Electronic Communications of the EASST

Pre-Proceedings of the International Colloquium on Graph and Model Transformation – On the occasion of the 65th birthday of Hartmut Ehrig (GraMoT 2010)

Guest Editors: Claudia Ermel, Hartmut Ehrig, Fernando Orejas, Gabriele Taentzer
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Preface

This report is a collection of invited papers and position statements presented at the **International Colloquium on Graph and Model Transformation**, February 11-12, 2010, at Technische Universität Berlin on the occasion of the 65th birthday of Hartmut Ehrig. After the colloquium all contributions will be peer-reviewed, and the final versions of the accepted contributions will be published in a special issue of the ”Electronic Communications of the European Association of Software Science and Technology” (ECEASST).

Scope and Objectives of the Colloquium

Graphs are a general kind of models which have been used in various fields of computer science. On the one hand, they are well-suited to formally describe complex structures. On the other hand, the underlying structure of models, especially visual models, can be described best by graphs, due to their multidimensional extension. Graphs can be manipulated by graph transformation in a rule-based manner. Considering current trends in software development such as model driven engineering and model-integrated computing, there is an emerging need to describe model manipulations, model evolution, model semantics, etc. in a precise way. Recent research has shown that graph transformation is a promising formalism to specify model transformations.

The goal of the colloquium is to foster interaction between the graph transformation and the model transformation community in order to facilitate exchange of results and challenging problems. The graph transformation research community has built up a significant body of knowledge over the past 30 years and in addition to the theoretical base several practical implementations have been created. The research area of model transformations has recently been identified as a key subject in model-driven development. Graph transformations could offer an elegant theory and powerful concepts for the model-driven engineering of software systems, while the software engineering community can generate interesting challenges for the graph transformation community. Therefore, there is a need for strong interaction and inter-operation between these communities: the interchange of ideas, problems, and solutions will lead to major advances in both fields.

Invited Papers and Position Statements for the Colloquium

The workshop program has been organized in six technical sessions, in two days:

Thursday, February 11, 2010	**Friday, February 12, 2010**
Graph Transformation Techniques, Modeling with Graph and Net Transformations, Verification and Constraints, Modeling of Chemical and Biochemical Reactions	Model Transformations, Software System Modelling (incl. Panel Discussion)

In **Session 1 (Graph Transformation Techniques)**, hierarchical graphs are used to model a service and session calculus in the first contribution by A. Corradini, U. Montanari et al. In the second one by F. Orejas, the new concept of *symbolic attributed graphs* is introduced to deal with attributed graphs in contrast and comparison with the standard approaches to attributed graph transformation.

The paper by K. Ehrig and C. Ermel in **Session 2 (Modelling with Graph and Net Transformations)** focuses on the role of graphs and graph transformations in practical application areas like molecular analysis, model transformations, and medical information systems. The second contribution by F. Parisi-Presicce et al. proposes a new termination criterion for graph transformation systems with negative application conditions. Finally, an integration of Petri nets and high-level replacement systems, known as *reconfigurable Petri nets*, is used in the third paper by K. Hoffmann, J. Padberg et al. for formal modeling and analysis of flexible processes in communication based systems, like mobile ad-hoc networks and Skype.

In **Session 3 (Verification and Constraints)**, the first paper by T. Margaria et al. presents a second order value numbering as new optimization method to be used in the $M2L(Str)$ verification tool set for monadic second-order logic on strings. This new method allows applications to transformations on directed acyclic graphs. In the second paper by L. Ribeiro et al., the Event-B formal method and its theorem proving tools are proposed to analyze graph grammars, especially reachable states. Finally, in the third contribution by A. Habel et al., graph conditions – in the sense of nested application conditions – are extended to graph conditions with variables in order to improve the expressive power, especially concerning non-local properties like ”there exists a path”.

In **Session 4 (Modelling of Chemical and Biochemical Reactions)**, the contribution by R. Heckel et al. presents a methodology for extracting ordinary differential equations from stochastic graph transformation systems, especially based on a model for chemical reactions. In the second contribution G. Rozenberg introduces the new concept of reaction systems as a formal framework for biochemical reactions.

The first paper by S. Glesner et al. in **Session 5 (Model Transformations)** presents an approach of the VATES project using process algebraic techniques in order to integrate modeling, implementation, transformation, and verification stages of embedded system development. The second contribution by F.Hermann, B. König et al. presents specification and verification techniques for model transformations based on triple graph grammars and the *Double-Pushout-Approach with Borrowed Context* (DPO-BC). Finally, H.-J. Kreowski et al. show how to use graph transformation units in order to handle composition and correctness of model transformations.

In **Session 6 (Software System Modelling)**, the first contribution by G. Engels et al. is a paper on test-driven language derivation using the specification technique *Dynamic Meta Modelling* which is based on graph transformations. The remaining contributions are position statements for the Panel Discussion *Software System Modelling : Past, Present and Future*. The position statement of H. Ehrig discusses the state of the art and role of formal specification techniques in these three periods, starting from single techniques in the past and leading to certified, integrated, and visual techniques and environments in the future. The statement of M. Löwe discusses the role of graph transformations for agile software development, including concepts like software refactoring, test-first, extreme programming, or dynamic systems development. B. Mahr points out in his statement that the modeling of software systems is a task in which a complex interaction of models is being created. Finally, G. Taentzer shows that the theory of algebraic graph transformation can be used to show important properties of model transformations, like type consistency, functional behavior, as well as conflicts and dependencies between transformation steps.

We would like to thank all invited speakers and panelists for their contributions presented at the colloquium. Moreover we are looking forward to the special issue of ECEASST, where - after the review process - the final versions of the accepted contributions will be published.

Berlin, February 2010

Claudia Ermel,
Hartmut Ehrig,
Fernando Orejas,
Gabriele Taentzer

PC of GraMoT 2010

Welcome Address on Behalf of the ICGT Steering Committee

This spring it will be 10 years since the first meeting of the steering committee of the International Conference on Graph Transformation. By now we have had four successful conferences, in Barcelona, Rome, Natal, and Leicester, with the fifth coming up in Enschede this Autumn. Graph transformation is firmly established as a discipline and recognised both for its foundational contributions and applications.

None of this would have happened without Hartmut's scientific contribution and services to the community. Recognising its potential, Hartmut was among those who first formalised the general mechanism of rewriting on graphs in the early 70ies, based on the algebraic or double-pushout approach which is still amongst the most popular today. He went on to make seminal contributions to its conceptual foundation and mathematical theory and it is not an overstatement to say that the majority of papers published in the area today in some way rely on foundations Hartmut helped to establish. For example, two thirds of the papers presented at the last ICGT in 2008 are directly based on the algebraic approach or variants of it.

Besides founding this Berlin school of graph transformation, Hartmut has also been the driving force behind the formation and organisation of the community of people behind what is now the ICGT conference, but evolved via a series of international workshops and with support from a number of European projects such as COMPUGRAPH, APPLIGRAPH, GETGRATS and SEGRAVIS. After having led two instalments of the European Working Group COMPUGRAPH in the late 80ies and early 90ies, Hartmut kept his role as coordinator of the community, motivating and guiding others like myself, until he became chair of the ICGT steering committee in 2000.

Apart from his scientific achievements and organisational contributions, Hartmut's main legacy are the people he educated in his culture of theoretical research inspired by practical phenomena, formalised in terms of graphs, algebra and category theory. Equipped with this background, we are now able to address today's problems in computer science and beyond, build tools, design languages and algorithms for analysis, etc. while relying on solid foundations.

Dear Hartmut, on behalf of your students, the ICGT steering committee, and the graph transformation community at large I would like to thank you for your invaluable contributions and the support and guidance you are giving us. We hope that you keep driving us forward both scientifically and as a community for many years to come and wish you many happy returns of your anniversary.

Leicester, February 2010

Reiko Heckel, University of Leicester (UK)
Chair of the ICGT Steering Committee

Table of Contents

Modeling a Service and Session Calculus with Hierarchical Graph Transformation[*]

Roberto Bruni[1], **Andrea Corradini**[1], **and Ugo Montanari**[1]

[1] [bruni,andrea,ugo]@di.unipi.it
Dipartimento di Informatica, Università di Pisa, Italy

Abstract: Graph transformation techniques have been applied successfully to the modelling of process calculi, for example for equipping them with a truly concurrent semantics. Recently, there has been an increasing interest towards hierarchical structures both at the level of graph-based models, in order to represent explicitly the interplay between linking and containment (like in Milner's bigraphs), and at the level of process calculi, in order to deal with several logical notions of scoping (ambients, sessions and transactions, among others). In this paper we show how to encode a sophisticated calculus of services and nested sessions by exploiting a suitable flavour of hierarchical graphs. For the encoding of the processes of this calculus we benefit from a recently proposed algebra of graphs with nesting.

Keywords: Hierarchical graphs, service oriented architecture, process calculi, CaSPiS

1 Introduction

The use of graphs or diagrams of various kinds is pervasive in Computer Science, as they are very handy for describing in a two-dimensional space the logical or topological structure of systems, models, states, behaviors, computations, metamodels, and several other entities of interest; well-known examples are the graphical presentations of data structures (like lists and trees), of entity-relationship diagrams, of various kinds of automata and labeled transition systems, of static and behavioral UML diagrams (like class, message sequence and state diagrams), of computational formalisms like Petri nets, and so on.

The advantage of drawing graphs or diagrams, rather than using their underlying set-theoretical definition or some term-like linear syntax, lies in the fact that graphs emphasize relevant topological features of the systems or models they describe, like adjacency and connectivity of components, sharing of data and structures, causal dependencies, hierarchical structuring, among others, making such features easily understandable and detectable also to non-specialists. In several cases graphs provide a representation of models or systems at the "right" level of abstraction: for example, as drawings are always understood "up to isomorphism", the order in which nodes and arcs are drawn is typically irrelevant (unless some tacit drawing convention is enforced) and if the concrete identity of certain syntactical entities is irrelevant (e.g., the names of the states of a finite state automata), it is sufficient not to depict them in the drawing.

The use of graphs as a domain for the visualization of algebraically-specified systems, in general, and process calculi, in particular, has been pursued in a vast literature of which is not

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

possible to give a comprehensive account here (see, e.g., [BL05] and references therein), but one striking example is the research on "optimal" implementations for functional calculi [AG98]. Here we restrict the attention to the analysis of the concurrent behavior of process calculi with name passing, in the style of [Gad03, BMM06]. To this aim, there are several "graphical specification frameworks" which provide general techniques and/or tools for the graphical description of systems and, possibly, of their behavior, including Graph Transformation [Roz97], Bigraphical Reactive Systems [Mil06] and Synchronized Hyperedge Replacement [FHL$^+$06].

Recently, there has been an increasing interest towards hierarchical structures both at the level of graph-based models, in order to represent explicitly the interplay between linking and containment (like in Milner's bigraphs), and at the level of process calculi, in order to deal with several logical notions of scoping (ambients, sessions, and transactions, among others). The goal of the work summarized in this paper is to show how to encode both the static aspects and the dynamics of CaSPiS, a sophisticated calculus of services and nested sessions [BBDL08], by exploiting a suitable flavor of hierarchical graphs and corresponding transformation rules.

Following a methodological approach that has been applied recently to provide a graphical encoding of the static aspects of a variety of formalisms (including process calculi, workflow languages, entity relationship diagrams and others, see [BGL09, BGLa]), we will not present the graph encoding of CaSPiS processes directly, but we will exploit instead as an intermediate language a recently proposed algebra of hierarchical designs, which allows to reduce the representation distance between the considered formalisms.

The algebra is defined by an equational signature, whose operator symbols are interpreted as operations on graphs, and where the axioms formalize suitable properties of such operators. Therefore the terms of the initial algebra can be interpreted as graphs, and the axioms can be shown to be sound and complete with respect to the interpretation, in the sense that two terms are equivalent if and only if they denote the same graph (up to isomorphism). The interesting fact, is that the interpretation of the terms of the algebra can be given over different kinds of graphs, resulting in different layouts. As a typical example, the nested structure of designs can be interpreted adequately in a class of truly hierarchical graphs, where subgraphs can be encapsulated in hyperedges, or also can be rendered by over-imposing to a standard, flat hypergraph, a tree representing the hierarchy.

Therefore, the advantages of the use of an intermediate algebra for the encoding are twofold:

- the algebra provides explicit operators for parallel composition, nesting of components, names representing shared resources, local and global restriction, as well as aliasing mechanisms: the richness of such operators makes the encoding of process algebras like CaSPiS quite intuitive, less error-prone and easy to understand;
- the various interpretations of the terms of the algebra as different kinds of graphs can be defined once and for all, and reused for the encoding of several other formalisms.

In the next sections we shall first account for the algebra of hierarchical graphs, showing, only at the informal level, how it can be interpreted over both hierarchical graphs and term graphs. Next we introduce the syntax and the reduction semantics of (a significant fragment of) CaSPiS, and show how the static aspects of CaSPiS can be encoded in the algebra. As far as the dynamics of CaSPiS processes is concerned, the work is still ongoing, but some interesting aspects will be discussed. In particular, as the CaSPiS reduction semantics allows for reactions in (static) contexts of arbitrary depth, the standard notion of graph transformation rule, which has a local effect only, is not sufficient to model it. We will sketch some possible approaches to overcome this problem.

Some preliminary work on the graphical encoding of CaSPiS and its behavioral semantics has been presented in [Ter08].

2 An algebra of hierarchical graphs

We introduce here our algebra of (typed) hierarchical graphs that we call *designs*. The algebraic presentation of designs is inspired by our previous work on *Architectural Design Rewriting* [BLMT08] (hence the name) and by the graph algebra of CHARM [CMR94].

Definition 1 (design) A *design* is a term of sort $\mathbb{D}$ generated by the grammar

$$\mathbb{D} \;::=\; L_{\overline{x}}[\mathbb{G}] \qquad\qquad \mathbb{G} \;::=\; \mathbf{0} \;\mid\; x \;\mid\; l\langle\overline{x}\rangle \;\mid\; \mathbb{G}\,|\,\mathbb{G} \;\mid\; (\nu x)\mathbb{G} \;\mid\; \mathbb{D}\langle\overline{x}\rangle$$

where l and L are drawn from disjoint vocabularies $\mathscr{E}$ and $\mathscr{D}$ of *edge* and *design labels*, respectively, x is taken from a global set $\mathscr{N}$ of *nodes*, and $\overline{x} \in \mathscr{N}^*$ is a list of nodes.

As a matter of notation, in the following, we let $\lfloor\overline{x}\rfloor$ denote the set of elements of a list $\overline{x}$ and overload $|\cdot|$ to denote both the length of a list and the cardinality of a set.

Terms generated by $\mathbb{G}$ and $\mathbb{D}$ are meant to represent hierarchical graphs and "edge-encapsulated" hierarchical graphs, respectively. The syntax has the following informal meaning: $\mathbf{0}$ represents the empty graph, x is a discrete graph containing node x only, $l\langle\overline{x}\rangle$ is a graph formed by an l-labeled (hyper)edge attached to nodes $\overline{x}$ (the i-th tentacle to the i-th node in $\overline{x}$, sometimes denoted by $\overline{x}[i]$), $\mathbb{G} \mid \mathbb{H}$ is the graph resulting from the parallel composition of graphs $\mathbb{G}$ and $\mathbb{H}$ (their disjoint union up to shared nodes), $(\nu x)\mathbb{G}$ is the graph $\mathbb{G}$ after making node x not visible from the outside (borrowing nominal calculus jargon we say that the node x is *restricted*), and $\mathbb{D}\langle\overline{x}\rangle$ is a graph formed by attaching design $\mathbb{D}$ to nodes $\overline{x}$ (the i-th node in the interface of $\mathbb{D}$ to the i-th node in $\overline{x}$).

A term $L_{\overline{x}}[\mathbb{G}]$ is a design labeled by L, with body graph $\mathbb{G}$ whose nodes $\overline{x}$ are exposed in the interface. To clarify the exact role of the interface of a design, we can use a programming metaphor: a design $L_{\overline{x}}[\mathbb{G}]$ is like a procedure declaration where $\overline{x}$ is the list of formal parameters. Then term $L_{\overline{x}}[\mathbb{G}]\langle\overline{y}\rangle$ represents the application of the procedure to the list of actual parameters $\overline{y}$; of course, in this case the lengths of $\overline{x}$ and $\overline{y}$ must be equal.

Restriction $(\nu x)\mathbb{G}$ acts as a binder for x in $\mathbb{G}$ and similarly $L_{\overline{x}}[\mathbb{G}]$ binds $\lfloor\overline{x}\rfloor$ in $\mathbb{G}$, leading to the usual notion of *free* nodes $fn(\mathbb{D})$ and $fn(\mathbb{G})$, defined inductively as follows:

$$\begin{array}{lll} fn(L_{\overline{x}}[\mathbb{G}]) = fn(\mathbb{G}) \setminus \lfloor\overline{x}\rfloor & fn(\mathbf{0}) = \emptyset \quad fn(x) = \{x\} & fn(l\langle\overline{x}\rangle) = \lfloor\overline{x}\rfloor \\ fn(\mathbb{G} \mid \mathbb{H}) = fn(\mathbb{G}) \cup fn(\mathbb{H}) & fn((\nu x)\mathbb{G}) = fn(\mathbb{G}) \setminus \{x\} & fn(\mathbb{D}\langle\overline{x}\rangle) = fn(\mathbb{D}) \cup \lfloor\overline{x}\rfloor \end{array}$$

The algebra includes the structural graph axioms of [CMR94] such as associativity and commutativity for $|$ with identity $\mathbf{0}$ (axioms DA1–DA3 in Definition 2) and restricted nodes (DA4–DA6). In addition, it includes axioms to α-rename bound nodes (DA7–DA8), an axiom for making immaterial the addition of a node to a graph where that same node is already free (DA9) and another one ensuring that global names are not localized within hierarchical edges (DA10).

Definition 2 ($\equiv_D$) The structural congruence $\equiv_D$ over well-formed designs and graphs is the least congruence satisfying the axioms in Fig. 1, where in axiom (DA7) the substitution is required to be a function (to avoid node coalescing) and substitutions are required to respect the typing.

$$
\begin{array}{rcll}
\mathbb{G} \mid \mathbb{H} & \equiv & \mathbb{H} \mid \mathbb{G} & & \text{(DA1)}\\
\mathbb{G} \mid (\mathbb{H} \mid \mathbb{I}) & \equiv & (\mathbb{G} \mid \mathbb{H}) \mid \mathbb{I} & & \text{(DA2)}\\
\mathbb{G} \mid \mathbf{0} & \equiv & \mathbb{G} & & \text{(DA3)}\\
(\nu x)(\nu y)\mathbb{G} & \equiv & (\nu y)(\nu x)\mathbb{G} & & \text{(DA4)}\\
(\nu x)\mathbf{0} & \equiv & \mathbf{0} & & \text{(DA5)}\\
\mathbb{G} \mid (\nu x)\mathbb{H} & \equiv & (\nu x)(\mathbb{G} \mid \mathbb{H}) & \text{if } x \notin fn(\mathbb{G}) & \text{(DA6)}\\
L_{\overline{x}}[\mathbb{G}] & \equiv & L_{\overline{y}}[\mathbb{G}\{^{\overline{y}}/_{\overline{x}}\}] & \text{if } \lfloor \overline{y} \rfloor \cap fn(\mathbb{G}) = \emptyset & \text{(DA7)}\\
(\nu x)\mathbb{G} & \equiv & (\nu y)\mathbb{G}\{^{y}/_{x}\} & \text{if } y \notin fn(\mathbb{G}) & \text{(DA8)}\\
x \mid \mathbb{G} & \equiv & \mathbb{G} & \text{if } x \in fn(\mathbb{G}) & \text{(DA9)}\\
L_{\overline{x}}[z \mid \mathbb{G}]\langle \overline{y} \rangle & \equiv & z \mid L_{\overline{x}}[\mathbb{G}]\langle \overline{y} \rangle & \text{if } z \notin \lfloor \overline{x} \rfloor & \text{(DA10)}
\end{array}
$$

Figure 1: Structural congruence axioms for designs

It is immediate to observe that structural congruence respects free nodes, i.e. $\mathbb{G} \equiv_D \mathbb{H}$ implies $fn(\mathbb{G}) = fn(\mathbb{H})$ for any $\mathbb{G}, \mathbb{H}$. Moreover, being $\equiv_D$ a congruence, we remark that $L_{\overline{x}}[\mathbb{G}] \equiv_D L_{\overline{x}}[\mathbb{H}]$ whenever $\mathbb{G} \equiv_D \mathbb{H}$.

Two different classes of models have been studied for our design algebra, as summarized in the next two subsections: these are in straight analogy with two common visual representations of file systems. The icon view, where each folder is a window recursively containing files and folders, represents a global view of the system taken “from the top”. Instead the tree-like view, where the whole hierarchy is presented as a tree whose nodes can be contracted and expanded and where containment is rendered, for example, through indentation, represents some sort of “side-view” of the system.

2.1 Top-view models

In [BGLb] we have proposed an original notion of hierarchical graphs with interfaces: roughly they extend ordinary hyper-graphs with the possibility to embed (recursively) a hierarchical graph within each edge, thus inducing a layered structure of nodes and edges. Notably, differently from the definition proposed in [DHP02], the nodes defined in one layer are also visible below in the hierarchy (but not above). The main result of [BGLb] shows that the encoding of design terms in hierarchical graphs is surjective and that the axiomatization of the design algebra is sound and complete w.r.t. the encoding. Moreover, in the presence of the extrusion axiom, which is introduced later, the encoding can be slightly modified in order to preserve the validity of the main results. The set-theoretical presentation of hierarchical graphs is quite heavy and out of the scope of this paper: we refer the interested reader to [BGLb] for all technical details and formal proofs.

The following example gives a better intuition of the algebra and the model of hierarchical graphs. For this purpose we use an informal, appealing visual notation.

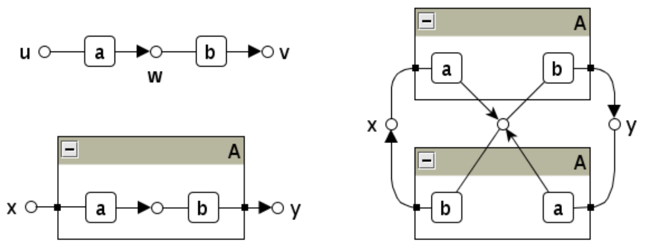


Figure 2: Some terms of the graph algebra

Example 1 *Let* $a,b \in \mathscr{E}$, $A \in \mathscr{D}$, $u,v,w,x,y \in \mathscr{N}$. *We depict in Fig.* 2 *the hierarchical graphs corresponding to a few terms of our algebra. Nodes are represented by circles and free nodes are annotated with their name. Edges are represented by rounded boxes, annotated inside with the edge label. Each design is represented by a square box with their label in a top bar, and encapsulating the body graph. Instead of numbering the tentacles of edges and designs, we use different kinds of lines and arrows: in this example the first tentacle of an edge is represented by a plain line, while the second one is denoted by a standard arrow.*

To avoid the node proliferation, we omit drawing the local interface nodes of a design and fuse them with the corresponding nodes to which the design is attached and mark the overlapping with small black boxes at the border of designs.

Figure 2 *includes the graphs corresponding to the following terms:* $\mathbb{G} = a\langle u,w\rangle \mid b\langle w,v\rangle$ *(left-top),* $A_{u,v}[(\nu w)\mathbb{G}]\langle x,y\rangle$ *(left-bottom), and* $(\nu w)(A_{u,v}[\mathbb{G}]\langle x,y\rangle \mid A_{u,v}[\mathbb{G}]\langle y,x\rangle)$ *(right). Note how the tentacles attached to* x *and* y *do actually cross the interface and are hence denoted by small black boxes in border of designs. This does not happen for tentacles attached to node* w*, because it is shared without being exposed in the interfaces of the design.*

The hierarchical graphs in Fig. 2 illustrate a global view of the system taken from the top. Another possibility is to take a side-view of the system, where containment is traced by dependencies between items in different layers.

2.2 Side-view models

In [BCG⁺] we have followed the tree-like analogy to define a second interpretation over a class of graphs already available in the literature, called *gs-graphs* [FM00]. Roughly, they are an extension of term-graphs [BEG⁺87] tailored to many-sorted hyper-signatures. The crucial fact is that the algebraic structure of gs-graphs has been formalized in terms of the so-called *gs-monoidal theories* [CG99]. The main idea is to define a signature whose sorts correspond to nodes of the type graph and whose operators correspond to the edges of the type graph. One additional sort $\bullet$ is introduced to represent *locations* within the hierarchy. Then each design label $L \in \mathscr{D}$ defines an operator that takes as arguments a location and the list of actual parameters and returns a location and the list of formal parameters (i.e., it provides the inner graph with the location where to reside and with a local environment). Edge labels $l \in \mathscr{E}$ provide no result (their coarity is ε,

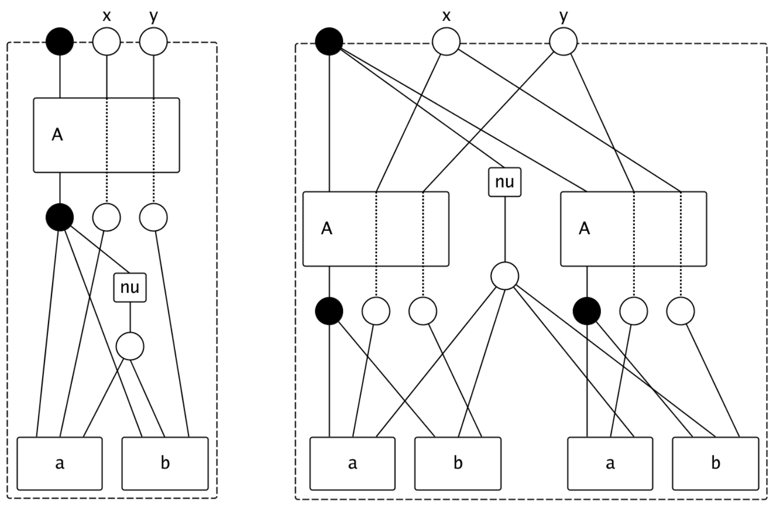


Figure 3: Hierarchical structure as gs-monoidal terms

the empty list of sorts). Then, the results in [BCG$^+$] define a sound and complete encoding of design terms in gs-graphs.[1] Again, we skip all technical details and we just sketch in Fig. 3 the gs-graphs corresponding to the two hierarchical graphs in Fig. 2: $A_{u,v}[(\nu w)\mathbb{G}]\langle x,y\rangle$ on the left, and $(\nu w)(A_{u,v}[\mathbb{G}]\langle x,y\rangle \mid A_{u,v}[\mathbb{G}]\langle y,x\rangle\,)$ on the right (for $\mathbb{G} = a\langle u,w\rangle \mid b\langle w,v\rangle$). The drawing is decorated with an external dashed line enclosing the gs-graph and emphasizing its boundary, on which the names of the available free nodes are placed; furthermore some dotted lines suggests the correspondence between actual and formal parameters of A-labeled edges. Such decorations are not part of the formal definition and have the only purpose of making easier the intuitive correspondence with Fig. 2.

2.3 Well-typedness and extrusion

In practice, it is very frequent that one is interested in disciplining the use of edge and design labels so to be attached only to a specific number of nodes (possibly of specific sorts) or to contain graphs of a specific shape. To this aim it is typically the case that: 1) nodes are sorted, in which case their labels take the form $n : s$ for n the *name* and s the *sort* of the node; 2) each label of $\mathscr{E}$ and $\mathscr{D}$ has a fixed arity and for each rank a fixed node sort; 3) designs can be partitioned according to their top-level labels (i.e. the set of design labels $\mathscr{D}$ can be seen as the set of sorts, with a membership predicate $\mathbb{D} : L$ that holds whenever $\mathbb{D} = L_{\overline{x}}[\mathbb{G}]$ for some $\overline{x}$ and $\mathbb{G}$). When this is the case, we say that a design (or a graph) is *well-typed* if for each sub-term $L_{\overline{x}}[\mathbb{G}]$ we have that the (lists of) sorts of $\overline{x}$ and L coincide, and similarly for sub-terms $\mathbb{D}\langle\overline{x}\rangle$ and $l\langle\overline{x}\rangle$.

In addition to the axioms of Fig. 1, another axiom that has been considered in the literature is the so-called *extrusion* axiom.

[1] Actually the construction in [BCG$^+$] is carried out for a slightly different algebra of design, but it is discussed how to extend the results to the algebra considered here.

Definition 3 (Extrusion axiom) The extrusion axiom extr is $L_{\overline{y}}[(\nu z)\mathbb{G}]\langle\overline{x}\rangle \equiv (\nu z)L_{\overline{y}}[\mathbb{G}]\langle\overline{x}\rangle$, for any $L \in \mathscr{D}$, where $z \notin \lfloor\overline{x}\rfloor \cup \lfloor\overline{y}\rfloor$.

The presence or absence of the extrusion axiom marks the distinction between the so-called global restriction (orthogonal to nesting) and located restriction stressed in [BCG+]. Our encoding of CaSPiS requires the presence of the extrusion axiom: roughly, at the level of gs-graphs, the difference between the two alternatives relies in the absence/presence of the arc connecting ν-labeled boxes to •-nodes (see e.g. Figures 3 and 6).

3 A calculus with nested structures and communication: CaSPiS

This section recalls the basics of CaSPiS [BBDL08], a session-centered calculus. We have chosen this calculus since it represents a non-trivial example of the interplay between nesting and linking introduced by nested sessions, pipelines and communication.

While referring the interested readers to [BBDL08] for an exhaustive description of CaSPiS, we remark that we focus here on the close-free fragment of the calculus and we present a slightly simplified syntax (without summation and pattern-matching). Both decisions are for the sake of a convenient and clean presentation only, and constitute no limitation on expressiveness.

CaSPiS is based on the following key computing entities: (i) service definitions $s.P$ and invocations $\overline{s}.Q$, whose synchronization establishes (ii) a fresh session name r shared by the two partner session sides $r \triangleright P$ and $r \triangleright Q$, where respective interaction protocols can interact in both directions by executing (iii) intra-session (synchronous) output $\langle u\rangle$ and input $(?x)$ prefixes. Moreover, (iv) session sides can be nested, and (v) a children side can execute an (extra-session) return prefix $\langle u\rangle^{\uparrow}$ for making u available to its parent session side as an output for intra-session communication with its corresponding partner side. Finally, (vi) in-side computation can be achieved using the pipeline operator $P > (?x)Q$, which redirects each output $\langle u\rangle$ from P to activate a corresponding new instance $Q\{^u/_x\}$ of Q. Notably, any such instance will run in parallel with $P > (?x)Q$. Summarizing all the above, each CaSPiS process can be thought of as running in an environment providing him different means of communication: one channel for “standard” input (expecting values from the partner session side), one channel for “standard” output (either directed to the partner session side or to an in-side pipeline) and one channel for returning values one level up (according to the nesting of session sides).

Definition 4 (CaSPiS syntax) Let $\mathscr{S}$ a set of service names, $\mathscr{R}$ be a set of session names, $\mathscr{V} \supseteq \mathscr{S}$ a set of value names (disjoint from $\mathscr{R}$), and $\mathscr{X} \subseteq \mathscr{V}$ a set of value variables. The set $\mathscr{P}$ of CaSPiS processes is the set of all the terms P generated by the grammar below

$$\begin{array}{lcl} P & ::= & \mathbf{0} \mid r \triangleright P \mid P > Q \mid (\nu w)P \mid P \,|\, P \mid A.P \\ A & ::= & s \mid \overline{s} \mid (?x) \mid \langle u\rangle \mid \langle u\rangle^{\uparrow} \end{array}$$

where $s \in \mathscr{S}$, $r \in \mathscr{R}$, $u \in \mathscr{V}$, $w \in (\mathscr{V} \cup \mathscr{R}) \setminus \mathscr{X}$ and $x \in \mathscr{X}$.

As usual, the restriction operator $(\nu w)P$ binds w in P, and similarly $(?x).P$ binds x in P, leading to straightforward definition of free names $fn(P)$ of a process P. Albeit the syntax allows a more general form of pipelines, for simplicity we restrict to consider pipelines of the form $P > (?x)Q$, sometimes written as $P > x > Q$, using a notation reminiscent of the Orc programming language [KCM06].

We assume that in any process P at most two session sides are present for the same session name and that the binary relation $\prec_P^+$ over session names is irreflexive, where we write $r \prec_P r'$ whenever in P a session side r' appears nested within a session side r and $\prec_P^+$ denotes the transitive closure of $\prec_P$.

The operational semantics is defined in terms of reduction rules over processes taken up to a suitable structural congruence.

Definition 5 ($\equiv_C$) The structural congruence for CaSPiS processes is the relation $\equiv_C \subseteq \mathscr{P} \times \mathscr{P}$, closed under process construction, inductively generated by the axioms in Fig. 4.

$$
\begin{array}{rcll l}
P \mid (Q \mid R) & \equiv & (P \mid Q) \mid R & & \text{(CA1)} \\
P \mid Q & \equiv & Q \mid P & & \text{(CA2)} \\
P \mid \mathbf{0} & \equiv & P & & \text{(CA3)} \\
(\nu n)(\nu m)P & \equiv & (\nu m)(\nu n)P & & \text{(CA4)} \\
(\nu n)\mathbf{0} & \equiv & \mathbf{0} & & \text{(CA5)} \\
P \mid (\nu n)Q & \equiv & (\nu n)(P \mid Q) & \text{if } n \notin fn(P) & \text{(CA6)} \\
((\nu n)Q) > P & \equiv & (\nu n)(Q > P) & \text{if } n \notin fn(P) & \text{(CA7)} \\
r \rhd (\nu n)P & \equiv & (\nu n) r \rhd P & \text{if } n \neq r & \text{(CA8)} \\
(\nu n)P & \equiv & (\nu m)P\{^m/_n\} & \text{if } m \notin fn(P) & \text{(CA9)} \\
(?x).P & \equiv & (?y).P\{^y/_x\} & \text{if } y \notin fn(P) & \text{(CA10)}
\end{array}
$$

Figure 4: Structural congruence axioms for CaSPiS.

Reduction rules make use of contexts; a context $C[\cdot]$ is simply a process term in which there is a single occurrence of a process variable X, called the hole of the context. With $C[P]$ we denote the process obtained filling the hole of the context with the process P (i.e. we substitute X with P). We can easily generalize such definition to n-holes: instead of a single process variable X, we will have n process variables $X_1, \ldots, X_n$.

Definition 6 (Static and dynamic operators) The operators $s.[\cdot]$, $\overline{s}.[\cdot]$, $\langle u \rangle.[\cdot]$, $(?x).[\cdot]$, $\langle u \rangle^{\uparrow}.[\cdot]$ and $P > [\cdot]$ are *dynamic*. The remaining operators ($r \rhd [\cdot]$, $[\cdot] > P$, $(\nu n)[\cdot]$ and $P|[\cdot]$) are *static*.

Intuitively the dynamic operators, like the prefixes in the π-calculus or in CCS, do not allow a transition to take place in their argument. Therefore we can define the contexts in which the various kinds of action prefixes are ready to be executed.

Definition 7 (Static and "immune" contexts) A context $C[\cdot]$ is *static* if its hole does not occur in the scope of a dynamic operator. A static context is *session-immune* if the hole does not appear in the scope of a session operator $r \rhd [\cdot]$. A static context is *pipeline-immune* if the hole does not appear in the scope of a pipeline operator $[\cdot] > P$.

Session-immune contexts are guaranteed not to interfere with inputs and returns of the process in their hole, while contexts that are both session- and pipeline-immune are also guaranteed not to interfere with outputs. Note that in the latter case, the hole can only appear under restriction and parallel composition. We are ready now to present the reduction semantics of CaSPiS.

Definition 8 (Reduction rules of CaSPiS) Given two CaSPiS processes P and Q we have $P \Rightarrow Q$ if and only if one of the five cases in Fig. 5 holds, for some static contexts $C[\cdot]$, $C[\cdot,\cdot]$, some static session-immune contexts $S_0[\cdot]$ and $S_1[\cdot]$, some processes P', P'', R and some names r, r', u and x.

$$\begin{array}{llcll}
(1) & P & \equiv & C[\, s.P' \,,\, \overline{s}.R \,] & (\mathit{ServiceSync}) \\
 & Q & \equiv & (\nu r)C[\, s.P' | r \triangleright P' \,,\, r \triangleright R \,] \text{ with } r \text{ fresh for } P', C[\cdot], R & \\[1ex]
(2) & P & \equiv & C[\, r \triangleright (P' | \langle u \rangle.P'') \,,\, r \triangleright S_0[(?x).R] \,] & (\mathit{SessionSync}) \\
 & Q & \equiv & C[\, r \triangleright (P' | P'') \,,\, r \triangleright S_0[R\{^u/_x\}] \,] & \\[1ex]
(3) & P & \equiv & C[\, r' \triangleright (P' | r \triangleright S_0[\langle u \rangle^{\uparrow}.P'']) \,,\, r' \triangleright S_1[(?x).R] \,] & (\mathit{SessionSyncRet}) \\
 & Q & \equiv & C[\, r' \triangleright (P' | r \triangleright S_0[P'']) \,,\, r' \triangleright S_1[R\{^u/_x\}] \,] & \\[1ex]
(4) & P & \equiv & C[\, (P' | \langle u \rangle.P'') > (?x).R \,] & (\mathit{PipelineSync}) \\
 & Q & \equiv & C[\, R\{^u/_x\} \mid ((P' | P'') > (?x).R) \,] & \\[1ex]
(5) & P & \equiv & C[\, (P' | r \triangleright S_0[\langle u \rangle^{\uparrow}.P'']) > (?x).R \,] & (\mathit{PipelineSyncRet}) \\
 & Q & \equiv & C[\, R\{^u/_x\} \mid ((P' | r \triangleright S_0[P'']) > (?x).R) \,] &
\end{array}$$

Figure 5: Possible cases for $P \Rightarrow Q$

The first rule models the invocation of a service: there is a definition of service s ($s.P'$) and a request of invocation of such service ($\overline{s}.R$) located somewhere else in the system. Then a new session r is created with the protocols P' and R of the server and of the client respectively. Note that differently from [BBDL08], here services are persistent: they are not discarded once invoked and thus they can serve other requests.

Rule ($\mathit{SessionSync}$) allows session partners to exchange messages, through a concretion and an abstraction. Technically, the concretion $\langle u \rangle$ can appear in an arbitrary session- and pipeline-immune context within the session operator, but since restrictions can be moved outside the session operator by structural congruence, this is equivalent to require that the concretion is in parallel with an arbitrary process P', as indicated in the rule. Instead the abstraction $(?x)$ can be at an arbitrary depth in the syntax tree, for example in the left-side of a pipeline, but not in a nested session operator: for this reason we use a static session-immune context $S_0[\cdot]$. The next rule ($\mathit{SessionSyncRet}$) can be used for returning a value computed by a nested session side to the session partner. One can view this rule as composed of two steps: first the value computed in session r is passed to the enclosing session side r', then such session side sends the value to its partner.

The pipeline rule $(PipelineSync)$ shows that a value computed by the left-hand side $P'|\langle u\rangle.P''$ can trigger an instance of the right-hand side $(?x)R$. Finally the rule $(PipelineSyncRet)$ describes the situation where a pipe can be activated through a value returned by a nested session side of the process on the left side of the pipeline.

4 Encoding CaSPiS into the algebra of designs

In [BGL09, BGLb] we have provided a sound and complete encoding of CaSPiS processes to our algebra of designs, exploiting the fact that reduction rules can then be directly interpreted over and applied to graphs instead of terms. Unfortunately, this way an interleaving semantics is obtained, not a truly concurrent one, because the whole graph is rewritten at each step (no standard notion of "preserved" nodes/edges is available).

