualization model*.
The structure model represents what we call networks (Figure 2 shows the core elements). Networks are structured similarly to mathematical graphs and substantially made up of components, nets and connection points (called as pins). To extend the graphs, components (which correspond to graph vertices) can also contain networks of their own. This means that hierarchically structured models can easily be represented. Such hierarchical component can be a node

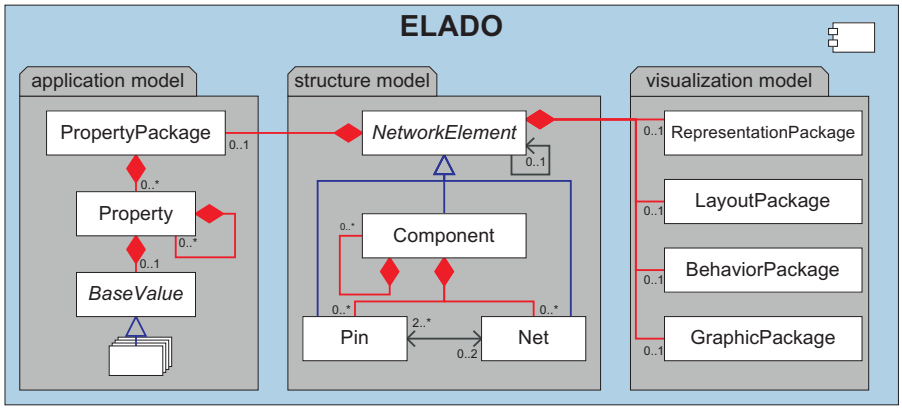


Figure 2: ELADO model parts (extract)

in a scheme or a simple project node (e.g. a folder). The latter is used for project structuring. Apart from the hierarchical structure, networks differ from graphs in that components have pins. Pins are the connection points for nets. Figure 3 shows a simple network. A outer component contains three components. These components are connected by a net. This net is owned by the upper component and associated with pins of the inner components. Additional, Figure 3 denotes other elements of the ELADO structure model. In the mathematical sense, the net between the inner components is a hyper edge. The three parts of the net (called as segments) are ending in a common branch point (shown as smal black rectangle in Figure 3). Further, each segment is modeled by segment parts which are delimited by bend points (shown as smal black cicles in Figure 3). Apart from the model elements featured above, other classes exist for group representation, annotations and special structure elements.

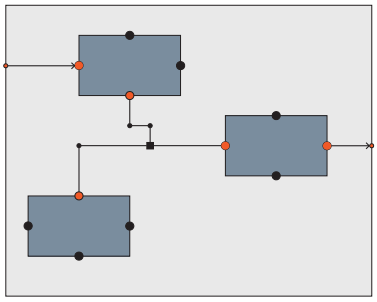


Figure 3: ELADO structure

Besides the structure model, ELADO consists of a visualization model. This part of ELADO is used to store information about the geometry of a network element, its shape as well as layout-specific behaviors. This means for instance, that standardized symbols from various fields of applications can be simply represented. For example in Figure 4, the net's segment representing a bus is drawn by a double line and in Figure 3 the component's pins are located on fixed positions on the component's shape and all connected pins are colored different. The latter could be a special layout behavior. In addition, each component can contain simple graphics (pictures, lines, text, etc.) as well as store its own layout information such as connection rules and routing behaviors. Therefore, in one application various visual languages could be embedded.

In order to be able to develop different applications and in particular to store their application-specific data (e.g. technology data, business data), ELADO has been augmented with a third model part, the application model. In contrast to the structure model, in which each relevant ele-

ment is modeled by a class, the application model is an abstract data structure. The data is stored in an associative property tree. Each node in the tree can be identified by a key and owns a value object and a subtree. As indicated in Figure 2, diverse types of basic values such as numbers, intervals and time series are implemented in ELADO as derived classes.
In conclusion, ELADO satisfies the requirement of an extendable meta model. The use of the same model for networks as well as for project structure modeling and the appropriate application data management makes CASTool different from standard drawing tools. As a matter of principle, all applications we develop underlie this model approach. Hence, we call such applications *graph-base engineering applications*. ELADO consists of approx. 65 classes. Based on ELADO, CASTool provides a few class libraries to develop graph-based engineering systems in an efficient way. Some of these are described below.

2.2 Extensibility and Configurability

In order to be able to develop diverse engineering solutions, the system architecture must be designed such that new functionality can be added and existing functionality extended or modified. For this purpose, the framework can be augmented by application-specific services such as special layout methods and engineering components like simulators etc.
Figure 1 shows the extension mechanism embedded into the system architecture. Additional software components can be added dynamically at run time via a defined mechanism. On principle, every user-controlled communication between components is based on the execution of commands. The use of such a command is based on the fabric pattern [GHJV95] whereas the creation is embedded in the command itself.
Due the abstract application data model (see Section 2.1), the handling of special data is more complicated and not high-performing. Therefore, every ELADO object (more precisely every CASTool object) can be decorated with interface objects. An interface object provides specific functions for model data access and manipulation, but can not store data persistently. This is not necessary, since all data can be stored in the application model. The registration of interface objects works in the same way as for commands by registering an alias at runtime.
Beside the system's extensibility, configuration mechanisms have been implemented within the system (configuration engine in Figure 1), which are primarily used for the design of GUI and layout management (in form of layout plans that formulate the calls of individual analysis, layout and evaluation methods) of an application. This can be done by modifying initialization files or, alternatively, by dynamic script interpretation at run time. Currently, interpreters for *JavaScript*, *Phyton*, *TCL* and *CSL* are integrated. All interpreters are extended with special functions for CASTool command calling, configuration access and model method invocation. For this purpose, a navigation language for object data, which is according to *XPath*, is implemented.
To sum up: the framework offers a convenient way of system extension by commands and interface objects integration as well as system configuration. This satisfies the implementation strategies described as *component orientation*, *generic programming* and *configuration management* above.

2.3 Additional Framework Features

CASTool's presentation module is responsible for outputting the model data corresponding to the application. To visualize the ELADO structure model and interact with it, a software component for drawing schematics is included (the presentation engine in Figure 1). To deal with ELADO application data, generic dialogs and forms have been developed. For that purpose, dialog elements for base data types are developed including optional GUI elements for addition information (units, bounds, intervals). A dialog generator arranges sets of these line by line on forms. Nodes of the application data tree rendered as tabs. Besides this features CASTool provides data exchange routines (XML formats for data exchange, SVG and pixel formats for schematics) as well as client-server components to run the framework in intranet/internet environments without visualization, in which case CASTool acts as a layout server.

3 Layout Methods

The framework includes a couple of basic layout methods such as the usual procedures for aligning selected components and simple routing methods like rubber band. All these methods work direct on ELADO. Furthermore, some special layout and placement methods (with special data structures) are included. These methods also operate with structure as well as geometry information and have been developed taking graph drawing techniques into account. Their purpose is to emphasize the structural properties of networks, and they also have to be clear and aesthetical. Moreover, layout methods need to be topologically and geometrically stable to prevent recognition problems during interactions on the part of the user [BP90]. Amongst others, the framework includes the following layout methods:

Orthogonal routing: This aids the interactive graphic design of networks by locally adapting the net routing whenever component positions are changed. As a special feature, this method finds long continuous connection parts (buses) and draws them as horizontal or vertical lines. This makes the schemes much easier to read because of the representation of one main flow. This bus routing method consists of three parts: pattern routing for two point connections, autonomous bus routing and post processing for bus routing. The pattern routing draws two-point connections in a sophisticated way by using a set of defined base patterns. The bus routing handles hyper-edges and embeds them as sub-optimal Steiner-Trees. While the autonomous bus routing disregards collisions, the post processing optimizes the results by regarding collisions. Figure 6 shows the bus routing in an application.

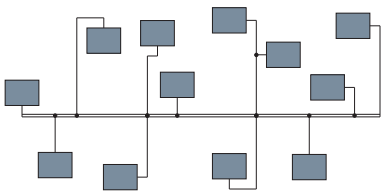
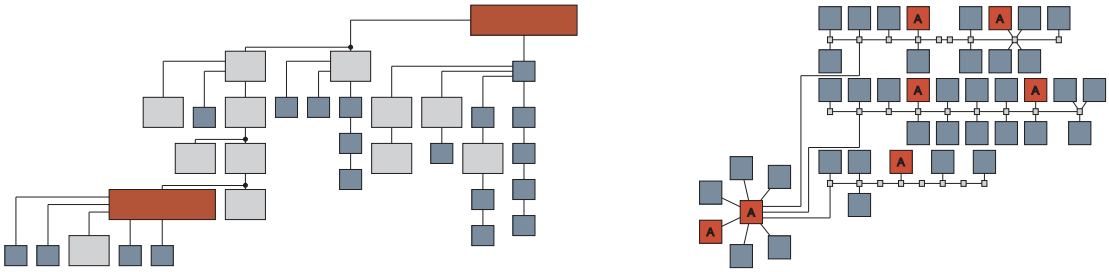


Figure 4: Bus routing

Level layout: The framework also contains a so-called level layout that is suitable for diagrams of logic networks and analogous control systems in which the components are arranged in vertical levels. A channel router is used to draw nets between the component levels [GPS03]. Level layout seams to be similar to the well known layer layout due to Sugiyama [STT81]. However, the level layout does not change the order of components within the levels.

Tree layout: This includes techniques that allow to draw networks with tree-like structures. Various basic principles regarding the direction of growth, adjustment and the order of components can be selected in accordance with corresponding parameters. The tree layout follows the principle of topological stability. For that purpose, the user can choose the root node of a tree as well as the order of subtrees by graphical interactions. Additional, special network features like stars and busses are supported.



(a) Bottom left tree (b) Tree with star and buses

Figure 5: Various tree layouts

4 Applications

In recent years, various different solutions have been developed on the basis of the domain framework presented here. Some of them are briefly described below.

4.1 TOP-Energy - Toolkit for Optimization of Industrial Energy Systems

The main aim of TOP-Energy is to support energy consultants in analyzing and optimizing industrial energy supply systems by providing modules for documentation, simulation and evaluation of energy systems with respect to energetic, economic and environmental aspects.
TOP-Energy consists of two major parts: CASTool and a set of modules. The former supplies the services of a modern GUI-application such as module-sensitive dialogs and presentations, flow sheet editing and report generation. Figure 6 shows the flow sheet editor in TOP-Energy for the modeling of energy supply systems. A flow sheet is TOP-Energy's visual language to model energy systems and can contain different kinds of energy components. Components represent technical objects (e.g. chiller) as well as non-material objects (e.g. energy rates). The flow sheets are used by a simulator module to calculate energy demands [AWKP04]. The visual language of flow sheets includes some connection rules to connect only pins of the same medium as well as rule definitions of hyper-nets and flow directions. Because of the marginal spatial comprehensiveness of these systems, a special placement procedure is not necessary. The included local alignment operations and collision detections are adequate. However, strong internal linking structures require an appropriate routing procedure.
Based on the first TOP-Energy application, the software was enhanced to a common CASTool-based framework for software solutions on the field of energy systems [WHA07].