Here we pursue a different objective, by establishing an encoding for which ordinary graph rewriting techniques can be used to recover the dynamics. In particular, as rewrites are forbidden under dynamic contexts of CaSPiS, we will expand dynamic operators only by need. This means that for each term P having a dynamic top operator, we introduce a corresponding edge label, sorted according to the free names of P, which are needed as parameters in the rewrite rule that will expand P to the corresponding graph after a reaction. Instead, the static contexts will be encoded in nested designs corresponding to the session and left-pipeline operators, while restriction operators will be encoded directly as restrictions of the algebra of designs.

In the following we assume that a standard total order on names is available, and for a set of names X we denote by $\lceil X\rceil$ the list of names in X ordered accordingly. Moreover, we assume the existence of a canonical set of totally ordered fresh names $\mathscr{C}$, together with a canonical (order preserving) renaming $\sigma_X : X \to \mathscr{C}$ for any $X \subseteq \mathscr{V} \cup \mathscr{R}$ such that whenever $|X| = |Y|$ then $\sigma_X(X) = \sigma_Y(Y)$. We denote by $can(P)$ the term $P\sigma_{fn(P)}$ obtained by renaming the free names of P according to $\sigma_{fn(P)}$.

Names (of services, sessions, etc.) are encoded as nodes of the algebra, thus we assume that the set of nodes is sorted accordingly, even if we do not make this formal. The set of edge labels is $\{\, \underline{A.P}\,\}$, i.e. it includes one standard representative label $\underline{A.P}$ for (the equivalence class up to structural congruence of) each CaSPiS process of the form $can(A.P)$. The tentacles of $\underline{A.P}$ are sorted according to $fn(A.P)$. The set of design labels includes $SES\_$ for session sides (exposing an anonymous session name $\_$), and one standard representative $\underline{x > Q}$ for each static context of the form $[\cdot] > (?x)Q$, exposing $n = |fn(Q) \setminus \{x\}|$ canonical fresh variables $\sigma_{fn(Q)\setminus\{x\}}(fn(Q) \setminus \{x\})$.

As a matter of notation, for a term $L_{\overline{y}}[\mathbb{G}]\langle\overline{x}\rangle$ we use the shorthand $L[\mathbb{G}]\langle\overline{x}\rangle$ if $\lfloor\overline{y}\rfloor \cap fn(\mathbb{G}) = \emptyset$.

Definition 9 (CaSPiS encoding) The interpretation of CaSPiS operators over the design algebra (with extrusion, i.e., with global restriction) is given by

$$\begin{array}{rcl}
[\![\mathbf{0}]\!] & \stackrel{\text{def}}{=} & \mathbf{0} \\
[\![A.P]\!] & \stackrel{\text{def}}{=} & \underline{A.P}\langle\lceil fn(A.P)\rceil\rangle \\
[\![r \rhd P]\!] & \stackrel{\text{def}}{=} & SES[\,[\![P]\!]\,]\langle r\rangle \\
[\![P > (?x)Q]\!] & \stackrel{\text{def}}{=} & \underline{x > Q}[\,[\![P]\!]\,]\langle\lceil fn(Q) \setminus \{x\}\rceil\rangle \\
[\![P \mid Q]\!] & \stackrel{\text{def}}{=} & [\![P]\!] \mid [\![Q]\!] \\
[\![(\nu w)P]\!] & \stackrel{\text{def}}{=} & (\nu w)[\![P]\!]
\end{array}$$

It is worth stressing that if one is interested in analyzing, through the encoding to the algebra of designs and the transformation of the corresponding graphs, a finite set of CaSPiS processes, then the resulting algebra will have a finite number of edge and design labels, determined by the set of sub-processes of those of interest. Instead, to be able to accommodate the encoding of all possible CaSPiS processes, denumerable sets of labels are needed.

Notably, structural congruence amounts to design equivalence, i.e. equivalent processes are mapped into isomorphic graphs.

Proposition 1 *For any* $Q,R \in \mathscr{P}$ *we have* $P \equiv_{\mathrm{C}} Q$ *iff* $[\![P]\!] \equiv_{\mathrm{D}} [\![Q]\!]$.

4.1 Transformation rules for CaSPiS reduction semantics

Given the encoding of CaSPiS processes as terms of the algebra of designs, and any suitable model of the algebra in terms of a class of graphs (like those presented in Sections 2.1 and 2.2), it is natural to try to lift the reduction semantics of CaSPiS, through these encodings, to a corresponding notion of transformation over the resulting graphs. Ideally, we would like to translate the reduction rules of Definition 8 to ordinary graph transformation rules, in order to exploit the rich theory of graph transformation and the corresponding analysis and verification tools, also accounting for concurrency aspects.

However, this is not possible in a direct way. In fact, the reduction rules of CaSPiS include suitable contexts in the left- and right-hand sides, which can be instantiated in arbitrary ways to match a subterm of the process to be reduced. In other words, each reduction rule can be considered as a rule schema, summarizing the common shape of infinitely many similar rules, obtained by consistently replacing the contexts with suitable terms. Quite obviously, if we are interested in reducing a single process (or a finite set of processes), we need to consider only a finite set of instances of the rules.

In Figures 6, 7 and 8 we depicted the graph transformation rule schemata corresponding to the reduction rules $(\mathit{ServiceSync})$, $(\mathit{SessionSync})$ and $(\mathit{PipelineSync})$, using the side-view of designs discussed in Section 2.2. Comparing them with the corresponding reduction rules in Fig. 5, we can note that: 1) we can now omit to specify the static context C under which the interacting redexes are found, because the left-hand side of a graph transformation rule can always be applied in larger graphs; 2) for the same reason, we can safely omit idle items that run in parallel, like process P' from rules $(\mathit{SessionSync})$ and $(\mathit{PipelineSync})$ in Fig. 5; 3) but we must still account for the presence of any admissible static session-immune context S_0, because it constraints the applicability of the rule (in general $[\![S_0]\!]$ can be a chain of pipeline-labeled boxes of arbitrary length, possibly 0). Even if we did not work out the corresponding definitions, we identified two graph transformation frameworks which can provide the means to turn each such rule schemata into a collection of graph rules, whose overall effect would be the expected one when applied to a graph representing a CaSPiS process.

Synchronized Hyperedge Replacement. In the SHR approach [FHL+06], the parallel application of a set of rules to a graph is controlled by a synchronization mechanism which requires a consistency check among the redex boundaries of the involved rules. This mechanism can be used to build (standard) rules with unbound left-hand sides, starting from a finite set of rules.

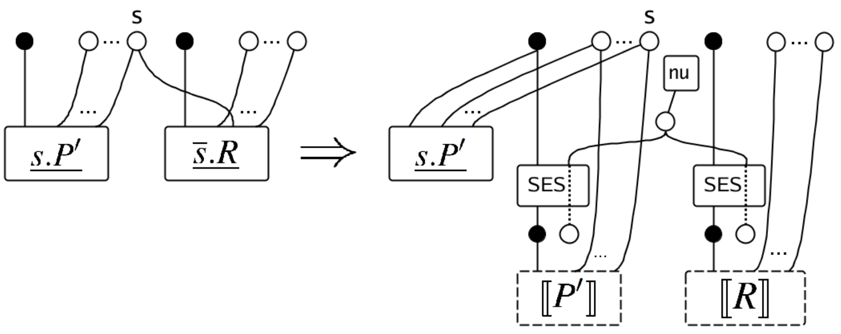


Figure 6: Rule $(ServiceSync)$

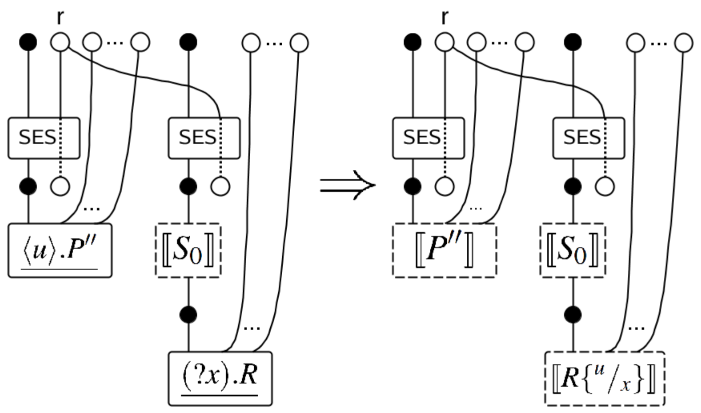


Figure 7: rule $(SessionSync)$

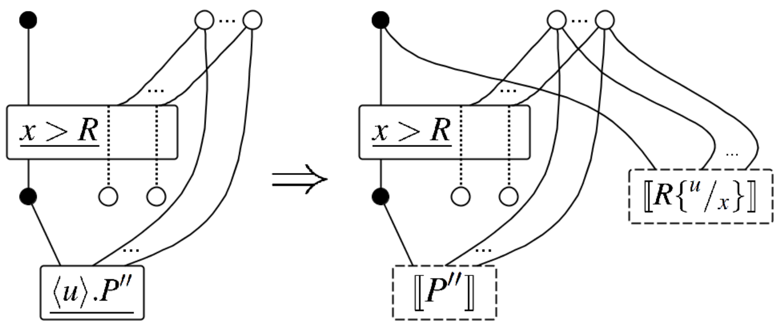


Figure 8: Rule $(PipelineSync)$

Therefore a CaSPiS rule schema could be implemented by a set of SHR rules, which should be able to induce the set of all its instantiations.

Graph Transactions. The notion of graph transaction proposed in [BCD+08] is based on a notion of "unstable" graph items. A transaction is a minimal derivation starting and ending in graphs not containing unstable items, up to shift equivalence, and the operational semantics of a transactional graph transformation system includes only derivations that are made of transactions. Therefore a CaSPiS rule schema could be translated into a collection of rules which simulate the navigation of the process in order to identify an occurrence of the left hand side. This can be done by generating unstable items in the graph: their presence conceptually inhibits the application of other rules in parallel. When the left pattern is recognized and the effect of the rule is applied, such unstable elements are deleted, resulting in the commitment of the transaction.

The study of these possible translations of CaSPiS rule schemata is an interesting topic for future research.

5 Conclusions

In this paper we have shown the main issues regarding the graphical encoding of a sophisticated process calculus with inherently hierarchical features. The encoding of processes can be written quite smoothly by exploiting a recently proposed algebra of graphs with nesting (see Definition 9), and it can be shown to preserve and respect the structural congruence of processes. On the other hand, the encoding of reduction rules as ordinary graph transformation rules requires some ingenuity, because the redexes can require the traversal/inspection of an unbound number of nesting levels due to the presence of static session-immune contexts in the rules of Fig. 5.

The main methodological innovation of the paper, with respect to other proposals of encoding process algebras into graph transformation systems, resides is the identification of an intermediate algebra of designs, which bridges the gap between the syntax of the process calculus and the set theoretical definition of the graphs. A direct translation of CaSPiS processes to, for example, gs-graphs, would be possible but more cumbersome. Furthermore, a sound and complete interpretation of the algebra into a class of graphs can be reused for different process calculi. For example, besides the top- and side-view graphs discussed in the paper, another natural graph model for the algebra are Milner's bigraphs [Mil06], which are naturally endowed with a notion of embedding and of linking.

The ultimate motivation in equipping CaSPiS with a graph transformation operational semantics is to exploit the rich theory of graph transformation and corresponding tools for the analysis and verification of relevant properties of CaSPiS processes. The intermediate design algebra provides one additional framework for such analysis, which could be performed by exploiting tools directly based on the algebra, which are currently under development (see http://www.albertolluch.com/adr2graphs/).

Acknowledgements: We want to thank Fabio Gadducci, Alberto Lluch Lafuente, Daniele Terreni and Liang Zhao for many interesting discussions and exchange of ideas regarding the graphical encoding of CaSPiS.

Bibliography

[AG98] A. Asperti, S. Guerrini. *The Optimal Implementation of Functional Programming Languages*. Cambridge University Press, 1998.

[BBDL08] M. Boreale, R. Bruni, R. De Nicola, M. Loreti. Sessions and Pipelines for Structured Service Programming. In Barthe and de Boer (eds.), *FMOODS 2008*. LNCS 5051, pp. 19–38. Springer, 2008.

[BCD$^+$08] P. Baldan, A. Corradini, F. Dotti, L. Foss, F. Gadducci, L. Ribeiro. Towards a Notion of Transaction in Graph Rewriting. In Bruni and Varró (eds.), *International Workshop on Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. ENTCS 211. Elsevier, 2008.

[BCG$^+$] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, U. Montanari. On GS-Monoidal Theories for Graphs with Nesting. Submitted.

[BEG$^+$87] H. Barendregt, M. van Eekelen, J. Glauert, J. Kennaway, M. Plasmeijer, M. Sleep. Term graph reduction. In *PARLE'87*. LNCS 259, pp. 141–158. Springer, 1987.

[BGLa] R. Bruni, F. Gadducci, A. Lluch Lafuente. An Algebra of Hierarchical Graphs. Submitted.

[BGLb] R. Bruni, F. Gadducci, A. Lluch Lafuente. An Algebra of Hierarchical Graphs and its Application to Structural Encoding. Submitted.

[BGL09] R. Bruni, F. Gadducci, A. Lluch Lafuente. A Graph Syntax for Processes and Services. In Jianwen and Laneve (eds.), *WS-FM 2009*. LNCS. Springer, 2009. To Appear.

[BL05] R. Bruni, I. Lanese. On Graph(ic) Encodings. In Koenig et al. (eds.), *Proceedings of Dagstuhl Seminar n. 04241, Graph Transformations and Process Algebras for Modeling Distributed and Mobile Systems*. Pp. 23–29. 2005.

[BLMT08] R. Bruni, A. Lluch Lafuente, U. Montanari, E. Tuosto. Style Based Architectural Reconfigurations. *Bulletin of the European Association for Theoretical Computer Science (EATCS)* 94:161–180, February 2008.

[BMM06] R. Bruni, H. Melgratti, U. Montanari. Event Structure Semantics for Nominal Calculi. In Baier and Hermanns (eds.), *CONCUR 2006*. LNCS 4137, pp. 295–309. Springer, 2006.

[CG99] A. Corradini, F. Gadducci. An Algebraic Presentation of Term Graphs, via GS-Monoidal Categories. *Applied Categorical Structures* 7:299–331, 1999.

[CMR94] A. Corradini, U. Montanari, F. Rossi. An Abstract Machine for Concurrent Modular Systems: CHARM. *Theoretical Computer Science* 122(1-2):165–200, 1994.

[DHP02] F. Drewes, B. Hoffmann, D. Plump. Hierarchical Graph Transformation. *Journal on Computer and System Sciences* 64(2):249–283, 2002.

[FHL$^+$06] G. L. Ferrari, D. Hirsch, I. Lanese, U. Montanari, E. Tuosto. Synchronised Hyperedge Replacement as a Model for Service Oriented Computing. In Boer et al. (eds.), *FMCO 2005*. LNCS 4111, pp. 22–43. Springer, 2006.

[FM00] G. L. Ferrari, U. Montanari. Tile Formats for Located and Mobile Systems. *Information and Computation* 156(1-2):173–235, 2000.

[Gad03] F. Gadducci. Term Graph Rewriting for the pi-Calculus. In Ohori (ed.), *APLAS 2003*. LNCS 2895, pp. 37–54. Springer, 2003.

[KCM06] D. Kitchin, W. R. Cook, J. Misra. A Language for Task Orchestration and Its Semantic Properties. In Baier and Hermanns (eds.), *CONCUR 2006*. LNCS 4137, pp. 477–491. Springer, 2006.

[Mil06] R. Milner. Pure bigraphs: Structure and dynamics. *Information and Computation* 204(1):60–122, 2006.

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

[Ter08] D. Terreni. Computational models based on hierarchical graphs: bigraphs and cogs-graphs. Master thesis, Dipartimento di Informatica, Università di Pisa. 2008.

Symbolic Attributed Graphs for Attributed Graph Transformation

Fernando Orejas[*]

orejas@lsi.upc.edu
Dpt. de Llenguatges i Sistemes Informátics
Universitat Politècnica de Catalunya, Barcelona, Spain.

Abstract: In this paper we present a new approach to deal with attributed graphs and attributed graph transformation. This approach is based on working with what we call symbolic graphs, which are graphs labelled with variables together with a formula that constrains the possible values that we may assigned to these ariables. In particular, in this paper we will compare in detail this new approach with the standard approach to attributed graph transformation.

Keywords: Graph Transformation, Attributed Graphs, Symbolic Graphs

1 Introduction

The study of graph grammars and graph transformation started 40 years ago. However, the first formal approach to deal with attributed graphs is much more recent [12], even if this kind of graphs are needed in many applications of the field. Actually, the development of the fundamental theory of graph transformation for the case of attributed graphs is quite recent [7]. The reason for this late development is probably that, even if the attributed case may seem to be a straightforward generalization of the standard case, it presents some difficulties which have hampered the development of this fundamental theory. One of these difficulties lies on the complication of putting together two theoretical frameworks, algebraic specification and graph transformation, even if both are algebraic and categorical frameworks. In fact, to avoid, to some extent, this problem, in [12] graphs are coded as algebras with the aim of having a uniform setting. The problem is that, in general, algebra transformation does not enjoy the right properties to ensure that the basic theory of graph transformation will hold. The approach taken in [12], based in the approach presented in [10] is quite different. In this case, an attributed graph is seen as a pair formed by an algebra, to define the values of the attributes of the graph, and a graph that includes all the values of the algebra as (a special kind of) nodes, In this way, the algebra part and the graph part of an attributed graph are kept separated up to a certain point. However, this approach still has some difficulties caused by the fact that, even if the graphs of interest are defined over the same data algebra, we have to consider categories including graphs over different algebras. The reason is that, must often, the algebras in the graphs occurring in the transformation rules are different from the algebras in the graphs to which we apply these rules. In particular, an aspect which is not completely obvious in this approach is how one should define the algebra associated to a graph transformation rule. In this sense, in this paper we introduce a specific way to do this by taking the initial algebra associated to a given specification.

[*] This work has been partially supported by the CICYT project (ref. TIN2007-66523) and by the AGAUR grant to the research group ALBCOM (ref. 00516).

However, the main aim of this paper is to present a new approach to deal with attributed graph transformation, which we believe is conceptually simpler than previous approaches, and moreover it is more powerful, as we show. The approach is partially inspired on how the clausal part and the data part are conceptually separated in Constraint Logic Programming [11, 14]. In particular, attributed graphs are presented as *symbolic graphs* consisting of a graph that includes as nodes some variables which represent the values of the attributes, together with a set of formulas that constrain the possible values of these variables. This means that the underlying algebra of values remains only implicit to define the satisfaction of these formulas. The idea underlying this approach was first introduced in [16, 17] to study graph constraints over attributed graphs and, then, used again with a similar aim to specify model transformations by means of patterns [8].

Symbolic graphs can be seen as specifications of attributed graphs. Actually, to compare the standard approach to attributed graph transformation, we define a semantics of symbolic graphs in terms of classes of attributed graphs and we show how attributed graphs can be identified with some specific kind of symbolic graphs, which we call grounded symbolic graphs. Then, to compare the expressive power of the two approaches with respect to attributed graph transformation, we first show that symbolic graphs, as it happens with attributed graphs [10], form an adhesive HLR category [13, 4] to ensure that symbolic graphs inherit the fundamental theory of graph transformation. A variant of this proof is already included in [17]. Finally, we show that attributed graph transformation systems can be coded into symbolic graph transformation systems but that the converse is not true in general.

The paper is organized as follows. In section 2 we provide a reminder of some notions that are used in the rest of the paper. In particular, first, we briefly enumerate some notions from algebraic specification; then, we present E-graphs which are used as the graph part for both attributed graphs and symbolic graphs; finally, we define the category of attributed graphs as presented in [4]. In Section 3 we present the category of symbolic graphs, showing that it is adhesive HLR. Section 4 is dedicated to relate the categories of attributed and symbolic graphs and Section 5 to compare the the expressive power of both approaches with respect to attributed graph transformation. In Section 6, we draw some conclusions. Finally, in an appendix some technical details and proofs are provided.

2 Preliminaries

We assume that the reader has a basic knowledge on algebraic specification and on graph transformation. For instance, we advise to look at [6] for more detail on algebraic specification or at [18, 4] for more detail on graph transformation.

2.1 Basic algebraic concepts and notation

As usual, a signature $\Sigma = (S, \Omega)$ consists of a set of sorts S, and a family of operation symbols of the form $op : s_1 \times \cdots \times s_n \to s$, denoted by Ω, where $n \geq 0$ and $s_1, \ldots, s_n, s \in S$. However, in this paper, signatures include also predicates. We can deal with this extended case in two ways. The first one is to consider that Σ consists, in addition, of a family of predicate symbols. The second one, which we will use, because it is simpler, is based in considering that there is a spacial sort in

S, which we could call *logical*, and that predicate symbols are just operation symbols with profile $s_1 \times \cdots \times s_n \rightarrow logical$. In this case, logical connectives can be treated as operation symbols over the logical sort. In addition, the truth values **t** and **f** may be seen as constants in the signature.

A Σ-algebra A consists of an S-indexed family of sets $\{A_s\}_{s \in S}$ and a function $op_A : A_{s_1} \times \cdots \times A_{s_n} \rightarrow A_s$ for each operation $op : s_1 \times \cdots \times s_n \rightarrow s$ in the signature. A Σ-homomorphism $h : A \rightarrow A'$ consists of an S-indexed family of functions $\{h_s : A_s \rightarrow A'_s\}_{s \in S}$ commuting with the operations. Σ-algebras and Σ-homomorphisms form the category $\mathbf{Alg}_{\blacksquare}$.

A congruence $\equiv$ on an algebra A is an S-indexed family of equivalence relations $\{\equiv_s\}_{s \in S}$ which are compatible with the operations. In this case, $A/\equiv$ denotes quotient algebra whose elements are equivalence classes of values in A. Between A and $A/\equiv$ there is a canonical homomorphism mapping every element in A into its equivalent class.

Given signatures Σ, Σ', with $\Sigma' \subseteq \Sigma$, every Σ-algebra can be seen as a Σ'-algebra, by *forgetting* all the sorts an operations which are not in Σ'. In particular this is called the Σ'-reduct of a Σ-algebra A and is denoted by $A|_{\Sigma'}$.

Given a signature Σ, we denote by T_Σ the term algebra, consisting of all the possible Σ-(ground) terms. T_Σ is initial in $\mathbf{Alg}_{\blacksquare}$, and the unique homomorphism $h_A : T\Sigma \rightarrow A$ yields the value of each term in A. Similarly, $T_\Sigma(X)$ denotes the algebra of all Σ-terms with variables in X, and given a variable assignment $\sigma : X \rightarrow A$, this assignment extends to a unique homomorphism $\sigma^\# : X \rightarrow A$ yielding the value of each term after the replacement of each variable x by its value $\sigma(x)$. In particular, when an assignment is defined over the term algebra, i.e. $\sigma : X \rightarrow T_\Sigma$, then $\sigma^\#(t)$ denotes the term obtained by substituting each variable x in t by the term $\sigma(x)$.

A Σ-algebra A is finitely generated if every element in A is the value of some ground term. It is not difficult to see that if A is finitely generated there is at most on homomorphism between A and any other algebra A'.

A specification $SP = (\Sigma, Ax)$ consists of a signature Σ and a set of axioms Ax, which may be seen as terms of logical sort. Equational specifications are a special case, where the only predicate symbol is the equality. Similarly, conditional equations may be considered as a special kind of terms. Given SP, $\mathbf{Alg_{SP}}$ denotes the full subcategory of $\mathbf{Alg}_{\blacksquare}$, consisting of all Σ-algebras A satisfying the axioms in the specification, i.e. $A \models Ax$. In the case where SP consists of equations or conditional equations there is an initial algebra in $\mathbf{Alg_{SP}}$, denoted by T_{SP}.

2.2 E-graphs

E-graphs are introduced in [4] as a first step to define attributed graphs. Intuitively, an E-graph is a kind of labelled graph, where both nodes and edges may be decorated with labels from a given set E. The difference with labelled graphs, as commonly understood, is that in labelled graphs it is usually assumed that each node or edge is labelled with a given number of labels, which is fixed a priori. In the case of E-graphs, each node or edge may have any arbitrary (finite) number of labels, which is not fixed a priori. Actually, in the context of graph transformation, the application of a rule may change the number of labels of a node or of an edge.

Formally, in E-graphs labels are considered as a special class of nodes and the labeling relation between a node or an edge and a given label is represented by a special kind of edge. Notice that, for instance, this means that the labeling of an edge is represented by an edge whose source is an edge and whose target is a node (a label).

Definition 1 (E-Graphs and morphisms) An *E-graph* over the set of labels L is a tuple $G = (V, L, E_G, E_{NL}, E_{EL}, \{s_j, t_j\}_{j \in \{G,NL,EL\}})$ consisting of:

- V and L, which are the sets of *graph nodes* and of *label nodes*, respectively.
- E_G, E_{NL}, and E_{EL}, which are the sets of *graph edges*, *node label edges*, and *edge label edges*, respectively.

and the source and target functions:

- $s_G : E_G \to V_G$ and $t_G : E_G \to V_G$
- $s_{NL} : E_{NL} \to V$ and $t_{NL} : E_{NL} \to L$
- $s_{EL} : E_{EL} \to E_G$ and $t_{EL} : E_{EL} \to L$

Given the E-graphs G and G', an *E-graph morphism* $f : G \to G'$ is a tuple, $\langle f_{V_G} : V_G \to V'_G, f_L : L \to L', f_{E_G} : E_G \to E'_G, f_{E_{NL}} : E_{NL} \to E'_{NL}, f_{E_{EL}} : E_{EL} \to E'_{EL} \rangle$ such that f commutes with all the source and target functions.

E-graphs and E-graph morphisms form the category $\mathbf{E-Graphs}$.

The following construction, which tells us how we can replace the labels of an E-graph, is used in the sections below.

Definition 2 (Label substitution) Given an E-graph $G = (V, L, E_G, E_{NL}, E_{EL}, \{s_j, t_j\}_{j \in \{G,NL,EL\}})$, a set of labels L', and a function $h : L \to L'$ we define the graph resulting from the substitution of L along h, denoted $h(G)$, the E-graph $h(G) = (V', L', E'_G, E'_{NL}, E'_{EL}, \{s'_j, t'_j\}_{j \in \{G,NL,EL\}})$ defined:

- $V = V', E = E', E_G = E'_G, E_{NL} = E'_{NL}, E_{EL} = E'_{EL}, \{s_j = s'_j\}_{j \in \{G,NL,EL\}}$, and $t_G = t'_G$
- For every $e \in E_{NL} : t'_{NL}(e) = h(t_{NL}(e))$
- For every $e \in E_{EL} : t'_{EL}(e) = h(t_{EL}(e))$

Moreover h induces the definition of the morphism $h^* : G \to G'$, with $h^* = \langle id_V, h, id_{E_G}, id_{E_{NL}}, id_{E_{EL}} \rangle$.

It is routine to see that G' is indeed an E-graph and h^* is an E-graph morphism. In addition, it should be obvious that if h is a bijection then h^* is an isomorphism.

2.3 Attributed Graphs

Following [4], an attributed graph is an E-graph whose labels are the values of a given data algebra that is assumed to be included in the graph.

Definition 3 (Attributed graphs and morphisms) Given a signature Σ an *attributed graph* over Σ is a pair $\langle G, D \rangle$, where D is a given Σ-algebra, called the data algebra of the graph, and G is an E-graph such that L_G the set of labels of G consists of all the values in D, i.e. $L_G = \biguplus_{s \in S} D_s$, where s is the set of sorts of the data algebra and $\uplus$ denotes disjoint union.

Given the attributed graphs over Σ $AG = \langle G, D\rangle$ and $AG' = \langle G', D'\rangle$ and G', an *attributed graph morphism* $h : AG \rightarrow AG'$ is a pair $\langle h_{graph}, h_{alg}\rangle$, where h_{graph} is an E-graph morphism, $h_{graph} : G \rightarrow G'$ and h_{alg} is a over Σ-homomorphism, $h_{alg} : D \rightarrow D'$ such that the values in D are mapped consistently by h_{graph} and h_{alg}, i.e. for each sort $s \in S$ the diagram below commutes:

$$\begin{tikzcd}
D_S \arrow[r, "h_{alg}"] \arrow[d, hook] & D'_{h_{alg(s)}} \arrow[d, hook] \\
L_G \arrow[r, "h_{graph}"'] & L'_G
\end{tikzcd}$$

Attributed graphs and attributed graph morphisms form the category **AttGraphs**. Moreover, given a data algebra D we will denote by $\mathbf{AttGraphs}_D$ the full subcategory of **AttGraphs** consisting of attributed graphs over D.

In [4] it has been proved that **AttGraphs** is an adhesive HLR category. Let us first recall this notion [4, 13]:

Definition 4 (Adhesive HLR category) A category **C** is adhesive HLR with respect to a class M of morphisms if:

1. M is a class of monomorphisms closed under isomorphism, composition (i.e. if $f : A \rightarrow B \in M$ and $g : B \rightarrow A \in M$ then $g \circ f \in M$), and decomposition (i.e. if $g \circ f \in M$ and $g \in M$ then $f \in M$).

2. **C** has pushouts and pullbacks along M-morphisms. Moreover, M-morphisms are closed by pushouts and pullbacks.

3. Pushouts in **C** along M-morphisms are van Kampen squares, i.e. for any commutative diagram as the one below, assuming that h_1 and g_2 are M-morphisms, if the bottom diagram is a pushout and the back faces are pullbacks then the top diagram is a pushout if and only if the front diagrams are pullbacks.

The key idea to show that **AttGraphs** is adhesive HLR is the choice of the right kind of M-morphisms. Actually, **AttGraphs** is not adhesive because it fails to satisfy the van Kampen property for arbitrary monomorphisms.

Theorem 1 **AttGraphs** *is adhesive HLR, with respect to the class of M-morphisms consisting of all monomorphisms* $\langle h_{graph}, h_{alg}\rangle$ *such that* h_{alg} *is an isomorphism.*

$$\begin{tikzcd}
 & & A'_0 \arrow[dll, "h'_1"'] \arrow[dr, "h'_2"] \arrow[dd, "f_0"] & \\
A'_1 \arrow[dr, "g'_1"] \arrow[dd, "f_1"'] & & & A'_2 \arrow[dll, "g'_2"'] \arrow[dd, "f_2"] \\
 & A'_3 \arrow[dd, "f_3"] & A_0 \arrow[dll, "h_1"'] \arrow[dr, "h_2"] & \\
A_1 \arrow[dr, "g_1"'] & & & A_2 \arrow[dll, "g_2"] \\
 & A_3 & &
\end{tikzcd}$$

3 The category of symbolic graphs

A symbolic graphs can be seen as the specification of an attributed graph (or of a class of attributed graphs). In particular, a symbolic graph consists of an E-graph G whose labels are variables, together with a set of formulas Φ that constrain the possible values of these variables. In this sense, we consider that a symbolic graph denotes the class of all attributed graphs where the variables in the E-graph have been replaced for values that make Φ true in the given data domain. For instance, below on the right, we can see an example of a very simple symbolic graph and, on the left, the (unique) attributed graph denoted by that symbolic graph.

$$\begin{tikzcd}
27 \arrow[r, "12"] \arrow[rr, bend left, "18"] & 45 \arrow[r, "15"] & 37
\end{tikzcd}
\qquad
\begin{tikzcd}
\mathrm{x} \arrow[r, "d_1"] \arrow[rr, bend left, "d_3"] & \mathrm{y} \arrow[r, "d_2"] & \mathrm{z}
\end{tikzcd}$$

with $x = 27 \wedge y = 45 \wedge z = 37$
$\wedge\ d_1 = 12 \wedge d_2 = 15 \wedge d_3 = 18$

However, as said above, a symbolic graph, in general denotes a class of graphs. For instance, the graph below specifies a class of attributed graph that includes the graph depicted above on the left, but it also specifies many other graphs.

$$\begin{tikzcd}
\mathrm{x} \arrow[r, "d_1"] \arrow[rr, bend left, "d_3"] & \mathrm{y} \arrow[r, "d_2"] & \mathrm{z}
\end{tikzcd}$$

with $d_3 \leq d_1 + d_2$

It may be noted that the class of attributed graphs denoted by a symbolic graph may be empty if the associated condition is unsatisfiable.

Therefore, let us define what is a symbolic graph over a given data algebra.

Definition 5 (Symbolic graphs and morphisms) A *symbolic graph* over the the data Σ-algebra D is a pair $\langle G, \Phi \rangle$, where G is an E-graph over a set of variables X, i.e. $L_G = X$, and Φ is a set of first-order Σ-formulas over the free variables in X and over the values in D.

Given symbolic graphs $\langle G_1, \Phi_1 \rangle$ and $\langle G_2, \Phi_2 \rangle$ over the same data algebra D, a symbolic graph morphism $h : \langle G_1, \Phi_1 \rangle \rightarrow \langle G_2, \Phi_2 \rangle$ is an E-graph morphism $h : G_1 \rightarrow G_2$ such that $D \models \Phi_2 \Rightarrow h^{\#}(\Phi_1)$, where $h^{\#}(\Phi_1)$ is the set of formulas obtained when replacing in Φ_1 every variable x_1 in the set of labels of G_1 by $h_L(x_1)$.

Symbolic graphs over D together with their morphisms form the category $\mathbf{SymbGraphs_D}$.

In what follows, to simplify notation, even if it may be considered an abuse of notation, we will write $h(\Phi)$ instead of $h^{\#}(\Phi)$. Moreover, also for simplicity, we may identify the set of formulas Φ with the formula consisting of the conjunction of all the formulas in Φ, even if that formula may be infinitary in the case where Φ is an infinite set.

Notice that, according to the above definition, given any E-graph G, if $D \models \Phi \Leftrightarrow \Phi'$ then $\langle G, \Phi \rangle$ and $\langle G, \Phi' \rangle$ are isomorphic in $\mathbf{SymbGraphs_D}$.

To show that symbolic graphs are an adhesive HLR category, first, we have to define our notion of M-morphism over symbolic graphs. We consider that M-morphisms are monomorphisms where the formulas constraining the source and target graphs are equivalent (in most cases they will just be the same formula). The intuition of this definition is based on the use of our category of symbolic graphs to define graph transformation. More precisely, we think that the most reasonable formulation of graph transformation rules in our context is based on defining a graph transformation rule as an E-graph transformation rule, together with a set of formulas that globally constrain and relate all the variables in the rule. This is equivalent to consider that the left and right-hand sides (and also the interface) of a rule are constrained by the same set of formulas.

Definition 6 (M-morphisms) An *M-morphism* $h : \langle G, \Phi \rangle \rightarrow \langle G', \Phi' \rangle$ is a monomorphism such that $L_G \equiv L_{G'}$, i.e. h_L is a bijection, and $D \models h(\Phi) \Leftrightarrow \Phi'$

It is not difficult to see that M-morphisms satisfy the required properties. Then, to define pushouts and pullbacks in $\mathbf{SymbGraphs_D}$ we use, respectively, pushouts and pullbacks in $\mathbf{E-Graphs}$. More precisely, the pushout of $\langle G_1, \Phi_1 \rangle \leftarrow \langle G_0, \Phi_0 \rangle \rightarrow \langle G_2, \Phi_2 \rangle$ is a graph $\langle G_3, \Phi_3 \rangle$, where G_3 is the pushout of $G_1 \leftarrow G_0 \rightarrow G_2$ and Φ_3 is the conjunction of Φ_1 and Φ_2. The case of pullbacks is similar, but the pullback of $\langle G_1, \Phi_1 \rangle \rightarrow \langle G_3, \Phi_3 \rangle \leftarrow \langle G_2, \Phi_2 \rangle$ is the graph $\langle G_0, \Phi_0 \rangle$, where is the pullback of $G_1 \rightarrow G_3 \leftarrow G_2$ and Φ_0 is the disjunction of Φ_1 and Φ_2. However, since the G_0 may include a strict subset of the variables of Φ_1 and Φ_2, in this case Φ_0 is existentially quantified by the variables not in G_0.

Proposition 1 $\mathbf{SymbGraphs_D}$ *has pushouts and pullbacks.*

To see that pushouts and pullbacks preserve M-morphisms we just have to do some basic logical deduction. If the diagram below is a pushout and h_1 is an M-morphism then we essentially have to prove that if $D \models \Phi_2 \Rightarrow h_2(\Phi_0)$ then $D \models \Phi_2 \Leftrightarrow (h_2(\Phi_0) \wedge \Phi_2)$, which is obvious. The case of the pullbacks is slightly more complex because of the existential quantifiers in Φ_0.

$$\begin{tikzcd}
\langle G_0,\Phi_0\rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1,\Phi_1\rangle \arrow[d, "g_1"] \\
\langle G_2,\Phi_2\rangle \arrow[r, "g_2"'] & \langle G_3,\Phi_3\rangle
\end{tikzcd}$$

Proposition 2 *Pushouts and pullbacks preserve M-morphisms.*

Finally, to prove the van Kampen property we show that a cube in $\mathbf{SymbGraphs_D}$ is a van Kampen square if and only if the underlying cube in in $\mathbf{E}-\mathbf{Graphs}$ is also a van Kampen square. To do this, again we just need to do some basic logical reasoning. As a consequence we have:

Theorem 2 $\mathbf{SymbGraphs_D}$ *is adhesive HLR.*

4 Symbolic graphs and attributed graphs

In this section we present the relation between the categories of symbolic and attributed graphs over a given data algebra. On one hand, we will see that every symbolic graph may be seen as denoting a class (a subcategory) of attributed graphs, which may be considered its semantics. On the other hand, we will see that every attributed graphs can be represented in a canonical way by a symbolic graph, which means that, for a given data algebra, the category of attributed graphs can be seen as a subcategory of the corresponding category of symbolic graphs.

Definition 7 (Semantics of symbolic graphs) Given a symbolic graph $\langle G,\Phi\rangle$ over a data algebra D, its semantics is a class of attributed graphs defined as follows:

$$Sem(\langle G,\Phi\rangle) = \{\langle\sigma(G),D\rangle \mid \sigma : L_G \to D \text{ and } D \models \sigma(\Phi)\}$$

where $\sigma(G)$ denotes the graph obtained according to Def. 2.

For example, given the symbolic graph below:

$$\begin{tikzcd}
x \arrow[r, "d_1"] \arrow[rr, bend left, "d_3"] & y \arrow[r, "d_2"] & z
\end{tikzcd}$$

with $d_3 \leq d_1 + d_2$

we have that its semantics would include the following attributed graphs:

$$\begin{tikzcd}
27 \arrow[r, "12"] \arrow[rr, bend left, "18"] & 45 \arrow[r, "15"] & 37
\end{tikzcd}
\qquad
\begin{tikzcd}
6 \arrow[r, "8"] \arrow[rr, bend left, "4"] & 7 \arrow[r, "25"] & 8
\end{tikzcd}$$

Conversely, we can identify every attributed graph AG with a *grounded* symbolic graph whose semantics consists only of AG. More precisely a grounded graph is a symbolic graph $\langle G, \Phi \rangle$ that includes a variable x_v for each element v of the data algebra and where the only substitution $\sigma : L_G \to D$ such that $D \models \sigma(\Phi)$ is defined for each variable x_v as $\sigma(x_v) = v$.

Definition 8 (Grounded symbolic graphs) A symbolic graph $\langle G, \Phi \rangle$ over a data algebra D is grounded if

1. L_G includes a variable, which we denote by x_v, for each value $v \in D$, and
2. For every substitution $\sigma : L_G \to D$, such that $D \models \sigma(\Phi)$, we have $\sigma(x_v) = v$, for each variable $x_v \in L_G$.

Moreover, we define **GSymbGraphs$_\mathbf{D}$** as the full subcategory of **SymbGraphs$_\mathbf{D}$** consisting of all grounded graphs.

Notice that if $\langle G, \Phi \rangle$ is grounded and $\sigma : L_G \to D$ is a substitution such that $D \models \sigma(\Phi)$ then $\sigma^* : G \to \sigma(G)$ is an isomorphism.
It should be obvious that the semantics of a grounded graph includes exactly one attributed graph and that grounded graphs are closed up to isomorphism. Moreover, we can see and that for every attributed graph AG there is a unique grounded symbolic graph (up to isomorphism) $GSG(AG)$ such that $Sem(GSG(AG))$ consists of AG. In particular, the E-graph associated to the symbolic graph is obtained substituting every data value v in a set of labels by a variable x_v. Then, the set of formulas in the symbolic graph consists of an equation $x_v = v$, for each value v in D.

Proposition 3

1. *If SG is grounded then Sem(SG) consists exactly of one attributed graph.*
2. *Grounded symbolic graphs are closed up to isomorphism.*
3. *For each attributed graph $AG = \langle G, D \rangle$ there is a unique grounded symbolic graph $GSG(AG)$ such that $AG \in Sem(GSG(AG))$, which means that $Sem(GSG(AG)) = \{AG\}$.*

As a consequence, we can identify each attributed graph with a grounded symbolic graph, and vice versa. Therefore, we may ask whether **AttGraphs**$_D$ is isomorphic or equivalent to some subcategory of **SymbGraphs$_\mathbf{D}$** consisting of all grounded graphs. The answer is negative since GSG cannot be made injective on morphisms as the following counter-example shows.

Example 1 *Let D be a data algebra consisting of two values, which we call a and b. Let AG be an attributed graph having no graph nodes and no graph edges (i.e. the graph structure of AG is empty, which means that it consists only of the label nodes a and b). As a consequence, $GSG(AG) = \langle G, x_a = a \wedge x_b = b \rangle$, where G is an E-graph consisting only of the label nodes x_a and x_b. Now, there are four morphisms, f_1, f_2, f_3 and f_4, from AG to itself:*

- $f_1(a) = a,\ f_1(b) = b.$

- $f_2(a) = a,\ f_2(b) = a.$
- $f_3(a) = b,\ f_3(b) = b.$
- $f_4(a) = b,\ f_4(b) = a.$

However the only morphism from $GSG(AG)$ to itself is the identity. For example we may see that the mapping $g : \{x_a, x_b\} \rightarrow \{x_a, x_b\}$, defined $g(x_a) = x_a$, $g(x_b) = x_a$, does not define a symbolic graph morphism. In particular, if g is a morphism it should hold that $D \models (x_a = a \wedge x_b = b) \Rightarrow g(x_a = a \wedge x_b = b)$. But this is equivalent to $D \models (x_a = a \wedge x_b = b) \Rightarrow (x_a = a \wedge x_a = b)$, which is obviously false.

The problem in the above counter-example is that we assume that we can define any mapping between the elements of the algebra, while for the variables of the grounded graphs we are forced to map the variable x_v associated to a value to the corresponding variable associated to the same value. This problem disappears if the value algebra is finitely generated. In that case, we know that the only homomorphism of an algebra into itself is the identity causing that morphisms on attributed graphs should be the identity on data values. This means that, if D is finitely generated then the categories $\mathbf{AttGraphs}_D$ and $\mathbf{GSymbGraphs_D}$ are equivalent. Moreover, this kind of restriction is quite reasonable since, otherwise, the algebra would include values which we cannot refer to. Nevertheless, as we will see in the following section, attributed graph transformation rules are usually defined over non-finitely generated algebras.

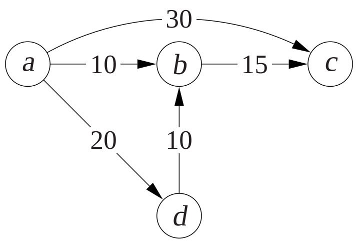
Proposition 4 *If D is finitely generated then $\mathbf{AttGraphs}_D$ and $\mathbf{GSymbGraphs_D}$ are equivalent.*

5 Symbolic graph transformation and attributed graph transformation

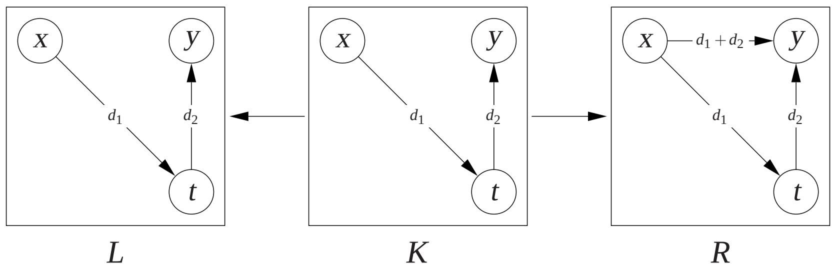
In this section we compare attributed graph transformation with symbolic graph transformation. This comparison may seem trivial: if attributed graphs may be seen as a special case of symbolic graphs then we can conclude that attributed graph transformation is a special case of symbolic graph transformation. However things are not so obvious. As we have seen, if the given data algebra D is finitely generated, we can identify attributed graphs over D with grounded symbolic graphs over D. This means, in that case, that if transformation rules are spans of M-morphisms in $\mathbf{AttGraphs}_D$ then these transformation rules can be considered equivalent to spans of M-morphisms in $\mathbf{GSymbGraphs_D}$, and the application of these rules to a graph AG in $\mathbf{AttGraphs}_D$ is equivalent to the transformation of $GSG(AG)$ by the corresponding rules in $\mathbf{GSymbGraphs_D}$. The problem is that if the graphs that we want to transform are in $\mathbf{AttGraphs}_D$, usually, the transformation rules will not be spans of M-morphisms in $\mathbf{AttGraphs}_D$, but in $\mathbf{AttGraphs}_{D'}$, where D' is some free algebra over D and, hence, different from D. This means that the rules for transforming graphs in $\mathbf{AttGraphs}_D$ are are not directly equivalent to rules in $\mathbf{SymbGraphs_D}$.

In order to compare attributed and symbolic graph transformation we start describing how we usually define attributed graph transformation rules by means of the following example.

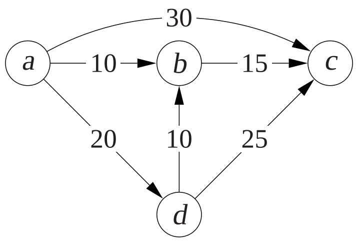
Example 2 *Let us suppose that we are dealing with a class of graphs whose edges have an attribute that represents the distance between the source and target nodes. For instance, the graph G below may be an example of a graph in this class:*



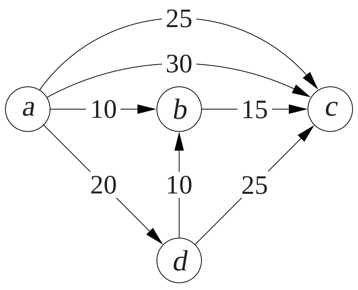
Let us also suppose that we want to compute the distance of the shortest paths between any two nodes. The rule p below could describe how a new distance can be computed:



Applying this rule, matching x with d, y with c and, t with b, we could transform G into G_1:



Similarly, matching x with a, t with b, and y with c, we would get the graph G_2:

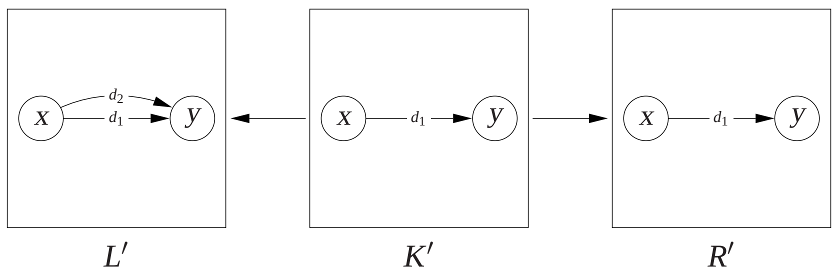


Let us analyze what happens in this example. G, G_1, and G_2 are attributed graphs over the *algebra D natural numbers, defined over the signature* Σ*:*

Sorts	$nat, bool$
Opns	$0 : nat$
	$suc : nat \to nat$
	$true, false : bool$
	$+ : nat \times nat \to nat$
	$\leq : nat \times nat \to bool$

This means that the graphs G, G_1*, and* G_2 *include the natural numbers and the booleans as label nodes, even if they are not depicted in the above figures. But the graphs* L, K*, and* R *in the transformation rule* p *are not defined over the same algebra. The reason is that the labels* d_1, d_2*, or* $d_1 + d_2$ *are not in* D*, since they are not natural numbers. The simplest solution that we can use here, is to consider that* L, K*, and* R *are attributed graphs over the algebra* $T_\Sigma(\{d_1, d_2\})$*, i.e. the term algebra over the variables* d_1 *and* d_2*. These graphs would include as label nodes all the possible* Σ*-terms over these two variables, even if they are not depicted explicitly. Now, according to the example, when we apply* p *to* G *we define a morphism* m *from* L *into* G *matching* x *with* D*,* y *with* C *and, and* T *with* B*. Obviously,* m *would also match the edges in* L *with the edges in* G *in the expected way and it would also match* d_1 *with* 10 *and* d_2 *with* 15*. But this is not all. The match* m *includes a* Σ*-homomorphism* m_{alg} *from* $T_\Sigma(\{d_1, d_2\})$ *to* D *matching not only* d_1 *with* 10 *and* d_2 *with* 15*, but also each possible term over* d_1 *and* d_2 *with its corresponding value, after assigning to* d_1 *and* d_2 *the values* 10 *and* 15*, respectively. This means that, for instance,* m *would also match* $suc(d_1)$ *with* 11 *or* $suc(d_1) \leq suc(suc(d_2))$ *to* $\mathbf{t}$*. In particular,* m *would also match* $d_1 + d_2$ *with* 25*, even if* $d_1 + d_2$ *is not explicitly depicted in* L*, i.e. we need to compute the resulting value of* $d_1 + d_2$ *when defining the match, before computing the transformation.*

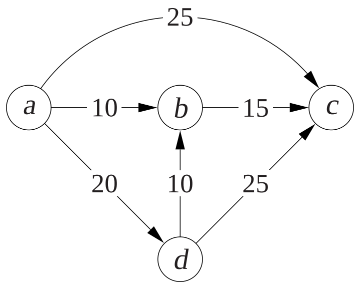
The fact that the values of the underlying algebra are considered (label) nodes of the attributed graphs, together with the fact that match morphisms must be homomorphisms for the algebra part, allow us to do some kind of conditional graph transformation without using a negative application condition (NAC) [3, 9]. For example, in G_2 *the nodes* A *and* C *are connected with two edges labelled with* 25 *and* 30*, respectively. If we want to compute the shortest distances on the given graph, we may like to delete the edge which includes the largest distance. In particular the following rule* p' *deletes the edge labelled with* d_2*:*



However, we would like to use this rule only if $d_1 \leq d_2$*. We can do this, as said above, without using a NAC. First we define a specification* SP *including the variables* d_1 *and* d_2 *as constants,* *and the desired condition as an axiom, i.e.:*

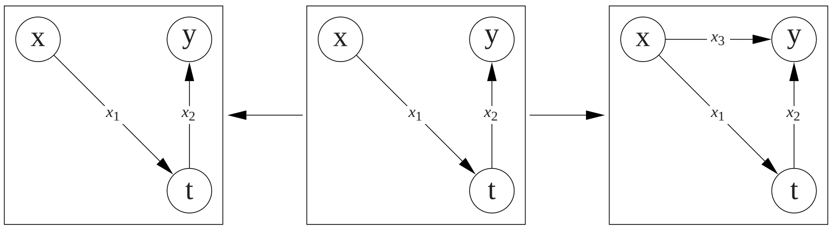
$$\begin{array}{rll} SP = & \textbf{Sorts} & nat, bool \\ & \textbf{Opns} & 0 : nat \\ & & d_1, d_2 : nat \\ & & suc : nat \to nat \\ & & true, false : bool \\ & & + : nat \times nat \to nat \\ & & \leq : nat \times nat \to bool \\ & \textbf{Axms} & (d_1 \leq d_2) = true \end{array}$$

Let T_{SP} be the initial algebra associated to SP, and $T_{SP}|_\Sigma$ its Σ-reduct. In T_{SP} the term $d_1 \leq d_2$ and the term $true$ denote the same element. This means that if h is a homomorphism from $T_{SP}|_\Sigma$ into D, $h(d_1) = n_1$, and $h(d_2) = n_2$, then n_1 must be smaller or equal than n_2. As a consequence, if we consider that L', K', and R' are attributed graphs on $T_{SP}|_\Sigma$ then any morphism from L' to G_2 matching node x to node A and node y to node C would necessarily match d_1 to 25 and d_2 to 30. Hence the application of r' to G_2 with that match would yield the graph:



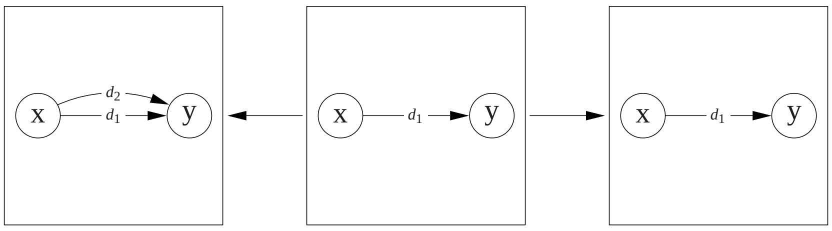
Therefore, we can consider that, in a transformation system for attributed graphs over a Σ-algebra D, each rule r is a span on the category $\textbf{AttGraphs}_{D_r}$, where $D_r = T_{SP_r}|_\Sigma$ and SP_r is a specification $SP_r = (\Sigma_r, \Phi_r)$, such that $\Sigma_r = \Sigma \cup X$ and Φ_r is a set of Σ_r-equations or conditional equations, since this ensures the existence of initial algebras. For simplicity, we assume that Φ_r only includes equations, since the case of conditional equations is similar. Under this assumption, we may see that attributed graph transformation systems can be seen as a special case of symbolic graph transformation system. The idea is that every rule r as above can be represented by a symbolic transformation rule r', where each term t in r has been replaced by a variable x_t and the set of formulas $\Phi_{r'}$ associated to r' consists of all the axioms in Φ_r plus all the equalities $x_t = t$.

For instance, the symbolic rules associated to the attributed graph transformation rules described above could look like:



$$\textbf{with} \;\; x_1 = d_1 \wedge x_2 = d_2 \wedge x_3 = d_1 + d_2$$

and



with $(d_1 \leq d_2) = true$

Definition 9 Given a specification $SP = (\Sigma \cup X, \Phi)$, such that there is an initial algebra T_{SP} in the category of SP-algebras, we say that $ch : T_{SP} \to T_{\Sigma \cup X}$ is a *choice function* for T_{SP} if for every element $|t'| \in T_{SP}$ if $ch(t) = t'$ then $T_{SP} \models t = t'$.

Given SP, an attributed graph $AG = \langle G, T_{SP}|_\Sigma \rangle$, and a choice function ch for T_{SP} we define the symbolic representation of AG with respect to ch, $SR_{ch}(AG)$ as the symbolic graph $\langle G', \Phi' \rangle$, where:

- The set of labels of the E-graph G' is $X \cup Y$, where Y is disjoint with X and it consists of a variable y_a for each element $a \in T_{SP}$ such that $a \notin \{|x| \mid x \in X\}$.
- $G' = f^*(G)$, where $f : T_{SP} \to Y$ is a substitution such that for every $a \in T_{SP}$, if $a \notin \{|x| \mid x \in X\}$ then $f(a) = y_{ch(a)}$. Otherwise, $f(|x|) = x$.
- $\Phi' = \Phi \cup \{y_e = t \mid y_e \in Y \wedge ch(e) = t\}$.

This means that G' includes as labels the variables in X and a variable y_a for every element a in T_{SP} which is not the congruence class of a variable in X. This means that the substitution f is a bijection. As a consequence, for every attributed graph $AG = \langle G, T_{SP}|_\Sigma \rangle$, if $SR_{ch}(AG) = \langle G', \Phi' \rangle$ then G and G' are isomorphic E-graphs, f^*.

It may be noted that we have not stated to which category $\mathbf{SymbGraphs_D}$ should $SR_{ch}(AG)$ belong. The reason is that we may consider that $SR_{ch}(AG)$ is in any category $\mathbf{SymbGraphs_D}$, for any Σ-algebra D.

We may extend the previous definition to define the symbolic representation of attributed graph transformation rules over the algebra $T_{SP}|_\Sigma$:

Definition 10 Given a specification $SP = (\Sigma \cup X, \Phi)$, such that there is an initial algebra T_{SP}, a choice function ch for T_{SP}, and an attributed graph transformation rule $r = (\langle L, T_{SP}|_\Sigma \rangle \hookleftarrow \langle K, T_{SP}|_\Sigma \rangle \hookrightarrow \langle R, T_{SP}|_\Sigma \rangle)$, we define the symbolic representation of r with respect to ch, $SR_{ch}(r)$ as the symbolic transformation rule $r' = \langle L' \hookleftarrow K' \hookrightarrow R', \Phi' \rangle$ where:

- $\langle L', \Phi' \rangle = SR_{ch}(\langle L, T_{SP}|_\Sigma \rangle)$, $\langle K', \Phi' \rangle = SR_{ch}(\langle K, T_{SP}|_\Sigma \rangle)$, and $\langle R', \Phi' \rangle = SR_{ch}(\langle R, T_{SP}|_\Sigma \rangle)$.
- The inclusions $K' \subseteq L'$ and $K' \subseteq R'$ are a consequence, first, of the fact that we assume that $K \subseteq L$ and $K \subseteq R$[1]; second, of the definition of how a label substitution is applied to an E-graph; and third of the fact that the use of the choice function ensures that the substitution of values in $T_{SP}|_\Sigma$ by variables in Y is the same on the three graphs L', R', and K'.

[1] Since the morphisms relating L and R are M-morphisms, without loss of generality, we may assume that they are the identity on the algebra part and an inclusion on the graph part

Moreover, it may be noticed that, by definition of the choice functions, diagrams (1), (2), (3), and (4) below are pushouts, where f^*_L, f^*_K, and f^*_R are, respectively, the isomorphisms relating L, K, and R with L', K', and R', and where f^{*-1}_L, f^{*-1}_K, and f^{*-1}_R are their inverses:

$$\begin{tikzcd}
L \arrow[d, "f^*_L"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, "f^*_K"'] \arrow[dr, phantom, "(2)"] & R \arrow[d, "f^*_R"] \\
L' & K' \arrow[l, hook'] \arrow[r, hook] & R'
\end{tikzcd}
\qquad
\begin{tikzcd}
L' \arrow[d, "f^{*-1}_L"'] \arrow[dr, phantom, "(3)"] & K' \arrow[l, hook'] \arrow[r, hook] \arrow[d, "f^{*-1}_K"'] \arrow[dr, phantom, "(4)"] & R' \arrow[d, "f^{*-1}_R"] \\
L & K \arrow[l, hook'] \arrow[r, hook] & R
\end{tikzcd}$$

Theorem 3 *Let $r = (AL \leftarrow AK \rightarrow AR)$ be an attributed graph transformation rule, where AL, AK, and AR are attributed graphs over $T_{SP}|_\Sigma$ and $SP = (\Sigma \cup X, \Phi)$, let ch be a choice function for T_{SP}, and let $r' = SR_{ch}(r)$, then for every attributed graph $AG = \langle G, D\rangle$ and every morphism $m : AL \rightarrow AG$ there is a morphism $m' : \langle SR_{ch}(AL), \Phi'\rangle \rightarrow GSG(AG)$ such that AG is transformed into AH by r with match m, i.e. $AG \Rightarrow^m_r AH$, if and only if $GSG(AG) \Rightarrow^{m'}_{r'} GSG(AH)$. Conversely, for every morphism $m' : \langle SR_{ch}(AL), \Phi'\rangle \rightarrow GSG(AG)$ there is a morphism $m : AL \rightarrow AG$ such that $GSG(AG) \Rightarrow^{m'}_{r'} GSG(AH)$ if and only if $AG \Rightarrow^m_r AH$.*

The theorem above shows that attributed graph transformation can be seen as a special case of symbolic graph transformation. One may wonder whether both kind of transformations can be considered equivalent in the sense that every symbolic graph transformation rule r can be coded into an attributed graph transformation rule r' such that the application of r to a grounded graph produces the same effect as the application of r' to the corresponding attributed graph. The answer is negative as the counter-example below shows, which means that symbolic transformation rules have more definitional power than attributed graph transformation rules.

Example 3 *Let us suppose that the following symbolic graph SG is the left-hand side of a symbolic graph transformation rule r:*

$$\begin{tikzcd}
x \arrow[r] & y
\end{tikzcd}$$

with $x = 0 \vee y = 0 \vee x = y$

where the signature of the data domain is:

Sorts	*nat*
Opns	$0 : nat$
	$suc : nat \rightarrow nat$

and where the given data algebra D is the algebra of natural numbers. This means that, if r could be represented by an attributed graph transformation rule r', then r' would include as a left-hand side an attributed graph AG like:

$$\begin{tikzcd}
a_1 \arrow[r] & a_2
\end{tikzcd}$$

where a_1 and a_2 are elements of some Σ-algebra A. Moreover, there should exist a match m from SG into any grounded symbolic graph SG' if and only if there exists a similar match m' from AG into the corresponding attributed graph. In particular, given the symbolic graph:

$$\begin{tikzcd} x \arrow[r] & y \end{tikzcd}$$

$$\textbf{with } x = n_1 \wedge y = n_2$$

where a and b are two natural numbers, there should exist a homomorphism from A to D mapping a_1 to n_1 and a_2 to n_2 if and only if $n_1 = 0$ or $n_2 = 0$ or $n_1 = n_2$. Let us see that this is impossible.

First, we may notice that we may assume without loss of generality that A satisfies the axiom:

$$e : s(x) = s(y) \Rightarrow x = y$$

since for every homomorphism $h : A \to D$ there is a unique homomorphism $h' : A/\equiv_e \to D$ such that the diagram below commutes:

$$\begin{tikzcd} A \arrow[rr, "h"] \arrow[d, "i"'] & & D \\ A/\equiv_e \arrow[urr, "h'"'] & & \end{tikzcd}$$

where $\equiv_e$ is the congruence on A defined by the axiom e and i is the canonical homomorphism from A into its quotient, mapping every element from A into its congruence class. Vice versa, for every homomorphism $h' : A/\equiv_e \to D$ there is a unique homomorphism $h : A \to D$ such that the diagram above commutes. Finally, we know that $h(a_1) = 0$ or $h(a_2) = 0$ or $h(a_1) = h(a_2)$ if and only if $h'(|a_1|) = 0$ or $h'(|a_2|) = 0$ or $h'(|a_1|) = h'(|a_2|)$.

Therefore, let us assume that A satisfies the above axiom. Now, let us notice that neither a_1 nor a_2 can be the value of some ground term $suc^n(0)$, for $0 \leq n$. The reason is that, otherwise, if $n_1 = n_2 \neq n$ the match would be impossible. We can also see that it is not possible that a_1 is the value of some term $suc^n(a)$, for $1 \leq n$ and any $a_0 \in A$. The reason is that, otherwise, if $n_1 = 0$ the match would be impossible, against the assumption, since if the match m' satisfies $m'(a_0) = n_0$ then $m'(a_1)$ would be $n + n_0$. For similar reasons, we know that it is not possible that a_2 is the value of some term $suc^n(a_0)$, for $1 \leq n$ and any $a_0 \in A$. As a consequence, we can see that

$$A' = A \setminus \{a \mid (a = suc^n_A(a_1)) \vee (a = suc^n_A(a_2)) \text{ for some } n \geq 0\}$$

is a subalgebra of A. Suppose, otherwise, that A' is not a subalgebra of A. This would mean that there is an element $a' \in A'$ such that $suc_A(a') \in A \setminus A'$. But this would mean that $suc_A(a') = suc^n_A(a_1))$ or $suc_A(a') = suc^n_A(a_2))$. But this would imply one of the following cases:

1. *$suc_A(a') = a_1$ or $suc_A(a') = a_2$. These two cases are impossible according to what we have proved above.*

2. *$suc_A(a') = suc^n_A(a_1))$ or $suc_A(a') = suc^n_A(a_2))$ for $n \geq 1$. However, since A is assumed to satisfy the axiom e, this means that $a' = suc^{n-1}_A(a_1))$ or $a' = suc^{n-1}_A(a_2))$, implying that $a' \in A \setminus A'$, against the hypothesis.*

As a consequence of the previous facts we know that every homomorphism $h : A \rightarrow D$ is uniquely determined by a homomorphism $h' : A' \rightarrow D$ and by the values of $h(a_1)$ and $h(a_2)$, in the sense that given h', there is a unique h extending h' satisfying $h(a_1) = n_1$ and $h(a_2) = n_2$, for any $n_1, n_2 \in D$, and vice versa. But this implies that there is a morphism $m' : A \rightarrow D$ satisfying $m'(a_1) \neq m'(a_2)$.

In general, a symbolic transformation rule $r' = \langle L' \hookleftarrow K' \hookrightarrow R', \Phi' \rangle$ over a Σ-algebra D can be simulated by an attributed graph transformation rule $r = (AL \leftarrow AK \rightarrow AR)$ over a Σ-algebra A, if the specification SP, whose signature is Σ plus the labels in r' (considered as constants), and whose set of axioms is Φ', has an initial algebra T_{SP}. The problem in the previous counter-example is that the associated specification has no initial algebra. In particular, to ensure the existence of initial algebras Φ' should include only equations and conditional equations.

6 Conclusion

In this paper we have presented a new approach to deal with attributed graphs based on the new notion of symbolic graphs, showing that the new category is adhesive HLR, which means that it is adequate to define graph transformation. Moreover we have compared this new approach with the most standard approach to deal with attributed graphs. In particular, there are essentially two kinds of approaches to define attributed graphs and attributed graph transformation. On one hand, we have the approaches [10, 7] where an attributed graph is a pair (G, D) consisting of a graph G and a data algebra D whose values are nodes in G. On the other hand, we have the approaches [12, 1] where attributed graphs are seen as algebras over a given signature $ASIG$, where $ASIG$ is the union of two signatures $ASSIG$, the graph signature and $DSIG$, the data signature, that overlap in the value sorts. In particular, $ASSIG$ may be seen as a representation of the graph part of an attributed graph. In [2] the two approaches are compared showing that they are, up to certain point, equivalent. However, only a complete theory of graph transformation has been formulated for [7] as a consequence of its characterization as an adhesive HLR category (for more detail see [4]). For these reasons, in this paper we have essentially used that approach to study it in connection with our approach based on symbolic graphs.

As we have seen, our approach can be considered an abstract version of [7], since we work at the specification level, rather than dealing directly with algebras to define the attributes. However, as we have shown, it has more expressive power than [7] for the definition of graph transformation rules. In addition to the expressive power, using symbolic attributed graphs has some other advantages. For instance, in [15] working with symbolic attributed graphs makes simple certain kinds of operations defined on transformation rules. For example, this is the case of the operation that, given two transformation rules r_1 and r_2, where r_1 is a subrule of r_2, yields a rule r_3 that computes the remainder of r_2 with respect to r_1, i.e. what has not been computed by r_1 but is computed by r_2. In particular, when working with symbolic graphs the attribute conditions of r_3 are just a simple combination of the attribute conditions of r_1 and r_2. However, if we would have worked with attributed graphs, computing the attributes for r_3 may involve some complex equation solving.

Moreover, we think that there are further aspects related to symbolic graph transformations that deserve some further study. In particular, Using logical conditions to specify the attributes of a graph, may allow us to postpone finding the solution to attribute constraints when making graph transformation. This can make attributed graph transformation more efficient. In addition, a generalization of this idea would allow us to define a certain form of narrowing that may be useful in connection to some kind of problems.

Bibliography

[1] M. Berthold, I. Fischer, M.Koch: Attributed Graph Transformation with Partial Attribution. Technical Report 2000-2 (2000).

[2] H. Ehrig: Attributed Graphs and Typing: Relationship between Different Representations. *Bulletin of the EATCS* 82: 175-190 (2004)

[3] H. Ehrig, A. Habel: Graph Grammars with Application Conditions. In *The Book of L* (Grzegorz Rozenberg and Arto Salomaa, Eds.), Springer (1986), 87–100.

[4] Hartmut Ehrig, Karsten Ehrig, Ulrike Prange, Gabriele Taentzer: *Fundamentals of Algebraic Graph Transformation*, Springer (2006).

[5] Hartmut Ehrig, Annegret Habel, Julia Padberg, Ulrike Prange: Adhesive High-Level Replacement Categories and Systems. In *Graph Transformations, Second International Conference, ICGT 2004*. Springer Lecture Notes in Computer Science 3256 (2004),144-160.

[6] H. Ehrig, B. Mahr: *Fundamentals of Algebraic Specifications 1: Equations and Initial Semantics*. Vol. 6 of EATCS Monographs on Theoretical Computer Science. Springer, 1985.

[7] H. Ehrig, U. Prange, G. Taentzer: Fundamental Theory for Typed Attributed Graph Transformation. In *Graph Transformations, Second International Conference, ICGT 2004*. Springer Lecture Notes in Computer Science 3256 (2004), 161-177.

[8] Esther Guerra, Juan de Lara, Fernando Orejas: Pattern-Based Model-to-Model Transformation: Handling Attribute Conditions. In *Theory and Practice of Model Transformations, Second International Conference, ICMT 2009*, Richard F. Paige (Ed.), Springer Lecture Notes in Computer Science 5563 (2009), pp. 83–99.

[9] A. Habel, R. Heckel, G. Taentzer: Graph Grammars with Negative Application Conditions. Fundam. Inform. 26(3/4): 287–313 (1996).

[10] R. Heckel, J. Küster, G. Taentzer: Towards Automatic Translation of UML Models into Semantic Domains. In Proc. *APPLIGRAPH Workshop on Applied Graph Transformation* 2002, pp. 11–22.

[11] J. Jaffar, M. Maher, K. Marriot, and P. Stukey. The semantics of constraint logic programs. *The Journal of Logic Programming*, (37):1–46, 1998.

[12] M. Löwe, M. Korff, A. Wagner: An Algebraic Framework for the Transformation of Attributed Graphs. In *Term Graph Rewriting: Theory and Practice*. John Wiley and Sons Ltd. (1993) 185199.

[13] S. Lack, P. Sobocinski: Adhesive Categories. In *Foundations of Software Science and Computation Structures, 7th International Conference, FOSSACS 2004* (Igor Walukiewicz, Ed.), Springer Lecture Notes in Computer Science 2987 (2004), 273–288.

[14] P. Lucio, F. Orejas, E. Pasarella and E. Pino. A Functorial Framework for Constraint Normal Logic Programming. In *Algebra, Meaning, and Computation, Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*, Kokichi Futatsugi, Jean-Pierre Jouannaud, José Meseguer (Eds.), Springer-Verlag Lecture Notes in Computer Science 4060 (2006), 555–577.

[15] M. Naeem, R. Heckel, F. Orejas, F. Hermann, Incremental Service Composition based on Partial Matching of Visual Contracts. Accepted FASE 2010.

[16] F. Orejas: Attributed Graph Constraints. In *Graph Transformations, 4th International Conference, ICGT 2008* (Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, Gabriele TaentzerEds.), Springer Lecture Notes in Computer Science 5214 (2008): 274–288

[17] F. Orejas: Attributed Graph Constraints for Attributed Graph Transformation. Submitted, 2009.

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

Appendix

In this appendix we provide the detailed proofs of the results presented in the paper. In some cases these proofs need some auxiliary results which are also included.

Proposition 5 *M-morphisms in* $\mathbf{SymbGraphs_D}$ *are closed under isomorphism, composition and decomposition*

Proof. Let us first prove that isomorphisms in $\mathbf{SymbGraphs_D}$ are M-morphisms. Let $h: \langle G, \Phi\rangle \to \langle G', \Phi'\rangle$ be an isomorphism. On one hand, this means that all its components are bijections, i.e. h_L is a bijection. On the other hand, it also means that there is another isomorphism $g: \langle G', \Phi'\rangle \to \langle G, \Phi\rangle$ such that $g \circ h$ and $h \circ g$ are identities. But, then, we have $D \models \Phi' \Rightarrow h(\Phi)$ and $D \models \Phi \Rightarrow g(\Phi')$ but, since h_L and g_L are bijections (i.e. bijective variable renamings) we have that $D \models \Phi' \Leftrightarrow h(\Phi)$.

The fact that isomorphisms are M-morphism means that, if we prove that M-morphisms are closed under composition, then M-morphisms would also be closed under isomorphism. So, let us see that M-morphisms are closed under composition. Let $g: \langle G, \Phi\rangle \to \langle G', \Phi'\rangle$ and $h: \langle G', \Phi'\rangle \to \langle G'', \Phi''\rangle$ be two M-morphisms. Obviously $h \circ g$ is a monomorphism and also $(h \circ g)_L$ is a bijection. Moreover, if $D \models \Phi' \Leftrightarrow g(\Phi)$ and $D \models \Phi'' \Leftrightarrow h(\Phi')$ then $D \models \Phi'' \Leftrightarrow h(g(\Phi))$.

Finally, let us see that M-morphisms are closed under decomposition. Let $g: \langle G, \Phi\rangle \to \langle G', \Phi'\rangle$ and $h: \langle G', \Phi'\rangle \to \langle G'', \Phi''\rangle$ be two monomorphisms where h and $h \circ g$ are M-morphisms. Obviously, this implies that g is a monomorphism and that g_L is a bijection. Moreover, if $D \models \Phi'' \Leftrightarrow h(\Phi')$ and $D \models \Phi'' \Leftrightarrow h(g(\Phi))$, since both h_L and $(h \circ g)_L$ are bijections then this means that the three formulas Φ, Φ' and Φ'' are equivalent modulo the corresponding variable renamings. Therefore, $D \models \Phi' \Leftrightarrow g(\Phi)$.

□

Before proving Proposition 1 we will present to auxiliary results characterizing pushouts and pullbacks in $\mathbf{SymbGraphs_D}$.

Proposition 6 *Diagram* (1) *below is a pushout if and only if diagram* (2) *is also a pushout and* $SP_{\mathscr{D}} \models \Phi_3 \Leftrightarrow (g_1(\Phi_1) \wedge g_2(\Phi_2))$.

$$\begin{tikzcd}
\langle G_0, \Phi_0\rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] \arrow[dr, phantom, "(1)"] & \langle G_1, \Phi_1\rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2\rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3\rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
G_0 \arrow[r, "h_1"] \arrow[d, "h_2"'] \arrow[dr, phantom, "(2)"] & G_1 \arrow[d, "g_1"] \\
G_2 \arrow[r, "g_2"'] & G_3
\end{tikzcd}$$

Proof. If diagram (1) is a pushout then we have to prove that if

$$\begin{tikzcd}
G_0 \arrow[r, "h_1"] \arrow[d, "h_2"'] & G_1 \arrow[d, "g_1'"] \\
G_2 \arrow[r, "g_2'"'] & G_3'
\end{tikzcd}$$

commutes then there is a unique morphism $h : G_3 \to G'_3$ satisfying that $h \circ g_1 = g'_1$ and $h \circ g_2 = g'_2$. Now, if this diagram commutes then it will also commute the following diagram:

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g'_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g'_2"'] & \langle G'_3, \{\mathbf{false}\} \rangle
\end{tikzcd}$$

But this means that there is a unique morphism $h : \langle G_3, \Phi_3 \rangle \to \langle G_3, \{\mathbf{false}\} \rangle$ satisfying that $h \circ g_1 = g'_1$ and $h \circ g_2 = g'_2$. Moreover, if $h' : G_3 \to G'_3$ is another morphism satisfying that $h' \circ g_1 = g'_1$ and $h' \circ g_2 = g'_2$. Then $h' : \langle G_3, \Phi_3 \rangle \to \langle G_3, \{\mathbf{false}\} \rangle$ would also be a morphism such that $h' \circ g_1 = g'_1$ and $h' \circ g_2 = g'_2$, and the uniqueness of h in $\mathbf{SymbGraphs_D}$ would imply $h = h'$ in $\mathbf{E} - \mathbf{Graphs}$.

Conversely, if diagram (2) is a pushout then we have that

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}$$

where $\Phi_3 = (g_1(\Phi_1) \wedge g_2(\Phi_2))$, is a commuting diagram in $\mathbf{SymbGraphs_D}$, since $(g_1(\Phi_1) \wedge g_2(\Phi_2)) \Rightarrow g_1(\Phi_1)$ and $(g_1(\Phi_1) \wedge g_2(\Phi_2)) \Rightarrow g_1(\Phi_1)$ are tautologies. Moreover, we know that if

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g'_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g'_2"'] & \langle G'_3, \Phi'_3 \rangle
\end{tikzcd}$$

commutes then there is a unique morphism $h : G_3 \to G'_3$ satisfying that $h \circ g_1 = g'_1$ and $h \circ g_2 = g'_2$. But this means that $h : \langle G_3, \Phi_3 \rangle \to \langle G'_3, \Phi'_3 \rangle$ is a morphism, since if $D \models \Phi'_3 \Rightarrow g'_1(\Phi_1)$ and $D \models \Phi'_3 \Rightarrow g'_2(\Phi_2)$ then $D \models (\Phi'_3) \Rightarrow (g'_1(\Phi_1) \wedge g'_2(\Phi_2))$. But we know that $g'_1(\Phi_1) \wedge g'_2(\Phi_2) = (h \circ g_1)(\Phi_1) \wedge (h \circ g_2)(\Phi_2) = h(g_1(\Phi_1) \wedge g_2(\Phi_2)) = h(\Phi_3)$ and this means that $D \models (\Phi'_3) \Rightarrow h(\Phi_3)$. Finally, if $h' : \langle G_3, \Phi_3 \rangle \to \langle G_3, \Phi'_3 \rangle$ is a morphism satisfying that $h' \circ g_1 = g'_1$ and $h' \circ g_2 = g'_2$ then, by the uniqueness of h in $\mathbf{E} - \mathbf{Graphs}$, we have that $h = h'$. $\square$

Pullbacks in $\mathbf{SymbGraphs_D}$ can be characterized similarly. However, the proposition below seems to be stated slightly more restrictively than Proposition 6. In particular, we require that the the variables (labels) in G_0 are disjoint with the variables in G_1 and G_2. The reason for this condition is that it simplifies substantially the proposition and its proof. However, it must be noted that this condition is not really a restriction. The reason is that two attributed graphs $\langle G, \Phi \rangle$ and $\langle G', \Phi' \rangle$ are isomorphic if $\langle G', \Phi' \rangle$ can be obtained from $\langle G, \Phi \rangle$ by a variable renaming. Therefore, if needed, to define the pullback of some attributed graphs we can, first, rename the variables involved and, then, use the results below.

Proposition 7 *If L_{G_0} is disjoint from $L_{G_1} \cup L_{G_2}$ then diagram (1) below is a pullback if and only if $SP_{\mathscr{D}} \models \Phi_0 \Leftrightarrow \big(\exists L_{G_1}(\Phi_1 \wedge eq(h_1)) \vee \exists L_{G_2}(\Phi_2 \wedge eq(h_2))\big)$ and diagram (2) is also a pullback, where $\exists L_{G_i}$ denotes an existential quantification over the variables in L_{G_i} and $eq(h_i)$ denotes the conjunction of equalities $\bigwedge_{x_0 \in L_{G_0}} x_0 = h_i(x_0)$*[2].