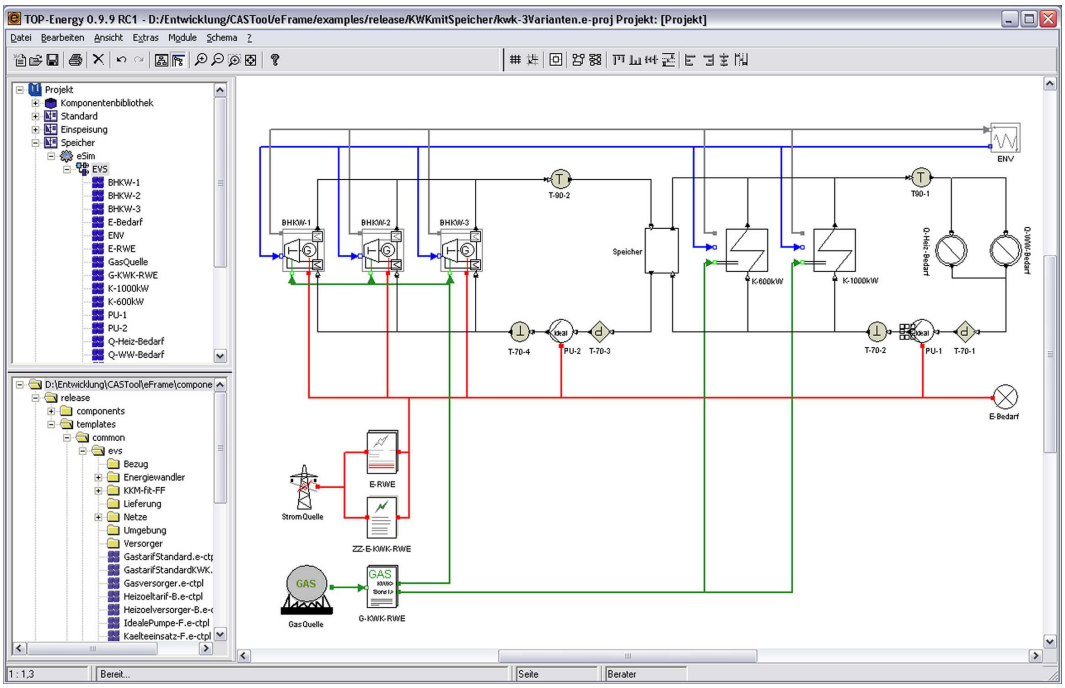


Figure 6: Flow Sheet in TOP-Energy

4.2 VotAn - Requirements Engineering Tool for Software of Technical Systems

VotAn is a tool for requirements engineering of software solutions in technical engineering environments, especially in the field of automation. It provides a model-based approach with focus on a product model. VotAn includes a few well-defined term structures (thesaurus, taxonomy) containing domain-based knowledge for some application areas. Besides product objects, this structure includes terms for functional and non-functional requirements in form of templates, so-called VotAn-Objects. In general, VotAn supports, among other things, the following activities:

- acquisition, specification, structuring, tracing and revision of requirements in a systematic way by further using of standard (third party) software
- adaptable methodical guidance for the structuring of the requirement specification including their documentation in different forms
- template creation for self-made reusable VotAn-objects

VotAn supports different schematic illustrations for the requirements modeling. Each VotAn-Object can be represented in such a scheme. For that purpose, a VotAn-Object possess several proxies and in fact, these proxies are assigned to a scheme. Besides representations to show structural information (e.g. UML use case diagrams), there exist also schemes that control dialog sequences (see Figure 7). The visual language for these sequences is designed according to UML activity diagrams wherein a acitivity is a VotAn proxy that opens a form. This form shows data

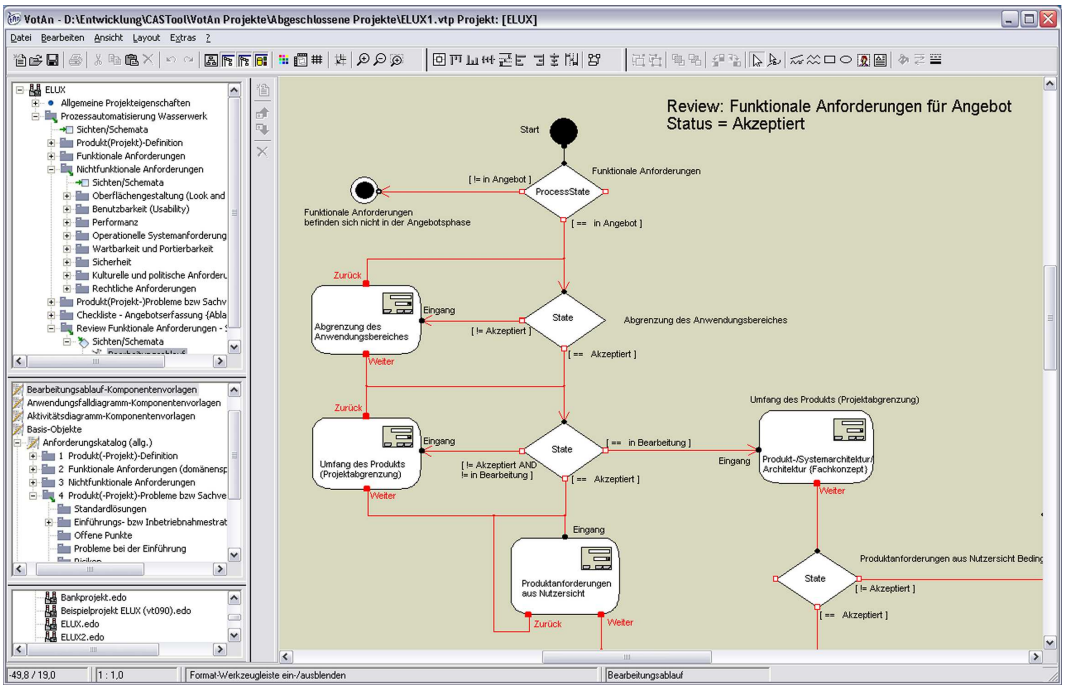


Figure 7: Activity Sequence Scheme in VotAn

of a VotAn-Object that the proxy belongs to, using the framework‘s generic dialogue generation. The conditions in that diagrams are proxies, too. These control the sequence in dependence of a value of a VotAn-Object’s parameter. Additional, some common diagram elements (e.g. message dialogs in form of activities and user controlled branchings in form of conditions) are available. As a result, the user can define wizard-like data administration sequences. Again, these can be stored as templates and reused as domain-specific process model for different projects.

4.3 SwitchLay - Switch Cabinet Layout

Switching stations can be found in modern factories and buildings everywhere. They are utilized as supervisory station to control and observe different technical processes. Electrical/ electromechanical engineering systems (ECAD-systems) are often used to design switchboards. As design result, circuit diagrams are mainly developed. But for the physical design of a switchboard they offer no assistance [VSP04].
SwitchLay possesses two methods for layout on the mounting plates of switch cabinets: placement and routing. These are quite different layout tasks as in the application descried before. In SwitchLay, a physical layout is required, in which the components have to be full-scale placed, and routing means wiring of cable in a physical environment.
The task of the placement technique is to accommodate a specified set of electrical devices on the mounting plate. For this purpose, a pattern of a channel frame can be selected for the mounting plate. The electrical devices must then be assigned to the individual facets between the channels.

Any affinities or incompatibilities between the electrical devices must be taken into account. Note that the pattern merely describes the basic architecture of the channel frame and the existence of certain rails. During placement, the channel frame needs to be resized depending on space requirements.
The key problem of routing is the decomposition of hypernets. The clamps involved in a hypernet must be connected by a tree that branches out only at the clamps. The clamps have limited valences for technical reasons. For each hypernet, the algorithm forms partial trees, which are successively connected. An extended form of Dijkstra's algorithm is used [VSP04].

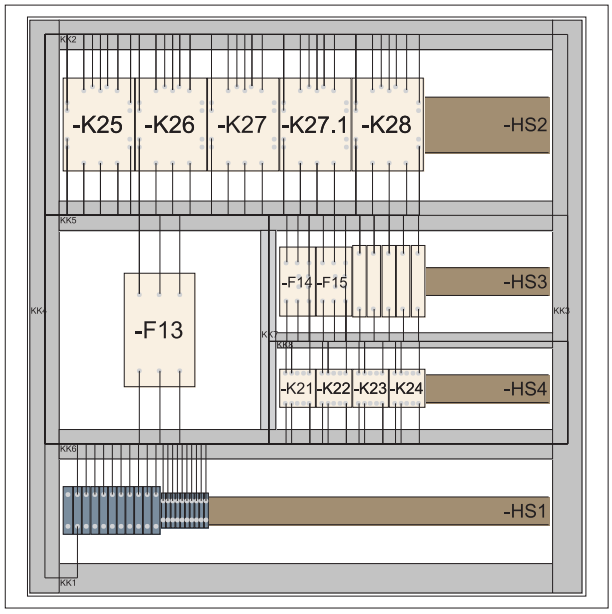


Figure 8: Switch Cabinet Layout

SwitchLay seems to be quite different to applications including logical net-like structures, but de facto, all SwitchLay components could be modeled by ELADO (see Section 2.1). Consequently, SwitchLay was developed as a prototype based on CASTool to investigate the layout methods and show the layout results. Currently, the routing method is included in a third party ECAD-system.

5 Experiences

We use the described framework in application development since a couple of years. More precisely, we have developed the framework from prototypes, which have been developed in research projects. Hence, we used a reactive strategy to develop our framework as a product line infrastructure. This way implicated some necessary redesign processes. As the result, we have now a powerful software basis for application development in the field of graph-based engineering systems. The main approach, the same data model for all applications, has the advantage of using common software components. Especially the GUI components for the ELADO structure model and generic dialogs and forms for the ELADO application model make rapid prototyping possible. Certainly for end products, special GUI elements especially for application data must be implemented. Furthermore, the same date model comprised the data exchange between different applications in a native way.
Based on the exiting results, we changed our framework development recently to a proactive strategy. We gain the same experience as [Dit04], that the implementation of a lot of functionalities in commands such as model manipulations makes the code difficult to read. Additional, by calling sub commands in commands, dependencies increased and as consequence, side-effects have a negative impact. Currently, in a re-design phase, we integrate some command stored

common functionalities in framework libraries and modules.
The system configurability and extensibility necessitate the abstraction level described here. Together with the abstract ELADO application data, the system's complexity increases, which is why particular attention had to be paid to system performance.
In order to cover areas of usage which are as large as possible, the system has been developed to run on various system environments by means of platform-independent libraries. All framework classes are written in C++ using open-source third party libraries such as wxWidgets [SHC05] and Apache Xerces. So far, the framework, including layout methods, comprises about 850 classes and approx. 300,000 lines of code.

6 Related Works

With the increasing popularity of UML in the end of 1990s, the use of visual techniques and graph-based tools increased. Therefore, a large number of (Meta)CASE tools have been developed. The main objective of these systems is to provide software modeling techniques as well as tools for source code generating. Independently of that, many systems for modeling, programming as well as simulating of technical applications with visual modeling techniques (e.g. LabView and Matlab Simulink) are widely used. These systems are designed for non-application specific languages and require special user knowledge.
In contrast, CASTool offers the development of software systems for engineering applications with user-defined, domain-specific, visual languages and, therefore, provides the modeling of technical systems in an easier way. This and the possibility to extend application-logic (e.g. simulators) enables the end-user to deal with complex issues in a domain-specific environment.
Besides many visual modeling tools [JM03], a few frameworks for such applications have been built up to date. The Graphical Modeling Framework (GMF [N06]) is such a framework. GMF is a bridge between the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF). Another framework for the development of graph-based applications is UPGRADE [BJSW02] that is not only used to develop software engineering tools too. It provides many of the framework features descried above (e.g customizability, extensibility as well as convenient layout methods). GMF provides a number of visual editors for the application development process. In contrast, CASTool contains only an editor for tree-structured configuration files and a graphical editor to create component appearances. To describe connection rules, CASTool includes a declarative way in form of configuration files. This is rather weak compared to the visual languages provided by GMF and PROGRES. A visual description language for design rules was developed in [FPA05] and will be integrated in the framework by future works.
The main difference between the frameworks above and CASTool is that CASTool is focused on engineering systems including net-like structures. For this purpose, CASTool's abstraction level is lower. Certainly, the GMF and UPGRADE are more general and powerful in the sense that these can be used for the development of various applications. But for such engineering systems which can be represented by ELADO and which need to store besides the structure information a lot of application data, the described framework is more suitable.

7 Conclusion

The framework described above enables custom-made engineering systems to be produced efficiently. The chosen approach of using a lower abstract meta model enables the implementation of many common functionalities for graph-based engineering systems in the software basis. The simple adaptability of the system supports the supply of ergonomic user interfaces and enables individual configurations. As a result, changing user requirements can easily be taken into account in the latter phases of application development and even after the system has been completed. The presented solutions demonstrate that the system approach can be used for application development, rapid prototyping as well as basis for further frameworks.
One can easily imagine that CASTool can be used not only for engineering systems but also for other similar systems (related to the data model and required system functionalities) of different domains. In particular, the possibility of being able to generically add functionality to the system and to integrate this functionality into the overall architecture opens up a broad sphere of applications for CASTool.
Future work will involve evolving research prototypes into products and developing other layout methods as well as searching for new fields of application.

Acknowledgements: The authors are grateful to the Federal Ministry of Economics and Technology of Germany as well as the Federal Ministry of Educations and Research of Germany for the financial support of several projects on this subject.

Bibliography

[AWKP04] E. Augenstein, G. Wrobel, I. Kuperjans, M. Pleßow. TOP-ENERGY - Computational Support for Energy System Engineering Processes. In Tsahalis (ed.), *Proceedings of the 1st International Conference "From Scientific Computing to Computational Engineering"*. Volume 3, pp. 1284–1291. Patras University Press, Athens, Greece, September 2004.

[BJSW02] B. Böhler, B. Jäger, D. Schleicher, B. Westfechtel. UPGRADE: A Framework for Building Graph-Based Interactive Tools. In Mens et al. (eds.), *Proceedings International Workshop on Graph-Based Tools (GraBaTs 2002)*. Electronic Notes in Theoretical Computer Science 72(2). Elsevier, Barcelona, Spain, October 2002.

[BP90] K.-F. Böhring, F. N. Paulisch. Using Constraints to Achieve Stability in Automatic Graph Layout Algorithms. In *ACM SIGCHI Conference on Human Factors in Computing Systems*. ACM SIGCHI, pp. 43–51. Seattle, WA, April 1-5 1990.

[BPPS01] T. Bartsch, M. Pleßow, M. Pocher, H.-W. Schmidt. Ein universelles System für die Projektierung von Prozeßleitsystemen. *ZwF Zeitschrift für wirtschaftlichen Fabrikbetrieb* 4(96):205–211, 2001.

[DETT99] G. Di Battista, P. Eades, R. Tamassia, I. G. Tollis. *Graph Drawing - Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[Dit04] K. Dittert. Softwarearchitekturen: Mythen und Legenden. *OBJELTspektrum* 3:34–39, Mai/Juni 2004.

[FPA05] R. Fröhling, M. Pocher, M. P. ans Alexej Lisounkin. Tools for Knowledge Acquisition, Modeling and Visualization Applied to Process Supervision. In Krüger et al. (eds.), *Industrial Simulation Conference*. Pp. 358 – 362. EUROSIS, June 2005.

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns*. Addison-Wesley, January 1995.

[GPS03] B. Goetze, M. Pleßow, P. Scheffler. Level-Layout für die Generierung grafischer Dokumentationen in der Leittechnik. *ZwF Zeitschrift für wirtschaftlichen Fabrikbetrieb* 3(98):97–101, 2003.

[JM03] M. Jünger, P. Mutzel. *Graph Drawing Software*. Mathematics and Visualization. Springer, 2003.

[N06] N. N. The Eclipse Foundation - Graphical Modeling Framework. http://www.eclipse.org/gmf/, 19.12.2006.

[PP98] M. Pleßow, M. Pocher. Intelligente Editoren - ein innovatives Konzept für die Erstellung von schematischen Darstellungen. In Dassow and Kruse (eds.), *Informatik '98*. Pp. 141 – 150. Springer-Verlag, Magdeburg, Germany, September 1998.

[PS89] M. Pleßow, P. Simeonov. Netlike Schematics and their Structure Description. In Menga and Kempe (eds.), *Workshop on Informatics in Indusrial Automation*. Pp. 144–163. CICIP, Berlin, GDR, Nevember 1989.

[SHC05] J. Smart, K. Hock, S. Csomor. *Cross-Platform GUI Programming with wxWidgets*. Bruce Peren's Open Source Series. Prentice Hall, USA, 2005.

[STT81] K. Sugiyama, S. Tagawa, M. Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics* SMC-11(2):109–129, 1981.

[VSP04] W. Vigerske, B. Stube, M. Pleßow. Automatic Wiring in Switch Cabinets. In Maropoulos and Schaefer (eds.), *Proceedings of the 1st International Conference on Electrical/Electromechanical Computer Aided Design and Engineering*. Pp. 90–93. The University of Durham, School of Engineering, Durham, UK, November 2004.

[WHA07] G. Wrobel, S. Herbergs, E. Augenstein. TOP-Energy - Ein Framework für Softwarelösungen in der Energietechnik. In *8. Internationale Tagung Wirtschaftsinformatik*. Karlsruhe, Germany, February/March 2007.

Imposing Hierarchy on a Graph

Brendan Sheehan, Benoit Gaudin and Aaron Quigley

Systems Research Group[1]
University College Dublin, Belfield, Dublin 4, Ireland
First.Last@ucd.ie

Abstract: This paper investigates a way of imposing a hierarchy on a graph representing data and their relationships with each other. The hierarchy is imposed by clustering. First a tree structure is imposed on the initial graph, then a *k*-partite structure is imposed on each previously obtained cluster. Imposing a tree exposes the hierarchical structure of the graph as well as providing an abstraction of the data. In this study, three kinds of merge operations are considered and their composition is shown to yield a tree with a maximal number of vertices in which vertices in the tree are associated with disjoint connected subgraphs. These subgraphs are subsequently transformed into k-partite graphs using similar merge operations. These merges also ensure that the obtained tree is proper with respect to the hierarchy imposed on the data.

A detailed example of the technique's application in exposing the structure of protein interaction networks is described. The example focuses on the MAPK cell signalling pathway. The merge operations help expose where signal regulation occurs within the pathway and from other signalling pathways within the cell.

Keywords: Graph visualization, clustering technique, tree, *k*-partite graphs.

1 Introduction

Graphs representing data often possess a large number of vertices and edges. As a result standard graph drawing techniques [2] cannot help the user in exploring the data and extracting useful information. This study deals with transforming a graph into a better understood graph structure that forms the basis by which it is more convenient to achieve these tasks. The main idea consists of computing a tree which possesses less vertices than the initial graph but also introduces a tree structure on the data.

Clustering the vertices of a graph introduces abstraction and a node hierarchy. This technique reduces the number of vertices rendering the graph more readable. The vertices can be clustered in different ways, achieving certain goals. In [7] for example, the authors cluster vertices to obtain a planar abstraction of the initial graph. The resulting graph does not contain any cross-edges, improving the readability. Planarity is a good criterion to impose, but some graphs are structured (like trees or grids), and this feature can be exploited to assist the user in understanding the structure of the graph. In [10] for example, the authors show that users prefer hierarchical layout when exploring the data set. For this reason, abstracting the graph as a tree is considered as a reasonable approach to graph exploration in this study. Extracting a tree from a graph is usually done by means of *spanning tree* computation techniques. But spanning trees do not satisfy all our goals. The number of nodes of a spanning tree is the same as the number of vertices of the graph. There have been attempts to visualize large networks using spanning trees [12] but such layouts do not minimize the number of vertices that the user has to read.

There exist other methods to expose the tree-like structure of a graph such as tree-decomposition [6, 3]. However the mapping of vertices to clusters is not one to one. This means that it is hard for the user to associate underlying vertices with such a decomposition.

[1] This work is supported by the Irish Research Council for Science, Engineering and Technology: funded by the National Development Plan in association with Microsoft Research

Reducing the number of vertices of the graph is clearly interesting when dealing with large graphs. Structuring the data set is also of great interest for the user to browse and select a relevant subset. The hierarchical structure of a tree is particularly relevant for data that already possesses an implicit hierarchy. Consider for instance a social network represented by a graph whose vertices model people and whose edges model the contacts between them. It is interesting to pick one person and visualize their degree of relatedness with others.In the case where there is no tree structure to exploit, if there is for instance no legitimate root, then a k-partite graph structure offers a good way to display the hierarchy of the data.

In this study, a tree is computed from the initial graph by means of operators called merges that are inspired by tree-width techniques ([3]). Intuitively, a merge collapses some vertices of the initial graph into clusters and updates the edges so that two clusters are connected if they share a common edge. Three merges are considered in this paper. The first one, introduced in [4], yields a tree whose number of vertices is maximal. This tree represents an abstraction of the initial graph, enabling the user to browse the smaller graph. If one of the vertices of the tree appears to be of interest, then the user may want to see the part of the initial graph to which it corresponds. Often it is desirable that clusters of the graph are connected. Since this is not ensured by the merge previously mentioned, a second merge operation is considered, which has to be applied together with the first merge operation (the two merge operations have to be composed). Finally, the third merge introduced enables visualization of the contents of clusters as k-partite graphs.

In section 2 some notation and preliminaries are introduced. The first merge operation is presented in Section 3, the second merge operation in Section 4 and the third merge is introduced in Section 5. It is shown in these sections that their composition yields a tree with maximal number of vertices. The nodes of this tree are obtained by clustering and the contents of a cluster is itself cluster to result in a k-partite graph. Finally, an example is provided based on protein-protein interaction data (see [1]) in Section 6.

2 Preliminaries

A simple graph G is a couple (V_G, E_G) where V_G is the set of vertices of G and E_G is its set of edges, such that $\{(x,x') \in V_G^2 | x = x'\} \subseteq E_G \subseteq V_G^2$. A graph G is said to be empty if $V_G = \emptyset$. Finally, G is said to be undirected if for all $(x,y) \in E_G$, it is also true that $(y,x) \in E_G$. The set of undirected simple graphs is denoted $\mathbf{G}$. In what follows, only undirected graphs will be considered.

Given a non-empty graph G and two vertices x,y of G, if $(x,y) \in E_G$ then x and y are said to be neighbors. Moreover, a path p of G between x and y is a sequence of edges $(x_i,x_{i+1})_{1\leq i\leq k}$ such that $x = x_1$, $y = x_{k+1}$ and for all $i \in \{1,\ldots,k\}$, $(x_i,x_{i+1}) \in E_G$. The empty path is denoted ε and represents the empty sequence of edges.

If $p = (x_i,x_{i+1})_{1\leq i\leq k}$ is a path of G, then any sub-sequence $(x_i,x_{i+1})_{k_1\leq i\leq k_2}$ with $k_1 \geq 1$ and $k_2 \leq k$ is called a sub-path of p and is denoted $p_{x_{k_1},x_{k_2}}$. For instance, if p equals

$$(x_1,x_2),(x_2,x_3),(x_3,x_4),(x_5,x_6),(x_6,x_7),(x_8,x_9)$$

then p_{x_3,x_6} equals $(x_3,x_4),(x_5,x_6)$. If p and p' are two paths and p (resp. p') is empty, then the concatenation of p and p', denoted $p.p'$, equals p' (resp. p). Now if neither p nor p' is empty and $p = ((x_i,x_{i+1}))_{1\leq i\leq k}$ and $p' = ((x'_j,x'_{j+1}))_{1\leq j\leq k'}$ and $x_{k+1} = x'_1$, then $p.p'$ is the path

$$(x_1,x_2).(x_2,x_3)\ldots(x_k,x_{k+1}).(x'_1,x'_2)\ldots(x'_{k'},x'_{k'+1})$$

The length of a path p, denoted $|p|$, is defined as the following:

$$|p| = \begin{cases} 0 & \text{if } p = \varepsilon \\ |p'| & \text{if } p = (x,x').p' \text{ with } x = x' \\ |p'|+1 & \text{if } p = (x,x').p' \text{ with } x \neq x' \end{cases}$$

For instance, considering the path p defined above, $|p| = 6$ and $|(x_1, x_1).p| = 6$ also.

Given a connected simple undirected graph G and a vertex r of G, the distance function $d_{G,r}(\cdot)$ is defined for all vertices x of G by

$$d_{G,r}(x) = \min\{|p| \in \mathbb{N} |\ p \text{ is a path in } G \text{ between } r \text{ and } x\} \tag{1}$$

As an example, for all vertices n of graph G given in Figure 1, the value of $d_{G,r}(n)$ is written down on the left side of Figure 1.

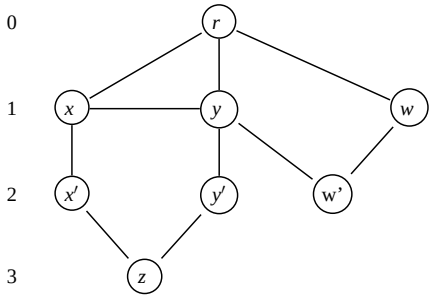


Figure 1: A graph G.

A graph is said to be k-partite if it is possible to partition vertices into k subsets such that two vertices belonging to the same partition are not neighbors. The goal of this study is to compute a tree and k-partite graphs with maximal number of vertices, by transforming a graph G. The kind of transformations under consideration are called *merges*:

Definition 1 A merge is an operator from **G** to **G** which possesses the following properties: if $G' \in \mathbf{G}$ is the result of a merge applied to $G \in \mathbf{G}$, then

(a) $V_{G'}$ is a partition of V_G (i.e $V_{G'} \subseteq 2^{V_G}$ and $\bigcup_{X \in V_{G'}} X = V_G$ and $\forall X, Y \in V_{G'}, X \cap Y = \emptyset$).

(b) for $X, Y \in V_{G'}$,
$$(X,Y) \in E_{G'} \Longleftrightarrow \exists x \in X, y \in Y, \text{ s.t } (x,y) \in E_G$$

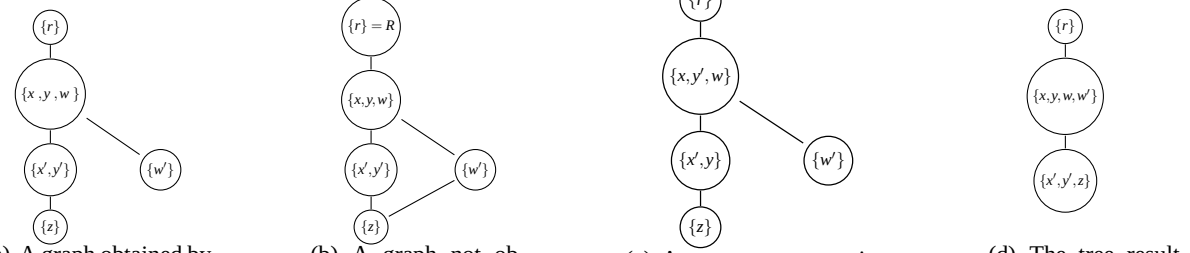
Thus a merge is an operator which collapses together several vertices of a graph to obtain a new vertex. The edges incident to the collapsed vertices then incident to the newly obtained vertex. Note that a graph obtained by merging, using the first merge operation, is different from a graph minor. A merge corresponds to the repeated applications of vertex-contractions, and a minor corresponds the repeated applications of edge-contractions and edge-deletions.