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] \arrow[dr, phantom, "(1)"] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}
\qquad
\begin{tikzcd}
G_0 \arrow[r, "h_1"] \arrow[d, "h_2"'] \arrow[dr, phantom, "(2)"] & G_1 \arrow[d, "g_1"] \\
G_2 \arrow[r, "g_2"'] & G_3
\end{tikzcd}$$

Proof. The proof is similar to the proof of Proposition 6. First, assuming that diagram (1) is a pullback, we have to prove that if

$$\begin{tikzcd}
G_0' \arrow[r, "h_1'"] \arrow[d, "h_2'"'] & G_1 \arrow[d, "g_1"] \\
G_2 \arrow[r, "g_2"'] & G_3
\end{tikzcd}$$

commutes then there is a unique morphism $h: G_0' \to G_0$ satisfying that $h_1 \circ h = h_1'$ and $h_2 \circ h = h_2'$. Now, if this diagram commutes then it will also commute the following diagram:

$$\begin{tikzcd}
\langle G_0', \{\mathbf{true}\} \rangle \arrow[r, "h_1'"] \arrow[d, "h_2'"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}$$

But this means that there is a unique morphism $h: \langle G_0', \{\mathbf{true}\} \rangle \to \langle G_0, \Phi_0 \rangle$ satisfying that $h_1 \circ h = h_1'$ and $h_2 \circ h = h_2'$. Moreover, if $h': G_0' \to G_0$ is another morphism satisfying that $h_1 \circ h = h_1'$ and $h_2 \circ h = h_2'$. Then $h': \langle G_0', \{\mathbf{true}\} \rangle \to \langle G_0, \Phi_0 \rangle$ would also be a morphism such that $h_1 \circ h' = h_1'$ and $h_2 \circ h' = h_2'$, and the uniqueness of h in $\mathbf{SymbGraphs_D}$ would imply $h = h'$ in $\mathbf{E-Graphs}$.

Conversely, if diagram (2) is a pullback, then we can prove that $h_i: \langle G_0, \Phi_0 \rangle \to \langle G_i, \Phi_i \rangle$, for $i \in \{1,2\}$, are morphisms in $\mathbf{SymbGraphs_D}$. In particular, we have to prove that $D \models \Phi_i \Rightarrow h_i(\Phi_0)$, for each $i \in \{1,2\}$. Let us suppose that $\sigma_1: L_{G_1} \to D$ is a substitution for the variables in L_{G_1} such that $D \models \sigma(\Phi_1)$. We have to show that $D \models \sigma(h(\Phi_0))$ or, equivalently, that $D \models \sigma_0(\Phi_0)$, where $\sigma_0 = \sigma_1 \circ h_1$. In particular, it is enough to prove that $D \models \sigma_0(\exists L_{G_1}(\Phi_1 \wedge eq(h_1)))$, but this is equivalent to prove that there is a substitution $\sigma_1': L_{G_1} \to D$ for the variables in L_{G_1} such that $D \models \sigma_1'(\sigma_0(\Phi_1 \wedge eq(h_1))))$. Notice that $\sigma_1'(\sigma_0(\Phi_1 \wedge eq(h_1)))) = \sigma_1'(\Phi_1) \wedge \sigma_1'(\sigma_0(eq(h_1))))$, since Φ_1 does not involve any variable from L_{G_0}. Now, if we take $\sigma_1' = \sigma_1$, we have, on one hand, that $D \models \sigma(\Phi_1)$ by assumption. On the other hand, if x_0 is a variable in L_{G_0} then, by definition, $\sigma_0(x_0) = \sigma_1 \circ h_1(x_0) = \sigma_1(h_1(x_0))$, which means that $D \models \sigma_1(\sigma_0(x_0 = h_1(x_0)))$, for every $x_0 \in L_{G_0}$, and therefore $D \models \sigma_1(\sigma_0(eq(h_1)))$. As a consequence, $D \models \sigma_1(h_i(\Phi_0))$, for each $i \in \{1,2\}$. This means that the diagram:

[2] Notice that in these equalities $h_i(x_0)$ denotes the variable in L_{G_i} which is the image of x_0 through h_i

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}$$

is a commuting diagram in $\mathbf{SymbGraphs_D}$. Moreover, we know that if

$$\begin{tikzcd}
\langle G'_0, \Phi_0 \rangle \arrow[r, "h'_1"] \arrow[d, "h'_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}$$

commutes then there is a unique morphism $h : G'_0 \rightarrow G_0$ satisfying that $h_1 \circ h = h'_1$ and $h_2 \circ h = h'_2$ in $\mathbf{E-Graphs}$. But then we can prove that $h : \langle G'_0, \Phi'_0 \rangle \rightarrow \langle G_0, \Phi_0 \rangle$ is a morphism in $\mathbf{SymbGraphs_D}$. Suppose that $\sigma_0 : L_{G_0} \rightarrow D$ is a substitution for the variables in G_0 such that $D \models \sigma_0(\Phi_0)$, we have to show that $D \models \sigma_0(h(\Phi'_0))$. Now, if $D \models \sigma_0(\exists L_{G_1}(\Phi_1 \wedge eq(h_1)) \vee \exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$, then $D \models \sigma_0(\exists L_{G_1}(\Phi_1 \wedge eq(h_1)))$ or $D \models \sigma_0(\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$. Without loss of generality, let us assume that $D \models \sigma_0(\exists L_{G_1}(\Phi_1 \wedge eq(h_1)))$ this means that there is a substitution $\sigma_1 : L_{G_1} \rightarrow D$ such that $D \models \sigma_1(\sigma_0(\Phi_1 \wedge eq(h_1)))$, i.e. $D \models \sigma_1(\Phi_1) \wedge \sigma_1(\sigma_0(eq(h_1)))$. Now, since $h'_1 : \langle G'_0, \Phi'_0 \rangle \rightarrow \langle G_1, \Phi_1 \rangle$ is a morphism and $D \models \sigma_1(\Phi_1)$ we have that $D \models \sigma_1(h'_1(\Phi'_0))$. But $h_1 \circ h = h'_1$ implies that $D \models \sigma_1((h_1 \circ h)(\Phi'_0))$ or, equivalently, $D \models \sigma'_1(h(\Phi'_0))$, where $\sigma'_1 = \sigma_1 \circ h_1$. On the other hand, since $D \models \sigma_1(\sigma_0(eq(h_1)))$, we have that for every variable $x_0 \in L_{G_0}$ we have $\sigma_1(\sigma_0(x_0)) = \sigma_1(\sigma_0(h_1(x_0)))$. Moreover, since σ_0 and σ_1 are defined over disjoint sets of variables, $\sigma_1(\sigma_0(x_0)) = \sigma_0(\sigma_1(h_1(x_0)))$ and, since x_0 is not in L_{G_1}, $h_1(x_0)$ is not in L_{G_0} and $\sigma'_1 = \sigma_1 \circ h_1$, we have $\sigma_0(x_0) = \sigma'_1(x_0)$. Therefore, $D \models \sigma_0(h(\Phi'_0))$.

Finally, if $h' : \langle G'_0, \Phi'_0 \rangle \rightarrow \langle G_0, \Phi_0 \rangle$ satisfying that $h_1 \circ h = h'_1$ and $h_2 \circ h = h'_2$ then, by the uniqueness of h in **LabGraphs**, we have that $h = h'$. □

Proof of Proposition 1. Direct consequence of Propositions 6 and 7 □

Before proving Proposition 2, let us first prove an auxiliary property that we use in some further proofs:

Proposition 8 *If $h_1 : G_1 \rightarrow G_2$ where L_{G_1} and L_{G_2} are disjoint, then $D \models \exists L_{G_2}(\Phi_2 \wedge eq(h_2)) \Rightarrow \Phi_1$*

Proof. Let us suppose that $\sigma_1 : L_{G_1} \rightarrow D$ is a substitution for the variables in G_1 such that $D \models \sigma_1(\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$, then we have to show that $D \models \sigma_1(\Phi_1)$.

If $D \models \sigma_1(\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$ then $D \models \exists L_{G_2}(\Phi_2 \wedge \sigma_1(eq(h_2)))$, since Φ_2 includes no variables from L_{G_1}. But this is equivalent to $D \models \sigma_2(\exists(L_{G_2} \setminus L'_2)(\Phi_2))$, where $L'_2 = \{h_2(x_1) \mid x_1 \in L_{G_1}\}$ and $\sigma_2 : L'_2 \rightarrow D$ is defined as follows : for every $x_1 \in L_{G_1}$: $\sigma_2(h_2(x_1)) = \sigma_1(x_1)$, i.e. $\sigma_2 \circ h_2 = \sigma_1$. It must be noticed that the definition of σ_2 is correct, even if h_2 is not necessarily injective. The reason is that, if we have that $h_2(x_1) = x_2 = h_2(x'_1)$, for $x_1, x'_1 \in L_{G_1}$ and $x_2 \in L_{G_2}$

then $x_2 = x_1$ and $x_2 = x'_1$ are part of $eq(h_2)$. But we are assuming that $D \models \sigma_1(eq(h_2)))$ implying $\sigma_1(x_1) = \sigma_1(x'_1)$.

Now, if $D \models \sigma_2(\exists(L_{G_2} \setminus L'_2)(\Phi_2))$, this means that there exists a substitution $\sigma'_2 : L_{G_2} \setminus L'_2 \to D$ such that $D \models \sigma_2(\sigma'_2(\Phi_2))$. We know that g_2 is an M-morphism, which means that $D \models \Phi_2 \Leftrightarrow \Phi_3$, and we also know that $D \models \Phi_3 \Rightarrow h_2(\Phi_1)$ since g_1 is a morphism and $(h_2)_L = (g_1)_L$. Therefore, we have that $D \models \sigma_2(\sigma'_2(h_2(\Phi_1)))$, which is equivalent to $D \models \sigma_2(h_2(\Phi_1))$, because in $h_2(\Phi_1)$ there are no variables in $L_{G_2} \setminus L'_2$. But this implies $D \models \sigma_1(\Phi_1)$ since, by definition, $\sigma_2 \circ h_2 = \sigma_1$. $\square$

Proof of Proposition 2. If the diagram below is a pushout,

$$\begin{tikzcd}
\langle G_0, \Phi_0 \rangle \arrow[r, "h_1"] \arrow[d, "h_2"'] & \langle G_1, \Phi_1 \rangle \arrow[d, "g_1"] \\
\langle G_2, \Phi_2 \rangle \arrow[r, "g_2"'] & \langle G_3, \Phi_3 \rangle
\end{tikzcd}$$

then we have to prove that $D \models g_2^\#(\Phi_2) \Leftrightarrow (g_1^\#(\Phi_1) \wedge g_2^\#(\Phi_2))$. Trivially, $D \models g_2^\#(\Phi_2) \Leftarrow (g_1^\#(\Phi_1) \wedge g_2^\#(\Phi_2))$, therefore we only have to prove that $D \models g_2^\#(\Phi_2) \Rightarrow (g_1^\#(\Phi_1) \wedge g_2^\#(\Phi_2))$, or equivalently that $D \models g_2^\#(\Phi_2) \Rightarrow g_1^\#(\Phi_1)$ and $D \models g_2^\#(\Phi_2) \Rightarrow g_2^\#(\Phi_2)$. The second implication is again trivial, so let us prove the first one. Since h_2 is a morphism, we know that $D \models \Phi_2 \Rightarrow h_2^\#(\Phi_0)$. This means that $D \models g_2^\#(\Phi_2) \Rightarrow g_2^\#(h_2^\#(\Phi_0))$. On the other hand, since h_1 is an M-morphism, we have $D \models h_1^\#(\Phi_0) \Leftrightarrow \Phi_1$ and, thus, $D \models g_1^\#(h_1^\#(\Phi_0)) \Leftrightarrow g_1^\#(\Phi_1)$. But, since the diagram commutes, $g_1^\#(h_1^\#(\Phi_0)) = g_2^\#(h_2^\#(\Phi_0))$ implying $D \models g_2^\#(\Phi_2) \Rightarrow g_1^\#(\Phi_1)$.

Let us suppose that the diagram above is a pullback and, without loss of generality, that the variables in L_{G_1} are disjoint from the variables in L_{G_2}. First we must notice that (for an adequate choice of L_{G_0} and h_1) we have that $L_{G_0} = L_{G_1}$ and $(h_1)_L$ is the identity, since pullbacks in $\mathbf{E}-\mathbf{Graphs}$ preserve monos. Therefore we have to show that $D \models \Phi_1 \Leftrightarrow \Phi_1 \vee (\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$, since $\exists L'_{G_1}(\Phi_1 \wedge \bigwedge_{x_1 \in L_{G_1}} x_1 = x'_1)$ is logically equivalent to Φ_1, where L'_{G_1} is a renaming apart of the variables in L_{G_1} and x'_1 is the corresponding renaming of x_1. Now, the implication $D \models \Phi_1 \Rightarrow \Phi_1 \vee (\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$ is trivial, therefore we just have to prove $D \models \Phi_1 \Leftarrow \Phi_1 \vee (\exists L_{G_2}(\Phi_2 \wedge eq(h_2)))$. Obviously, $D \models \Phi_1 \Leftarrow \Phi_1$, therefore we have to show $D \models \Phi_1 \Leftarrow \exists L_{G_2}(\Phi_2 \wedge eq(h_2))$. But this is shown in Proposition 8.

$\square$

Before proving Theorem 2, let us characterize van Kampen squares in $\mathbf{SymbGraphs_D}$.

Proposition 9 *Let us assume that h_1 in the diagram below is an M-morphism, then this diagram is a van Kampen square:*

$$\begin{tikzcd}
 & & \langle G'_0, \Phi'_0 \rangle \arrow[dll, "h'_1"'] \arrow[dr, "h'_2"] \arrow[dd, "f_0"] & \\
\langle G'_1, \Phi'_1 \rangle \arrow[dr, "g'_1"] \arrow[dd, "f_1"'] & & & \langle G'_2, \Phi'_2 \rangle \arrow[dll, "g'_2"'] \arrow[dd, "f_2"] \\
 & \langle G'_3, \Phi'_3 \rangle \arrow[dd, "f_3"] & \langle G_0, \Phi_0 \rangle \arrow[dll, "h_1"'] \arrow[dr, "h_2"] & \\
\langle G_1, \Phi_1 \rangle \arrow[dr, "g_1"'] & & & \langle G_2, \Phi_2 \rangle \arrow[dll, "g_2"] \\
 & \langle G_3, \Phi_3 \rangle & &
\end{tikzcd}$$

if and only if the diagram below is also a van Kampen square.

$$\begin{tikzcd}
 & & G'_0 \arrow[dll, "h'_1"'] \arrow[dr, "h'_2"] \arrow[dd, "f_0"] & \\
G'_1 \arrow[dr, "g'_1"] \arrow[dd, "f_1"'] & & & G'_2 \arrow[dll, "g'_2"'] \arrow[dd, "f_2"] \\
 & G'_3 \arrow[dd, "f_3"] & G_0 \arrow[dll, "h_1"'] \arrow[dr, "h_2"] & \\
G_1 \arrow[dr, "g_1"'] & & & G_2 \arrow[dll, "g_2"] \\
 & G_3 & &
\end{tikzcd}$$

Proof. If the first diagram is a van Kampen square then the second diagram is also a van Kampen square, as a consequence of Propositions 6 and 7.

Conversely, suppose that the diagram in $\mathbf{E-Graphs}$ is a van Kampen square, then we have two prove that if, in the diagram in $\mathbf{SymbGraphs_D}$, the square at the bottom is a pushout and the backfaces are pullbacks then the square at the top is a pushout if and only if the front faces are pullbacks. So, let us assume that the square at the top is a pushout.

By Proposition 5, we know that g_2, h'_1 and g'_2 are M-morphisms, therefore we may assume without loss of generality that $L_{G_0} = L_{G_1}$, $L_{G'_0} = L_{G'_1}$, $L_{G_2} = L_{G_3}$, and $L_{G'_2} = L_{G'_3}$ and that $(h_1)_L$, $(g_2)_L$, $(h'_1)_L$, and $(g'_2)_L$ are the identity. Moreover, to avoid name conflicts, let us assume that, otherwise, the sets of variables used as labels in the graphs are pairwise disjoint (if necessary, they may be renamed apart). In addition, we also know that $D \models \Phi_0 \Leftrightarrow \Phi_1$, $D \models \Phi_2 \Leftrightarrow \Phi_3$, $D \models \Phi'_0 \Leftrightarrow \Phi'_1$, and $D \models \Phi'_2 \Leftrightarrow \Phi'_3$. Now we have to prove that the front faces of the first diagram are pullbacks and, according to Proposition 7, this is equivalent to showing that:

a) $D \models \Phi'_1 \Leftrightarrow \big(\exists L_{G_1}(\Phi_1 \wedge eq(f_1)) \vee \exists L_{G'_3}(\Phi'_3 \wedge eq(g'_1))\big)$ and

b) $D \models \Phi'_2 \Leftrightarrow \big(\exists L_{G_2}(\Phi_2 \wedge eq(f_2)) \vee \exists L_{G'_3}(\Phi'_3 \wedge eq(g'_2))\big)$

a) We know that $D \models \Phi'_1$ is equivalent to $D \models \Phi'_0$. Moreover, since the back squares are pullbacks, we have $D \models \Phi'_0 \Leftrightarrow \big(\exists L_{G_0}(\Phi_1 \wedge eq(f_0)) \vee \exists L_{G'_2}(\Phi'_3 \wedge eq(h'_2))\big)$. But we are assuming that h_1 and g'_2 are M-morphisms and this means that $\big(\exists L_{G_0}(\Phi_1 \wedge eq(f_0)) \vee \exists L_{G'_2}(\Phi'_3 \wedge eq(h'_2))\big)$ and $\big(\exists L_{G_1}(\Phi_1 \wedge eq(f_1)) \vee \exists L_{G'_3}(\Phi'_3 \wedge eq(g'_1))\big)$ are the same formula. Therefore, we have that $D \models \Phi'_1 \Leftrightarrow \big(\exists L_{G_1}(\Phi_1 \wedge eq(f_1)) \vee \exists L_{G'_3}(\Phi'_3 \wedge eq(g'_1))\big)$.
b) First of all, we may notice that $\exists L_{G'_3}(\Phi'_3 \wedge eq(g'_2)))$ is logically equivalent to Φ'_3, since g'_2 is an M-morphism. Moreover, for the same reason, we know that $D \models \Phi'_2 \Leftrightarrow \Phi'_3$. Therefore, we have to prove that $D \models \Phi'_2 \Leftrightarrow \big(\exists L_{G_2}(\Phi_2 \wedge eq(f_2)) \vee \Phi'_2\big)$. But this is equivalent to prove $D \models \Phi'_2 \Leftarrow \exists L_{G_2}(\Phi_2 \wedge eq(f_2))$, and this is proved in Proposition 8.

Now, suppose that the front faces of the above diagram in $\mathbf{SymbGraphs_D}$ are pullbacks, let us prove that the square at the top is a pushout. This means proving $D \models \Phi'_3 \Leftrightarrow (g'^{\#}_1(\Phi'_1) \wedge g'^{\#}_2(\Phi'_2))$. Again, by By Proposition 5, we know that g_2, h'_1 and g'_2 are M-morphisms, therefore we know that $D \models \Phi_0 \Leftrightarrow \Phi_1$, $D \models \Phi_2 \Leftrightarrow \Phi_3$, $D \models \Phi'_0 \Leftrightarrow \Phi'_1$, and $D \models \Phi'_2 \Leftrightarrow \Phi'_3$, and we may assume without loss of generality that $L_{G_0} = L_{G_1}$, $L_{G'_0} = L_{G'_1}$, $L_{G_2} = L_{G_3}$, and $L_{G'_2} = L_{G'_3}$ and that $(h_1)_L$, $(g_2)_L$, $(h'_1)_L$, and $(g'_2)_L$ are the identity. Also, as above, we assume that, otherwise, the sets of variables used as labels in the graphs are pairwise disjoint.

Hence, we have to prove that $D \models \Phi'_3 \Leftrightarrow (g'^{\#}_1(\Phi'_1) \wedge \Phi'_3)$, and this means proving $D \models \Phi'_3 \Rightarrow g'^{\#}_1(\Phi'_1)$, but this is a consequence of the fact that g'_1 is a morphism in $\mathbf{SymbGraphs_D}$.

□

Proof of Theorem 2. Direct consequence of Propositions 5, 1, 2, and 9.

□

Proof of Proposition 3.

1. If $SG = \langle G, \Phi \rangle$ is grounded, by definition we know that Sem(SG) is not empty, since Φ is satisfiable in D. Moreover, If $AG1, AG2 \in Sem(\langle G, \Phi \rangle)$ then this means that there are substitutions $\sigma, \sigma' : L_G \to D$ such that $D \models \sigma(\Phi)$ and $D \models \sigma'(\Phi)$. But if $\langle G, \Phi \rangle$ is grounded this means that $L_G = \{x_v \mid v \in D\}$ and for each $v \in D$: $\sigma(x_v) = v$ and $\sigma'(x_v) = v$. But this implies that $\sigma = \sigma'$ and therefore $AG_1 = AG_2$.

2. Let $SG = \langle G, \Phi \rangle$ be a grounded graph and let $SG' = \langle G', \Phi' \rangle$ be isomorphic to SG. This means that there is an E-graph isomorphism $h : G \to G'$ such that $D \models \Phi \Leftrightarrow h(\Phi')$. But this implies that $h_L : L_G \to L_{G'}$ is a bijection and if $\sigma' : L_{G'} \to D$ is a substitution such that $D \models \sigma'(\Phi')$ then $\sigma' \circ h : L_G \to D$ is a substitution such that $D \models \sigma' \circ h(\Phi)$ impliying that for every $v \in D$ $\sigma' \circ h(x_v) = v$. Therefore, if for every $v \in D$ we call y_v the variable $h(x_v)$ then we have that, for each $v \in D$, $\sigma'(y_v) = v$, which means that SG' is grounded.

3. Let $AG = \langle G, D \rangle$ be an attributed graph and let $\mathscr{X}_D$ be a set of variables consisting of a variable x_v for each $v \in D$. We define $GSG(AG) = \langle G', \Phi_{AG}$ as follows:

   - $G' = f^*(G)$, where $f : D \to \mathscr{X}_D$ is a substitution defined for every $v \in D$ as $f(v) = x_v$.
   - $\Phi_{AG} = \{x_v = v \mid v \in D\}$.

Now, it should be obvious that, by construction, $GSG(AG)$ is grounded and, moreover, $AG \in Sem(GSG(AG))$.

Now, suppose that $SG_0 = \langle G_0, \Phi_0 \rangle$ is a symbolic graph such that $AG = \langle G, D \rangle \in Sem(SG)$. Let us prove that SG and $GSG(AG) = \langle G', \Phi_{AG} \rangle$ are isomorphic. First of all, we know that $G = \sigma_0^*(G_0)$ for a substitution σ_0 such that $D \models \sigma_0(\Phi_0)$. But, since SG_0 is grounded, σ_0^* is an isomorphism. For similar reasons, we know that $f^* : G \to G'$ is also an isomorphism therefore $f^* \circ \sigma_0^* : G_0 \to G'$ is an E-graph isomorphism. Finally, it is easy to see that $D \models \Phi \Leftrightarrow f \circ \sigma_0(\Phi_0)$. In particular, if σ is a substitution such that $D \models \sigma(\Phi)$ we have to prove that $D \models \sigma \circ f \circ \sigma_0(\Phi_0)$ or, equivalently, that $\sigma \circ f \circ \sigma_0 = \sigma_0$. But this is obvious since, on one hand, by construction, $v \in D$: $f(v) = x_v$, and, on the other we know that for every $v \in D$: $\sigma(x_v) = v$, which means that $f = \sigma^{-1}$. Conversely, if σ is a substitution such that $D \models \sigma \circ f \circ \sigma_0(\Phi_0)$ we can prove similarly that this implies that $D \models \sigma(\Phi)$.

□

Proof of Proposition 4. First, we will show that GSG can be extended to a functor and, then, that GSG is full, faithful and essentially surjective. Let $f : \langle G_1, D \rangle \to \langle G_2, D \rangle$ be an attributed graph morphism. Since D is finitely generated, f_{alg} is the identity and f_{graph} is an E-graph morphism. Hence, if $f : D \to \mathscr{X}_D$ is a substitution defined for every $v \in D$ as $f(v) = x_v$, and $\Phi = \{x_v = v \mid v \in D\}$ we know that $GSG(\langle G_1, D \rangle) = \langle f(G_1), \Phi \rangle$ to $GSG(\langle G_2, D \rangle) = \langle f(G_2), \Phi \rangle$. We define $GSG(f)$ as follows:

- For every $x \in \{V_G, E_G, E_{NL}, E_{EL}\}$: $GSG(f)_x = f_x$.
- $GSG(f)_L$ is the identity.

Then, it is routine to prove that $GSG(f)$ is indeed a symbolic graph morphism.

To prove that GSG is full, we have to show that if $AG_1 = \langle G_1, D \rangle$ and $AG_2 = \langle G_2, D \rangle$ are two attributed graphs and $h : GSG(AG_1) \to GSG(AG_2)$ is a symbolic graph morphism then there exists an attributed graph morphism $f : AG_1 \to AG_2$ such that $GSG(f) = h$. But it is enough to define f as follows:

- For every $x \in \{V_G, E_G, E_{NL}, E_{EL}\}$: $f_x = h_x$.
- f_{alg} (and, therefore, f_L is the identity.

To prove that GSG is faithful we have to show that GSG is injective on morphisms, but this is straightforward by construction. Finally, to prove that GSG is essentially surjective, we have to show that for every grounded graph SG there is another grounded graph SG', which isomorphic to SG, and an attributed graph AG such that $GSG(AG) = SG'$. But by Prop 3 we know that $Sem(SG)$ is not empty and that if $AG \in Sem(SG)$ satisfies that $GSG(AG) = SG$. □

Proof of Theorem 3. Let us assume that $AL = \langle L, T_{SP}|_\Sigma \rangle, AK = \langle K, T_{SP}|_\Sigma \rangle, AR = \langle R, T_{SP}|_\Sigma \rangle)$, and $r' = \langle L' \hookleftarrow K' \hookrightarrow R', \Phi' \rangle$, and let us consider the following diagram in $\mathbf{E-Graphs}$:

$$\begin{tikzcd}
L' \arrow[d, "f_L^{*-1}"'] \arrow[dr, phantom, "(3)"] & K' \arrow[l, hook'] \arrow[r, hook] \arrow[d, "f_K^{*-1}"'] \arrow[dr, phantom, "(4)"] & R' \arrow[d, "f_R^{*-1}"] \\
L \arrow[d, "m"'] \arrow[dr, phantom, "(5)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(6)"] & R \arrow[d] \\
G \arrow[d, "g_L^*"'] \arrow[dr, phantom, "(7)"] & I \arrow[l, hook', "h_1"] \arrow[r, hook, "h_2"'] \arrow[d, "g_K^*"'] \arrow[dr, phantom, "(8)"] & H \arrow[d, "g_R^*"] \\
G' & I' \arrow[l, hook', "h'_1"] \arrow[r, hook, "h'_2"'] & H'
\end{tikzcd}$$

where (5) and (6) are the pushouts defining the application of r to AG with match m, $\langle G',\Psi\rangle = GSG(AG), \langle I',\Psi\rangle = GSG(\langle I, T_{SP}|_\Sigma\rangle)$, $\langle H',\Psi\rangle = GSG(\langle H, T_{SP}|_\Sigma\rangle)$, g_L^*, g_K^*, g_R^* are, respectively, the isomorphisms relating G, I, H with G', I', H', and finally the morphisms h'_i, for $i = 1,2$ are defined as follows. For every element $e \in I'$ which is not a label, $h'_i(e) = h_i(e)$, and for every label e, $h'_i(e) = e$. It is routine to see that, by definition of GSG and as a consequence of the fact that h_1 and h_2 are M-morphisms, h'_1 and h'_2 are morphisms and, moreover, diagrams (7) and (8) not only commute but are pushouts. Therefore, since we know that diagrams (3) and (4) are pushouts, then diagrams (3)+(5)+(7) and (4)+(6)+(8) are also pushouts. Therefore, if we define $m' = g_L^* \circ m \circ f_L^{*-1}$ and we show that m' is a morphism in $\mathbf{SymbGraphs_D}$ the first part of the theorem will be proved. Therefore, we have to prove that $D \models \Psi \Rightarrow m'(\Phi')$.

We now that

$$\Phi' = \Phi \cup \{y_a = t \mid y_a \in Y \wedge ch(a) = t\} \text{ and } \Psi = \{x_v = v \mid v \in D\}$$

We also know that the only substitution σ such that $D \models \sigma(\Psi)$ is defined $\forall v \in D : \sigma(x_v) = v$. Therefore, we have to show that $D \models \sigma(m'(\Phi'))$ or, equivalently, $D \models \sigma(m'(\Phi))$ and $D \models \sigma(m'(\{y_e = t \mid y_e \in Y \wedge ch(e) = t\}))$. Finally, by definition, on the one hand, we have that for every $a \in T_{SP}|_\Sigma$ we have $m'(y_a) = x_v$, where $m(a) = v$, which means that $\sigma(m'(y_a)) = m(a)$, and on the other, for each $x \in X$, $\sigma(m'(x)) = m(x)$.

Now, let $t_1 = t_2$ be an equation in Φ. Since m_alg is a Σ-homomorphism and T_{SP} satisfies this equation, we have that $D \models m(t_1) = m(t_2)$, implying $D \models \sigma(m'(t_1)) = \sigma(m'(t_2))$.

Let $y_a = t$ and $ch(a) = t$. Then, on the one hand, we have that $\sigma(m'(y_a)) = m(a)$ and, on the other, since $ch(a) = t$ we have that in $a = |t|$, implying $\sigma(m'(t)) = m(a)$. Therefore, $\sigma(m'(y_a)) = \sigma(m'(t))$.

The proof of the second part of the theorem is similar to the proof of the first part. We only have to consider the diagram below:

$$\begin{tikzcd}
L \arrow[d, "f_L^*"'] \arrow[dr, phantom, "(1)"] & K \arrow[l, hook'] \arrow[r, hook] \arrow[d, "f_K^*"'] \arrow[dr, phantom, "(2)"] & R \arrow[d, "f_R^*"] \\
L' \arrow[d, "m'"'] \arrow[dr, phantom, "(9)"] & K' \arrow[l, hook'] \arrow[r, hook] \arrow[d] \arrow[dr, phantom, "(10)"] & R' \arrow[d] \\
G' \arrow[d, "g_L^{*-1}"'] \arrow[dr, phantom, "(11)"] & I' \arrow[l, hook', "h'_1"] \arrow[r, hook, "h'_2"'] \arrow[d, "g_K^{*-1}"'] \arrow[dr, phantom, "(12)"] & H' \arrow[d, "g_R^{*-1}"] \\
G & I \arrow[l, hook', "h_1"] \arrow[r, hook, "h_2"'] & H
\end{tikzcd}$$

□

Graph Modelling and Transformation: Theory meets Practice

Karsten Ehrig[1] **and Claudia Ermel**[2]

[1] Federal Institute for Materials Research and Testing
(Bundesanstalt für Materialprüfung), Berlin, Germany
karsten.ehrig@bam.de

[2] Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
claudia.ermel@tu-berlin.de

Abstract: In this paper, we focus on the role of graphs and graph transformation for four practical application areas from software system development. We present the typical problems in these areas and investigate how the respective systems are modelled by graphs and graph transformation. In particular, we are interested in the usefulness of theoretical graph transformation results and graph transformation tools in order to solve the typical problems. Finally, we characterize concepts and tool features which are still missing in practice to solve the presented and related problems even better.

Keywords: graph modelling, graph transformation, graph transformation tools

1 Introduction

Graphs are one of the key concepts for modelling. Since the early days of mankind, graphs are used to depict the relationship between two or more entities as abstractions of real world systems and processes. The visual nature of graphs makes them an intuitive language for human beings to think and discuss about partitioning systems into different components, and about processes of running systems which can be drawn as related but changing system state graphs. Throughout the history of software engineering, graph models have been used for software system design, such as entity-relationship diagrams for databases, class diagrams for static software structure, and the diagram types offered by the Unified Modeling Language (UML) [OMG07] to model different static and dynamic system aspects. Yet, when it came to programming, often enough a yawning gap opened between what the modellers meant when designing their graph models and what the programmers encoded using standard textual programming languages, where the graph models played the role of a rough guideline for programmers. Ambitious programming projects resulted in failure, went over their budgets or proved to be unstable over time.

"Everything is a Model" [Béz05]: It is common knowledge today that the need for a high quality software production has to be faced from a different perspective: model-driven development (MDD). The objective of MDD is generating code from a higher-level visual system model. This means that, for software developers the abstraction level is now raised. No longer do they need to worry about technical details and features of programming languages but can concentrate on

more creative parts of software engineering: analysis, design and validation, all based on models. The MDD perspective raises the importance of graph models and calls for rigorous methods to capture the semantics of models and their evolution over time.

"Mathematics are useful to solve real world problems" [BGL09]: In computer science, using mathematics means using formal methods, referring to the use of a formal notation to represent system models during software development. Typically, first a specification is written in a formal notation, then refined step by step into code. Correctness of the refinement steps guarantees that the code satisfies the specification. In some methods, developers can check correctness of the refinement steps using a theorem prover for the method's underlying formal logic, but other methods remain manual because it is difficult to automate the notation used. Experience shows that many problems in using formal methods in software development arise because the formal notation and the problem domain are too far apart. Since any software system is situated in a particular social context, this context (domain) should be represented also in models based on formal notations. Here, again, graph models with their visual nature are a good candidate for uniting the domain-specific and the formal aspects of real-world problems. Domain specific languages based on graphs use a graphical concrete syntax with adequate intuitive symbols which are manipulated to model dynamic system aspects. Thus, the system behaviour can be animated in a domain-specific visualization to validate system properties by domain experts. Furthermore, the fundamental notions behind graph models have been captured long ago by mathematical terms, thus allowing for rigorous reasoning at model level with purely abstract objects and structures.

"Graph changes everywhere" [Eng00]: Real world systems evolve, and hence, formal graph system models need to model evolution as well. Algebraic graph transformation is a formally defined calculus based on graphs and graph transformation rules [EEPT06]. For ages, rules have proven to be extremely useful for describing computations by local transformations. Areas like language definition, logic, functional programming, algebraic specification, term rewriting and expert systems have rules as key concepts. Graph transformation, also known as graph rewriting or graph reduction, combines the potential and advantages of both graphs and rules into a single computational paradigm. For nearly 40 years, graph transformation has been studied in a variety of approaches, motivated by application domains such as pattern recognition, semantics of programming and visual modelling languages, specification of distributed systems etc. [EEKR99, EKMR99, BTMS99].

A detailed presentation of different graph transformation approaches, is given in volume 1 of the *Handbook of Graph Grammars and Computing by Graph Transformation* [Roz97]. The *algebraic approach* is based on pushout constructions, where pushouts are used to model the gluing of graphs. In fact, there are two main variants of the algebraic approach, the double and the single pushout approach. The double pushout (DPO) approach [EEPT06], is the formal basis for visual modelling of behavioural models and model transformations considered in this article. The DPO approach is based on category theory: a graph transformation rule is a pair of morphisms in the category of graphs with total graph morphisms as arrows: $r = (L \leftarrow K \rightarrow R)$ where $K \rightarrow L$ is injective. Graph K is called *gluing graph*. Another graph morphism $m\colon L \rightarrow G$ models an occurrence of L in G and is called a *match*. Practical understanding of this is that L is a subgraph that is matched to G, and after a match is found, the rule can be applied.

A *direct transformation* or application of rule r to graph G is defined by two pushout diagrams (see the diagram to the right). Applying the rule, $m(L)$ is replaced with $m^*(R)$ in graph G, leading to the transformed graph H. A *graph transformation*, or, more precisely, a graph transformation sequence, consists of zero or more direct transformations, written $G_0 \overset{*}{\Longrightarrow} G_n$. A set of graph rules is called *graph transformation system*. A *type graph* defines a set of types which can be used to assign a type to the nodes and edges of a graph. The typing itself is done by a graph morphism from the graph to the type graph. A *typed graph transformation system* $GTS = (TG,P)$ consists of a type graph TG and a set P of typed graph rules. A *(typed) graph grammar* $GG = (GTS,S)$ consists of a (typed) graph transformation system GTS and a (typed) start graph S. The *(typed) graph language* L of GG is defined by $L = \{G \mid \exists$ (typed) graph transformation $S \overset{*}{\Longrightarrow} G\}$. The key idea of *attributed* graph transformation is to model graphs with node and edge attributes, i.e. an attributed graph is a pair $AG = (G,A)$ of a graph G and a data type algebra A. *Typed attributed* graph transformation, combining process and data modelling proved to be well-suited to define and analyse visual models and model transformations [EEPT06, MVVK05].

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

A variety of tools for graph transformation exist [TEG$^+$05] to be used as transformation engine and for analysis purposes, to reason about issues such as conflicts and dependencies of actions as well as consistency of object structures.

In this paper we summarize a few selected case studies from recent literature which have been modelled by graphs and graph transformation. In particular, we focus on four case studies from different application areas: a medical information system (Section 2), a model transformation between two different modelling notations (Section 3), a metabolic pathway analysis (Section 4), and a self-healing system (Section 5). For each application area, we ask the following questions:

1. What are typical problems in these areas?

2. How can they be modelled by graphs or graph transformation?

3. What kind of graph transformation results can be applied to solve these problems?

4. What are missing graph modelling and transformation concepts and results?

In the evaluation (Section 6), we summarize the experiences gained from the case studies and state what kinds of concepts and results we find still missing.

2 Case Study 1: Medical Information System

Problem

Information systems are very common nowadays in almost all common application areas of software systems. In health care, data from different domains like admission, physical examination, medical record archive, etc. have to be coordinated and presented to the employees. Data manipulations, like the admission of a new patient, have to be supported intuitively.

Aim of the Model

An interactive visual application with a suitable graphical user interface shall be modelled. Instead of complex textual data, visual symbols shall be used to support the necessary information system operations. Yet, the operations shall be modelled in a precise, unambiguous way.

Technique to solve the problem / realize the aim

We use typed, attributed graphs to model the abstract syntax of the information systems, and graph transformation rules to model the operations to be performed by the clinical staff. Moreover, we combine the abstract syntax elements with concrete syntax symbols to visualize graphs in an adequate, domain-specific way. Constraints and application conditions are used to check the consistency of the model and the operations to be performed.

Overview of the model

Figure 1 shows icons for patients, beds, rooms, admission and discharge (from left to right) used in our information system. In Figure 2, the current ward patient allocation diagram shows bed icons inside the room icons to represent the number of available beds in the ward rooms. A patient icon is connected with a bed if occupied, otherwise the bed is left empty. Patients currently not associated with a bed are shown next to the admission or discharge symbols. This requires a user action. Dragging the female icon onto the Admission symbol evokes rule *Admission* (Figure 3). Applying this rule, the user can move the female patient figure to her corresponding bed, unless there are male patients in the same room (modelled by a NAC). With a graphical rule editor, the information system designer may define new rules and user policies according to the needs and standards of the hospital.

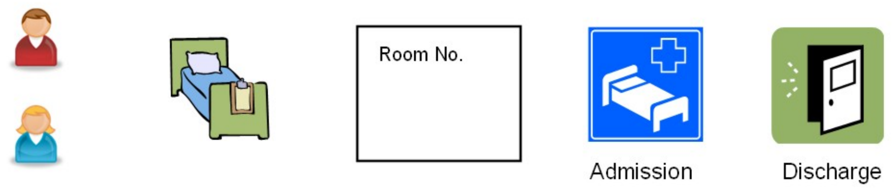


Figure 1: Graphical Symbols for Medical Information System

Tool Support

Tiger2 [BEEH09] is an EMF-based generator of modeling tool environments for visual domain specific languages. In the modelling environment, a set of EMF transformation rules called editing rules define the editing commands of the generated visual editor, i.e. the model syntax; on the other hand, a set of simulation rules describe a model's operational semantics.