Figure 3(a) represents a graph obtained by merging from the graph G of Figure 1. But Figure 3(b) represents a graph which cannot be obtained from G by merging. There is no edge between w' and z in G, and there is one between $\{w'\}$ and $\{z\}$.

Considering a particular vertex r of a graph G as well as the distance function $d_{G,r}$, a hierarchy is explicitly introduced over the vertices of G in order to facilitate the understanding of the data by the user: r is the higher element in the hierarchy and the further from r a vertex is the lower it is in the hierarchy[2]. It is usually desirable that a merge applied to G *preserves the hierarchy*. Formally, M *preserves the hierarchy* of G according to r if $\forall x, y \in V_G, \forall X, Y \in V_{M(G)}$ s.t $x \in X$ and $y \in Y$,

$$d_{G,r}(x) \leq d_{G,r}(y) \Rightarrow d_{M(G),R}(X) \leq d_{M(G),R}(Y) \tag{2}$$

[2] Note that if $d_{G,r}(x)$ is greater than $d_{G,r}(y)$ and $d_{G,r}(y)$ is greater than $d_{G,r}(x)$ then $x = y$ does not hold in general.



(a) A graph obtained by merging.

(b) A graph not obtained by merging.

(c) A merge not preserving the hierarchy.

(d) The tree resulting from applying M_1 and M_2.

Figure 2:

where R is the vertex of $M(G)$ containing r. The graph G' represented in Figure 3(a) is obtained by a merge preserving the hierarchy of G (see Figure 1) according to r. This is not the case for the merge resulting in the graph of Figure 3(c). If R denotes the vertex $\{r\}$, then $d_{G',R}(\{x,y',w\}) < d_{G',r}(\{x',y\})$ and $d_{G,r}(x') < d_{G,r}(y')$. This means that x' is lower than y' in the hierarchy involved by G', although it is the opposite in G. In the case of a social network for instance, a merge like the one yielded in G' would then reverse the fact that the person x' plays a more important role than the person y'. This seems awkward in some situations. For this reason, merges avoiding this kind of reversion are considered in this study.

Merging vertices of a graph represents an interesting way to provide an abstraction. If this abstraction is done in a reasonable way, merging could provide a basis by which to browse the graph. Indeed one could browse the result of a merge and decide to enter into details by browsing the part of the initial graph represented by a vertex obtained by merging. Such a part of the initial graph is called an *associated graph*.

Definition 2 Let G be a graph and M be a merge. Given a vertex X of $M(G)$, the graph associated to X according to G, denoted $G'(X)$ represents the sub-graph G' of G defined by:

- $V_{G'} = \{x \in V_G | x \in X\}$
- $E_{G'} = \{(x,y) \in E_G | x \in X \wedge y \in X\}$

Even if G is a connected graph, this property does not hold for any associated graph. But when considering entering into details of a vertex of a graph obtained by merging, it is often preferable that a connected sub-graph of G is associated with it. For this reason, we will consider in section 4 merges which ensure that all associated sub-graphs are connected. In what follows, such a graph will be called a *deeply connected graph* according to G.

Lemma 1 is an interesting lemma showing that the existence of path in a graph G is preserved by merging.

Lemma 1 *Let G and G' be two graphs such that G' is obtained from G by applying a merge operator. Let $x,y \in V_G$ and $X,Y \in V_{G'}$ such that $x \in X$ and $y \in Y$. If there exists a path in G between x and y, then there exists a path in G' between X and Y.*

Section 3 aims at providing a means of imposing an optimal tree from any undirected connected simple and non empty graph. Section 4 aims at providing an algorithm to be applied on the previous tree so that a deeply connected one is obtained. Section 5 deals with imposing a k-partite graph on the associated graph of the previously obtained clusters.

3 Imposing a Tree

In this section how using merges to impose a tree on a graph is considered. In [4] a merge yielding a tree is introduced. We show in this section that this merge, denoted M_1 in what follows, preserves the hierarchy introduced by the distance function from a given vertex and yields a tree with maximal number of vertices. M_1 is actually defined from an equivalence relation $\mathbf{R}$ and is then shown to merge as few vertices as possible to result in a tree (see theorem 1).

If G is an undirected simple and non-empty graph and r is a vertex of G, the relation $\mathbf{R}$ on V_G according to r is defined as the following:

$$x\mathbf{R}y \iff (d(x) = d(y)) \wedge \big(\exists z, p, p',\, p_{x,z} = p \wedge p'_{y,z} = p' \wedge \forall x' \in p, \forall y' \in p',\, d(x) \leq d(x') \wedge d(y) \leq d(y')\big) \tag{3}$$

Basically, two vertices are related according to $\mathbf{R}$ if there exists a path from each of them leading to the same vertex by only traversing vertices that are further away from r. Considering that r would be the root of a tree, then the distance function from r would represent the depth of each vertex in this tree. Two vertices would then be related by $\mathbf{R}$ if there exists a vertex whose depth is greater and from which there is a path to both of them. The idea is then to merge iteratively two such vertices together into a single one to obtain a tree.

Note that the relation $\mathbf{R}$ is reflexive and symmetric. This implies that the transitive closure $\mathbf{R}^+$ of $\mathbf{R}$ is an equivalence relation over the set of vertices of $\mathcal{G}$. By definition, $x\mathbf{R}^+y$ if and only if there exists a sequence $(x_i)_{1\leq i\leq k}$ such that $x\mathbf{R}x_1$, $x_k\mathbf{R}y$ and for all $i \in \{1,\ldots,k-1\}$, $x_i\mathbf{R}x_{i+1}$.

It is possible to compute the classes of equivalence associated to $\mathbf{R}^+$ by searching for pairs of vertices related to each other by $\mathbf{R}$. The idea would then be to merge all the vertices of a same class to obtain a tree. Considering Figure 1 with $d_{G,r}(r) = 0$, $d_{G,r}(x) = d_{G,r}(y) = d_{G,r}(w) = 1$, $d_{G,r}(x') = d_{G,r}(y') = d_{G,r}(w') = 2$ and $d_{G,r}(z) = 3$, the corresponding classes of equivalence are $\{x,y,w\}$, $\{x',y'\}$, $\{w'\}$ and $\{z\}$. An algorithm implementing the merge M_1 can be found in [4].

Definition 3 Let G be a simple undirected non-empty graph and let r be a vertex of G. $M_1(G,r)$ denotes the merge defined by:

- $V_{M_1(G,r)} = \{\text{classes of equivalence of } \mathbf{R}^+\}$
- $E_{M_1(G,r)} = \{(X,Y) \in V^2_{M_1(G,r)} \,|\, \exists x \in X, y \in Y,\, (x,y) \in E_G\}$

As an example, the graph of Figure 3(a) represents $M_1(G,r)$ where G is the graph of Figure 1.
The operator defined by Definition 3 is clearly a merge according to Definition 1. Moreover, $M_1(G,r)$ is a tree rooted in $\{r\}$ since no path from two vertices at the same level can lead to a deeper vertex. It is also worth noting that each vertex of the obtained tree $M_1(G,r)$ consists of a set of vertices of G. Moreover, given $X \in V_{M_1(G,r)}$, if $r \in R$ then

$$\forall x \in X,\; d_{M_1(G,r),R}(X) = d_{G,r}(x)$$

This equation implies in particular that M_1 preserves the hierarchy according to r.

Theorem 1 is actually the main result of this section. It shows that M_1 is a merge which yield a tree whose number of vertices is maximal, while preserving the hierarchy introduced by the distance function.

Theorem 1 *Let G be a undirected simple non-empty graph and r be a vertex of G. $M_1(G,r)$ is the maximal (in the sense of the number of vertices) tree rooted in r, preserving the hierarchy, and obtained from G by merging.*

Proof. (proof of Theorem 1)
If $M_1(G,r)$ was not an optimal tree, then there exists an optimal merge M such that $M(G)$ is a tree as well as two different vertices X and Y of $M_1(G,r)$ and two vertices x and y of G such that $x\mathbf{R}^*y$ but $x \in X$ and $y \in Y$. Since the merge M also preserves the hierarchy, $d_{M(G),R}(X) = d_{M(G),R}(Y)$ holds. Now since $x\mathbf{R}^*y$, then it exists a path between x and y such that only vertices, whose depth is greater than $d_{G,r}(x)$, are traversed. Now according to Lemma 1, there exists such a path between X and Y in $M(G)$. In that case $M(G)$ can not be a tree. This contradicts the assumptions and so $M_1(G,r)$ is not optimal.

□

4 On the Computation of a Deeply Connected Tree

The tree obtained after applying merge M_1 to a graph does not necessarily yield a tree whose vertices abstract connected sub-graphs of G. In this section a merge denoted M_2 is then introduced to achieve this goal. This merge is performed on $M_1(G,r)$. The idea is to merge couples of vertices for which the associated sub-graphs are not connected in the initial graph.

Definition 4 Given a graph G and a tree T obtained applying a merging operator M on G, M_2 is the maximal (in number of vertices) merging operator such that

$$\forall X \in V_{M_2(T)}, \forall x,y \in V_G, x,y \in X \Rightarrow$$
$$\exists \text{ a path } p \text{ in } G \text{ between } x \text{ and } y$$

The following algorithm gives a way to merge the vertices so that for each vertex of the obtained tree, its associated sub-graph is connected.

Algorithm 1

- **Input:** a tree T obtained by applying merge M_1
- **Output:** a tree T' which the graph associated to any vertex is connected.
- Determine for each vertex of T if the associated graph is connected (denoted 1-vertex) or not (denoted 0-vertex).
- From the lower levels to the upper one,
  (1) Merge the 0-vertices with its parent.
  (2) performs action 1 till the obtained tree is deeply connected.

This principle relies on the fact that if two vertices x and x' are merged to yield a vertex whose associated graph is connected, then x is the parent of x' or vice-versa. If it was not the case then any vertex of n would not be a neighbor of a vertex of x' in the initial graph. This implies that more merges should be performed to obtain a connected graph associated to the vertex $\{x,x'\}$.

It is also worth noting that if x is the parent of x' then the graph of the vertex $\{x,x'\}$ resulting from their merging is not necessarily connected. But this becomes true if the graph associated to x is connected (because x is the parent of x' and then each vertex of x' is connected to a vertex of x).

Theorem 2 *Algorithm 1 implements Merge M_2 and then yields a deeply connected tree with maximal number of vertices, when applied to $M_1(G,r)$.*

Proof. First, note that Algorithm 1 always terminates. This holds because the merging applied in action 1 provides a tree whose number of nodes is strictly lower than the initial tree. Moreover the tree, obtained from G by merging and containing only one node, is deeply connected.

Note also that Algorithm 1 always yields a tree. This is due to the fact that the merges performed in action 1 correspond to edges-contraction and cannot then add cycles to the tree.

The optimality of the solution provided by Algorithm 1 is ensured by two features:

- First, any 0-vertex has to be merged with an other node.
- Moreover, according to the Definition 3, any vertex x of G is either a neighbor of vertices belonging to the same vertex X of $M_1(G,r)$ as x, or belonging to a neighbor of X. This means that any 0-vertex has to be merged with its parent or one of its children. If a 0-vertex is merge with one of its children, then the obtained node will be a 1-node, but the number of 1-node will remain the same as in the previous tree. If now a 0-vertex is merge with its parents:
  - if the parent is a 1-vertex, the case is similar as the first one: the obtained vertex is a 1-node and the number of 1-node remains the same.
  - If the parent is a 0-node, it might happen that the resulting node is a 1-node and then the number of 1-nodes would increase.

  We can then deduce that merging a 0-node with its parent is more beneficial to obtain an optimal solution.

□

Figure 3(d) shows the resulting tree obtained by applying successively merges M_1 and M_2 to the graph G given in Figure 1, considering Vertex r as a root for the imposed tree. According to Theorem 2, this tree is the maximal deeply connected one and obtained by merging from G, considering r as a root. Moreover, M_2 clearly preserves the hierarchy since only edge-contractions are performed.

5 Imposing a k-partite Graph on an Associated Graph

When applying the merges considered in section 3 and 4, the initial graph is clustered so that a tree is obtained. This provides greater readability which is particularly relevant if the initial graph is dense and possesses a large number of vertices. But it is also of great interest for the user to visualize what is inside a cluster, *i.e* the associated graph. Since the initial graph can be dense and large, so may be the associated graphs. It is then of interest to apply merges to clusters and then offer a more readable visualization of the associated graphs. However, imposing a tree on them which preserves the hierarchy is not possible. There are indeed at least two nodes with minimal depth, in each cluster. There is then no legitimate root to form the basis to impose a tree.

In this section k-partite graphs are considered to be imposed on the associated graphs. Indeed it can be shown that it is possible to cluster each associated graph so that a k-partite graph, with maximal number of vertices and preserving the hierarchy, is obtained. With that aim, the merge M_3 is introduced.

Definition 5 Let G be a non empty connected undirected graph and r be a vertex of G. Let us denote G' an associated graph of $M_2(M_1(G,r))$. Merge M_3 is defined such that

- $V_{M_3(G')} = \{A \subseteq V_G |\ \forall x,y \in A,\ (x,y) \in E_G \text{ and } d_{G,r}(x) = d_{G,r}(y)\}$
- $E_{M_3(G')} = \{(X,Y) \in V^2_{M_3(G')} |\ \exists x \in X \text{ and } y \in Y \text{ with } (x,y) \in E_G\}$

As shown by Theorem 3, applying merge M_3 to each cluster of $M_2(M_1(G,r))$ yields a k-partite graph preserving the hierarchy according to r with maximal number of vertices.

Theorem 3 *Let G be a non empty connected undirected graph and r be a vertex of G. Let us denote G' an associated graph of $M_2(M_1(G,r))$. $M_3(G')$ is a k-partite preserving the hierarchy of G according to r with maximal number of vertices.*

Proof. First, let us introduce sets $(V_i)_i$ of vertices of G' such that x and y are in the same set V_i if and only if $d_{G,r}(x) = d_{G,r}(y)$. The only edges preventing G' from being a k-partite graph according to sets V_i are the ones between two edges of the same depth. But merge M_3 merges vertices so that these edges are hidden in a cluster. M_3 clearly preserves the hierarchy according to r since only vertices of the same depth are merged. Plus the sets $(V_i)_i$ are defined such that they reflect this hierarchy, partitioning the vertices according to their depth. Merge M_3 is then optimal since only vertices linked to an other vertex belonging to the same set V_i are merged.

□

Theorem 3 offers a way to impose a k-partite graph structure on each associated graph. Referring to Figure 1 and only considering the levels 1, 2 and 3, the resulting graph is not a k-partite graph since x and y are neighbors and on the same level. The merge operation M_3 results in the same graph except that x and y are merged.

The hierarchy introduced in Section 2 to impose a tree is preserved. This ensures that users are not confused when entering into details and reading the contents of a cluster. Since the clustered associated graphs are k-partite, a layered layout can be applied so that two vertices of the same layer are not neighbors. But the readability of the obtained k-partite graphs is even better in this case than a typical k-partite graph. If the chosen layering consists in gathering in the same layer the vertices with same depth, then no edge crosses any layer. This property comes from the fact that the layers are defined according to the depth of the vertices.

6 Example: Exploration of Intra-Cellular Signaling Cascades

The merge operations described in the previous sections are quite general in nature and so their primary use is for the purpose of exploration with little *a priori* knowledge about the general structure of the graphical data set being investigated. A question that might be asked of such data is to what extent the graphical data set adheres to a particular structure. In the context of the above merge operations it is useful to ask to what extent a data set adheres to a hierarchical structure. Examples of such data sets are protein-protein interaction networks that describe the observed interactions between proteins resulting from biological experimental methods (e.g [8, 13]). In this section we will describe the properties of such data sets, what sort of questions biologists may ask of such data sets and how imposing a hierarchical structure as described in the previous sections on such data can assist in understanding the underlying structure of the data.

6.1 Properties of Protein-Protein Interaction Data

Molecular interactions play an important role in determining the behavior of cells. The involved molecules can be proteins and the study of protein interactions helps to understand activities such as differentiation, development and proliferation of cells. Protein-protein interaction (PPI) networks are graphical data sets that describe the observed interactions between proteins and other molecules using particular experimental methods. Proteins and other molecules (e.g. Ca^{2+}) correspond to the nodes of the graph while observed interactions correspond to edges. Proteins interact with each other typically to alter the behaviour of another protein. Many of the interactions in these networks are well documented and have

been confirmed repeatedly through various experiments. Some sequences of interactions are well known and play a direct role in particular cell behaviors. These sequences are called *pathways*. Examples of such pathways include the MAPK (involved in the cell proliferation process) and JNK-c-JUN (responding to cell stress) pathways. However, other less well understood proteins are involved in controlling these pathways. These proteins are known as regulatory or scaffolding proteins. An example of such a protein is Ste5p [9] which is found in yeast (*S.cerevisiae*). The absence of Ste5p results in the yeast cell becoming sterile. Regulatory proteins are also involved in coordinating activity between various pathways.

It is important to know, given the current data, to what extent we can understand the behavior of these networks. The development of new experimental methods to detect possible protein-protein interactions has vastly increased the biologist's understanding of how cells function. However these *high-throughput* techniques for detecting interactions can result in many false negatives and positives. While there is an abundance of well organized and easily accessible interaction data sets [15, 14, 1] there is little additional information associated with the interactions such as level of confidence, locus or dynamics of reactions with which to judge the authenticity of such interactions. These experimental methods also rarely reproduce the same results which results in data sets from each experiment having little overlap. The ultimate goal of these experiments is to produce a mechanistic understanding of the activities in these networks. As it stands however there is little scope to achieve such a result given the current data sets and so it is best to focus on the structural properties of such networks.

6.2 Possible Queries on PPI Data

Given that we are restricted to discussing structural properties of protein interaction networks, what sort of questions can a biologist ask of such data? Often the questions are graph theoretic in nature. For example, a set of proteins interacting with each other corresponds to a clique in the network. Existence of such structures allows the investigator to abstract and simplify the network [5].The biologist may also try to infer behaviour or roles from the structural evidence. An investigator may ask which interactions are core to a particular pathway and which are regulatory or otherwise. Another common approach is to apply understanding of the structure of one interaction network such as yeast to the structure of a less well understood network such as the human cell [9].

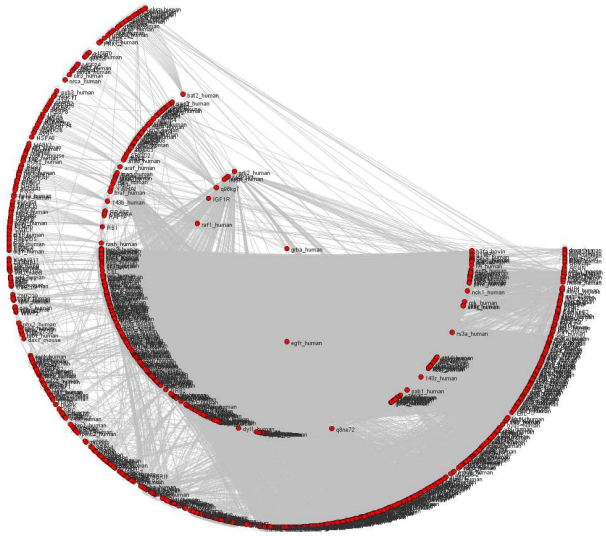


Figure 3: The initial protein interaction network before imposing a tree

6.3 Applying Hierarchy to PPI data

Our hierarchical merge operations can be applied to analyze the structure of protein interaction networks.It is natural to assume in many cases that a hierarchical structure describes the sequence of interactions in the cell. For example when the human growth hormone receptor (grba_human) is activated on the surface of the cell, many possible sequences of interactions are activated. Of interest is where the tree assumption is violated, indicating the presence of regulatory proteins. One can imagine a tree-like cascade of possible interactions with the activated human growth hormone receptor as the root. The first merge operation, M_1, can be used to induce such a structure on the data. It is a reasonable argument, given the data, that the resulting unconnected clusters are not meaningful protein interaction clusters. To produce biologically more meaningful (connected) clusters the merge M_2 is performed. Finally if the user wishes to investigate the structure inside a cluster with respect to the chosen root, merge M_3 produces a suitable abstraction.

6.4 Visualizing Protein Interaction Networks

We represent the protein-focused visualization using the standard node-link representation. Clusters are either represented as:

- An ellipse whose size is proportional to the number of underlying vertices it contains if all vertices are on the same level
- An ellipse containing a smaller radial drawing of the underlying graph whose underlying vertices are clustered using the first merge operation

We use a radial tree layout [2] since it handles broad, shallow trees quite well. The user may focus on visualizing the underlying graph within each cluster by either drawing the underlying graph using a force-directed layout [2] if all vertices are on the same level of the underlying hierarchy or as a layered Sugiyama style drawing [11] if the underlying vertices are found on multiple levels of the underlying hierarchy in which connected vertices on the same level are clustered together to create a k-partite graph.

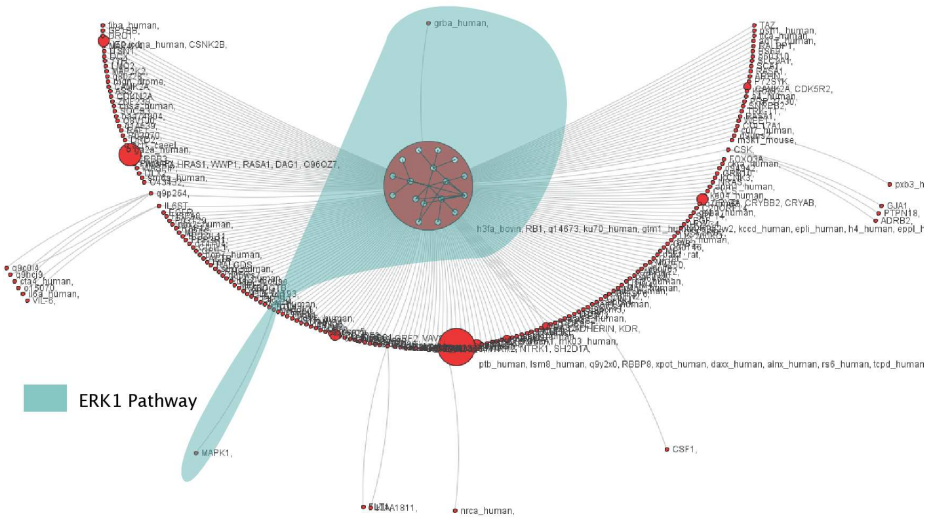


Figure 4: Imposing a tree on a protein interaction network

6.5 Data Source

We retrieved the data from the CPath protein interaction database using the protein human growth factor receptor-bound protein 10 as the focus protein. We constructed the underlying graph using vertices that were at most three interactions away from the focus protein. Our queries to the CPath web service were of the form:

http://cbio.mskcc.org/cpath/webservice.do?version=1.0&cmd=get_by_interactor_id&
q=CPATH_ID&format=psi_mi&startIndex=0&organism=9606&maxHits=50

This resulted in a graph containing 875 vertices. The imposed tree algorithm was applied to the underlying graph resulting in a clustered graph containing 210 vertices. The resulting graph is displayed in figure 4.

6.6 Discussion

The user is presented with in interface to scroll, zoom and investigate clusters in the graph. There is also a search facility to identify where a particular protein is in the visualization. Upon investigation it was found that there is no regulatory protein interacting with all proteins of the ERK1 pathway (involving the MAPK1 protein). Regulation of the ERK1 pathway, based on the evidence provided, occurs higher up the pathway. ERK1 is involved with cell proliferation. It is known from the study of the yeast cell that there exist what are known as scaffolding proteins, such as Ste5p, that interact with all parts of the ERK1 pathway in yeast and help regulate it. Such a protein does not exist for the human ERK1 pathway. So figure 4 shows that the behavior of the human cell is different from the behavior of the yeast cell, at least regarding proliferation.

In addition a significant number of proteins particularly on the leaf nodes of level 2 in figure 4 could be identified as being potentially spurious if they interact with only one other protein in the network and their function is unclear. Other spurious interactions have been absorbed into various clusters and investigation inside clusters aids in the further refinement of the tree.

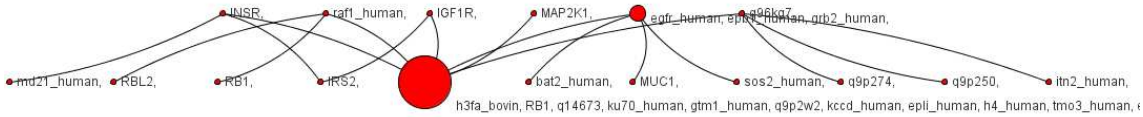


Figure 5: Inside a cluster

7 Conclusion

This study introduces a method to impose a tree on graph, by merging vertices. This method consists of two steps. The first one provides a tree with maximal number of vertices and preserves the hierarchy. The second one is meant to be applied to the previous tree. It provides a new tree whose nodes are associated to connected part of the initial graph. The composition of these two steps then yields a tree with maximal number of nodes such that the associated graphs are connected. Moreover the contents of each nodes of the tree, is itself clustered to result in a k-partite graph preserving the hierarchy introduced by the imposed tree.

We applied our approach to visualize protein interactions in human cells. It has been shown that although some proteins regulate all parts of the proliferation pathway of the yeast cell, there is no evidence of such a protein playing a similar role in the human cell. In the process of applying graph transformations to this data we discovered that dealing with biological data poses significantly different challenges

when compared to transforming software structure for example. The very fact that the data is inherently uncertain requires that graph transformation techniques need to handle precision and certainty if they are to be relevant to transforming such data.

As other future work, other structures to impose on the initial graph could be considered. Since a graph that does not possess a significant hierarchical structure is likely to possess a grid like structure imposing grids on a graph should be considered. Alternatively, we could consider extending the hierarchical notion further by investigating imposing polytrees on the initial graph. It would also make sense to look at how other more established graph transformation techniques can be applied to this problem.