Figure 4 shows the abstract syntax of the case study modelled in Tiger2. A *patient* is associated with a *bed* located in a *room* of the ward numbered with the attribute *number* of data type *String* to allow for combinations of letters and numbers (e.g. 'room A15'). Node *patient* is an abstract node, specialized to nodes *female_patient* and *male_patient*. The *patient* attribute *health_record_id* of is used for unique identification of the current health record in the system

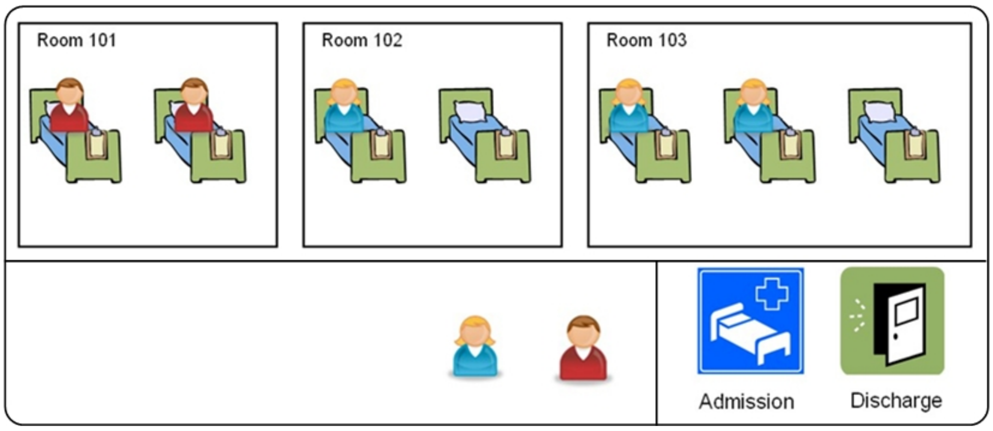


Figure 2: Sample User Interface Diagram for Medical Information System

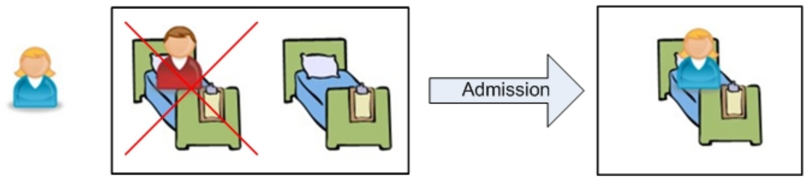
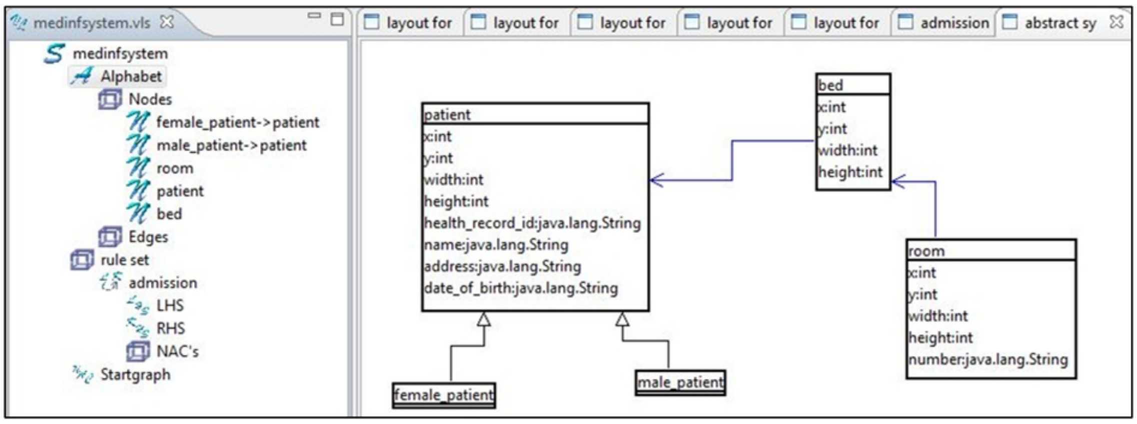


Figure 3: Sample Rule *Admission*



medinfsystem.vls
layout for | layout for | layout for | layout for | layout for | admission | abstract sy

- medinfsystem
  - Alphabet
    - Nodes
      - female_patient->patient
      - male_patient->patient
      - room
      - patient
      - bed
    - Edges
  - rule set
    - admission
      - LHS
      - RHS
      - NAC's
  - Startgraph

```
patient
x:int
y:int
width:int
height:int
health_record_id:java.lang.String
name:java.lang.String
address:java.lang.String
date_of_birth:java.lang.String

bed
x:int
y:int
width:int
height:int

room
x:int
y:int
width:int
height:int
number:java.lang.String

female_patient
male_patient
```

Figure 4: Abstract Syntax in Tiger2

database. One patient may acquire more than one health record ids for different admissions. The attributes *x*, *y*, *width*, and *height* are used for icon visualization.

Related Work

While common domain specific languages like the Ecore diagram editor as part of the Graphical Modeling Framework (GMF) [GMF07] and class and activity diagram editors as part of

UML2Tools [Fou09] are already part of the common Eclipse plug-in packages, more sophisticated domain specific language editors are rarely to find.

Starting with an EMF domain model GMF provides a code generation facility for a graphical editor with basic editor operations for inserting graphical objects and links between them. Apart from GMF [GMF07], also the TOPCASED modeler generator of the OPENEMBEDD [Ope09] MDE platform provides graphical patterns for common parts of user specific EMF domain models and thus allows to easily create a basic graphical editor.

Unsolved Problems

Multi-view graphical editors need a more flexible and user friendly way to define domain specific editor environments including editor operations, simulation and model transformation tools. These requirements can be more adequately fulfilled by graph transformation tools [Tae06] providing a graphical way to define complex operations for editing, simulation, and model transformation of domain specific languages based on a well-defined theoretical background [EEPT06]. Up to now, a comprehensive generation framework combining graph transformation and EMF-based meta-modeling for visual environment generation has not yet been implemented.

3 Case Study 2: Business Process Model Transformation

Problem

The Business Process Modelling Notation (BPMN) [Whi04] is a graph-oriented language in which control and action nodes can be connected almost arbitrarily. It defines a Business Process Diagram (BPD), which is a kind of flowchart incorporating constructs tailored to business process modelling, such as AND-split, AND-join, XOR-split, XOR-join. It is supported by various modelling tools but so far no systems can directly execute BPMN models. The Business Process Execution Language for Web Services (BPEL) [IBM03], on the other hand, is a mainly block-structured language. BPEL is emerging as a de-facto standard for implementing business processes on top of web services technology. Numerous platforms support the execution of BPEL processes.

Aim of the Model

The aim of this case study is to define the BPMN2BPEL model transformation at an adequate abstraction level. A challenge in formalizing the particular model transformation is the translation of BPMN *And* and *Xor* constructs to the corresponding BPEL language elements *Flow* and *Switch*. Translating those constructs with ordinary graph transformation rules requires a complex control structure for guidance. We aim for an intuitive, visual description of the model transformation where arbitrary many branches of *And* and *Xor* constructs can be treated in parallel.

Technique to solve the problem / realize the aim

We use typed, attributed graph transformation based on an integrated type graph TG_I. This type graph consists of the type graphs for the source and target language, and, additionally, reference nodes with arcs mapping source elements to target elements. We express model transformations directly by TG_I-typed graph transformation rules $L \leftarrow K \rightarrow R$ where L basically represents source model elements, and R represents the corresponding generated target model elements. The model transformation starts with graph G_S typed over TG_S. As TG_S is a subgraph of TG_I, G_S is also typed over TG_I. During the model transformation process, the intermediate graphs $G_S = G_1,..,G_n$are all TG_I-typed. To delete all items in G_n which are not typed over TG_T we can construct a restriction (a pullback in the category **Graphs**), which deletes all those items in one step.

$$\begin{tikzcd}
TG_S \arrow[rr, hook, "inc_S"] & & TG_I & TG_T \arrow[l, hook', "inc_T"'] \\
G_S \arrow[u, "type_{G_S}"] \arrow[r, Rightarrow, "r_1"] & \cdots \arrow[r, Rightarrow, "r_n"] & G_n \arrow[u, "type_{G_n}"'] & G_T \arrow[l] \arrow[u, "type_{G_T}"']
\end{tikzcd}$$

In addition to normal graph transformation rules, we also use rule schemes to express parallel transformation of arbitrary many similar model element patterns. The application of rule schemes is defined by the concept of amalgamated graph transformation [BFH87].

Overview of the Model Transformation

The complete model transformation case study is described in [BE09]. The type graph integrating the BPMN source model (left-hand part), the reference part connecting source and target model (the node type F2ARef and its adjacent edge types bpmn and bpel), and the BPEL target model (right-hand part) is shown in Figure 5.

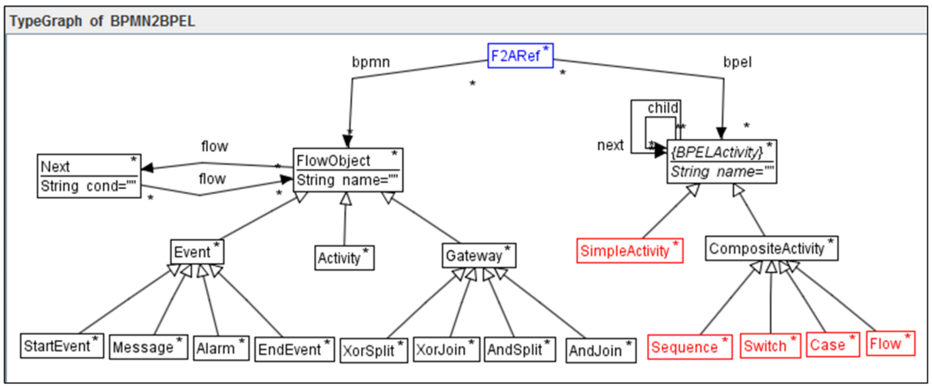


Figure 5: BPMN2BPEL type graph

As an example we consider a BPMN diagram which models a person's interaction with an ATM (see Figure 6 where the concrete and abstract syntax of the diagram are depicted). In the upper part, the ATM machine accepts and holds the card of the person while simultaneously contacting the bank for the account information. (The language elements AndSplit and AndJoin are used to model parallel actions.) Afterwards, the display prompts the user for the PIN. Depending on the user's input there are three alternative actions possible: (1) the user enters the correct PIN and can withdraw money, (2) a wrong PIN is entered – a message is displayed, (3) the operation is aborted – an alarm signal is given.

We give one example for a model transformation rule scheme (in abstract syntax) to translate And constructs. All other rules and rule schemes can be found in [BE09]. An And construct (a number of branches surrounded by an AndSplit and AndJoin element) is translated to a *Flow* container node which contains a child for each branch emerging from the AndSplit. Since the number of branches can be arbitrary, a normal graph transformation rule or any finite number of

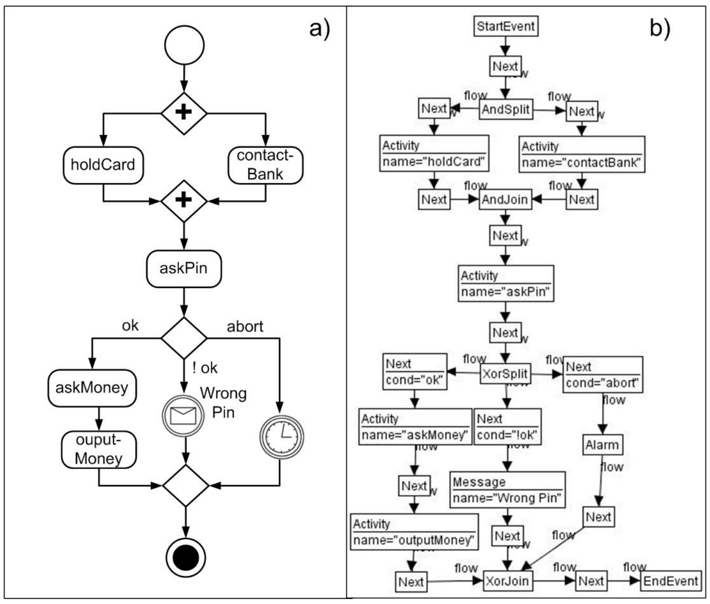


Figure 6: ATM machine in BPMN in concrete syntax (a) and abstract syntax (b)

rules would not be sufficient to express this situation. Therefore, we here use amalgamated graph transformation to express parallel execution. A common subaction is modelled by the application of a common *kernel rule* of all additional actions (modelled by *multi rules*). For example, in order to process an And construct, the common subaction is to process one branch only. Independent of the number of additional branches, this is the kernel action which will always happen. Hence, we model this action by the kernel rule in the upper part of Figure 7, where one branch surrounded by an AndSplit and AndJoin is translated to a BPEL Flow node with one child. Additional subactions now are modelled by multi rules. Each multi rule contains the kernel rule and specifies an additional action which is executed as many times as matches for this additional part can be found in the host graph. The multi rule for processing And constructs is shown in the bottom part of Figure 7. It extends the kernel rule by one more branch. Formally, the synchronization possibilities of kernel and multi rules are defined by an *interaction scheme* consisting of a kernel rule and a set of rules called multi rules such that the kernel rule is embedded in each of the multi rules. The subrule embedding morphism from the kernel rule to the multi rule is indicated in Figure 7 by corresponding numbers of some of the graph objects.

For applying an interaction scheme, first, a match of the kernel rule is selected. Then, multi rule instances are constructed, one for each new match of a multi rule in the current host graph that overlaps with the kernel match. At last, all multi rule instances are glued at their corresponding kernel rule objects which leads to a new rule, the *amalgamated rule*. The application of the amalgamated rule is called *amalgamated graph transformation*. For this case study, a theoretical result is applied which allows us to show parallel independence of amalgamated graph transformations by analyzing the underlying multi rules. Hence, we may translate the And construct and the Xor construct in arbitrary order. After applying our transformation rules and schemes starting with the ATM model in Figure 6, we get the resulting integrated graph shown in Figure 8 (b).

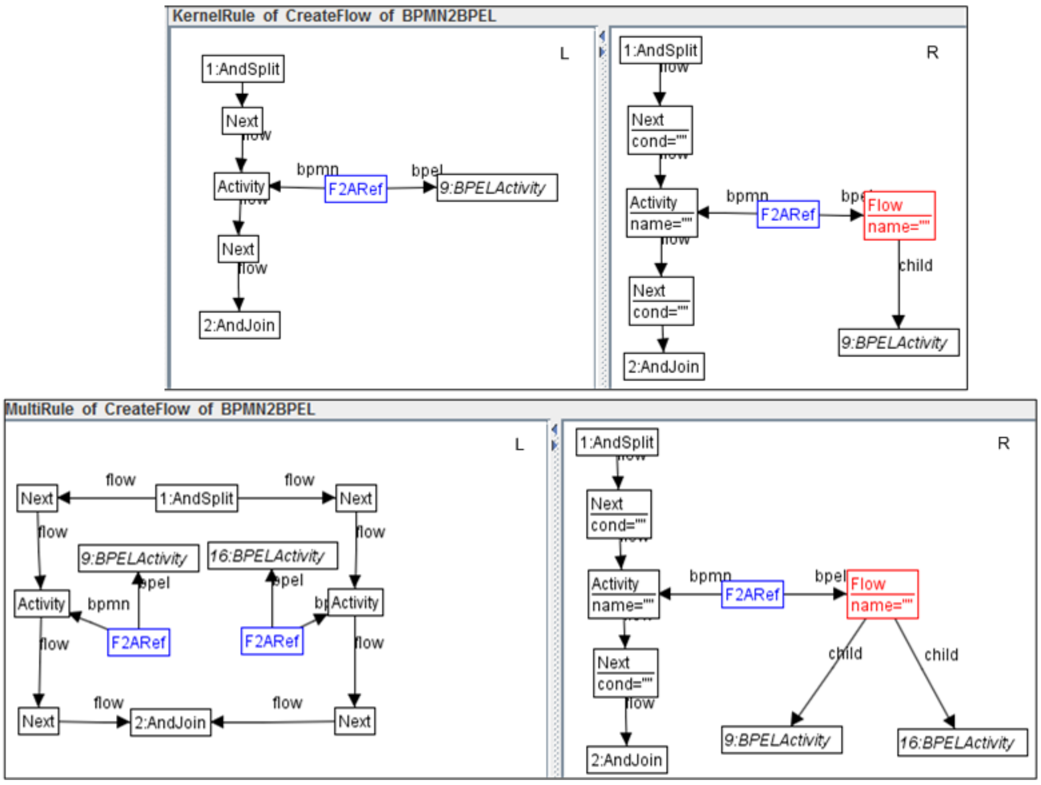
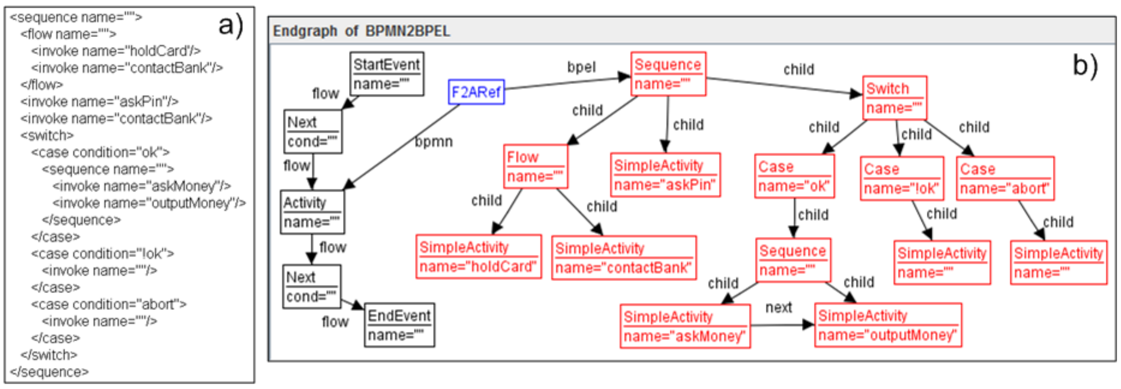


Figure 7: Interaction scheme *CreateFlow*

The abstract syntax of the BPEL expression is the red tree with root node Sequence. The concrete syntax of the BPEL model corresponding to this tree is shown in Figure 8 (a).



```
<sequence name="">
  <flow name="">
    <invoke name="holdCard"/>
    <invoke name="contactBank"/>
  </flow>
  <invoke name="askPin"/>
  <invoke name="contactBank"/>
  <switch>
    <case condition="ok">
      <sequence name="">
        <invoke name="askMoney"/>
        <invoke name="outputMoney"/>
      </sequence>
    </case>
    <case condition="!ok">
      <invoke name=""/>
    </case>
    <case condition="abort">
      <invoke name=""/>
    </case>
  </switch>
</sequence>
```

Figure 8: ATM machine after transformation: (a) in concrete BPEL syntax, (b) in abstract syntax

Tool Support

We implemented the case study in our tool AGG [AGG09, BEL+10], supporting the definition of type graphs, typed attributed graph rules and constraints. AGG has been extended recently by support for defining and applying amalgamated graph transformation. All screenshots in this section are taken from the AGG editors for rules and interaction schemes. Moreover, AGG supports verification of model transformations w.r.t. termination and confluence (functional behaviour). Further graph transformation tools tuned for domain-specific model transformations are VIATRA2 [BNS+05] and the Graph Rewriting and Transformation Language (GReAT) [SAL+03].

Related Work

A related model transformation approach based on graph transformation are triple graph grammars (TGGs) [Sch94] which transform pairs of related models simultaneously while maintaining their consistency. TGGs generate languages of *triple graphs*, consisting of a source graph G^S and a target graph G^T, together with a correspondence graph G^C "between" them. A triple graph is typed by a meta-model triple which contains the source and target meta-models, and declares the types of mappings between the elements of both languages. A *triple rule* tr consists of triple graphs $L = (SL \leftarrow CL \rightarrow TL)$ and $R = (SR \leftarrow CR \rightarrow TR)$, and an injective triple graph morphism $tr = (s,c,t) : L \rightarrow R$, representing a non-deleting rule which adds target elements.

Unsolved Problems

An open problem is the semantical correctness of model transformations. In order to be semantically correct, a model transformation should lead to target models which behave equivalently w.r.t. the corresponding source models. This is an important property of e.g. code generators for behavioural models. In the case that a model is more abstract than the code, semantical properties are defined explicitly, and it has to be shown that these properties are fulfilled by the respective pairs of source and target models.

4 Case Study 3: Metabolic Pathway Analysis

Problem

Metabolic pathway analysis is one of the tools in biology and medicine in order to understand chemical reaction cycles in living cells. The problem is that often, reactions are analysed at the level of structural formulae only, thus summarising the number of atoms of certain types in a compound without keeping track of their identity.

Aim of the Model

This case study [EHL06] aims at understanding chemical reactions at the level of individual atoms or component molecules. In particular, we are interested in the analysis of causal dependencies between biochemical reactions. Given a metabolic pathway (a sequence of reactions) we would like to be able to trace the history of particular atoms or molecules. This is relevant, for example, when trying to anticipate the outcome of experiments using radioactive isotopes of such atoms. Such questions have been crucial to the detailed understanding of the nature of reactions like the citric acid cycle.

Techniques used to solve the problem / realize the aim
Biological systems and chemical reactions are characterized by their inherent concurrency, allowing reactions to take place simultaneously as long as they involve different resources and to keep track of causal dependencies and conflicts between them. Graph transformation systems provide concurrency concepts which are suitable to be applied in this area. For modeling the metabolic pathway, we propose a new hypergraph model for chemical compounds which refines the classical representation in terms of structural formulae in two different ways.

- Our representation keeps track of the identity of atoms or molecular components by means of the identities of hyperedges. In contrast, when writing down chemical reactions with structural formulae, the identities of the reacting atoms are not explicitly represented in the notation. In situations where several atoms of the same element are involved, this lack of information leads to ambiguity as to where a new atom is placed in the resulting molecule. Our graph transformation-based model allows to track atom identities by graph homomorphisms between the graphs representing the compounds before and after the reaction.

- Modelling atoms as hyperedges, each connected to an ordered sequence of nodes, the relative spatial orientation of different molecular components is recorded through the ordering of the nodes connected to a hyperedge.

Using this model we are able to trace the dependencies between different steps in the reaction based on individual atoms and their spatial arrangement.

Formally, given a ranked set of labels $\mathscr{A} = (\mathscr{A}_n)_{n\in\mathbb{N}}$, an $\mathscr{A}$-labelled hypergraph (V,E,s,l) consists of a set V of vertices, a set E of edges, a function $s : E \to V^*$ assigning each edge a sequence of vertices in V, and an edge-labelling function $l : E \to \mathscr{A}$ such that, if $length(s(e)) = n$ then $l(e) = A$ for $A \in \mathscr{A}_n$, i.e., the rank of the labels determines the number of nodes the edge is attached to. A morphism of hypergraphs is a pair of functions $\phi_V : V_1 \to V_2$ and $\phi_E : E_1 \to E_2$ that preserve labels and assignments of nodes, that is, $l_2 \circ \phi_E = l_1$ and $\phi_V^* \circ s_1 = s_2 \circ \phi_E$. A morphism thus has to respect the atom represented by an edge and also its chemical valence (number of bonds). Labelled hypergraphs can be considered as hierarchical graph structures. As shown by Löwe [Löw93], pushouts can be computed elementwise for all hierarchical graph structures and therefore the standard graph transformation approaches can be applied.

Overview of the Model
We consider as an example the citric acid cycle, a classical, but non-trivial reaction for energy utilisation in living cells [ZPV95]. Our approach supports a molecular analysis of the cycle, tracing the flow of individual carbon atoms based on a simulation. This cycle is a series of chemical reactions of central importance in all living cells that utilise oxygen as part of cellular respiration. Starting with *acetyl-CoA*, one of the resulting products of the chemical conversion of carbohydrates, fats and proteins, the citric acid cycle produces fast usable energy in the form of *NADH*, *GTP*, and *FADH*$_2$ which are precursors of the well known *adenosine-tri-phosphate (ATP)*.

Figure 9 shows reaction 2 of the citric acid cycle. The input agent of reaction 2, *citrate*, has two CH_2COO^- groups, one on the top and one on the bottom.To fit into the enzyme *aconitase* catalysing reaction 2 (see Figure 9), only the CH_2COO^- group marked with 3 is able to fit into the enzyme due to 3-dimensional spatial relations.

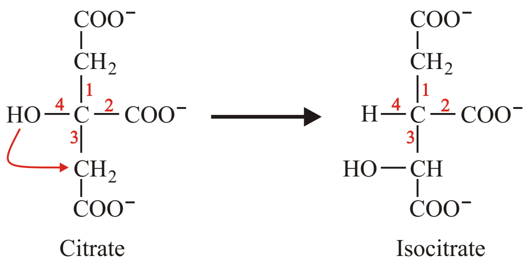
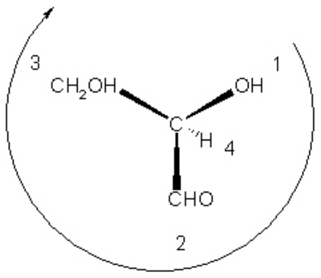


Figure 9: Reaction 2 of the citric acid cycle

In our hypergraph model, we interpret the hyperedges as atoms and the nodes as bonds between them. The string $s(e)$ of vertices incident to an edge $e \in E$ gives the specific order of the bonds to other atoms, coding also their spatial configuration, as we will see. As ranked set of labels, we use $\mathscr{A}_1 = \{\mathrm{H}, \mathrm{CH_3}, \mathrm{OH}, \ldots\}$, $\mathscr{A}_2 = \{\mathrm{O}, \mathrm{CH_2}, \mathrm{S}, \ldots\}$, $\mathscr{A}_3 = \{\mathrm{CH}, \mathrm{N}, \ldots\}$, $\mathscr{A}_4 = \{\mathrm{C}, \mathrm{S}, \ldots\}, \ldots$ to denote elements of the periodic system or entire chemical groups. The rank of a label models the valence of an atom. For instance, a carbon atom with $l(e) = \mathrm{C}$ always has $s(e) = v_1v_2v_3v_4$, a word of length 4. Hence, we define C as a label of rank 4. For elements with more than one possible valence (e.g. sulphur), the corresponding label can belong to several of the sets $\mathscr{A}_n$.

Given an organic molecule, we represent the 3-dimensional configuration of the ligands of a C atom as a hypergraph by relating it to D-glyceraldehyde, one of the simplest chiral organic compounds. We impose a numbering on the ligands of a carbon atom such that a substitution of ligand 1 by OH, ligand 2 by CHO, ligand 3 by CH_2OH, and ligand 4 by H would result in D-glyceraldehyde.



This convention defines the spatial arrangement of the ligands unambiguously. Substitution of ligands may change the angles between the ligands, and they often differ from the regular tetrahedral angle of $109°28'$, but the so called *angle strain* [Mos96] does not affect the uniqueness of the molecule represented by our notation.

As an example, we give the representation of the prochiral molecule *citrate* as a hypergraph (see Fig 10):

$$V = \{v_1, v_2, \ldots, v_6\}, E = \{e_1, e_2, \ldots, e_7\},$$

$$s(e_1) = v_1, s(e_2) = v_1v_2, s(e_3) = v_3, s(e_4) = v_2v_3v_4v_5, s(e_5) = v_4,$$

$$s(e_6) = v_5v_6, s(e_7) = v_6$$

$$l(e_1) = \mathrm{COO}^-, l(e_2) = \mathrm{CH_2}, s(e_3) = \mathrm{OH}, s(e_4) = \mathrm{C}, s(e_5) = \mathrm{COO}^-,$$

$$s(e_6) = \mathrm{CH_2}, s(e_7) = \mathrm{COO}^-$$

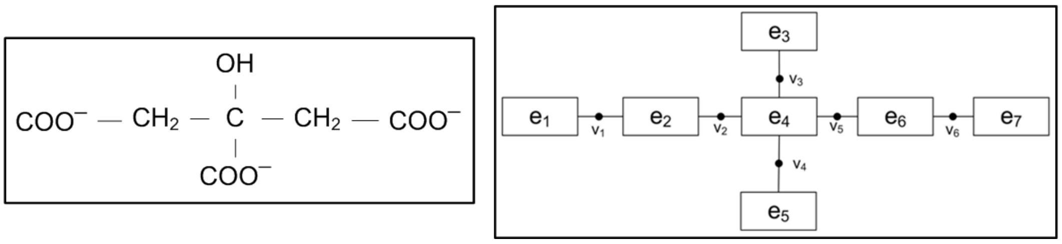

OH
$COO^- — CH_2 — C — CH_2 — COO^-$
COO^-
e_1 e_2 e_3 e_4 e_5 e_6 e_7
v_1 v_2 v_3 v_4 v_5 v_6

Figure 10: Structural formula (left) and hypergraph representation (right) of citrate.

Tool Support

We provide an encoding of the model in terms of attributed bipartite graphs that can be implemented in the AGG system for simulation and analysis [Tae04, AGG09].

Figure 11 shows reaction 2 of the citric acid cycle (see Figure 9) modelled in AGG. The enzyme *aconitase* accepts only the source agent *citrate* with the indicated *o* edge attribute order of the *1:C* atom in the left-hand side of Figure 11. In this reaction the OH group of the *1:C* atom is exchanged with the OH group of the *3:C* atom. This leads to the new agent *isocitrate*.

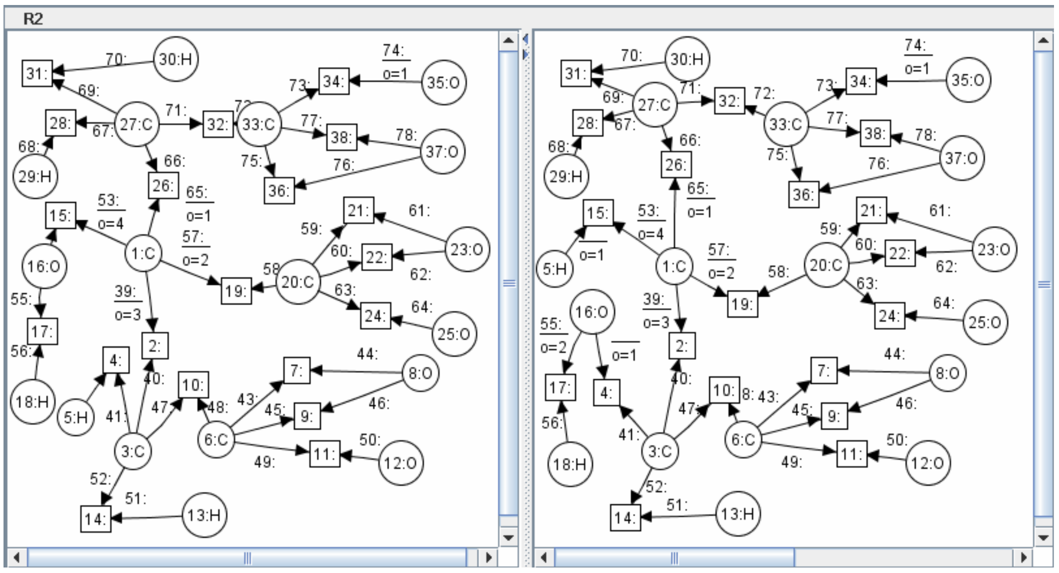


Figure 11: Reaction 2 of the citric acid cycle in AGG.

Related Work

The use of graph transformation for biological systems has a long history (see [RV05]), but early applications were mostly devoted to the field of morphogenesis. Our approach focuses on biochemistry, a field which gained much importance in the last decades because of the growth of biotechnology. Providing automated assistance for analyzing biochemical reactions can help in understanding the principles which govern the processes in living cells.

Unsolved Problems
The citric acid cycle is a very common cycle for energy utilization in living cells. However, biological systems are very complex and hard to understand, so most of the biological pathways are still not completely understood. For analyzing more complex pathways, big computer clusters are needed. Modelling with graph transformations might produce an overhead of data structures for the internal representation and computation with graphs. In general, the graph transformation problem is NP-complete. Putting several reactions together, the system might be unsolvable in a usable time frame.

5 Case Study 4: Self-Healing Automated Traffic-Light

Problem
Self-healing (SH-)systems are characterized by an automatic discovery of system failures, and techniques how to recover from these situations. The problem is that failures can occur at any time during system operation. It is very important for such systems that recovery actions can always be applied after a failure has occurred and that they always lead to a system that works as expected.

Aim of the Model
The aim of our model is to verify that SH-systems have certain self-healing properties, so that they are always able to fulfill the system requirements in case that a failure has occurred.

Technique to solve the problem / realize the aim
In this case study, we model SH-systems by typed attributed graph transformation systems enriched with graph constraints expressing their consistency w.r.t. their operational properties. We make use of theoretical results, i.e. sufficient static conditions for self-healing properties, deadlock-freeness and liveness of SH-systems.

Overview of the Model
The complete case study is given in [EER+10]. We model an automated Traffic Light System (TLS). The traffic light technology is based upon electromagnetic spires buried some centimeters underneath the asphalt of car lanes. The spires register traffic data and send them to other system components. The TLS is connected to cameras which record videos of the violations and automatically send them to the center of operations. In addition to the normal behavior, we may have failures caused by a loss of signals between a traffic light or a camera and the supervisor component. For each of the failures there are corresponding repair actions which can be applied after monitoring the failures during run-time.

We define the Traffic Light SH-system *TLS* by a type graph TG, an initial state, a set of normal rules R_{norm} (modelling the ideal behaviour), a set of failure rules R_{fail} modelling failures, a set of repair rules R_{repair}, which are the inverse repair rules, and sets of constraints that characterize properties of states being either consistent or failure states.

In our example, we model a single traffic crossing with two traffic lights in directions north-south and east-west. In the initial state (see Figure 12), both traffic lights are red and there are no cars at the crossing. The TL nodes represent the traffic lights, connected to a crossing supervisor component, and to cameras which are currently not in use (onCamera=false). The infraction attribute becomes true in the case that a car runs a red light.

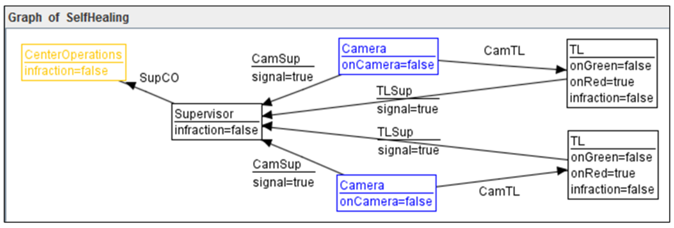


Figure 12: TLS initial state G_{init}

Normal rules model the behaviour of cars arriving at the crossing and leaving it, as well as cars running a red light and being filmed by a camera. Failure rules model the loss of a signal of either a traffic light (in this case the signal attribute of a TLSup edge changes to false), or of a camera (here, the signal attribute of a CamSup becomes false). The repair rules model the recovery from the respective signal loss. In [EER$^+$10], we formalize operational properties, including self-healing (i.e. each failure state can be repaired), and deadlock-freeness, and we provide static conditions for them based on rule set analysis.

Tool Support
For the automatic analysis of the static conditions ensuring the self-healing properties we use AGG, in particular to check dependencies and conflicts of rules. All properties are verified for our traffic light system.

Related Work
Different related approaches exist, either based on graph transformation [6,14,15,16,17,18,19] or on temporal logics and model checking [20,21,22]. In many cases, though, the state space of behavioral system models becomes too large or even infinite, and in this case model checking techniques have their limitations.

Unsolved Problems
A helpful extension of the formal approach would be the analysis and verification of consistency properties using the theory of graph constraints and nested application conditions in [EHL10]. Moreover, we will investigate how far the techniques in this paper for SH-systems can be used and extended for more general self-adaptive systems.

6 Evaluation and Conclusion

The table in Figure 13 summarizes the problem domains and modelling features and results for our four case studies. In the last line, we state concepts which, from our point of view, are missing not only for the particular case study presented in this paper but rather in general for the respective application domain.

	(1) Medical Information System	**(2) Business Process Model Transformation**	**(3) Self-Healing Automated Traffic Light System**	**(4) Metabolic Pathway Analysis**
Problem	Adequate visualization of clinical processes	Source-to-Target model transformation	System modelling with failures and recovery	Molecular ana-lysis of chemical reactions
GraTra Model	Visual language modelling by typed attributed GraTra	GraTra based on source-target type graph inclusion $TG_S \rightarrow TG_I \leftarrow TG_T$	GraTra with different rule sets R_{normal}, $R_{failure}$, R_{repair} and constraints	Hypergraph transformations with simulation
GraTra Results	Graph constraint satisfaction after transformation	Parallel independence of amalgamated GraTra	Static analysis of self-healing properties	Simulation, embedding and extension
Missing Concepts	Advanced tool support for visual user interfaces	Semantical correctness of model trafos	Critical pair analysis for general conditions	Scalability of graph representation

Figure 13: Comparison of Case Studies

In the area of visual language modelling (e.g. for **case study (1)**), the concept of typed attributed graph transformation, which is close to meta-modelling, proved to be suitable for defining syntax and semantics of domain-specific languages. But to be useful in the context of larger systems, these principles should be integrated in tools that are used in practical applications. In our case study, a suitable user interface should hide the formal representation of abstract graph and rule syntax, and the underlying model needs to be linked to the clinic information system. Here, advanced tool support integrating graph transformation tools to existing tools used in practice is one aim for graph transformation technology transfer.

Model transformations from domain-specific models to more machine-centric formats (like **case study (2)**) have become a necessary step towards unified and standards-based development environments. Here, important results have been achieved in recent years concerning the syntactical correctness of model transformations and their functional behaviour, i.e. termination and uniqueness. Also, for triple graph grammars, properties concerning the consistency of source and target models w.r.t. triple rules can be shown formally. An open problem for model transformations remains the semantical correctness, i.e. how can be shown in general that the behaviour of the source and the target model are equivalent (see also [Erm09]).

Often, a validation by simulation is helpful to provide new insights on behavioural system properties. **Case study (3)** showed that a simulation by graph transformation, supported by tools, can help to find a suitable abstraction level and visualize model features (like molecule identities) which are not easily seen using standard techniques and tools. Here, the problem arises that in contrast to standard tools, a graph representation might lead to a larger memory consumption than e.g. the standard format for chemical formulae. The general problem of scalability of the graph representation of models and rules has been tackled already by improving the performance of existing graph transformation engines and experimenting with different data formats. Here, future work will be necessary for further optimizations.

Many verification results for graph transformation systems are based on critical pair analysis. This kernel technique is also used in **case study (4)**, where we analyse conflicts and dependencies of rules to show self-healing properties. Recently, general (nested) conditions on graphs have been defined by Habel and Pennemann [HP09]. These conditions allow for a very flexible modelling of graph rules. In this context, it remains to provide the formal background for critical pair analysis of rules with nested application conditions.

Already, some of the "missing concepts" are topics of ongoing research projects[1]. We are confident that the visibility of graph transformation technology in practice will be further enhanced and that meetings between theory and practice, aided by good tool support, will be the rule rather than the exception.

Bibliography

[AGG09] TFS-Group, TU Berlin. AGG. 2009. http://tfs.cs.tu-berlin.de/agg.

[BE09] E. Biermann, C. Ermel. Transforming BPMN to BPEL with EMF Tiger. In *Proc. Graph-based Tools (GraBaTs'09)*. 2009. http://is.tm.tue.nl/staff/pvgorp/events/grabats2009/submissions/

[BEEH09] E. Biermann, K. Ehrig, C. Ermel, J. Hurrelmann. Generation of Simulation Views for Domain Specific Modeling Languages based on the Eclipse Modeling Framework. In *Automated Software Engineering (ASE'09)*. IEEE Press, 2009.

[BEL$^+$10] E. Biermann, C. Ermel, L. Lambers, U. Prange, G. Taentzer. Introduction to AGG and EMF Tiger by Modeling a Conference Scheduling System. *Software Tools for Technology Transfer*, 2010. To appear.