Bibliography

[1] Cancer pathway database: http://cbio.mskcc.org/cpath.

[2] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[3] Hans L. Bodlaender. A tourist guide through treewidth. *Acta Cybernetica*, 11:1–21, 1993.

[4] F. Boutin and M.Hascot. Focus dependent multi-level graph clustering. *Proceedings of the Conference on Advanced Visual Interface, AVI04*, 2004.

[5] D. Bu, Y. Zhao, L. Cai, H. Xue, X. Zhu, H. Lu, J. Zhang, S. Sun, L. Ling, L. Zhang, G. Li, and R. Chen. Topological structure analysis of the protein-protein interaction network in budding yeast. *Nucleic Acids Research*, 31(9):2443–50, May 2003.

[6] R. Diestel. *Graph Theory*. Springer-Verlag Heidelberg, 2005.

[7] Q. Feng. *Algorithms for Drawing Clustered Graphs*. PhD thesis, University of Newcastle, 1997.

[8] S. Fields and PL. Bartel. The two-hybrid system. a personal view. *Methods Mol. Biol.*, 177:3–8, 2001.

[9] G.Pearson, F.Robinson, T.B.Gibson, Bing-E Xu, M.Karandikar, K.Berman, and M.H.Cobb. Mitogen-activated protein (map) kinase pathways : Regulation and physiological functions. *Endocrine reviews (Endocr. rev.)*, 22, no2:153–183, 2001.

[10] W. Huang, S.H. Hong, and P.Eades. Layout effects on sociogram perception. *Graph Drawing*, pages 262–273, 2005.

[11] S.Tagawa K.Sugiyama and M.toda. Methods for visual understandins of hierarchical system structures. *IEEE Transactions on Systems, Man. and Cybernetics, SMC-11(2)*, pages 109–125, 1981.

[12] K.A. Lehmann, S. Kottler, and M. Kaufmann. Visualizing large and clustered networks. *Graph Drawing 06*, Septembre 2006.

[13] M. Mann, RC. Hendrickson, and A. Pandey. Analysis of proteins and proteomes by mass spectrometry. *Ann. rev. Biochem.*, 70:437–473, 2001.

[14] H. W. Mewes, C. Amid, R. Arnold, D. Frishman, U. Gldener, G. Mannhaupt, M. Mnsterktter, P. Pagel, N. Strack, V. Stmpflen, J. Warfsmann, and A. Ruepp. Mips: analysis and annotation of proteins from whole genomes. *Nucleic Acids Res*, 32 Database issue, January 2004.

[15] A. Zanzoni, Montecchi L. Palazzi, M. Quondam, G. Ausiello, Helmer M. Citterich, and G. Cesareni. Mint: a molecular interaction database. *FEBS Lett*, 513(1):135–140, February 2002.

The Jury is still out: A Comparison of AGG, Fujaba, and PROGRES

Christian Fuss, Christof Mosler, Ulrike Ranger, and Erhard Schultchen

[fuss|mosler|ranger|schultchen]@i3.informatik.rwth-aachen.de
http://www-i3.informatik.rwth-aachen.de
Department of Computer Science 3 (Software Engineering)
RWTH Aachen University, Germany

Abstract: Graph transformation languages offer a declarative and visual programming method for software systems with complex data structures. Some of these languages have reached a level of maturity that allows not only conceptual but also practical use. This paper compares the three widespread graph transformation languages AGG, Fujaba, and PROGRES, considering their latest developments. The comparison is three-fold and regards conceptual aspects, language properties, and infrastructure features. Because of the different relevance of these aspects, we do not determine a clear winner but leave it to the reader.

Keywords: Graph Transformation Languages, AGG, Fujaba, PROGRES

1 Introduction

Graph transformation languages are one branch of visual programming languages, which provide good concepts for developing software tools. Some languages reached a level of maturity that allows utilization in practice.

In this paper, we compare the three widespread languages AGG, Fujaba, and PROGRES. Our goal is to point out main differences in conceptual aspects, language properties, and infrastructure features. There exist some comparisons of graph transformation languages dating back several years, e.g. [BTMS99, FNT98]. Comparisons that are more recent focus only on particular application areas, e.g. [Agr04] focuses on model integration aspects. In [VSV05], the runtime efficiency of the generated applications is examined. We do not only want to update the older comparisons, considering recent developments, but also lay focus on practical aspects. This paper shall support users when deciding, which language is most appropriate for his or her application.

This paper is structured as follows: In Section 2, we describe the aspects of the graph transformation languages that we examine and introduce a running application example. In the following sections, we study how each of the languages AGG, Fujaba, and PROGRES meets the stated requirements and describe specific aspects. Finally, Section 6, summarizes our comparison and points out strengths and weaknesses of each language.

2 Compared Aspects

The graph transformation languages are compared by different aspects, which are introduced in this section. We compare the theoretical concepts building each language's background, the language features when specifying a graph transformation system, and the infrastructure offered to edit and run these systems.

2.1 Theoretical Concepts

Graphs are clear and intuitive data structures, whose fundamentals are mathematically founded. Since the late 1960s, different approaches to *graph grammars* have been developed, which differ e.g. in their graph model, the expressiveness of transformation rules, and the definition of semantics. Basically, two main approaches can be distinguished, which are briefly described in the following.

The *algebraic approach* considers a graph as a 2-sorted algebra, where nodes and edges are typed, attributed, and identified. The *derivation* of a graph by applying a graph transformation rule is defined by *pushouts* known from category theory. The approach allows formal and easy to understand proofs of properties on graphs and on graph transformation rules, e.g. the amalgamation of graph transformation rules. Two different branches concerning the derivation have been evolved within the algebraic approach, namely the *double pushout approach* (DPO) [CEH+97] and the *single pushout approach* (SPO) [EHK+97]. In DPO, a derivation is constructed by two pushouts using a gluing graph between the left-hand side and the right-hand side of a transformation rule, which enables to reverse transformations. A graph transformation rule can only be applied if all edges incident to its match in the working graph and to the context graph are specified within the transformation rule, which leads to complex specifications. For this reason, SPO has been developed, which overcomes this restriction and constructs only one pushout for a derivation. Thus, the graph transformation rules are easier, but the theoretical properties of SPO are limited.

The *set-theoretic approach* [Nag79] offers an intuitive understanding of graph transformation systems, but does not provide a theoretical foundation that is as powerful as in the algebraic approach. Graphs are described as sets of nodes and edges and the effect of applying a graph transformation rule is defined by set-theoretic operations. In contrast to the algebraic approach, edges are considered as relations between nodes and thus are neither identified nor attributed. The approach allows more expressiveness within graph transformation rules, e.g. embedding rules, which enable user-defined embedding of a rewritten sub-graph in its context graph. Furthermore, the application of graph transformation rules can be managed by control structures offering a *backtracking* mechanism for determining matches of transformation rules. The approach does not provide any means for describing static and derived graph properties. These aspects are integrated in the *logic-oriented approach* [Sch91], which is an enhancement of the set-theoretic approach. The approach allows to define an explicit graph schema and uses *predicate logic formulas* for defining graphs and graph transformation rules.

Besides the fundamental approach, a graph language may be based on different programming paradigms. As all presented graph languages offer means for typing graph elements, we will analyze in how far they support the object-oriented paradigm. This includes providing type-specific attributes and methods, inheritance relations between types, and polymorphism.

2.2 Language Properties

In this subsection, we examine properties of graph transformation languages, concerning the *graph model* and *graph transformations*, in general. Some of the properties are similar to those compared in [BTMS99], some are owed to new developments in AGG, Fujaba, and PROGRES. Table 1 shows a feature matrix listing all properties. The sections on AGG, Fujaba, and PROGRES describe properties implemented for each language in detail.

Table 1: Feature matrix with language properties

Property			AGG	Fujaba	PROGRES
Graph Model	graphs	kind	directed, attributed, labeled	directed, attributed, labeled	directed, attributed, labeled
		graph schema	unchecked type graph	UML class diagram	graph schema with static rule-check
		integrity constraints	global event-condition rules with manual application	—	global and node-local ECA rules, schema constraints
	nodes	kind	typed, attributed, identified	typed, attributed, identified	typed, attributed, identified
		derived node types	multiple inheritance	[multiple] inheritance	multiple inheritance
	edges	kind	labeled, attributed, identified, directed, binary, between nodes	labeled, directed, binary, between nodes	labeled, directed, binary, between nodes
		derived edge types	—	paths (textual)	paths (materializable)
		constraints	—	ordered	—
	attributes	value types	Java objects/standard types	Java objects/standard types, node types	internal standard types, C types, node types, sets
		expressions	parsed Java expressions	unparsed Java expressions	parsed C or PROGRES expressions
		derived attributes	—	simulated using methods	directed equations
		meta attributes	—	const, static	const, static
Transformations	matching	homo-/isomorphic	global option	explicit folding per rule element	explicit folding per rule element
		multiple matches	[amalgamated subrules]	set nodes, for-each patterns	set nodes, star rules
	conditions	subgraphs	nodes, edges	nodes, optional nodes, set nodes, edges, paths, constraints	nodes, optional nodes, set nodes, edges, paths, restrictions, constraints
		NACs	neg. subgraphs	neg. nodes, neg. edges, neg. constraints	neg. nodes, neg. edges, neg. paths, neg. constraints
		attribute conditions	yes	yes	yes
	gluing/embedding		gluing	—	embedding
	signature		in parameters	in parameters, return value	in/out parameters
	control programming	mechanisms	iteration over layers	conditional, iteration, sequence, collaboration stmts, method calls	conditional, iteration, sequence, non-deterministic choice, transformation calls
		transactions	—	—	yes
		backtracking	—	—	yes

Graph Model

Graphs. The working graphs of all discussed languages are *directed, attributed, node- and edge-labeled*. The structure of the working graphs is constrained by *graph schemas* that define node and edge types and their relations. Transformation rules should be checked against the schema to avoid syntactical errors at specification time. *Integrity constraints* are used to prohibit certain patterns in the working graph. They are checked at runtime. Transformations of *hierarchical graphs* can be found in literature but are not implemented in any of the languages.

Nodes. In the three languages, nodes are generally *typed, attributed, and identified* elements. Node types can be derived from other types by *inheritance*.

Edges. Edges are typed, directed and connect two nodes in all three languages. Edges might be identifiable graph objects or represent an unidentified relation of graph objects. Further properties of edges are attribution and constraints (e.g. ordered or sorted edges). Derived edges in the form of paths can be used to simplify otherwise very complex rules. Edges between edges, inheritance of edges, and n-ary edges are supported in neither language.

Attributes. Besides the type label, graph elements might carry attributes, which are defined by the element type. Value types can be standard types, often borrowed from host languages like Java or C (evaluation of expressions might also be borrowed). *Derived attributes* are not set directly, but evaluated according to an equation that might reference other graph elements. Additionally, sets and graph elements are useful attribute values.

Graph Transformation Rules

Graph transformation rules describe possible transformations of the working graph. They can be divided into *compound rules*, combining other rules by control structures and *simple rules*. Simple rules have a left-hand side (LHS) and a right-hand side (RHS). If the LHS is found in the working graph (i.e. it can be matched), the match is replaced by the RHS.

Matching. A rule match is a morphism that maps a rule's LHS elements to elements from the working graph. If LHS elements are mapped to only one working graph element, the morphism is a homomorphism. Non-homomorphic constructs are e.g. set nodes, amalgamated rules (AGG), star-rules (PROGRES). If each LHS element is mapped to a different element from the working graph, the morphism is injective (default). Whether the matching is non-injective (i.e. one working graph element can play multiple transformation roles) might be determined per graph grammar, per rule, or per rule element.

Conditions. Conditions define constraints for rule applications. Structural conditions are found in the LHS of a rule and include nodes, optional nodes, set nodes, paths, and restriction expressions. *Restrictions* constrain the match of a rule node by attribute or structure conditions. Attribute conditions are defined by expressions referring to element attributes. Negative application conditions (NACs) [HHT96] define structures that must not be found in the working graph, if a rule is applied; these might be integrated into the LHS or separated and range from simple negative nodes and edges to negative paths and complete negative partial graphs.

Gluing/Embedding. Gluing means the merging of two nodes into one, which owns all incident edges and all non-conflicting attributes of both. Embedding is somehow similar: it allows the redirection of incident edges from one node to another.

Signature. Procedure-like signatures support the use of graph transformation rules in a way known from imperative programming. Input parameters allow the parameterization of rules, while output parameters let transformation results influence following rules.

Control Structures. With control structures, the definition of *compound rules* is possible by combination through conditional, iteration, and chaining statements. Statements with non-deterministic behavior and backtracking allow the convenient specification of many graph algorithms. The chaining of rules should be accompanied by transactions, in order to rollback a chain of rules if one fails.

2.3 Infrastructure

Besides concepts and language properties, the infrastructure, offered to edit, analyze, and run the graph transformation system is crucial to its applicability. A graph language environment should provide a *visual and textual editor* for specifications. It should allow free-hand as well

as syntax-directed editing. At least some *analyzing functions*, e.g. a sophisticated type checker, should be integrated to detect and explain inconsistencies with respect to the language's static semantics. Basic *layout algorithms* for the rules should be available in the editor.

For testing a specification, the language environment should provide an *interpreter*. During an interpreter session, the environment performs a sequence of graph transformations and visualizes the working graph. Different application strategies for transformation rules should be possible, e.g. a debugging mode allowing step-by-step execution. Additionally, a code generator should produce compilable source code for a general programming language to support the development of stand-alone applications. The generator's backend should be sufficiently flexible to allow the extension to further programming languages. A graphical framework providing access to the specified graph transformation rules should be available to obtain an executable application.

To store large graphs and support efficient manipulation of graph structures, a database should be provided. It should also support undo/redo of transformation rules and provide persistence for the working graph. Another requirement concerns the extensibility of the language environments. Monolithic architectures are hard to extend, while plug-in structures are more flexible.

Sometimes the user is confronted with limited choices concerning the platform for installation of the language environment. Therefore, the language environments should be available for at least the most common operating systems, and offer an easy and fast installation. Ideally, the environment should be implemented in a platform independent language like Java and be freely available. As all presented languages are distributed under the terms of the GNU (Lesser) General Public License.

2.4 Example

To explain the different aspects of each graph transformation system in the next sections, we introduce a simple example of a Shipping Company. The Shipping Company resembles the example used in [ERT99]. Its graph schema is illustrated in Figure 1 as a class diagram. In the example, Pallets of different weights are kept in Stores. Every Pallet has to be brought to a certain City by a Truck, which is modeled by a toDestination-edge storing also the due date. A Truck has a maximum loading weight (maxLoad) and stores its current weight (load). The order of a Truck's target cities is determined by a route, which is modeled by *ordered* onRoute-edges. The Truck is drivenBy an Employee of a Store. The boolean attribute onDuty indicates whether the Employee is at work.

Figure 2 shows the sample graph transformation rule loadUrgentPallet, which is used for loading

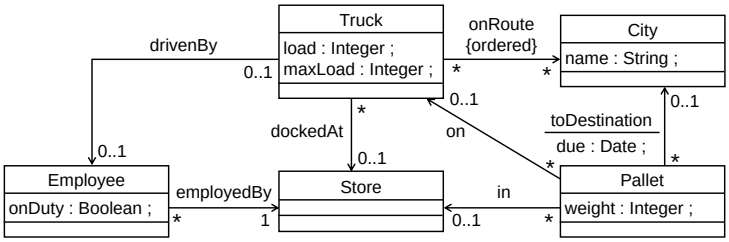


Figure 1: Graph schema of the Shipping Company

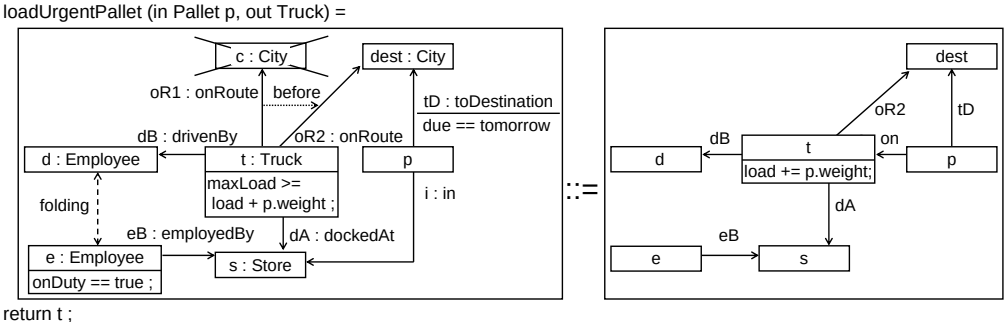


Figure 2: Graph transformation loadUrgentPallet

a given Pallet p in a suitable Truck t. The match for loadUrgentPallet is determined by the following constraints: The working graph is searched for the destination City dest and the current Store s of the given Pallet p, which is due tomorrow. Additionally, a Truck t dockedAt Store s has to be found, whose first target City is equal to the destination dest of Pallet p. This is modeled by the negative node City c, i.e. there exists no City c, which is before City dest on the route of Truck t. Furthermore, the maximum load of Truck t must not be exceeded by the weight of Pallet p. To load Pallet p on Truck t, an Employee is needed, which is onDuty. As even the driver d of Truck t may help to load Pallet p, if he is employed by Store s, Employees d and e are connected by a folding-construct. This enables the *non-injective* matching of the driver and the store employee in the working graph. A match found for the LHS is transformed according to the RHS: The in-edge incident to Pallet p is deleted and a new on-edge is created connecting Pallet p and Truck t. The load-attribute of Truck t is updated and t is returned.

3 AGG

Conceptually, AGG (Attributed Graph Grammar) [ERT99] follows the algebraic approach to graph transformation and implements single-pushout behavior. The implementation is based on the Colimit library [Wol98], which provides colimit construction for category theory of signatures and graph structures. Colimit could easily be used for the transformation of hierarchical graphs, but AGG does not support this.

An AGG graph grammar consists of a type graph, a start graph, and simple rules. Figure 3 shows a graph grammar for the Shipping Company example from Subsection 2.4.

The *type graph* contains an object-oriented description of node types, edge types, and their relations. Node types can be derived from other node types by multiple inheritance. Attributes can be defined for node and edge types. All constraints (attribute types, edges' source and target types and multiplicities) have to be checked manually within the rules. AGG does neither support derived edges (e.g. paths), derived attributes, nor meta attributes (e.g. *constant* or *static*). Although the Colimit library would allow complex edges, the language only supports binary edges between nodes. Edge constraints like ordering or sorting are not supported either, thus the ordered onRoute-edge has to be modeled as *edge-node-edge construct* with an ordering before-edge in the example's type graph (see Figure 3, top left). The AGG feature of graph constraints is not used in the example, with it one can define graph patterns and their conclusion to check

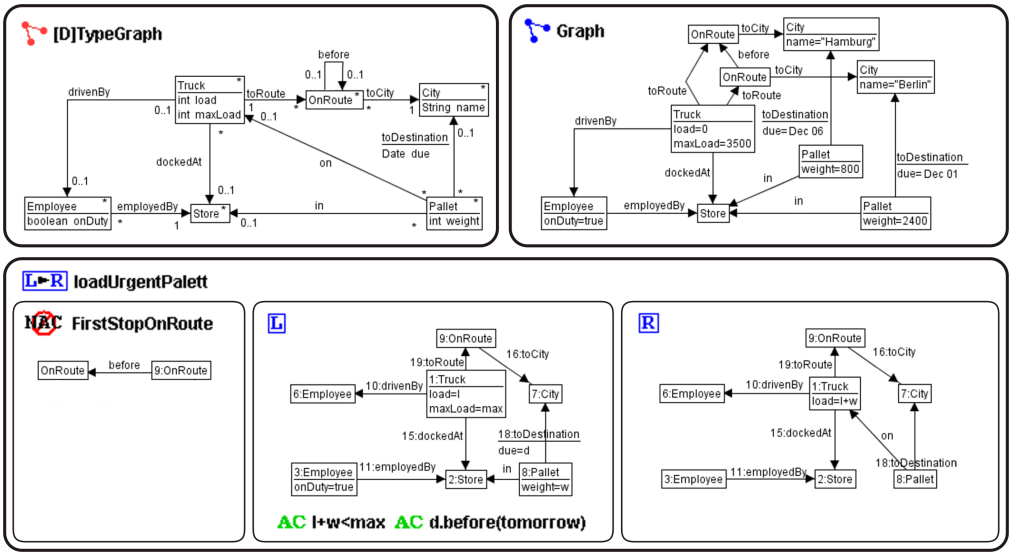


Figure 3: A simple AGG graph grammar for the Shipping Company example

structural properties of the working graph.

The *start graph* defines an initial working graph. All nodes and edges in a working graph are typed, identifiable, and might be attributed. Figure 3 (top right) shows a simple start graph for the example, with a small 3.5t Truck having two cities on his route (Berlin before Hamburg) and a driver, who is an Employee of the Store. Two Pallets with different weights are stored in the store, with destinations Berlin and Hamburg. They have to arrive on December 1st resp. 6th.

AGG only supports simple *rules*. The LHS consists only of nodes and edges (no other elements are available). Injective matching can be switched on and off globally[1]. With non-injective matching, the employee node from the working graph can be matched for the depicted rule in Figure 3 as node 6 (driver) *and* node 3 (store worker). NACs are subgraphs defined outside the LHS that must not be fulfilled. Here there must not be another OnRoute node before node 9. Attribute conditions, e.g. d.before(tomorrow)[2], are defined in a special attribute editor (not depicted) and can contain arbitrary Java expressions. The match is determined by the LHS, NACs, and attribute conditions. One feature not shown in the example is gluing, i.e. two nodes are merged into one node. The resulting node owns all non-conflicting attributes and edges. Conflicts have to be solved interactively by the user.

The execution of rules can be programmed slightly, by defining *layers* for the rules. Then the execution loops over the sequence of all rules on one layer, until none is executable anymore, then the loop is executed on the next layer until the last. Additionally, single rules can be selected for execution manually.

The editing of graph grammars is done by a graphical editor, which is completely built in Java and easily installed on different platforms. The editor has a GUI that is intuitive, but does not offer much support for syntax-directed editing. Positive is the good integration of the interpreter

[1] Non-homomorphic matching, i.e. multiple matches for one rule element, can be obtained in AGG with amalgamated subrules [TB94], which is an extension not yet publicly available.

[2] d is of type java.util.Date and the Java method before compares this date with another date (tomorrow).

into the AGG editor. The generation of executable code from the graph grammar is not possible but grammar specifications can be exported to XML files.

The TIGER framework [EEHT05] allows the generation of visual editors for an AGG graph grammar. For that, the graph grammar has to be decorated by a visual concrete syntax for all elements. The generated editors are GEF-based Eclipse-plugins, where the user can pick single rules for execution. The editors use AGG's Java API to interpret the graph grammar.

AGG is based on a very sound theory and the editor is simple to use and install. This allows easy testing of prototypical specifications. [MTR06] gives a good example of a small prototypical reengineering editor specified with AGG, relying on the notion of critical pair analysis. For an application in larger projects, code generation and control structures are missing.

4 Fujaba

Originally, the focus of Fujaba (From UML to Java And Back Again) was to provide a visual modeling tool based on UML diagrams and to generate Java code from these models. Meanwhile, Fujaba has been adapted to other metamodels like the Meta-Object-Facility (MOF) and other output formats like the EMF.

In Fujaba, graph schemas are modeled using simplified UML class diagrams, resembling the one shown in Figure 1. Classes can be attributed and any Java class or ordinal type is supported as attribute type. Derived attributes are not directly offered, but can be simulated by a method replacing the getter-method generated for the attribute. We therefore model the getLoad method to derive the truck's load. Thus, this attribute does not require manual update when pallets are loaded on the truck. Inheritance of classes is supported, although multiple inheritance is restricted to interfaces. Overloading of methods and polymorphism is handled by the Java environment at runtime. Attributed associations, inheritance on associations or n-ary relations are not supported. Fujaba provides ordered associations, which impose a total ordering on the link instances during runtime. This feature is well-suited to model the onRoute association.

The behavior of applications is modeled using so-called *Story Diagrams* [FNTZ98] which combine UML-collaboration with activity diagrams. From each Story Diagram, Fujaba generates a Java method operating according to the modeled transformation rule. Story Diagrams consist of one start and at least one stop activity, and an arbitrary number of *Story Patterns* operating on the runtime graph. These elements are connected through transitions. Story Patterns correspond to rules in AGG, but incorporate LHS and RHS into one diagram using the stereo-

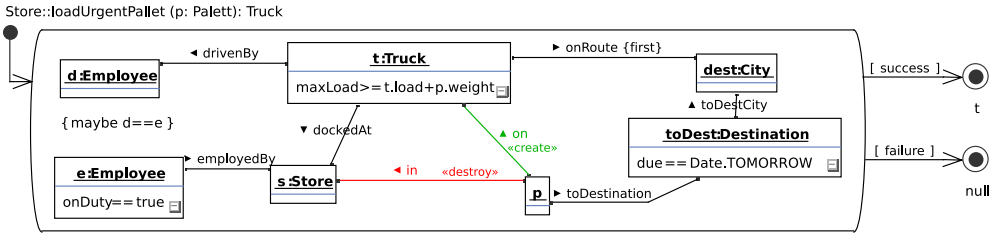


Figure 4: Fujaba Story Diagram loadUrgentPallet

types «create» and «destroy». For pattern matching, Fujaba offers obligatory, optional, set and negative node *variables*. By default, Fujaba creates injective morphisms from variables to objects, so that two variables are never bound to the same object. This behavior can be disabled per pair of variables. Attribute assertions may constrain the matched objects by an unparsed (thus arbitrary) Java expression. Furthermore, Fujaba supports obligatory, optional and negative edges between variables and textual path expressions. For ordered associations, additional constraints can be specified for the matching. Every pattern requires at least one *bound variable* e.g. provided by a parameter of the Story Diagram, the this object the method is invoked on, or variables bound in preceding patterns. From these bound variables, the other variables of the pattern are bound to objects from the runtime graph by traversing links of given type. Transformation rules are conducted after the complete pattern has been matched, and may create and delete elements, set attributes and call methods on matched objects.