[Béz05] J. Bézivin. On the unification power of models. *Software and System Modeling* 4(2):171–188, 2005.

[BFH87] P. Böhm, H.-R. Fonio, A. Habel. Amalgamation of graph transformations: a synchronization mechanism. *Computer and System Sciences (JCSS)* 34:377–408, 1987.

[BGL09] K. Biermann, M. Grötschel, B. Lutz-Westphal (eds.). *Besser als Mathe: Moderne angewandte Mathematik aus dem MATHEON zum Mitmachen*. Vieweg, 2009.

[BNS$^+$05] A. Balogh, A. Németh, A. Schmidt, I. Rath, D. Vágó, D. Varró, A. Pataricza. The VIATRA2 Model Transformation Framework. In *Proc. European Conference on Model Driven Architecture (ECMDA'05)*. 2005.

[1] See e.g. our project *Behaviour Simulation and Equivalence of Systems Modelled by Graph Transformation* (supported by the German Research Council) at http://www.tfs.tu-berlin.de/menue/forschung/#BehaviourGT.

[BTMS99] R. Bardohl, G. Taentzer, M. Minas, A. Schürr. Application of Graph Transformation to Visual Languages. In [EEKR99].

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

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theor. Comp. Science. Springer, 2006.

[EER$^+$10] H. Ehrig, C. Ermel, O. Runge, A. Bucchiarone, P. Pelliccione. Formal Analysis and Verification of Self-Healing Systems. In *Proc. Fundamental Aspects of Software Engineering (FASE'10)*. Springer, 2010. To appear.

[EHL06] K. Ehrig, R. Heckel, G. Lajios. Molecular Analysis of Metabolic Pathway with Graph Transformation. In *Proc. Graph Transformation (ICGT'06))*. LNCS 4178. Springer, 2006.

[EHL10] H. Ehrig, A. Habel, L. Lambers. Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. In *Manipulation of Graphs, Algebras and Pictures: Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday*. ECEASST, 2010. To appear.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation. Vol 3: Concurrency, Parallelism and Distribution*. World Scientific, 1999.

[Eng00] G. Engels. Graph Changes are Everywhere: The Role of Graph Transformations in Software Engineering. In *Proc. Joint APPLIGRAPH and GETGRATS Workshop on Graph Transformation Systems (GraTra'00)*. TU Berlin, TR 2000-2, 2000.

[Erm09] C. Ermel. Visual Modelling and Analysis of Model Transformations based on Graph Transformation. *Bulletin of the EATCS* 99:135 – 152, 2009.

[Fou09] Eclipse Modeling Foundation. MDT-UML2Tools. 2009. http://wiki.eclipse.org/MDT-UML2Tools.

[GMF07] Eclipse Consortium. Eclipse Graphical Modeling Framework (GMF). 2007. http://www.eclipse.org/gmf.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Comp. Science* 19:1–52, 2009.

[IBM03] IBM, BEA Systems, Microsoft, SAP AG, Siebel Systems. Business Process Execution Language for Web Services version 1.1. May 2003. http://www.ibm.com/developerworks/library/ws-bpel/.

[Löw93] M. Löwe. Algebraic Approach to Single-Pushout Graph Transformation. *TCS* 109:181–224, 1993.

[Mos96] G. Moss (ed.). *IUPAC Basic Terminology of Stereochemistry*. Volume 68(12). Pure & Applied Chemistry, 1996.

[MVVK05] T. Mens, P. Van Gorp, D. Varrò, G. Karsai. Applying a Model Transformation Taxonomy to Graph Transformation Technology. In *Proc. Graph and Model Transformation (GraMoT'05)*. ENTCS 152, pp. 143–159. Elsevier Science, 2005.

[OMG07] OMG. Unified Modeling Language: Superstructure – Version 2.1.1. 2007. formal/07-02-05, http://www.omg.org/technology/documents/formal/uml.htm.

[Ope09] OpenEmbeDD: Model Driven Engineering open-source platform for Real-Time & Embedded systems. 2009. http://openembedd.org.

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

[RV05] F. Rosselló, G. Valiente. Graph Transformation in Molecular Biology. In *Formal Methods in Software and System Modeling, LNCS 3393*. Pp. 116–133. Springer, 2005.

[SAL$^+$03] J. Sprinkle, A. Agrawal, T. Levendovszky, F. Shi, G. Karsai. Domain model translation using graph transformations. In *Int. Conf. on Engineering of Computer-Based Systems*. Pp. 159–168. 2003.

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

[Tae04] G. Taentzer. AGG: A Graph Transformation Environment for Modeling and Validation of Software. In Pfaltz et al. (eds.), *Application of Graph Transformations with Industrial Relevance (AGTIVE'03)*. LNCS 3062, pp. 446 – 456. Springer, 2004.

[Tae06] G. Taentzer. Characterizing Tools for Visual Modeling Techniques. In Ehrig et al. (eds.), *Lecture Notes of SegraVis Advanced School on Visual Modelling Techniques*. 2006.

[TEG$^+$05] G. Taentzer, K. Ehrig, E. Guerra, J. de Lara, L. Lengyel, T. Levendovsky, U. Prange, D. Varro, S. Varro-Gyapay. Model Transformation by Graph Transformation: A Comparative Study. In *Proc. Workshop Model Transformation in Practice*. 2005. http://tfs.cs.tu-berlin.de/publikationen/Papers05/TEG+05.pdf

[Whi04] S. White. Business Process Modeling Notation (BPMN) Version 1.0. BPMI.org, 2004.

[ZPV95] G. Zubay, W. Parson, D. Vance. *Principles of Biochemisty*. Volume 2. McGraw-Hill College, 1995.

A Termination Criterion for Graph Transformations with NACs

Paolo Bottoni, Francesco Parisi Presicce

Dipartimento di Informatica, ”Sapienza” Università di Roma, Italy

Abstract: Termination of graph rewriting is in general undecidable, but it is possible to prove it for specific systems by checking for sufficient conditions. In the presence of rules with negative application conditions, the difficulties increase. In this paper we propose a different approach to the identification of a (sufficient) criterion for termination based on the construction of a labeled transition system whose states represent overlaps between the negative application condition and the right hand side that can give rise to cycles.

Keywords: DPO, termination, label transition system, model transformation

1 Introduction

Model transformations are an essential component of the model-driven approach to software development. Graphs are a natural and intuitive way to describe models (e.g., class diagrams in UML) and graph transformations provide a rule-based approach to their modifications. Sometimes a particular transformation needs to be applied to the target graph/model as long as matchings of its left hand side can be found. In such cases, it is necessary to be able to determine that such a repeated application will eventually reach a state where the transformation is no longer applicable. More generally, the term *termination* refers to the problem of determining whether a set of rules can generate a graph/model to which none of the rules is still applicable. Ad hoc methods have been applied to show termination of specific rewriting systems (e.g. [KHE03]).

Termination properties can be (and have been) studied for specific rewriting systems, following the classical approach – as introduced by Dershowitz and Manna in [DM79] – of proving termination by constructing a monotone measure function on some multiset associated to the object to be rewritten, and showing that the value of such a function decreases with each application of the rule. Further termination criteria use polynomial orderings, recursive path orderings, etc. [Der87].

In a previous paper [BHPT05], we have identified an abstract notion of termination criterion for high-level replacement (HLR) systems, i.e. algebraic rewriting systems operating on objects and morphisms in adhesive HLR categories [EPPH06], in which rewriting is guided by control expressions. The approach is based on a generic measure function $\mathscr{F} : G \to N$, called a *(*termination criterion) if it satisfies the property $\mathscr{F}(A +_C B) = \mathscr{F}(A) +_{\mathscr{F}(C)} \mathscr{F}(B)$ for morphisms $C \to A$ and $C \to B$ in a specific subclass $\mathscr{M}$. A termination criterion for a rule $p : L \leftarrow K \to R$ is such a function with $\mathscr{F}(\mathrm{L}) \succ \mathscr{F}(\mathrm{R})$.
However, we have subsequently shown in [BHP06] how the extension of this notion to rules with negative application conditions (NACs) encounters several difficulties. In particular, we have presented examples of pairs of rules for which no application criterion can differentiate between a terminating and a non-terminating rule.

In this paper, we propose a different approach to the identification of a (sufficient) termination criterion for rules with NACs, based on the construction of a Labelled Transition Systems, where states correspond to classes of matches of a rule with respect to all the possible intermediate graphs between the left-hand side of a rule and a negative application condition.

2 Formal Background

We use the DPO (Double PushOut) approach to graph transformation [EEPT06], although the approach can be adapted to the SPO (Single PushOut) approach [EL93] as well.

A graph $G = (V, E, s, t)$ consists of a set of *nodes* $V = V(G)$, a set of *edges* $E = E(G)$, a *source* and a *target* function, $s, t : E \to V$. In a *type graph* $TG = (V_T, E_T, s^T, t^T)$, V_T and E_T are sets of node and edge types, while the functions $s^T : E_T \to V_T$ and $t^T : E_T \to V_T$ define source and target node types for each edge type. A typed graph on $TG = (V_T, E_T, s^T, t^T)$ is a graph $G = (V, E, s, t)$ equipped with a graph morphism $type\colon G \to TG$, composed of two functions $type_V : V \to V_T$ and $type_E\colon E \to E_T$, preserving the *source* s^T and the *target* t^T functions, i.e. $type_V(s(e)) = s^T(type_E(e))$ and $type_V(t(e)) = t^T(type_E(e))$.

A DPO rule consists of three graphs, called left- and right-hand side (L and R), and interface graph K. Two injective morphisms[1] $l\colon K \to L$ and $r\colon K \to R$ model the embedding of K (containing the elements preserved by the rule) into L and R. Figure 1 shows a DPO direct derivation diagram. Square (1) is a pushout modeling the deletion from G of the elements of L not in K, while pushout (2) models the addition to G of the elements present in R but not in K. Figure 1 also illustrates the notion of *negative application condition* (NAC), of the form $n\colon L \to N$ that a match $m\colon L \to G$ must satisfy. A rule is applicable with match $m\colon L \to G$ if there is no morphism $q\colon N \to G$ such that $q \circ n = m$.

$$\begin{tikzcd}
N \arrow[rd, bend right, "q"'] & L \arrow[l, "n"'] \arrow[d, "m"'] \arrow[ld, phantom, "\neq"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"'] \arrow[l, phantom, "(1)", shift left=6] & R \arrow[d, "m^*"] \arrow[l, phantom, "(2)", shift left=6] \\
 & G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 1: DPO Direct Derivation Diagram for rules with NAC.

3 A Termination Criterion

We study termination of single rules with a single NAC; we leave it to future work to extend the results to the case of multiple NACs and of rule sequences.

As we consider only non-deleting rules, we omit the K component of rules and write a rule with a single NAC as $p : N \leftarrow L \to R$.

Consider the simple example in Figure 2.

The rule is a non-deleting rule, so it is not clear how to apply the standard approach based on

[1] In this paper, when we speak of morphisms, we mean injective.

the ’consumption’ of some finite quantity. Nevertheless, the rule can only be applied a finite (two) number of times at most, regardless of the matching chosen for the left hand side. What decreases after each application is the difference between the left hand side and the negative application condition.

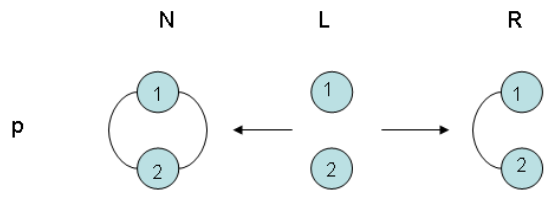


Figure 2: A simple terminating rule.

Consider now the slightly different rule in Figure 3.
In this rule, the edges have a direction and it is no longer true that its application must always terminate. After the first application, if the roles of the 2 nodes are reversed in the matching, the remaning part of the negative application condition is generated. But it is also possible to continue adding edges from node 1 to node 2, without ever generating the NAC to prevent further applications. Notice that these additional edges do not affect the applicability or not of the rule.

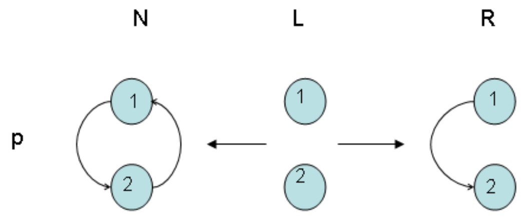


Figure 3: A simple non terminating rule.

In both cases, the rule generates a graph ’between’ the left hand side and the NAC. We now abstract from the specific examples.

Let $p : N \xleftarrow{n} L \xrightarrow{r} R$ be a rule. Let $\mathscr{H}^p = \{H^p_1, \dots, H^p_k\}$ be the set of all graphs and $\langle^p = \{h^i_j : H^p_i \to H^p_j\}$ be the set of associated morphisms such that

- for each $i = 1, \dots, k$, there exist morphisms $L \xrightarrow{h^L_i} H^p_i \xrightarrow{h^N_i} N$.
- for each $i, j = 1, \dots, k$, $h^L_j = h^L_i \circ h^i_j$ and $h^N_i = h^i_j \circ h^N_j$.

Note that the set is not empty since it includes $L = H_1^p$ and $N = H_k^p$ with the identity morphisms and $h_k^1 = n$.

Let V_L be the set of nodes in L. Let $M_L = \{m_1, \dots, m_r\}$ be the set of matches of V_L into itself, including the identity $id_{V_L} = m_1$.

We now construct a Labelled Transition System $\mathscr{L}^p = (S, \Lambda, \longrightarrow)$ as follows:

1. S contains a state s_i for each graph $H_i^p \in \mathscr{H}^p$. Each s_i induces a classification function c_i for matches of p such that $c_i(m_j) = true$ iff m_j can be extended to a match on H_i^p, but not to a match on any other $H_j^p \in \mathscr{H}_p \setminus (\{H_1^p, H_i^p\} \cup \{H_t^p | \exists h_i^t : H_t^p \to H_i^p\})$, i.e. H_i^p is the biggest graph to which m_l can be extended.

2. Λ contains a label p^i, $i = 1, \dots, r$ for each morphism in M_L.

3. $\longrightarrow \subset S \times S$ is such that $s_i \xrightarrow{p^l} s_j$ if the application of p with match m_l on graph H_i^p produces a graph for which $c_j(m_l) = true$.

We say that p:

- *should terminate simply* if there exists a chain from s_1 to s_k in $\mathscr{L}^p$ with all transitions labeled with p_1.

- *may terminate simply* if there exists a chain from s_1 to s_k in $\mathscr{L}^p$ with at least one label different from p_1.

- *does not terminate simply* if there is no chain from s_1 to s_k in $\mathscr{L}^p$.

In the definition above, by chain we mean a path which does not contain the same state twice.

We now consider the variation on the number of possible matches induced by the application of rule p on its minimal context. To this end, let $@ \mid \cdot \mid : S \to N$ be a function which associates with each state the number of matches for p on the graph H_i^p prior to application of p and with $\mid \cdot \mid : S \to N$ the function defining the number of matches on H_i^p after the application of p.

We use these functions to identify the effect of each transition from a path in $\mathscr{L}^p$ on the number of matches for the graph represented by the state.

We say that p:

- *must terminate* if it should terminate simply and for all states s_i in the path $\mid s_i \mid < @ \mid s_i \mid$.

- *may terminate* if it may terminate simply and for all states on the path $\mid s_i \mid < @ \mid s_i \mid$ after each transition on the path.

- *does not terminate* if it does not terminate simply or it should or may terminate simply, but there is at least one state on a path from s_1 to s_k for which $\mid s_i \mid \geq @ \mid s_i \mid$, for some transition on the path.

4 Examples

We present here some examples to illustrate the different cases which may occur. We use meaningful labels for states to describe the corresponding graph H_i^p. The state s_1 is indicated with an arrow, and the state s_k with a pair of concentric ovals.

In the Figures, we report the variations of the cardinality of matches for the source states of the transitions, or for states whose number of matches is affected by the transition, even if the transition does not involve the state. All other cardinalities are assumed to remain equal.

Figure 4 re-proposes the case (already seen in Figure 2) of a rule which must terminate, since there is a path with all transitions labeled p^1 and for all the states in the path the cardinality of the matches decreases.

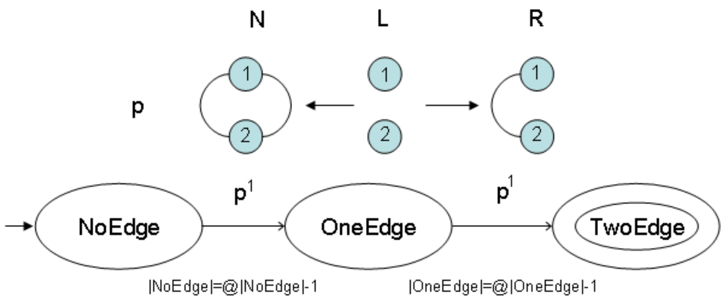


Figure 4: A terminating rule.

Figure 5 also presents the case (already seen in Figure 3) of a rule which may terminate. In this case, there is a path from s_1 to s_k and for all the states in the path the cardinality of the matches decreases if the transition making the path progress is taken. But the loops on the middle 2 nodes indicate that the application may not terminate (on the loops, the cardinality of the matches does not decrease).

Figure 6 presents a case of a rule which should terminate simply, as there is a path which consumes the possibility of rule application on the original match, but does not terminate, as the number of matches for the rule increases at each application of p.

The same situation occurs in Figure 7. However, it is interesting to notice that in the rule of Figure 6 there is no relation between R and N, while in the rule of Figure 7 we have $N \subset R$.

However, Figure 8 shows how the existence of an injection of N into R is not sufficient to discriminate between terminating and non-terminating rule. Indeed, rule p in Figure 8 presents a situation in which the rule must terminate. It is important to observe how no single function F from graphs to natural numbers, which is a termination criterion for rules without NACs could be used to discriminate between the two cases. The difference between the two cases is in fact that in the rule of Figure 7, the number of matches increases, whereas in Figure 8 it decreases.

A rule which must terminate, where $R \subset N$, is presented in Figure 9. Compare this to the case in Figure 10, where again $R \subset N$, but the rule may terminate, as choosing a different match after iterating application of the rule with the same match, leads to the state H_k^p, labeled here as

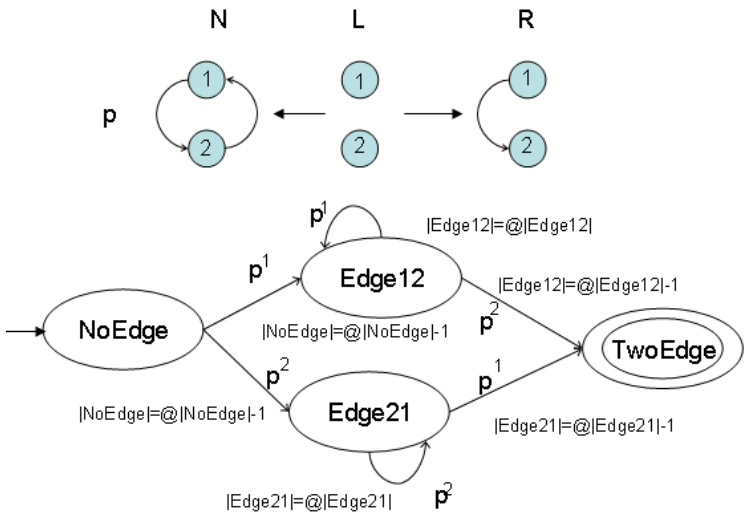


Figure 5: A rule which may terminate.

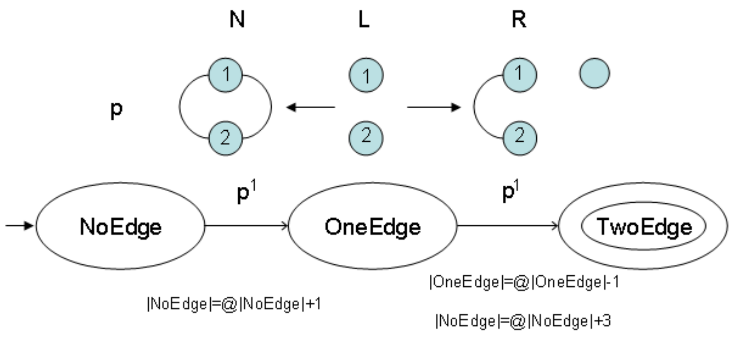


Figure 6: A rule which should terminate simply, but does not terminate.

`TwoLoop`.

5 Related Work

Termination of (string) rewriting systems has been studied for over 30 years (see [DM79] for example). Much more recent is the interest in termination of graph transformation systems.

One of the earlier applications is to program optimization and can be found in [Ass00], (submitted for publication a few years earlier) where termination criteria are defined for 2 specific types of rules. One kind is a deleting rule, which must remove at least one item from a specific subgraph: since graphs are finite, the removal must eventually stop. The other kind is a nondeleting rule that must add at least one edge incident to a node with a specific label: since no pair of nodes can have more than one edge with the same label, the addition must eventually stop and

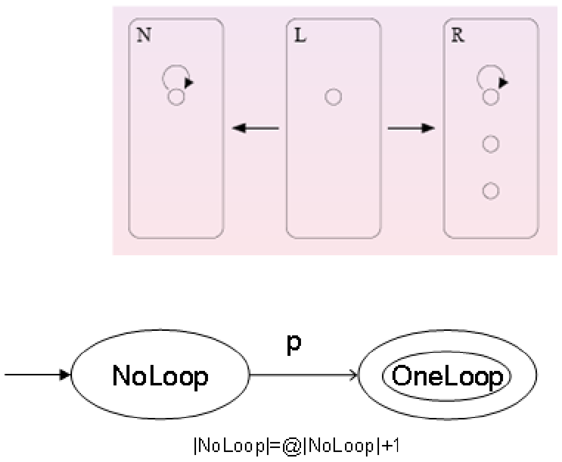


Figure 7: A rule which should terminate simply, but does not terminate, with $N \subset R$.

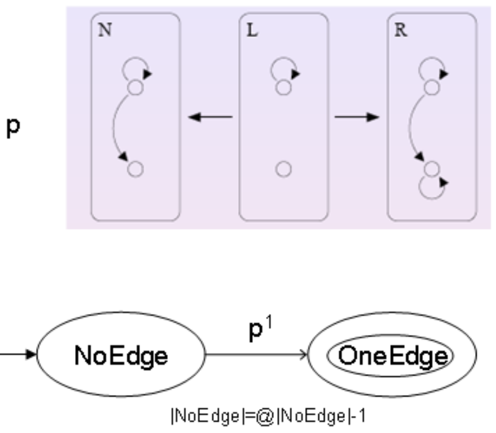


Figure 8: A terminating rule, with $N \subset R$.

so is the applicability.

The general problem of termination for graph rewriting has been tackled by Detlef Plump in [Plu98], where he proves that it is an undecidable problem. Although the framework deals only with 'plain' transformation rules (i.e., without application conditions), we expect the result to hold in general, for example by using trivial conditions alwasy satisfied.

Ad hoc sufficient conditions have been analyzed for special cases. In layered graph transformation systems [HKJ$^+$06] the different types of rules are grouped, establishing an application order. In each of the 2 kinds of layers (deleting and non-deleting) there are no infinite derivation sequences with injective matchings. Each rule in a deletion layer must delete at least one item, but not a newly created one. Each rule in a non-deletion layer cannot delete items, cannot be

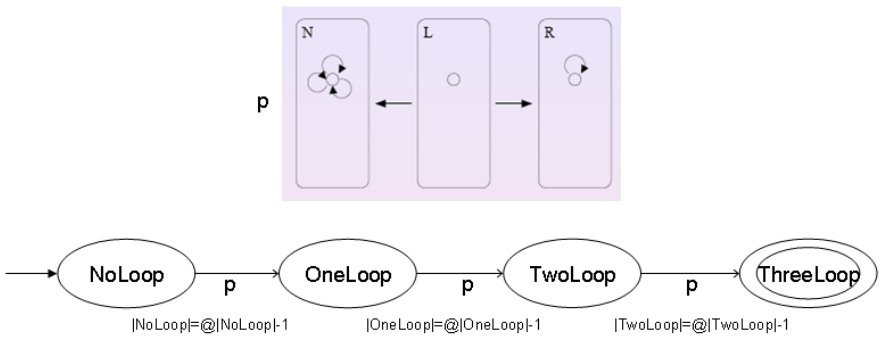


Figure 9: A terminating rule, with $R \subset N$.

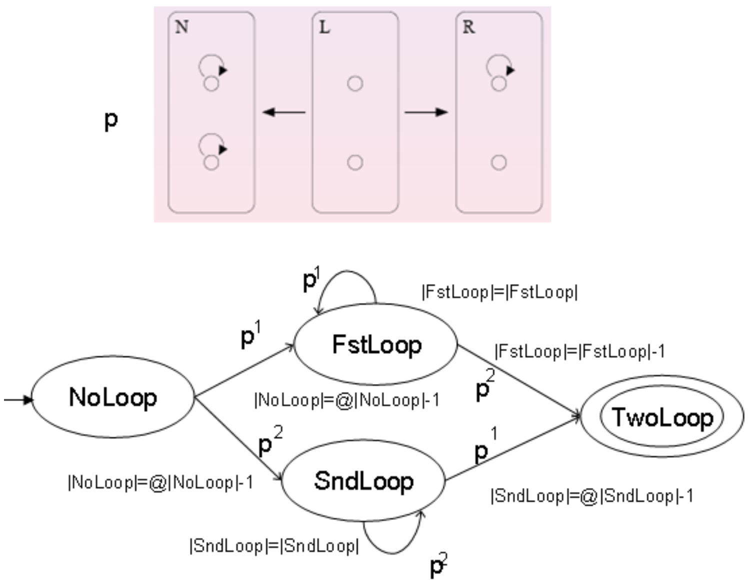


Figure 10: A rule which may terminate, with $R \subset N$.

applied twice with the same match and cannot use a newly created item for the match. A finite number of layers and a finite initial graphs guarantee termination.

More recent research [DSH$^+$06] uses a similar idea to the one presented here. A Graph Transformation System is abstracted by ignoring certain structure in a graph and used to define a Petri Net to represent the number of elements of a certain type. Transitions correspond to rule application with 'consumption' of elements (and reduction of tokens). Termination of the GTS corresponds then to the Petri Net exhausting its tokens.

Comments on some of these approaches and others can be found in [Asz07].

6 Concluding Remarks

In this paper we have discussed an approach to analyze termination properties of transformations. We have focused on the termination of a single rule expression of the form **asLongAsPossible** R , for a non-deleting rule R. Termination of plain transformation rules (i.e., rules without application conditions) usually depends upon a function which measures the consumption of a finite commodity and whose value decreases at each application of the rule. When application conditions are present, we can also measure the (hopefully decreasing) distance between the context and the negative application condition. This is what the steps in the LTS represent.

The examples presented in this paper are necessarily small. What we have not investigated (yet) is the feasibility of the approach to real problems, and in particular the complexity of the LTS relatively to the size of the negative application conditions. A systematic (hence automatic) way to construct the LTS would also be necessary.

Although the discussion and the examples are stated in terms of graphs, no specific properties of graphs are used, but only morphisms and their extensions. The approach can easily be extended to model transformations in high-level replacement (HLR) systems, i.e. algebraic rewriting systems operating on objects and morphisms in adhesive HLR categories [EPPH06]. The extension to multiple NACs should also be straigthforward (with appropriate combinations of the 'measuring' functions) while we expect rule sequences to require other ideas.

Bibliography

[Ass00] U. Assmann. Graph rewrite systems for program optimization. *ACM Trans. Program. Lang. Syst.* 22(4):583–637, 2000.
doi:http://doi.acm.org/10.1145/363911.363914

[Asz07] M. Asztalos. Comparison of Termination Criteria for Graph Transformation Systems. In *Automation and Applied Computer Science Workshop (AACS)*. Budapest, Hungary, 2007.

[BHP06] P. Bottoni, K. Hoffmann, F. P. Presicce. Termination of Algebraic Rewriting with Inhibitors. In Karsai and Taentzer (eds.), *Proc. GraMoT 2006*. ECEASST 4. 2006.

[BHPT05] P. Bottoni, K. Hoffmann, F. Parisi-Presicce, G. Taentzer. High-Level Replacement Units and their Termination Properties. *Journal of Visual Languages and Computing* 16:485–507, 2005.

[Der87] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation* 3(1& 2):69–115, 1987. Corrigendum: 4,3 (Dec. 1987), 409-410.

[DM79] N. Dershowitz, Z. Manna. Proving termination with multiset orderings. *Commun. ACM* 22(8):465–476, 1979.
doi:http://doi.acm.org/10.1145/359138.359142

[DSH$^+$06] D.Varro, S.Varro-Gyapay, H.Ehrig, U.Prange, G. Taentzer. Termination analysis of Model transformations by Petri Nets. In *Proc. ICGT 2006*. LNCS 4178, pp. 260–274. Springer, 2006.

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

[EL93] H. Ehrig, M. Löwe. Parallel and distributed derivations in the single-pushout approach. *Theor. Comput. Sci.* 109(1-2):123–143, 1993. doi:http://dx.doi.org/10.1016/0304-3975(93)90066-3

[EPPH06] H. Ehrig, J. Padberg, U. Prange, A. Habel. Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation. *Fundam. Inf.* 74(1):1–29, 2006.

[HKJ$^+$06] H.Ehrig, K.Ehrig, J.deLara, G. Taentzer, D.Varro, S.Varro-Gyapay. Termination Criteria for Model transformation. In *Proc. FASE 2005*. LNCS 3442, pp. 49–63. Springer, 2006.

[KHE03] J. M. Küster, R. Heckel, G. Engels. Defining and validating transformations of UML models. In *Proc. HCC 2003*. Pp. 145–152. IEEE Computer Society, 2003.

[Plu98] D. Plump. Termination of graph rewriting is undecidable. *Fundamenta Informaticae* 33(2):201–209, 1998.

Formal Modeling and Analysis of Flexible Processes using Reconfigurable Systems

K. Hoffmann[1*], **T. Modica**[2], **J. Padberg**[1]

Hochschule für angewandte Wissenschaften, Hamburg[1]
Technische Universität Berlin, Germany[2]

Abstract: In emergency scenarios we can obtain a more effective coordination among team members constituting a mobile ad hoc network (MANET) through the use of reconfigurable systems. This means that cooperative work can be adequately modeled by low level and high level Petri nets with initial markings and the net structure can be adapted to new requirements of the environment during run time by a set of rules. In this paper we give main requirements for flexible processes in MANETs and show how to realize them using the formal notions of reconfigurable systems. The main part presents a case study in the area of emergency management and demonstrates the advantages of our approach which allows the dynamic adaptation of processes in mobile environments. In this context we also discuss the main results achieved for reconfigurable systems and outline some interesting aspects of future work.

Keywords: mobile ad hoc network, reconfigurable system, Petri net, rule based transformation, algebraic higher order net

1 Introduction

As the adaptation of systems to changing environments gets more and more important processes that can be modified at run time have become a significant topic in the recent years especially in the area of mobile ad hoc networks (MANETs). MANETs are networks of mobile devices that communicate with one another via wireless links without relying on an underlying infrastructure e.g. as in emergency/disaster scenarios where an effective coordination is crucial among teams and team members to stabilize the situation and reduce the probability of secondary damage as well as to provide emergency assistance for victims.

As noticed in the context of the research project WORKPAD[1] the situation in such scenarios is complicated by the fact that the common goal is reached by different teams belonging to different organizations. Moreover each team member should carry on specific activities while the different teams collaborate through the interleaving of all the different processes. Normally processes in mobile environments are not fixed once and for all at build time but constantly adapted at run time e.g. to predict situations of disconnection or to restructure specific parts and activities.

For the effective coordination among teams and team members a suitable process definition language is desirable that supports an adequate modeling of processes and their modifications.

[*] This work has been partly funded by the research project forMA$_l$NET (see tfs.cs.tu-berlin.de/formalnet/) of the German Research Council.

[1] www.workpad-project.eu

But as recognized e.g. in the context of the graduate school METRIK[2] the workflow oriented view on processes in emergency/disaster scenarios is a novel line of research and up to now there exists only a few approaches especially designed for such an application area.

In [HEM05, PHE^{+}07, EHPP07, EKPE07, BHP07] the rule based approach of reconfigurable place/transition (P/T) systems is introduced, so that the modification of processes is realized at run time by a set of rules. The formalism of algebraic higher order systems follows the paradigm ”nets and rules as tokens” and represents a meta model for reconfigurable P/T systems where process execution and process modification is distinguished by the use of specific transitions.

This paper is organized as follows: in Section 2 we give a characterization of main requirements for flexible processes in emergency/disaster scenarios in order to review the formal notions and results of reconfigurable systems in Section 3 and compare them with the listed requirements. To demonstrate the advantages of our approach we illustrate in Section 4 reconfigurable systems by a case study in the area of pipeline emergencies. Finally in Section 5 we conclude with a discussion of some interesting aspects of future work.

2 Flexible Processes in MANETs

This section presents a characterization of main requirements for flexible processes in emergency/disaster scenarios. Based on the fundamental requirements for process definition languages called perspective in [AWW03] these perspectives are improved to fit in our intended application area. Summarizing a process definition language should cover the process perspective, informational perspective, organizational perspective, functional perspective, and operational perspective [AWW03].

The *process perspective* concentrates on the control flow, i.e. the start conditions and the order of activities that have to be executed. The Workflow Management Coalition[3] identifies some basic types of relationship between activities: sequential, parallel, conditional, and iterative routing. Following the approach in [KFP06] in a completely decentralized system as in MANETs each activity could be in addition in one of the following states :

- Received: a start conditions has arrived from the previous team member and is waiting until all conditions are true and the current team member is available to start running it.
- Initiated: a new process instance has just started, this is where the team member starts it because all start conditions are true.
- Running: the team member is running the activity.
- Aborted: the team member failed to complete the activity either because the team member is disconnected or for any other reason.
- Completed: the team member completed the activity.
- On-Hold: the activity is completed but the next team member is not available yet to receive his/her start conditions.
- Rejected: the team member rejects to complete the given activity.

[2] metrik.informatik.hu-berlin.de/grk-wiki
[3] www.wfmc.org

Moreover in a mobile environment movement activities concerning the network connectivity can be separated from activities concerning the intended process.

The *informational perspective* concentrates on the data flow, so that data dependencies between activities are characterized by input and output parameters. On the one hand control data is used for process management purposes and on the other hand production data subsumes information objects like documents, questionnaires and forms. In MANETs information about the geographic area is especially important e.g. to localize positions of team members or to predict situations of disconnection.

The *organizational perspective* is typically defined by roles, groups and other artifacts clarifying organizational issues. Because in emergency/disaster scenarios different teams belong to different organizations, the inter-organizational aspect should be respected. In addition, in MANETs the network topology typically represented as topology graphs [AZ03] both influences and is influenced by the process.

The *functional perspective* prescribes the decomposition of a process into smaller units often represented by a hierarchical structure.

Finally, the *operational perspective* depends on the technical environment, so that elementary operations are performed by resources and applications. Based on the observation in [KFP06] in a mobile environment the team member can be on line, i.e. he/she can receive new work, or off line, where the team member is not available to receive new work. In this case new activities may be on hold until the team member returns on line or even allocated to alternative team members. Team members before permanently leave may notify this otherwise the team leader may decide to treat any other team member failing to respond as permanent. For activities where the team member is temporarily off line, the execution of the process will continue, if possible. In this case when the team member returns some synchronization may be required or alternatively the execution will have to wait until the team member returns.

From a practical point of view processes in MANETs often have to be restructured e.g. because of unforeseen events or to maintain the network connectivity resulting in a highly dynamical modification of processes. In [AWW03] three issues to dynamic change of processes are addressed. By *constrained flexibility* certain properties should be preserved during process adaption while *instance change* refers to the modification of process instances at run time. Finally *instance migration* are based on simultaneous changes of both process schemes and process instances.

In addition dynamic changes are grouped into ad hoc changes, i.e. changes are responses to unforeseen exceptions, and pre-planned and evolutionary changes, i.e. changes are known at build time (see e.g [AWW03, RRD04]). Besides others in [SMO00, Ros07] a minimal set of change operations are characterized:

- inserting a new activity where also bridging actions may be used to keep network connectivity,
- removing an existing activity,
- modifying the order of activities, and
- modifying activity properties like data requirements, underlying applications, temporal constraints, resource allocation, or reassignment of activities from one team or member to another.

Processes have to be analysed (see e.g. [AWW03]) for *verification* purposes, so that some form of correctness criteria, i.e. different properties on a syntactical and/or semantical level, has to be satisfied and can be checked. In contrast *validation* verifies processes with respect to the intended and typically informally formalised process and *performance analysis* is realized by simulating processes to detect e.g. potential deadlocks or livelocks.

.

3 Reconfigurable Systems

In this section we compare reconfigurable systems with the requirements listed in the last section and present the results achieved for reconfigurable place/transition (P/T) systems in [HEM05, PHE$^+$07, EHPP07, EKPE07, BHP07].

A P/T system is a P/T net with an initial marking. P/T nets, P/T systems and their variants are an established process definition language (see e.g. [Ell79, vdA03]) providing constructs of the process perspective. While P/T nets represent process schemes, P/T systems describe the behavior of process instances due to their initial markings. Activities are modeled by transitions while the control flow is reflected by arcs between places and transitions. Places can be seen as pre and post conditions for activities and source places with an empty pre domain can be used as start condition for the process. The Workflow Patterns Initiative[4] [AHKB00] presents a number of patterns for the relationship between activities following not only the basic types identified by the Workflow Management Coalition but also more advanced constructs.

The concept of reconfigurable P/T systems was introduced for modeling changes of the net structure by rule based transformations while the system is kept running. For rule based transformations of P/T systems we use the framework of net transformations [EEPT06, EHPP07] following the double pushout (DPO) approach of graph transformation systems [Roz97]. The basic idea behind net transformation is the stepwise development of P/T systems by given rules. Think of these rules as replacement systems where the left hand side is replaced by the right hand side while preserving a context. In reconfigurable P/T systems not only the follower marking can be computed but also the net structure can be changed by rule applications and we obtain new P/T systems that are more appropriate with respect to some requirements of the environment. In detail a reconfigurable P/T system $((PN_1,M_1),RULES)$ consists of a P/T system (PN_1,M_1), where PN_1 is a P/T net with initial marking M_1, and a set of rules $RULES$.

Rules and transformations in the DPO approach are based on morphisms preserving on the one hand firing steps and requiring on the other hand that the initial marking at corresponding places is increasing or even stronger. An application of a rule is called a transformation step and describes how an object is actually changed by the rule. In general a rule $prod = ((L,M_L) \xleftarrow{l} (K,M_K) \xrightarrow{r} (R,M_R))$ is given by three P/T systems called left hand side, interface and right hand side, respectively, and a span of two P/T morphisms l and r. We additionally need a match morphism $(L,M_L) \xrightarrow{m} (PN_1,M_1)$ that identifies the relevant parts of the left hand side (L,M_L) in the P/T system (PN_1,M_1). Now a direct transformation $(PN_1,M_1) \overset{(prod,m)}{\Longrightarrow} (PN_2,M_2)$ via $prod \in RULES$ and m can be constructed in two steps. We delete in a first step those elements

[4] www.workflowpatterns.com

from (PN_1, M_1) which are identified by the match m but not preserved by the interface (K, M_K) leading to the intermediate P/T system (PN_0, M_0). In a second step we glue together the P/T systems (PN_0, M_0) and (R, M_R) along the interface resulting in the new P/T system (PN_2, M_2).

The DPO approach does not allow the treatment of unmatched transitions at places which should be deleted. In this case the so called gluing condition forbids the application of rules. Furthermore items which are identified by a non injective match must be preserved by rule applications. Note that a positive check of the gluing condition makes sure that the intermediate P/T system is well defined.