Figure 4 shows a Story Diagram implementing the loadUrgentPallet transformation rule. The required bound variable is provided by the method parameter p, from which the other variables are bound. Attribute assertions are used to check if the Truck t is not overloaded and the given Pallet p needs urgent delivery (due attribute denotes tomorrow). Injective matching is disabled for variables d and e by adding the {maybe d==e} constraint. For the ordered onRoute association, {first} retrieves the first link from t to a City. If pattern matching succeeds, the runtime graph is transformed by removing the Pallet's in edge to the Store and creating an on edge to the Truck.

To model the control flow, Story Patterns may hold transitions to multiple successors. In the depicted example, two stop activities exist. By the transition guard [success], the left one is called when the transformation rule succeeds and returns the matched truck as return value. Otherwise, the right stop activity returns null. Transitions may form loops, causing repeated execution of Story Patterns. Also, *for-each*-patterns allow to process every match of a Story Pattern instead of only one match.

The formal background of Story Patterns is obtained from the logic-oriented approach described in Subsection 2.1. However, some of their semantic aspects are only incompletely defined (cf. [TMG06]). The Fujaba environment also performs very limited checks on the modeled diagrams, so the specifier is often not warned about erroneous specifications.

The generated source code can easily be integrated into existing projects or used in rapid-prototyping frameworks. eDOBS is a plugin for the Eclipse IDE which visualizes the runtime graph of a Fujaba-generated application. With the help of the CoObRA framework, generated applications are able to store their runtime states persistently. Recently, the graph-oriented database DRAGOS and the related UPGRADE framework were adapted to support Fujaba. Being entirely written in Java, Fujaba works on multiple platforms and is easy to set up. Besides the regular stand-alone application, an Eclipse-plugin embedding Fujaba into the IDE is under development.

Fujaba's advantage is its extensible architecture and the use of the well-known UML. For example, Fujaba has been applied in [BGS05] to model real-time systems, including appropriate extensions of the modeling language. Major disadvantages are the lack of a complete semantic definition and the rare validity checks.

5 PROGRES

PROGRES (PROgrammed GRaph REwriting System) [SWZ99] is the eldest of the presented graph languages and environments. The logic-oriented approach [Sch91] forms the basis of PROGRES, which offers a proprietary language allowing the specification of a graph schema and consistent graph transformation rules.

PROGRES provides various constructs for defining a *graph schema* of a specification. For node types, three different types of attributes can be defined: Intrinsic attributes, whose values are assigned directly, meta attributes, which constitute class attributes and thus have the same value for every instance, and derived attributes. Values of derived attributes are computed dependent on attribute values of other nodes and are automatically updated when their values are invalid. For example, the node type Truck shown in Figure 5 owns a derived load-attribute, whose value is the sum of all loaded Pallet weights. The Pallet weights are obtained by traversing the incoming on-edges of the Truck. PROGRES supports the object-oriented paradigm regarding node types, which includes inheritance relations between node types, polymorphism, type-specific attributes and methods. *Edge types* define the type name, the source and target node types, and their cardinalities. *Paths* may be modeled allowing complex navigations through the working graph, traversing arbitrary edges of different types. PROGRES also enables the specification of *graph constraints*, e.g. there are at most *n* instances of a certain node type within the working graph. If such a constraint is violated, an appropriate repair action can be executed. Based on the schema, *incremental analyzes* check the specification for inconsistencies and show appropriate error messages.

Besides the graph schema, PROGRES offers modeling of graph queries and graph transformation rules, which may have several input and output parameters. A *graph query* defines a test for the existence of a graph pattern in the working graph. A *graph transformation rule* modifies the working graph. For their execution, the underlying graph database DRAGOS [Böh04] provides transactions for graph operations (ensuring ACID-properties). For every transformation rule, *pre- and postconditions* may be specified, which imply constraints on the working graph before

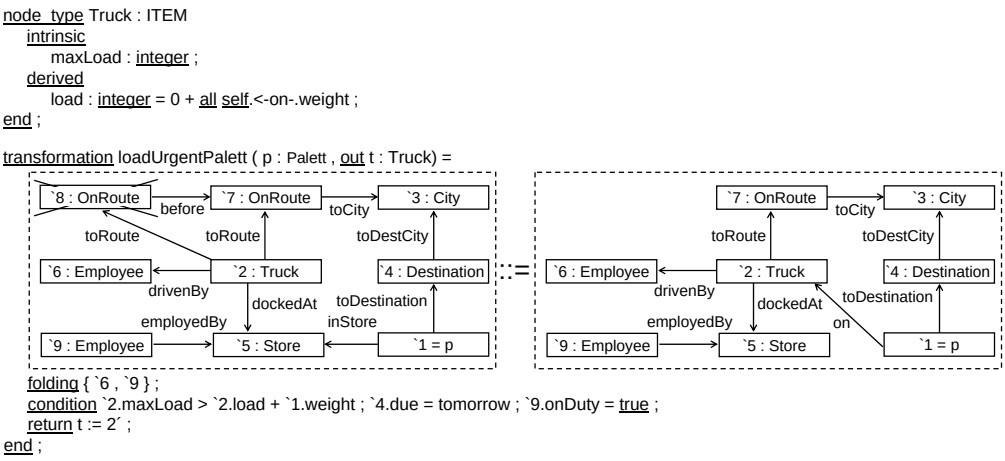


Figure 5: PROGRES transformation rule loadUrgentPallet

resp. after the execution of the rule. Furthermore, a *qualifier* determines if a transformation rule should be applied to one match or to all possible matches in parallel. Graph transformation rules are classified as *production* (simple rule) or *transaction* (compound rule).

Productions are similar to AGG rules and Story Patterns in Fujaba. They are visually specified and allow to create and delete nodes and edges. They are described by a LHS and a RHS, which may contain obligatory nodes and edges, paths, optional nodes, set nodes, and restrictions on nodes. NACs are modeled by negative nodes, edges, paths, and restrictions. Additionally, a production may have a *condition*- and a *transfer*-part to imply conditions on attribute values resp. to change the value of node attributes. PROGRES allows the specification of *embedding rules* for redirecting edges incident to deleted nodes and embedding new nodes into the working graph. The *folding*-statement enables the non-injective mapping of two nodes in the production to the same node in the working graph.

Figure 5 shows the PROGRES production loadUrgentPallet introduced in Subsection 2.4. The production uses two edge-node-edge constructs for the ordered onRoute-edge and the attributed toDestination-edge, as these sorts of edges are not supported by PROGRES. The folding-construct, the attribute conditions and the return-statement are represented as textual statements. As the load-attribute of Truck t is defined as derived attribute, its value is not assigned explicitly.

In contrast to productions, transactions contain control structures for combining transformation rules and queries. This includes to sequence transformation rules and to execute one of a set of rules non-deterministically. Furthermore, loop- and condition-statements may be used.

PROGRES is the most expressive graph language of the three presented languages and offers extensive support for modeling big software systems. But the proprietary language is fairly complex and difficult to learn. From a specification, C and Java source code can be generated. This code can be used for rapid prototyping by applying the UPGRADE-framework. AHEAD [JSW00] is a good example of an industrial-sized project specified with PROGRES.

The syntax-directed PROGRES editor, that also features an interpreter, guides the user well, but is not really intuitive. In addition, it is only available for Linux. A further disadvantage of PROGRES is its monolithic architecture, which makes the development and implementation of new language concepts difficult.

6 Summary

With the algebraic approach, AGG offers a graph transformation language with a sound theoretical basis. This offers convenient implementation possibilities for projects relying on theoretical notions. It provides a well-developed environment which can easily be installed and applied. The main disadvantage is the lack of control structures. Therefore, AGG still has to prove that it can be applied in large-scale projects.

The biggest advantage of Fujaba is its use of UML, which requires only little learning effort from the user. In addition, the vivid community is working intensively on improvements and further extensions. However, the language lacks a formal definition and shows some weak points when dealing with more complex graph transformation rules. Due to the lack of analyzes the user is not sufficiently guided during the specification process, often leading to malfunctioning.

PROGRES offers the most sophisticated language and an infrastructure with the highest level

of maturity. The experience with industrial-sized projects proves the practical usability. However, the environment does not conform to up-to-date standards, requiring a painstaking installation process and providing a relatively inconvenient interface, particularly to new users.

The jury is still out: Because of the different relevance of the compared aspects, we cannot give final advice, but leave it to the reader to decide which language to use.

Bibliography

[Agr04] A. Agrawal. Model Based Software Engineering, Graph Grammars and Graph Transformations. Area paper, EECS at Vanderbilt University, 2004.

[BGS05] S. Burmester, H. Giese, W. Schäfer. Model-Driven Architecture for Hard Real-Time Systems: From Platform Independent Models to Code. In *Proc. of the European Conf. on Model Driven Architecture - Foundations and Applications (ECMDA-FA'05), Nürnberg, Germany*. LNCS 3748, pp. 25–40. Springer, 2005.

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

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, A. Schürr. *Application of Graph Transformation to Visual Languages*. In [EEKR99], pp. 105–180, 1999.

[CEH+97] A. Corradini, H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner. *Algebraic Approaches to Graph Transformation – Part I: Basic Concepts and Double Pushout Approach*. In [Roz97], pp. 163–245, 1997.

[EEHT05] K. Ehrig, C. Ermel, S. Hänsgen, G. Taentzer. Generation of visual Editors as Eclipse Plug-ins. In *20th IEEE/ACM Int. Conf. on Automated Software Engineering, ASE'05*. Pp. 134–143. ACM Press, New York, 2005.

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

[EHK+97] H. Ehrig, R. Heckel, M. Korff, M. Löwe, L. Ribeiro, A. Wagner, A. Corradini. *Algebraic Approaches to Graph Transformation – Part II: Single Pushout Approach and Comparison with Double Pushout Approach*. In [Roz97], pp. 247–312, 1997.

[ERT99] C. Ermel, M. Rudolf, G. Taentzer. *The AGG Approach: Language and Environment*. In [EEKR99], pp. 551–603, 1999.

[FNT98] T. Fischer, J. Niere, L. Torunski. Konzeption und Realisierung einer integrierten Entwicklungsumgebung für UML, Java und Story-Driven-Modeling. Master Thesis, University of Paderborn, 1998.

[FNTZ98] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In Ehrig et al. (eds.),

6^{th} *Int. Workshop on Theory and Application of Graph Transformation (TAGT)*. LNCS 1764, pp. 296–309. Springer, 1998.

[GW06] H. Giese, B. Westfechtel (eds.). *Fujaba Days 2006*. Technical Report tr-ri-06-275. University of Paderborn, Germany, 2006.

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

[JSW00] D. Jäger, A. Schleicher, B. Westfechtel. AHEAD: A Graph-Based System for Modeling and Managing Development Processes. Pp. 325–339 in [NSM00].

[MTR06] T. Mens, G. Taentzer, O. Runge. Analysis Refactoring Dependencies using Graph Transformation. *Software Systems Modeling (SoSyM)*, 2006.

[Nag79] M. Nagl. *Graph-Grammatiken: Theorie, Anwendungen, Implementierung*. Vieweg Verlag, 1979.

[NSM00] M. Nagl, A. Schürr, M. Münch (eds.). *Int. Workshop on Applications of Graph Transformations with Industrial Relevance, AGTIVE'99*. LNCS 1779. Springer, 2000.

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

[Sch91] A. Schürr. *Operationales Spezifizieren mit programmierten Graphersetzungssystemen*. PhD-Thesis, RWTH Aachen University, 1991.

[SWZ99] A. Schürr, A. J. Winter, A. Zündorf. *The PROGRES Approach: Language and Environment*. In [EEKR99], pp. 487–550, 1999.

[TB94] G. Taentzer, M. Beyer. Amalgamated Graph Transformations and Their Use for Specifying AGG - an Algebraic Graph Grammar System. In *Int. Workshop on Graph Transformations in Computer Science*. Pp. 380–394. Springer, 1994.

[TMG06] M. Tichy, M. Meyer, H. Giese. On Semantic Issues in Story Diagrams. Pp. 10–14 in [GW06].

[VSV05] G. Varró, A. Schürr, D. Varró. Benchmarking for Graph Transformation. In *2005 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Pp. 79–88. IEEE Computer Society, 2005.

[Wol98] D. Wolz. *A Colimit Library for Graph Transformations and Algebraic Development Techniques*. PhD-Thesis, TU Berlin, 1998.