The rule based approach of reconfigurable P/T systems supports dynamic changes in the sense that the concept of instance change is formalised by the application of appropriate rules realising the insertion of new activities, removing of existing activities or changing the order of activities. Because rules are fixed at build time the concept of reconfigurable P/T system supports pre-planned and evolutionary changes. To support constraint flexibility the set of rules can be restricted to property preserving rules [PU03], so that safety and liveness properties are preserved by rule applications.

The main result in [EHPP07] concerns the formal foundation for transformations of P/T systems based on the framework of adhesive high level replacement (HLR) systems [EEPT06, EHPP06]. Adhesive HLR systems have been recently introduced as a new categorical framework for graph transformation in the DPO approach. They combine the well known framework of HLR systems with the framework of adhesive categories introduced in [LS05]. The main concept behind adhesive categories are the so called van Kampen squares. These ensure that pushouts along monomorphisms are stable under pullbacks and, vice versa, that pullbacks are stable under combined pushouts and pullbacks. Note that a pushout can be seen as a gluing construction of two objects over a specific interface, while a pullback is dual to a pushout in the sense that a pullback construction extracts the common part of two objects. In the case of adhesive HLR categories the class of all monomorphisms is replaced by a subclass of monomorphisms closed under composition and decomposition.

Within the framework of adhesive HLR systems there are many interesting results concerning the applicability of rules, the embedding and extension of transformations, parallel and sequential dependence and independence, and concurrency of rule applications. The concept of parallel independence states that two transformation steps are not in conflict while two consecutive transformation steps are sequentially independent if they are not causally dependent. Provided that the relevant conditions are satisfied two alternative transformation steps may be swapped and each of them can still be applied after the other has been performed. Since we have shown in [EHPP07] that P/T systems form a weak adhesive HLR category, we can apply these results to reconfigurable P/T systems.

Based on the observation of parallel and sequential independence of rule applications the main results in [EKPE07] deals with conflict situations between transformation and token firing. The traditional concurrency situation in P/T systems without capacities is that two transitions with overlapping pre domain are both enabled and together require more tokens than available in the current marking. As P/T systems can evolve in two different ways the notions of conflict and concurrency become more complex. Assume that a given P/T system represents a certain system state. The next evolution step can be obtained not only by token firing but also by the application of one of the rules available. Hence the question arises whether each of these evolution steps

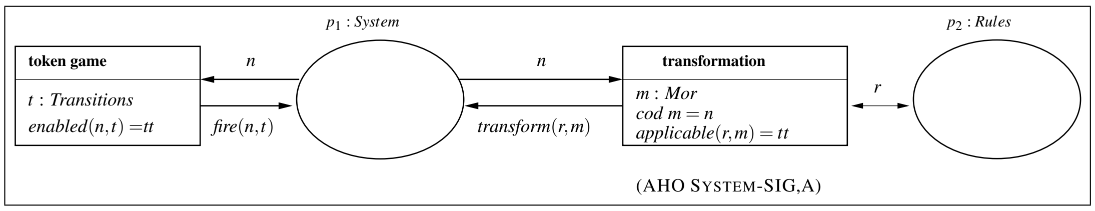


Figure 1: Algebraic higher order system

can be postponed after the realization of the other, yielding the same result, and if they can be performed in a different order without changing the result.

In [EKPE07] we have presented conditions for (co-)parallel and sequential independence and we have shown that in specific cases firing and transformation steps can be performed in any order, yielding the same result. We have correlated these conditions, i.e. that parallel independence implies sequential independence and, vice versa, sequential (coparallel) independence implies parallel and coparallel (parallel and sequential) independence. The advantage of the presented conditions is that they could be checked at a syntactical and local level instead of semantical and global one. Thus they are also applicable in the case of complex reconfigurable P/T systems.

In [HEM05] we have introduced the paradigm ”nets and rules as tokens” by a high level model with suitable data type part. The model called algebraic higher order (AHO) system exploits some form of control not only on rule application but also on token firing. In general an AHO system is defined by an algebraic high level net [PER95] with system places and rule places as for example shown in Fig. 1 where a marking can be given by suitable P/T systems and rules, respectively, on these places. For a detailed description of the data type part, i.e. the AHO SYSTEM-signature and corresponding algebra A, we refer to [HEM05].

In the following we review the behavior of AHO systems according to [HEM05]. With the symbol $Var(t)$ we indicate the set of variables of a transition t, i.e. the set of all variables occurring in pre- and post domain and in the firing condition of t. The marking M determines the distribution of P/T systems and rules in an AHO system which are elements of a given higher order algebra A. Intuitively P/T systems and rules can be moved along AHO system arcs and can be modified during the firing of transitions. The follower marking is computed by the evaluation of net inscriptions in a variable assignment $v : Var(t) \to A$. The transition t is enabled in a marking M, if and only if (t,v) is consistent, that is if the evaluation of the firing condition is fulfilled. Then the follower marking after firing of transition t is defined by removing tokens corresponding to the net inscription in the pre domain of t and adding tokens corresponding to the net inscription in the post domain of t.

The transitions in the AHO system in Fig. 1 realize on the one hand firing steps and on the other hand transformation steps as indicated by the net inscriptions $fire(n,t)$ and $transform(r,m)$, respectively. To compute the follower marking of P/T systems we use the transition *token game* of the AHO system while the transition *transformation* is provided for changing the structure of P/T systems. In this way process execution and process modification is distinguished by these two transitions.

The pair (or sequence) of firing and transformation steps discussed in [EKPE07] is reflected by firing of the transitions one after the other in our AHO system. Thus these results are most important for the analysis of AHO systems.

Using P/T systems as tokens AHO systems focus on the process perspective. To integrate the informational perspective we can use high level nets as tokens themselves, i.e. the data type part is extended by algebraic high level nets and corresponding rules. Analogously the organizational and operational perspectives can be added following e.g. the approach in [AW01]. So activity properties like data requirements and the reassignment of activities from one team member to another can be modified by the applications of suitable rules. For the functional perspective the formalism of AHO systems can be adapted using the hierarchy concept of Coloured Petri Nets (see [Jen96]).

To consider ad hoc changes of processes the modification of rule tokens requires an extension not only of the data type part but also of the net structure as introduced in [HPM05], so that the definition of new rules by reusing existing rules is supported at run time by different operations like inheritance [PP01].

While the AHO system in Fig. 1 deals with one layer for reconfigurable P/T systems, in [PHE$^+$07] we follow the observation that processes in MANETs consists of different aspects. Thus we separate movement activities from general activities and allow a local view of team members. This leads to an AHO system with different layers each of them equipped with its own P/T system and set of rules. Moreover the notion of layer consistent environment states that the views in each layer fit together realizing one form of instance migration. In [BHP07] we extend this approach to allow the introduction of new team members by more advanced changes at each layer.

Because reconfigurable P/T systems and AHO systems are formalized on a rigorous mathematical foundation and have a clear formal semantics, several results as described above are provided to analyse systems in the sense of formal verification. These results present a line of research and there is a large amount of most interesting and relevant open questions directly related to the work presented here. We plan to develop a tool to support simulation and analysis aspects for our approach. For the application of net transformation rules this tool will provide an export to AGG[5], a graph transformation engine as well as a tool for the analysis of graph transformation properties like termination and rule independence. Furthermore the token net properties could be analyzed using the Petri Net Kernel [KW01], a tool infrastructure supporting different Petri net classes.

4 Emergency/Disaster Scenario

In this section we illustrate the main idea of reconfigurable systems by a case study of a pipeline emergency scenario where an unknown source of a natural gas leak is detected in a residential area[6]: A postal worker delivering mail in a residential street smells a strong odor of gas. She immediately notifies the fire department. A single engine company is dispatched by the fire department with four firefighters leaded by one company officer. At the scene the postal worker meets the company officer and describes the problem. He calls the gas company and requests an additional law enforcement officers to control traffic into the area. While three firefighters

[5] tfs.cs.tu-berlin.de/agg

[6] www.pipelineemergencies.com

evacuate the homes in the immediate area and afterwards deny entry to this area, another one reads the gas indicator and detects that the gas is highest in front of a home located on 114 Maple Street. After electricity and gas lines are shut off to each home the fire department stand by with fully charged hose lines and wait for the arrival of the gas company.

The cooperative process enacted by the firefighter company is depicted as P/T system (PN_1,M_1) in Fig. 2. To start the activities of the firefighter team the follower marking of the P/T system (PN_1,M_1) is computed by firing the *and-split*-transition and we obtain the new P/T system (PN_1,M_1') in Fig. 3.

Next we focus on dynamic changes while the process is running. The three firefighters responsible for the evacuation process need more detailed information how to proceed. So the company officer gives the instruction that first of all the residents are notified of the evacuation. Afterwards the firefighters should assist handicapped persons and guide all of them to the extend possible. To introduce the refinement of the *Evacuate homes*-transition into the P/T system (PN_1,M_1') we provide the rule $prod_{evacuate}$ in Fig. 4. The marking M_{L_1} of the P/T system in the left hand side of $prod_{evacuate}$ demands that the evacuation process is not yet started because there is one token in the pre domain of the *Evacuate homes*-transition. The application of the rule is given as follows: the match morphism m_1 is given by the obvious inclusion and identifies the relevant parts of the left hand side (L_1,M_{L_1}) of rule $prod_{evacuate}$ in (PN_1,M_1'); next, the *Evacuate homes*-transition is deleted and we obtain an intermediate P/T system (PN_0,M_0); then, the transitions *Notify residents*, *Assist handicapped persons* and *Guide persons* together with their (new) environment are added leading to the P/T system (PN_2,M_2) in Fig. 5. Thus we obtain the transformation step $(PN_1,M_1') \overset{(prod_{evacuate},m_1)}{\Longrightarrow} (PN_2,M_2)$. Afterwards the firefighter company proceed with their activities and we obtain the P/T system (PN_2,M_2') in Fig. 6 by firing the corresponding transitions.

After the problem identification the odor of gas grows stronger and the firefighter takes an additional reading of the gas indicator and informs the company officer about the result, so that the company officer is able to determine if the atmosphere in the area is safe, unsafe, or dangerous. To extend our process by these additional activities we use the rule $prod_{analyse}$ in Fig. 7 where the marking M_{L_2} in the left hand side indicates that the problem location is identified. By the application of the rule we obtain the transformation step $(PN_2,M_2') \overset{(prod_{analyse},m_2)}{\Longrightarrow} (PN_3,M_3)$ where the new P/T system (PN_3,M_3) is depicted in Fig. 8.

Based on the additional results of the gas indicator the company officer analyses that the atmosphere in this area is over the lower explosive limit and thereby more dangerous than expected. He determines that the best course of action is to call for additional resources to maintain the isolation perimeter and expand the area of evacuation as a precaution. So, in a next step the follower marking of the P/T system (PN_3,M_3) is computed by firing the *Additional reading*- and *Analyse*-transitions leading to the P/T system (PN_3,M_3') in Fig. 9. Afterwards the rule $prod_{expand}$ depicted in Fig. 10 is applied to the P/T system (PN_3,M_3') resulting in the new P/T system (PN_4,M_4) in Fig. 11.

Summarizing, at the beginning our reconfigurable P/T system consists of the P/T system (PN_1,M_1) in Fig. 2 and the set of rules depicted in Figs. 4, 7 and 10. Let the reconfigurable P/T system be the initial marking of the AHO system in Fig. 1, i.e. the P/T system (PN_1,M_1) is on the place p_1 while the marking of the place p_2 is given by the set of rules. To compute the fol-

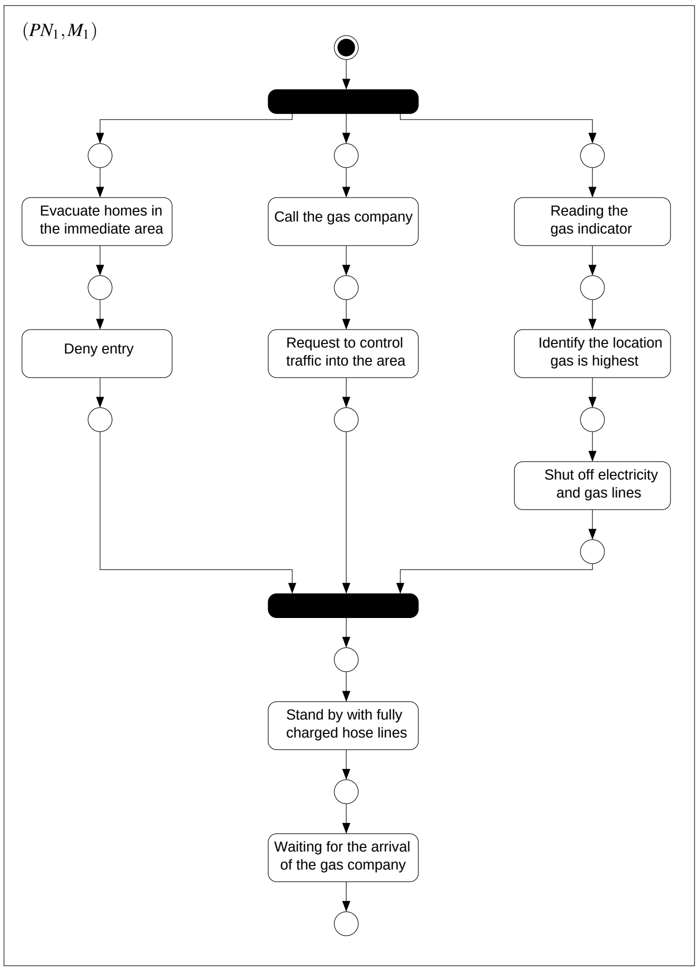

(PN_1, M_1)
Evacuate homes in the immediate area
Call the gas company
Reading the gas indicator
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Shut off electricity and gas lines
Stand by with fully charged hose lines
Waiting for the arrival of the gas company

Figure 2: Process (PN_1, M_1)

lower marking of the P/T system we use the transition *token game* of the AHO system. First the variable n is assigned to the P/T system (PN_1, M_1) and the variable t to the *and-split*-transition that is enabled, so that the firing condition is fulfilled. Due to the evaluation of the term $fire(n,t)$ we obtain the new P/T system (PN_1, M_1') in Fig. 2.

For changing the structure of P/T systems the transition *transformation* is provided in Fig. 1. Again we have to give an assignment v for the variables of this transition, i.e. variables n, m and r, where $v(n) = (PN_1, M_1')$, $v(m) = m_1$ is a suitable match morphism and $v(r) = prod_{evacuate}$ (see Fig. 4). The firing condition $cod\ m = n$ ensures that the codomain of the match morphism is equal to (PN_1, M_1') while the second condition $applicable(r,m)$ checks the gluing condition, i.e. if the rule $prod_{evacuate}$ is applicable with match m_1. Afterwards the transformation step is computed by the evaluation of the net inscription $transform(r,m)$ and the effect of firing the transition *transformation* is the removal of the P/T system (PN_1, M_1') from place p_1 and adding the P/T system (PN_2, M_2) in Fig. 5 to it.

Analogously we proceed with the computation of the follower markings and dynamic adaption of our process as described above. After several firing steps of the transitions *token game* and *transformation* we obtain the reconfigurable P/T system consisting of the P/T system (PN_4, M_4) (see Fig. 11) and the original set of rules.

To analyse the reconfigurable P/T systems we apply the results presented in [EKPE07] and described in the previous section. For example the transformation step $(PN_1, M_1') \overset{(prod_{evacuate}, m_1)}{\Longrightarrow} (PN_2, M_2)$ is parallel independent of the firing step given by the *Reading gas indicator*-transition because the transition is not deleted by the transformation step and the marking of the P/T system (PN_1, M_1') is unchanged by the application of the rule $prod_{evacuate}$. Moreover the pair of transformation and firing steps is sequentially independent because the *Reading gas indicator*-transition is not created by the transformation step. Thus the pair of steps may be swapped and each of them can be applied after the other has been performed leading to the same result.

In the context of our AHO system in Fig. 1 this observation is reflected by an independent firing of the transitions *token game* and *transformation*, i.e. the sequential firing of these transitions leading to the same result independent of the order these transitions are fired.

The pair of consecutive steps given by firing the *and-split*-transition in (PN_1, M_1) and the transformation $(PN_1, M_1') \overset{(prod_{evacuate}, m_1)}{\Longrightarrow} (PN_2, M_2)$ is sequentially dependent because the marking of the left hand side of $prod_{evacuate}$ demands a token in the pre domain of the *Evacuate homes*-transition.

Further situations of independent and dependent firing and transformation steps are illustrated in Fig. 12 where, however, the traditional concurrency situation of transitions and transformations, respectively, is not shown. Note that e.g. the two consecutive transformations $(PN_1, M_1^2) \overset{(prod_{evacuate}, m_1)}{\Longrightarrow} (PN_2, M_2^2)$ and $(PN_2, M_2^2) \overset{(prod_{analyse}, m_2)}{\Longrightarrow} (PN_3, M_3^2)$ are sequentially independent because the overlapping of the right hand side of $prod_{evacuate}$ and the left hand side of $prod_{analyse}$ in (PN_2, M_2^2) is included in the intersection of the interfaces.

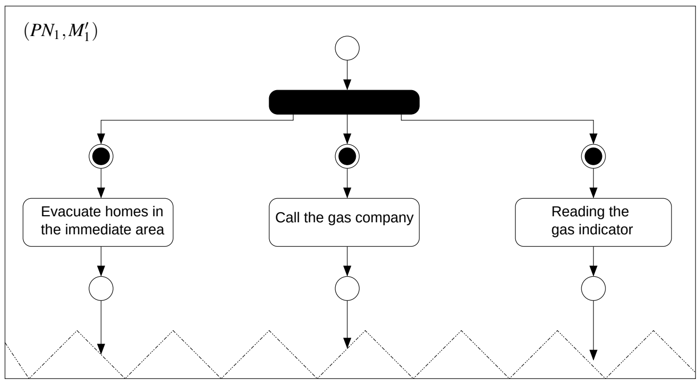

(PN_1, M'_1)
Evacuate homes in the immediate area
Call the gas company
Reading the gas indicator

Figure 3: Relevant part of process (PN_1, M'_1)

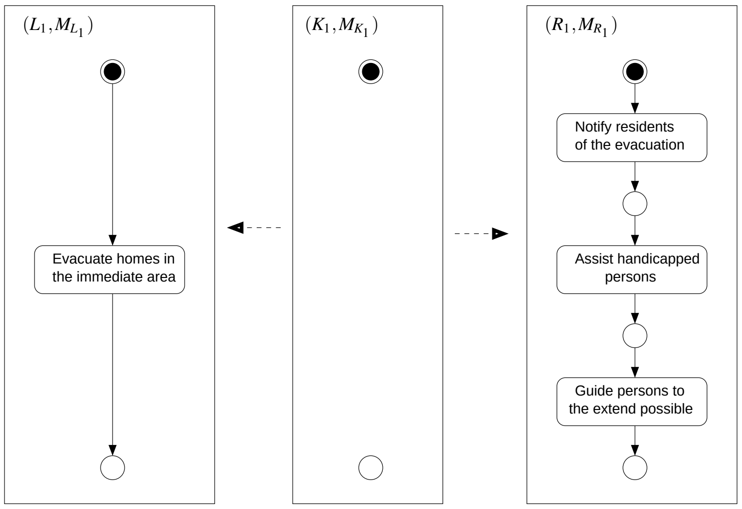

(L_1, M_{L_1})
Evacuate homes in the immediate area
(K_1, M_{K_1})
(R_1, M_{R_1})
Notify residents of the evacuation
Assist handicapped persons
Guide persons to the extend possible

Figure 4: Rule $prod_{evacuate}$

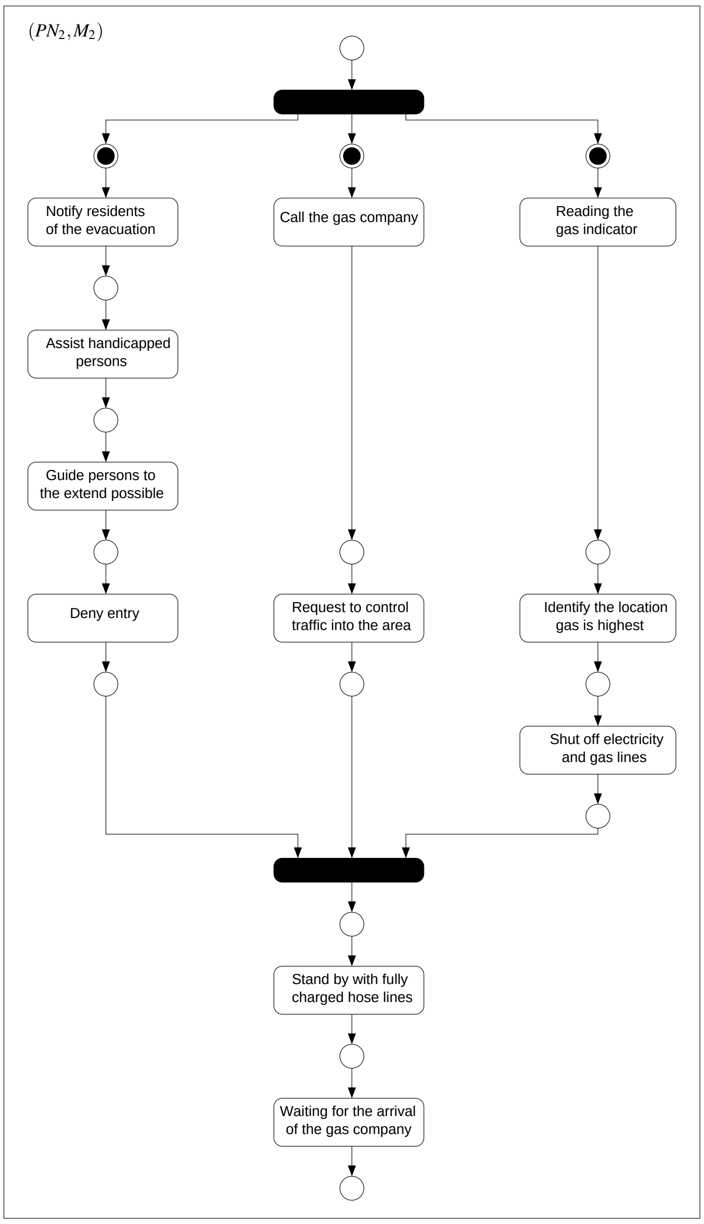

(PN_2, M_2)
Notify residents of the evacuation
Call the gas company
Reading the gas indicator
Assist handicapped persons
Guide persons to the extend possible
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Shut off electricity and gas lines
Stand by with fully charged hose lines
Waiting for the arrival of the gas company

Figure 5: Process (PN_2, M_2)

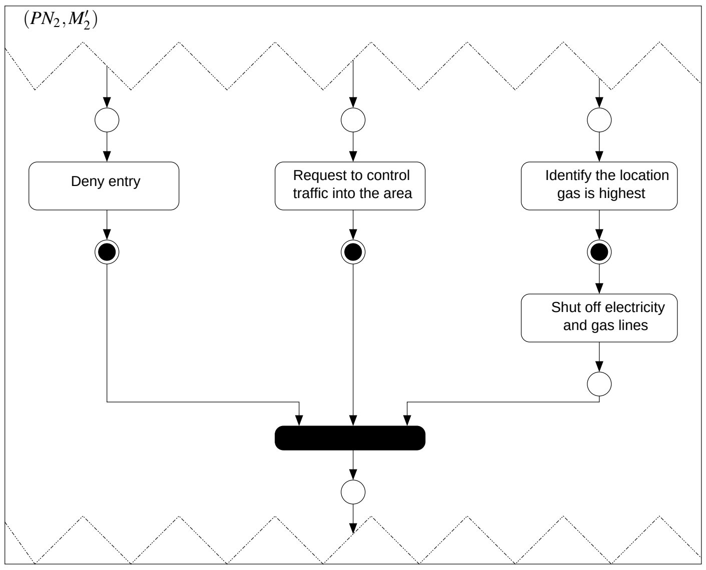

(PN_2, M'_2)
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Shut off electricity and gas lines

Figure 6: Relevant part of process (PN_2, M'_2)

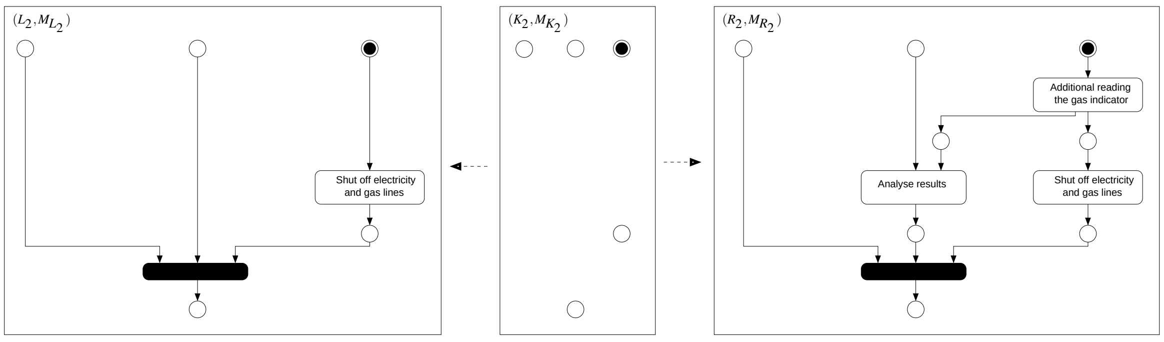

(L_2, M_{L_2})
Shut off electricity and gas lines
(K_2, M_{K_2})
(R_2, M_{R_2})
Additional reading the gas indicator
Analyse results
Shut off electricity and gas lines

Figure 7: Rule $prod_{analyse}$

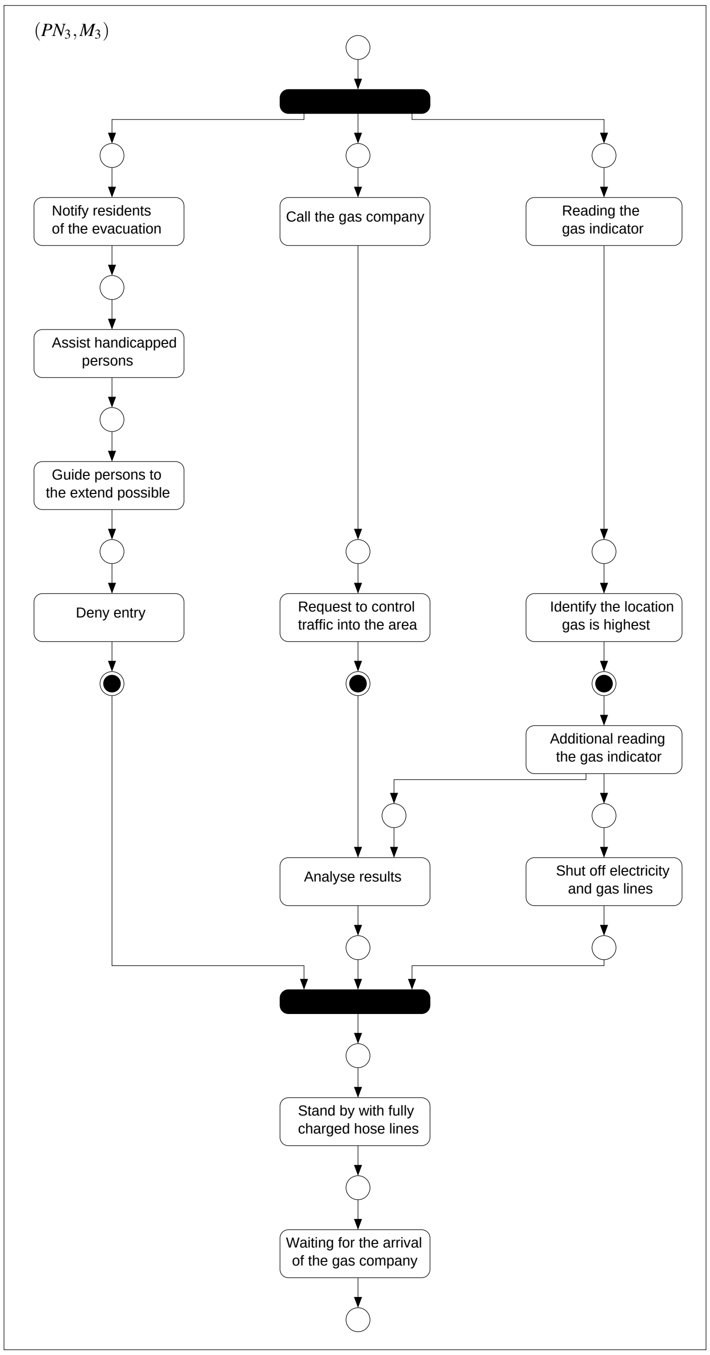

(PN_3, M_3)
Notify residents of the evacuation
Call the gas company
Reading the gas indicator
Assist handicapped persons
Guide persons to the extend possible
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Additional reading the gas indicator
Analyse results
Shut off electricity and gas lines
Stand by with fully charged hose lines
Waiting for the arrival of the gas company

Figure 8: Process (PN_3, M_3)

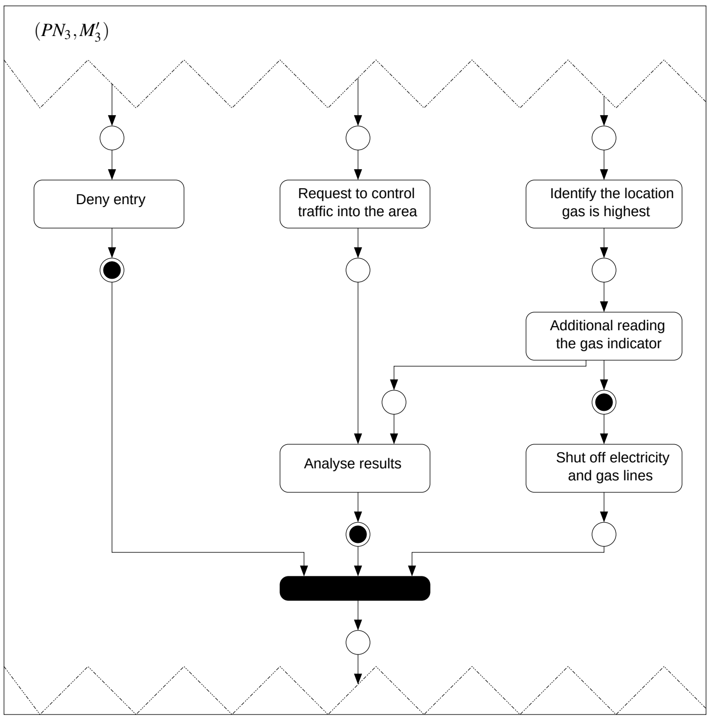

(PN_3, M'_3)
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Additional reading the gas indicator
Analyse results
Shut off electricity and gas lines

Figure 9: Relevant part of process (PN_3, M'_3)

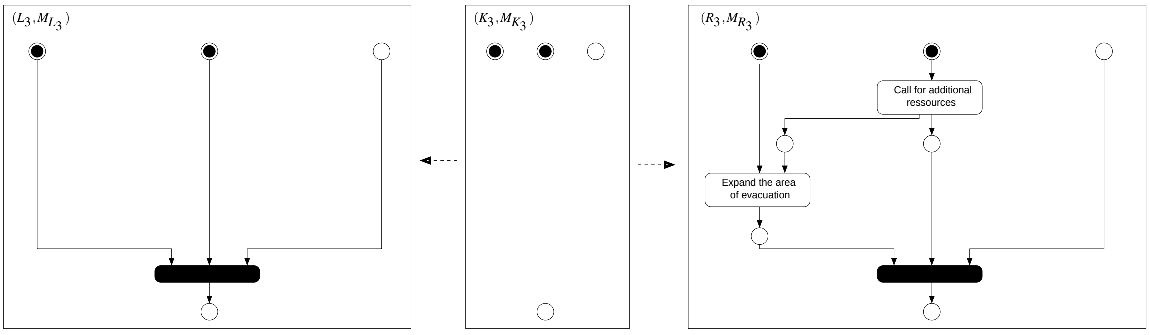
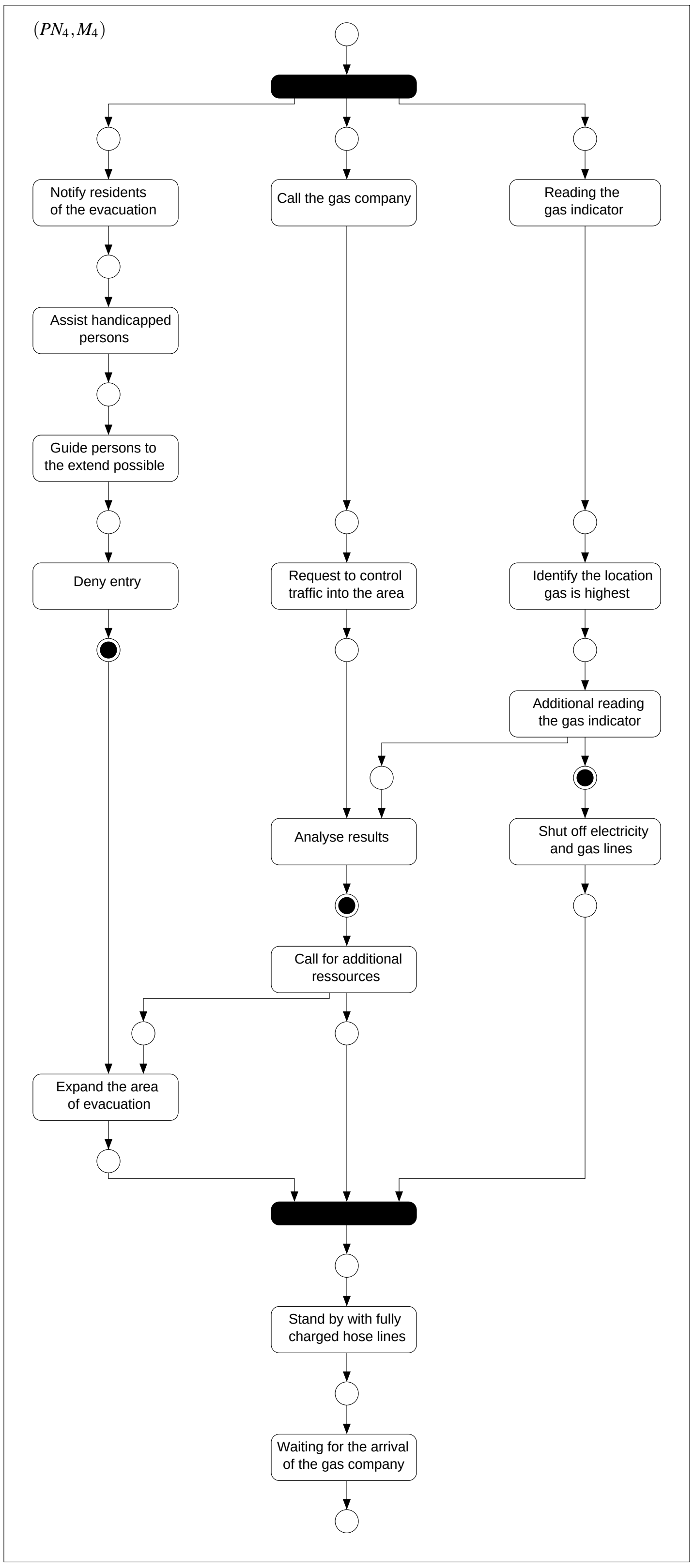


Figure 10: Rule $prod_{expand}$


(PN_4, M_4)
Notify residents of the evacuation
Call the gas company
Reading the gas indicator
Assist handicapped persons
Guide persons to the extend possible
Deny entry
Request to control traffic into the area
Identify the location gas is highest
Additional reading the gas indicator
Analyse results
Shut off electricity and gas lines
Call for additional ressources
Expand the area of evacuation
Stand by with fully charged hose lines
Waiting for the arrival of the gas company

$$\begin{tikzcd}
(PN_1,M_1) \arrow[d, "\mathit{and\text{-}split}"'] \arrow[dr, dotted, dash, "\mathit{dependent}"] & & & \\
(PN_1,M_1') \arrow[r, Rightarrow, "(prod_{evacuate},m_1)"] \arrow[d, "\mathit{Reading\ gas\ indicator}"'] & (PN_2,M_2) \arrow[d, "\mathit{Reading\ gas\ indicator}"] & & \\
(PN_2,M_1^1) \arrow[r, Rightarrow, "(prod_{evacuate},m_1)"] \arrow[d, "\mathit{Identify\ the\ location}"'] & (PN_2,M_2^1) \arrow[d, "\mathit{Identify\ the\ location}"] \arrow[dr, dotted, dash, "\mathit{dependent}"] & & \\
(PN_1,M_1^2) \arrow[r, Rightarrow, "(prod_{evacuate},m_1)"] \arrow[d, "\mathit{Call\ the\ gas\ company}"'] & (PN_2,M_2^2) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Call\ the\ gas\ company}"] & (PN_3,M_3^2) \arrow[d, "\mathit{Call\ the\ gas\ company}"] & \\
(PN_1,M_1^3) \arrow[r, Rightarrow, "(prod_{evacuate},m_1)"] \arrow[d, "\mathit{Request\ to\ control\ traffic}"'] & (PN_2,M_2^3) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Request\ to\ control\ traffic}"] & (PN_3,M_3^3) \arrow[d, "\mathit{Request\ to\ control\ traffic}"] & \\
(PN_1,M_1^4) \arrow[r, Rightarrow, "(prod_{evacuate},m_1)"] \arrow[dr, dotted, dash, "\mathit{dependent}"'] & (PN_2,M_2^4) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Notify\ residents}"'] & (PN_3,M_3^4) \arrow[d, "\mathit{Notify\ residents}"] & \\
& (PN_2,M_2^5) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Assist\ handicapped\ persons}"'] & (PN_3,M_3^5) \arrow[d, "\mathit{Assist\ handicapped\ persons}"] & \\
& (PN_2,M_2^6) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Guide\ persons}"'] & (PN_3,M_3^6) \arrow[d, "\mathit{Guide\ persons}"] & \\
& (PN_2,M_2^7) \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[d, "\mathit{Deny\ entry}"'] & (PN_3,M_3^7) \arrow[d, "\mathit{Deny\ entry}"] & \\
& (PN_2,M_2') \arrow[r, Rightarrow, "(prod_{analyse},m_2)"] \arrow[dr, dotted, dash, "\mathit{dependent}"'] & (PN_3,M_3) \arrow[d, "\mathit{Additional\ reading}"'] & \\
& & (PN_3,M_3^1) \arrow[d, "\mathit{Analyse\ results}"'] \arrow[dr, dotted, dash, "\mathit{dependent}"] & \\
& & (PN_3,M_3') \arrow[r, Rightarrow, "(prod_{expand},m_3)"] & (PN_4,M_4)
\end{tikzcd}$$

Figure 12: Independence and dependence of Firing and transformation steps

5 Conclusion

In this paper we have given main requirements for flexible processes in emergency/disaster scenarios in order to show that most of them are realized by reconfigurable systems, a rule based formalism based on the one hand on low level and high level Petri nets with a suitable marking and on the other hand on the categorical framework of weak adhesive high level replacement systems. As future work, it would be important to investigate and verify additional requirements necessary for flexible processes in emergency/disaster scenarios and mobile environments.

The main part of this paper presents the case study in the area of pipeline emergencies where dynamic changes of the process are realised at run time by rule applications to express the refinement and insertion of activities. Note that our processes focus on the intended activities and exclude movement activities because the network connectivity is assured due to the limited perimeter of the affected area and the use of cell phones and radio devices. Nevertheless, the scenario could be extended in such a way that the problem is located beyond the range of these equipment and several team members have to follow other ones to avoid a situation of disconnection.

One aspect of future work is integration of the informational and organizational perspectives into our formalism because within our case study these aspects become most relevant. In fact process modifications in our case study depend on the exchange of messages and data concerning a detailed instruction of the evacuation process, the results of reading the gas indicator and the final analysis of these results by the company officer. In addition the processes enacted by the gas company and the law enforcement officer have to be taken into account, so that the different teams collaborate through the interleaving of all the different processes to achieve the common goal.

Bibliography

[vdA03] W. van der Aalst. The Application of Petri nets to Workflow Management. *Journal of Circuits, Systems and Computers* 8(1):21–66, 2003.

[AHKB00] W. Van der Aalst, A. ter Hofstede, B. Kiepuszewski, A. Barros. Workflow Patterns. In *Proc. Cooperative Information Systems (CoopIS)*. LNCS 1901, pp. 18–29. Springer, 2000.

[AW01] W. M. P. van der Aalst, M. Weske. The P2P Approach to Interorganizational Workflows. In *Proc. Advanced Information Systems Engineering (CAiSE)*. LNCS 2068, pp. 140–156. Springer, 2001.

[AWW03] W. van der Aalst, M. Weske, G. Wirtz. Advanced Topics in Workflow Management: Issues, Requirements, and Solutions. *Journal of Integrated Design and Process Science* 7(3), 2003.

[AZ03] D. Agrawal, Q. Zeng. *Introduction to Wireless and Mobile Systems*. Thomson Brooks/Cole, 2003.

[BHP07] E. Biermann, K. Hoffmann, J. Padberg. Layered Architecture Consistency for MANETs: Introducing New Team Members. In *Proc. Integrated Design and Process Technology (IDPT)*. 2007.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs in Theoretical Computer Science. Springer, 2006.

[EHPP06] H. Ehrig, A. Habel, J. Padberg, U. Prange. Adhesive High-Level Replacement Systems: A New Categorical Framework for Graph Transformation. *Fundamenta Informaticae* 74(1):1–29, 2006.

[EHPP07] H. Ehrig, K. Hoffmann, U. Prange, J. Padberg. Formal Foundation for the Reconfiguration of Nets. Technical report 2007-01, Technical University Berlin, Fak. IV, 2007.

[EKPE07] H. Ehrig, J. P. K. Hoffmann, U. Prange, C. Ermel. Independence of Net Transformations and Token Firing in Reconfigurable Place/Transition Systems. In *Proc. Application and Theory of Petri Nets (ATPN)*. LNCS 4546, pp. 104–123. Springer, 2007.

[Ell79] C. Ellis. Information Control Nets: A Mathematical Model of Office Information Flow. In *Proc. Simulation, Measurement and Modelling of Computer Systems*. Pp. 225–240. ACM Press, 1979.

[HEM05] K. Hoffmann, H. Ehrig, T. Mossakowski. High-Level Nets with Nets and Rules as Tokens. In *Proc. Application and Theory of Petri Nets (ATPN)*. LNCS 3536, pp. 268–288. Springer, 2005.

[HPM05] K. Hoffmann, F. Parisi-Presicce, T. Mossakowski. Higher-Order Nets for Mobile Policies. In *Workshop on Petri Nets and Graph Transformation (PNGT)*. Electronic Notes in Theoretical Computer Science 127, pp. 87–105. Elsvier, 2005.

[Jen96] K. Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. EATCS Monographs in Theoretical Computer Science. Springer, 1996.

[KFP06] E. Kyriacou, G. Fakas, V. Pavlaki. A Completely Decentralized Workflow Management System for the Support of Emergency Telemedicine and Patient Monitoring. In *Proc. IEEE EMBS Annual International Conference*. 2006.

[KW01] E. Kindler, M. Weber. The Petri Net Kernel - An Infrastructure for Building Petri Net Tools. *Software Tools for Technology Transfer* 3(4):486–497, 2001.

[LS05] S. Lack, P. Sobocinski. Adhesive and Quasiadhesive Categories. *Theoretical Informatics and Applications* 39(5):511–546, 2005.

[PP01] F. Parisi-Presicce. On modifying high level replacement systems. *Electronic Notes in Theoretical Computer Science* 44(2), 2001.

[PER95] J. Padberg, H. Ehrig, L. Ribeiro. Algebraic High-Level Net Transformation Systems. *Mathematical Structures in Computer Science* 5:217–256, 1995.

[PHE$^+$07] J. Padberg, K. Hoffmann, H. Ehrig, T. Modica, E. Biermann, C. Ermel. Maintaining Consistency in Layered Architectures of Mobile Ad-hoc Networks. In *Proc. Fundamental Approaches to Software Engineering (FASE)*. LNCS 4422, pp. 383–397. Springer, 2007.

[PU03] J. Padberg, M. Urbasek. Rule-Based Refinement of Petri Nets: A Survey. In *Advances in Petri nets: Petri Net Technologies for Modeling Communication Based Systems*. Lecture Notes in Computer Science 2472, pp. 161–196. Springer, 2003.

[Ros07] F. D. Rosa. *Adaptive process management in mobile and dynamic scenarios*. PhD thesis, SAPIENZA - Universita di Roma, Department of Computer Science, 2007.

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

[RRD04] S. Rinderle, M. Reichert, P. Dadam. Correctness criteria for dynamic changes in workflow systems - a survey. *Data Knowl. Eng.* 50(1):9–34, 2004.

[SMO00] S. Sadiq, O. Marjanovic, M. Orlowska. Managing change and time in dynamic workflow processes. *Journal of Cooperative Information Systems* 9(12), 2000.

Second-Order Value Numbering

Tiziana Margaria[1], **Bernhard Steffen**[2] **and Christian Topnik**[2]

Chair Service and Software Engineering, University of Potsdam
[1]margaria@cs.uni-potsdam.de,
Chair Programming Systems, TU Dortmund
[2]steffen@cs.uni-dortmund.de

Abstract: We present second-order value numbering, a new optimization method for suppressing redundancy, in a version tailored to the application for optimizing the decision procedure of jMosel, a verification tool set for monadic second-order logic on strings (M2L(Str)). The method extends the well-known concept of value numbering to consider not merely *values*, but *semantics transformers* that can be efficiently pre-computed and help to avoid redundancy at the 2nd-order level. Since decision procedures for M2L are non-elementary, an optimization method like this can have a great impact on the execution time, even though our decision procedure comprises the analysis and optimization time for second-order value numbering. This is illustrated considering a parametric family of hardware circuits, where we observed a performance gain by a factor of 3. This result is surprising, as the design of these circuits exploits already structural similarity.

Keywords: Program Analysis and Optimization, (second order) Value Numbering

1 Introduction

Value numbering is a well-known compiler optimization technique used to efficiently detect and eliminate redundant code by identifying equality of values [CS70, AWZ88]. Considering this (first order) concept there is a natural generalization to second-order (or even higher-order in general): rather than considering just values, one could lift the analysis to second-order by considering *semantics transformers*, which may then be efficiently pre-computed and help to avoid redundancy at the second-order level.

In this paper we introduce second-order value numbering and illustrate its impact by applying it to improve the decision procedure of jMosel [TWMS06], a verification toolset for monadic second-order logic on strings (M2L(Str)). M2L [Chu63] is an extremely expressive specification language with a non-elementary decision procedure. This makes jMosel a good candidate for our new optimization technique, as there is room even for ambitious optimizations due to the huge leverage potential. Our experiments support this judgement: we observed a performance gain of a factor of three when analyzing a parametric family of hardware circuits, despite the fact that the optimized decision procedure includes the analysis and optimization time for 2nd-order value numbering as well. This result is surprising, as the design of these circuits exploits already structural similarity. - Please note that our technique is quite general, and not restricted to the considered application domain.

This paper is organized as follows: Section 2 provides an introduction to the jMosel toolset including the definition of its syntax and semantics. First-order value numbering in the jMosel context is explained in Section 3, while Section 4 introduces second-order value numbering together with a detailed discussion of a minimal example. Subsequently, Section 5 illustrates our new method along a realistic case study, before we conclude with Section 6.

2 jMosel

jMosel is a toolset for monadic 2nd-order logic on strings (*M2L(Str)*) that computes the semantics of a formula in terms of a finite state automaton. In this sense, it can be seen as a compiler form this logic into automata models. A detailed presentation of the tool can be found e.g. in [TWMS06]. Its underlying concepts and the predecessor MoSeL have been presented in [Mar96, KMMG97]. The following subsections summarize the required background about jMosel and M2L.

2.1 Syntax

jMosel's several user-level logics are built on top of the following *Minimal Logic*, which already provides the full expressive power of M2L (on strings):

```
T ::= Id
A ::= subseteq(T,T)  |  shifteq(T,T)
F ::= A  |  ˜F  |  F & F  |  ex Id:  F  |  (F)
```

In this BNF, the non-terminal T denotes 2nd-order terms in form of (2nd-order) variables Id. Atomic predicates A allow comparisons in terms of subset relation and equality after bit-shifting, while jMosel's minimal logic formulas, denoted by the non-terminal and start symbol F, may be constructed using the standard operators of (a minimal) first-order logic.

2.2 Semantics

In (*M2L(Str)*) formulas are interpreted as sets of (ordered) positions in a string of arbitrary, but finite length, which can be conveniently described as finite bitvectors, i.e. a finite word over the alphabet $\{0,1\}$. One often refers to the interpretation of these bitvectors as characteristic functions that describe subsets of a given ordered set. Typical is their interpretation as finite set of natural numbers, illustrated in Figure 1.

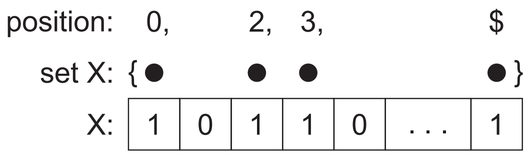


Figure 1: A set of positions set X and the corresponding bit vector X.

The bit vector corresponding to a string variable X has value 1 at position n iff $n \in \mathbb{N}_0$ is included in X, and value 0 otherwise. The figure shows a set X containing the positions 0, 2, 3 and \$, its representation as a characteristic set, and its corresponding bit vector. Here, the symbol \$ stands for the last position in the parametric string, and therefore marks the last bit in the bit vector; a special symbol for this last position is necessary since M2L(Str) allows reasoning about strings of finite but *arbitrary* length, a convenient model for parametric hardware components.

The following development will entirely foot on the bit vector interpretation of M2L(Str), which we formally define below.

Semantics of jMosel formulas

jMosel translates formulas into complete and deterministic finite automata (DFA) in such a way that the language recognized by one such automaton corresponds to the formula's interpretation as a bit vector. The semantics of a formula is defined by the function $[\![\ ]\!] : \varphi \longrightarrow \alpha$, where φ is the set of all jMosel formulas and α is the set of all complete DFAs.

Definition 1 (Boolean Automaton)
A Boolean Automaton A of α is defined as $A = (\Sigma, S, s_0, F, \delta)$, where

- Σ is the set of all edge labels, which themselves denote subsets of the set of free variable V in the considered formula. They are represented as bitvectors of length $|V|$.
- S is the set of all states.
- s_0 is the initial state, $s_0 \in S$.
- F is the set of accepting states $F \subseteq S$.
- δ is the transition function defined as $\delta : S \times \Sigma' \longrightarrow S$.

The edge labels determine for every string variable the Boolean value at position n, whenever this label is taken as nth step of an accepting run. The number of edge labels is exponential in the size of the formula's free variables, since the value of every variable $v \in V$ has to be checked for equality with 0 or 1. Therefore, each label consists of a bit vector of length $|V|$.

Boolean Automata typically have very many edges between two nodes. We therefore construct the following equivalent Symbolic Automaton $\mathscr{A}_s$, whose edges are labelled with Boolean functions and therefore compactly represent a set of edges of the original automaton.

Definition 2 (Symbolic Automaton)
A symbolic automaton $\mathscr{A}_s$ is defined as $\mathscr{A}_s = (\mathscr{L}, \mathscr{S}, s_0, \mathscr{F}, \delta)$, where

- $\mathscr{L}$ is the set of all possible edge labels, consisting of Boolean functions.
- $\mathscr{S}$ is the set of all states.
- s_0 is the initial state, $s_0 \in \mathscr{S}$.
- $\mathscr{F}$ is the set of all accepting states, $\mathscr{F} \subseteq \mathscr{S}$.

T \ t	0	1	2	…	$\$-1$	$\$$
X	x_0	x_1	x_2	…	$x_{\$-1}$	$x_\$$
Y	y_0	y_1	y_2	…	$y_{\$-1}$	$y_\$$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$	$\vdots$
Z	z_0	z_1	z_2	…	$z_{\$-1}$	$z_\$$

Figure 2: Values for 2nd order variables

T \ t	…	i	…
W	…	0	…
X	…	1	…
Y	…	0	…
Z	…	1	…

Figure 3: Representation of edge label `~w & x & ~y & z`

- δ is the transition function defined as $\delta : \mathscr{S} \times \mathscr{L} \longrightarrow \mathscr{S}$.

To describe the transformation from A to $\mathscr{A}_s$, we observe that values for a jMosel formula's 2nd order variables can be represented in table form, where a variable X is expressed as bit vector with position literals $x_0, x_1, \dots, x_{\$-1}, x_\$$. The ordering of variables is arbitrary but fixed.

The *rows* for variables X in Fig. 2 represent a word of the language $[\![X]\!]$. Every column of the table specifies one input symbol of A and must therefore match an appropriate edge label. The position j in this label corresponds to the variable at position j in the ordering of variables. To convert A into an Automaton $\mathscr{A}_s$ with Boolean formulas as edge labels, the labels of A are first transformed as shown in Fig. 3. Subsequently, edges sharing the same source and target state are merged; the resulting edge is labelled with the disjunction of the merged edges' labels.

The formulas for the edge labels resulting from this transformation may be large, but their BDD representations are canonical and typically nice and concise [Bry86]. The jMosel toolset supports various BDD libraries to optimally exploit this observation.

Semantic Completeness:
Note that a symbolic automaton $\mathscr{A}_s$ composed this way is typically not complete: its input alphabet consists of all Boolean functions, but not every state considers the input of every possible Boolean function. However, the automaton is complete *at a semantical level*: the automaton A with bit vector labels it represents is always complete. It is this semantic notion of completeness and determinism which we will refer to in the sequel of the paper.

Convention:
In the following sections, the semantics of jMosel formulas will always be given in terms of symbolic automata $\mathscr{A}_s$. In this section we used the index "s" to better distinguish between the two types of automata, but we omit it from now on. In the figures depicting automata, the

following applies: an arrow marks the initial state, accepting states are denoted as double circles, non-accepting states as plain circles.

In the following, we first present the classical (first-order) value numbering for this application domain, before we lift to second order in Section 4.

3 First-Order Value Numbering

First-order value numbering is an analysis method that allows the detection and removal of redundant computations from a program [CS70]. This goal is achieved by assigning abstract identification values to computations that imply equality: As soon as an identification value reappears, it is certain that the corresponding computation has been already performed before, thus the previously computed result may be reused instead of performing the computation again. This ‘classical optimization is called DAGification in [KMS02].

3.1 Characterization of 1st-order Value Numbering

Given a syntax tree T of a jMosel formula in terms of

- $\mathscr{L}$ is the set of all labels for predicates, operators, and variables,
  $\mathscr{L} = \{\texttt{subseteq}, \texttt{shifteq}, \tilde{}, \&, \texttt{ex}\} \cup \{X, Y, Z, ...\}$
- $\mathscr{N}$ is the set of nodes of the syntax tree T under consideration
- $l : \mathscr{N} \longrightarrow \mathscr{L}$ maps every syntax node to its label.

the assignment of abstract identification values can be given by any function $v_{1st} : \mathscr{N} \longrightarrow \mathbb{N}_0$ that satisfies the following two characteristics:

- For all nodes $n_1, n_2 \in \mathscr{N}$ of the syntax tree,
  $v_{1st}(n_1) = v_{1st}(n_2)$ implies $l(n_1) = l(n_2)$,
  i.e. the coincidence of their syntactic labels. In addition we require
- for all (internal) nodes $n_1, n_2 \in \mathscr{N}$ with children $c^1_1, ..., c^1_i \in \mathscr{N}$ and $c^2_1, ..., c^2_j \in \mathscr{N}$, respectively
  $i = j \wedge \forall k \in \{0, ..., i\} \, .v_{1st}(c^1_k) = v_{1st}(c^2_k)$

3.2 Example

As an example for the process of first-order value numbering, we consider the following jMosel formula:

$$F = \texttt{(subseteq(X,Y)\&shifteq(A,B))|(subseteq(X,Y)\&shifteq(A,B))}$$

Fig. 4 shows its syntax tree after computation of the value numbers.

At compilation, the compiler can benefit from the fact that the subformulas with value numbers 3, 6, and 7 all occur twice by only calculating each of them once, storing the result of the computation, and referring to it when the corresponding value number occurs for the second time. We will illustrate the impact of this optimization in in Section 5.

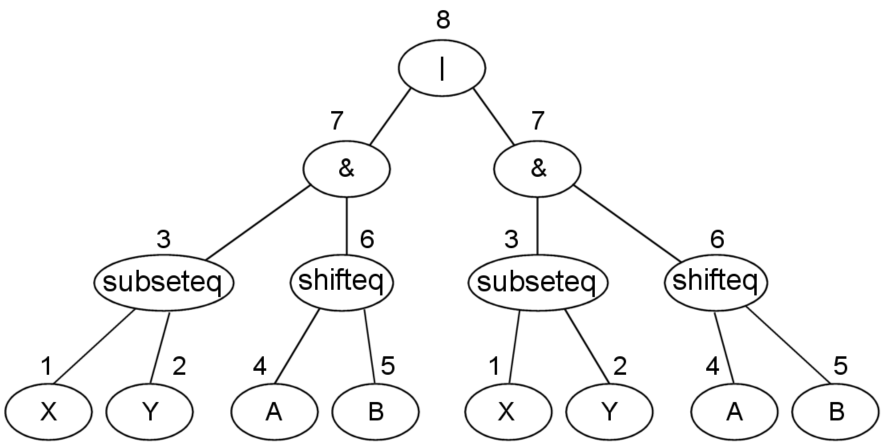


Figure 4: 1st order value numbering applied to the jMosel formula F.

4 Second-Order Value Numbering

While first-order value numbering is used to identify redundant computations and replacing them by previously computed results, the goal of second-order value numbering is to identify redundant *transformations* - the reason for the use of second-order here. As will be clear below, this analysis and its corresponding optimizations is only a bit more involved than in the first-order case, but has a far bigger impact, see Section 5.

4.1 Characterization of 2nd-order Value Numbering

The only difference in the characterization of the labelling function $v_{1st} : \mathscr{N} \longrightarrow \mathbb{N}_0$ concerns the treatment of atomic predicates, i.e., of $\mathscr{A} = \{\texttt{subseteq}, \texttt{shifteq}\}$. Their labelling does no longer require the second clause for internal nodes. This results in the following slightly modified characterization:

- For all nodes $n_1, n_2 \in \mathscr{N}$ of the syntax tree,
  $v_{1st}(n_1) = v_{1st}(n_2)$ implies $l(n_1) = l(n_2)$,
  i.e. the coincidence of their syntactic labels. In addition we require

- for all (internal) nodes $n_1, n_2 \in \mathscr{N}$ with children $c_1^1, ..., c_i^1 \in \mathscr{N}$ and $c_1^2, ..., c_j^2 \in \mathscr{N}$, respectively
  $i = j \quad \wedge \quad \forall k \in \{0, ..., i\}. v_{1st}(c_k^1) = v_{1st}(c_k^2)$ unless they are labelled with $\mathscr{A} = \{\texttt{subseteq}, \texttt{shifteq}\}$

After this labelling, nodes sharing the same value number can be replaced by calls to a semantics transformer. However note that transformers should only be created for subtrees containing at least one logical operator, as otherwise the effect of the transformation is vacuous.

4.2 Semantics Transformers

When implementing second-order value numbering for jMosel, the semantics transformers can be implemented in terms of custom predicates similar to the atomic formulas `subseteq` and `shifteq`. This means that every identification of redundancy results in the automatic definition of a custom predicate. This process can be seen as an "on-the-fly enhancement" of the logic with newly identified predicates with multiple occurrences.

For the definition of semantics transformers and calls to these transformers, the syntax of jMosel is enhanced by the `let`-construct

$$\texttt{let}\ \ <predicatename>\texttt{(}<argumentlist>\texttt{)}\ \texttt{=}\ <definition>$$
$$\texttt{in}\ \ <formula>$$

that allows one to formulate formulas like:

```
let pred(X,Y) =subseteq(X,Y) & shifteq(Y,X)
      in pred(A,B) <-> pred(S,T).
```

where a new predicate `pred` with arguments `X` and `Y` is defined by the formula `subseteq(X,Y) & shifteq(Y,X)` and instantiated twice in the formula `pred(A,B) <-> pred(S,T)`.

Definition 3 (Semantics of the *let*-Construct)
For formulas $f_1, f_2 \in F$ and a predicate `Pred(A1,...,An)` $\in P$, the semantics of the *let*-construct is defined as follows:

$$[\![\texttt{let Pred(arg1,...,argn) = } f_1 \texttt{ in } f_2]\!] =_{df} [\![f_2[f_1/\texttt{Pred(x1,...,xn)}]]\!]$$

where $\cdot[\cdot/\cdot] : F \times ID \times ID \longrightarrow F$ denotes the usual syntactic substitution.

We use the `let`-construct to implement second-order value numbering for jMosel. There, the definitions of and calls to semantics transformers are automatically inserted into the considered formula according to the value numbers assigned to the individual computations.

In the following we first illustrate on a very simple example how a semantics transformer is identified and inserted into the formula, then we consider a more complex case study in Section 5.

4.3 Example

As a short example for the process of second-order value numbering, we consider the following jMosel formula:

```
(subseteq(A,B) & shifteq(C,D)) | (subseteq(X,Y) & shifteq(V,W))
```

This formula is similar to the one of section 3.2, but cannot benefit from first-order value numbering, since the atomic formulas `subseteq` and `shifteq` are called with different parameters. This is a very frequent case in practice: in hardware design, for example, circuits are composed of a small number of component types, each instance of which has the same abstract function, but is connected differently. Circuits would thus not be eligible for first order value numbering, but are an excellent application for second-order value numbering.

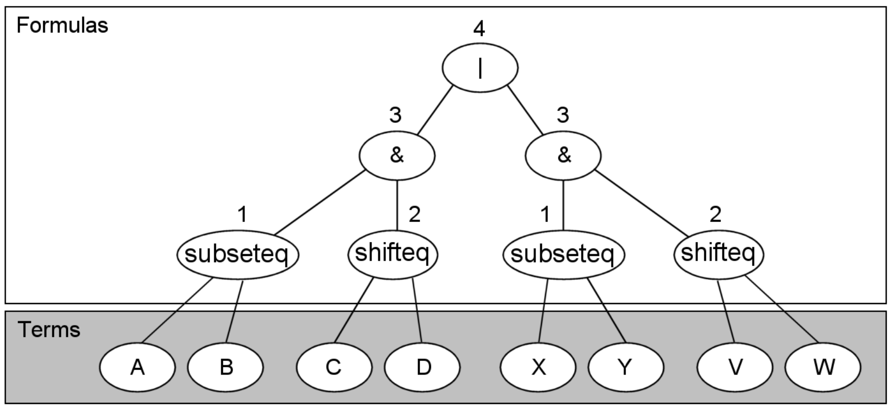


Figure 5: 2nd order value numbering applied to a jMosel formula.

The formula's syntax tree after computation of second-order value numbers is shown in Fig. 5. The nodes of the tree have been divided into two sets, *formulas* and *terms*; nodes representing terms have not been numbered by the 2nd-order value-numbering procedure.

The two nodes labeled with "&" share the value number 3; this means they both perform the same set of computations and can therefore be replaced by calls to a same semantics transformer `st_3`. The nodes with the value numbers 1 and 2 are not taken into account, since they are labeled with atomic predicates.

The definition of the `st_3` transformer is isomorphic to the subtrees labeled with value number 3, but all occurring variables are replaced by fresh variables "`arg_n`". This transformer is inserted via let-construct into the formula, and the subtrees labeled with 3 are replaced by calls to `st_3` (see the corresponding syntax tree in Fig. 6), resulting in the formula:

```
let st_3(arg_1,arg_2,arg_3,arg_4) =
        subseteq(arg_1,arg_2) & shifteq(arg_3,arg_4)
   in st_3(A,B,C,D) | st_3(X,Y,V,W)
```

When compiling this formula, the conjunction of the predicates `subseteq` and `shifteq` is only computed once and stored as a semantics transformer, opposed to the original formula, where the conjunction is computed twice. The detailed course of the optimization and compilation is described in the next section.

4.4 The Optimizing Transformation

The optimization of the syntax tree for the formula

```
(subseteq(A,B) & shifteq(C,D)) | (subseteq(X,Y) & shifteq(V,W))
```

is performed in the following steps:

- Perform the numbering of the syntax tree, resulting in the labelling shown in Fig. 5.
- Identify the good targets for optimization: the nodes labelled with 3 qualify, as their exist more than once, and the corresponding subtrees contain logical operators.

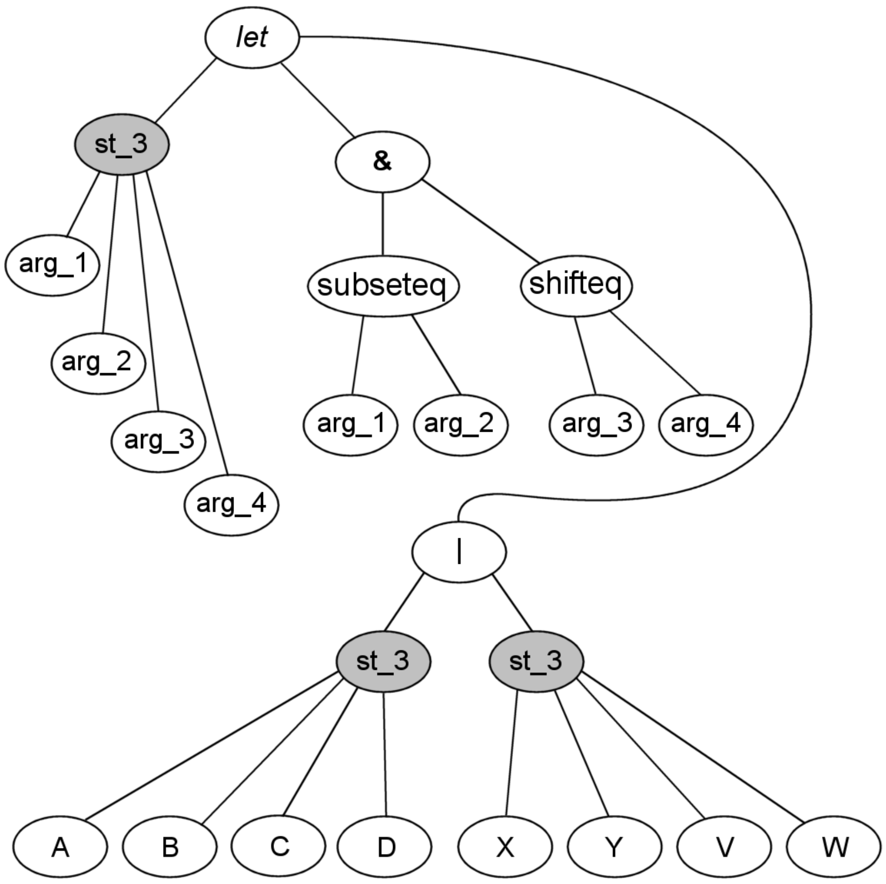


Figure 6: Syntax tree after optimization.

- Create a semantics transformer `st_3` for the nodes labelled with 3 by duplicating one of the syntax trees of the corresponding subfunction.
- Replace the two occurrences of syntax nodes labelled with 3 by calls to the newly created semantics transformer `st_3` (Fig. 7).
- Add the definition of `st_3` to the top of the syntax tree (Fig. 6).

The compiler operates on the modified syntax tree as follows:

- At the ”let” construct it compiles the semantics transformer’s definition, identified by the subtree of the second child node of ”let”.
- At the nodes representing the atomic predicates
  `subseteq(arg_1,arg_2)` and `shifteq(arg_3,arg_4)`
  it constructs the corresponding basic automata $\mathbf{a}_1$ and $\mathbf{a}_2$.

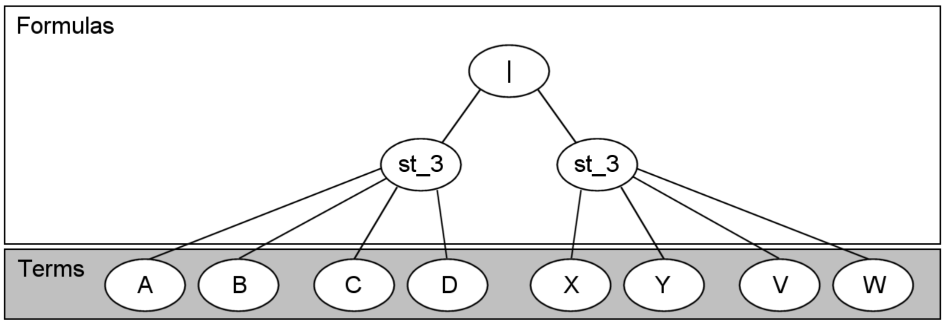
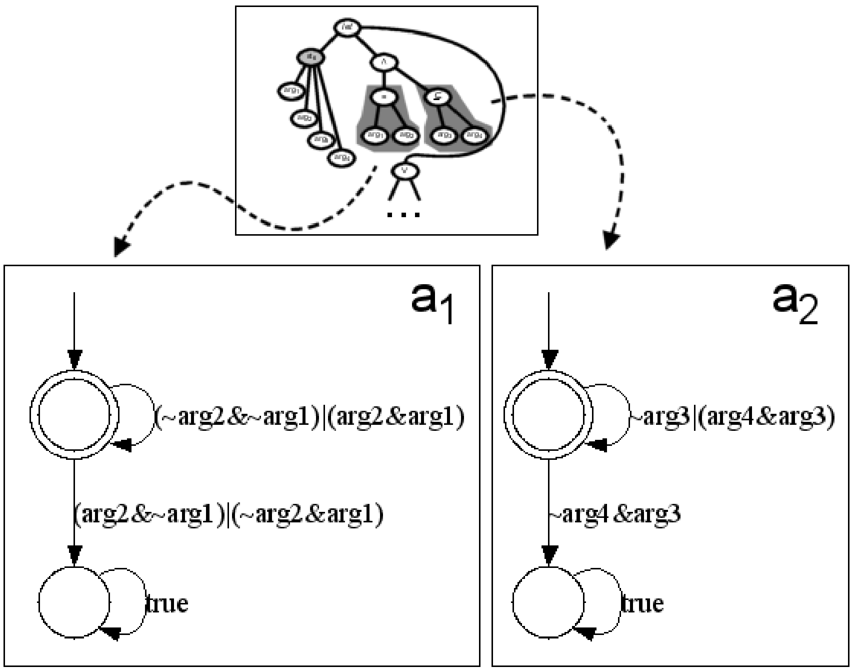


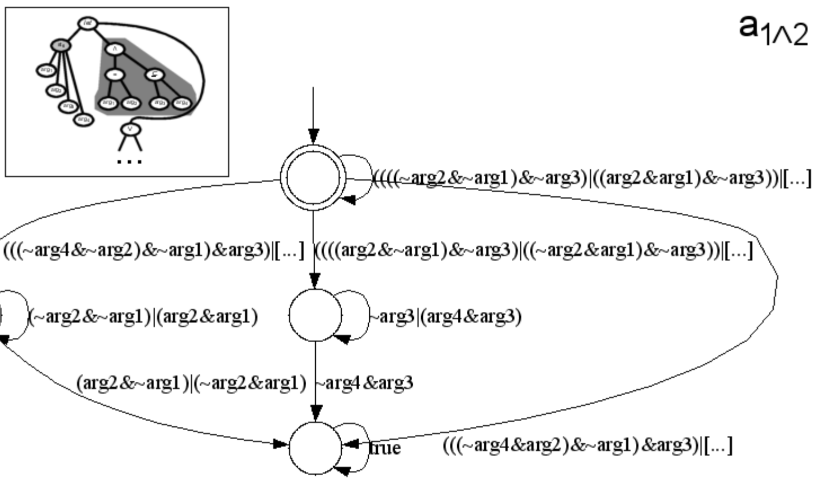
Figure 7: Syntax tree with calls to the semantics transformer.



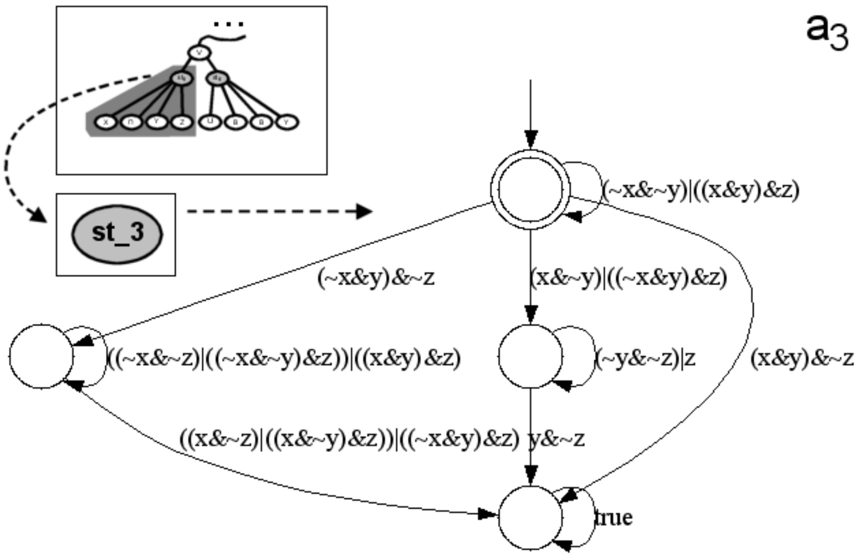
- At the node representing the formula

```
subseteq(arg_1,arg_2) & shifteq(arg_3,arg_4)
```

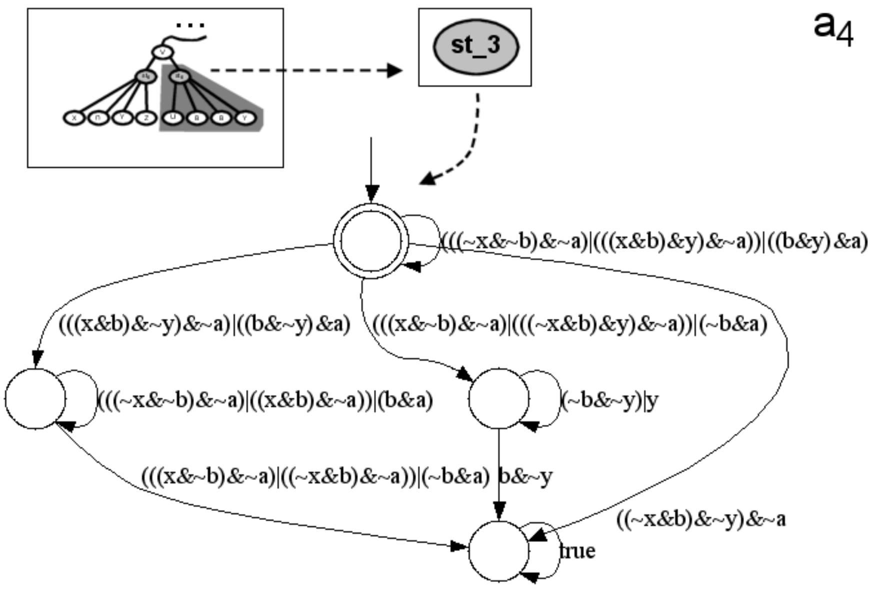
it constructs the product automaton $\mathbf{a}_{1\wedge 2}$ representing the conjunction of $\mathbf{a}_1$ and $\mathbf{a}_2$.


a1∧2
(((~arg2&~arg1)&~arg3)|((arg2&arg1)&~arg3))|[...]
(((~arg4&~arg2)&~arg1)&arg3)|[...]
(((arg2&~arg1)&~arg3)|((~arg2&arg1)&~arg3))|[...]
(~arg2&~arg1)|(arg2&arg1)
~arg3|(arg4&arg3)
(arg2&~arg1)|(~arg2&arg1)
~arg4&arg3
true
(((~arg4&arg2)&~arg1)&arg3)|[...]

- It stores the resulting automaton as a semantics transformer named `st_3` with arguments `arg_1,...,arg_4`.
- The compilation continues with the subtree of the third child node of ”let”.
- At the node representing the call of a semantics transformer `st_3(A,B,C,D)` the pre-computed definition of `st_3` is copied, replacing the arguments `arg_1, ...,arg_4` in the edge labels with the terms `A,B,C,D` to yield the result automaton $\mathbf{a}_3$.



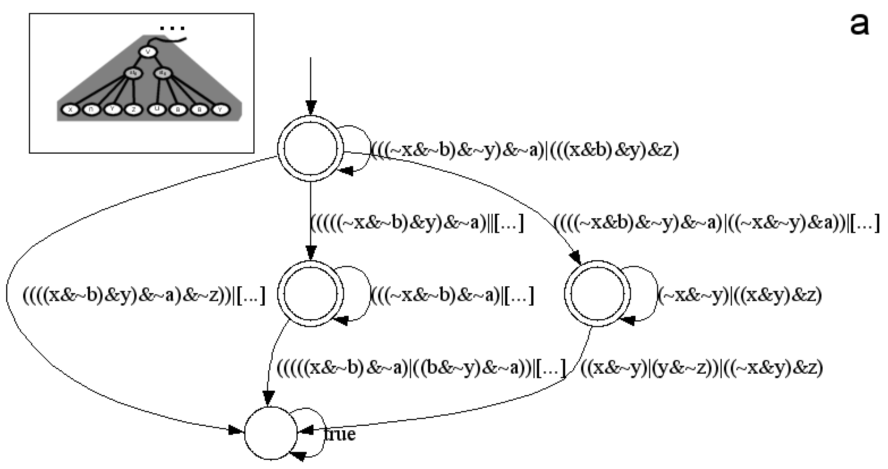
- At the node representing the call of a semantics transformer `st_3(X,Y,V,W)` the pre-computed definition of `st_3` is copied again, this time the arguments `arg_1,..., arg_4` in the edge labels are replaced with the terms `X,Y,V,W` to form the result automaton $\mathbf{a}_4$.


a4
st_3
(((~x&~b)&~a)|(((x&b)&y)&~a))|((b&y)&a)
(((x&b)&~y)&~a)|((b&~y)&a)
(((x&~b)&~a)|(((~x&b)&y)&~a))|(~b&a)
(((~x&~b)&~a)|((x&b)&~a))|(b&a)
(~b&~y)|y
(((x&~b)&~a)|((~x&b)&~a))|(~b&a)
b&~y
((~x&b)&~y)&~a
true

- At the node representing the formula

```
st_3(A,B,C,D) | st_3(X,Y,V,W)
```

it constructs the product automaton **a** representing the disjunction of $\mathbf{a}_3$ and $\mathbf{a}_4$ and returns it as the compilation's result.


a
(((~x&~b)&~y)&~a)|(((x&b)&y)&z)
(((((~x&~b)&y)&~a)||[...]
((((~x&b)&~y)&~a)|((~x&~y)&a))|[...]
((((x&~b)&y)&~a)&~z))|[...]
(((~x&~b)&~a)|[...]
(~x&~y)|((x&y)&z)
(((((x&~b)&~a)|((b&~y)&~a))|[...]
((x&~y)|(y&~z))|((~x&y)&z)
true

5 Application and Performance Measuring

One of jMosel's main application areas is the specification and verification of parametric hardware systems. We tested the presented optimization with a "real-world" example, applying it to the structural description of a parametric adder that describes the family of adder circuits for bit vectors of length *n*.

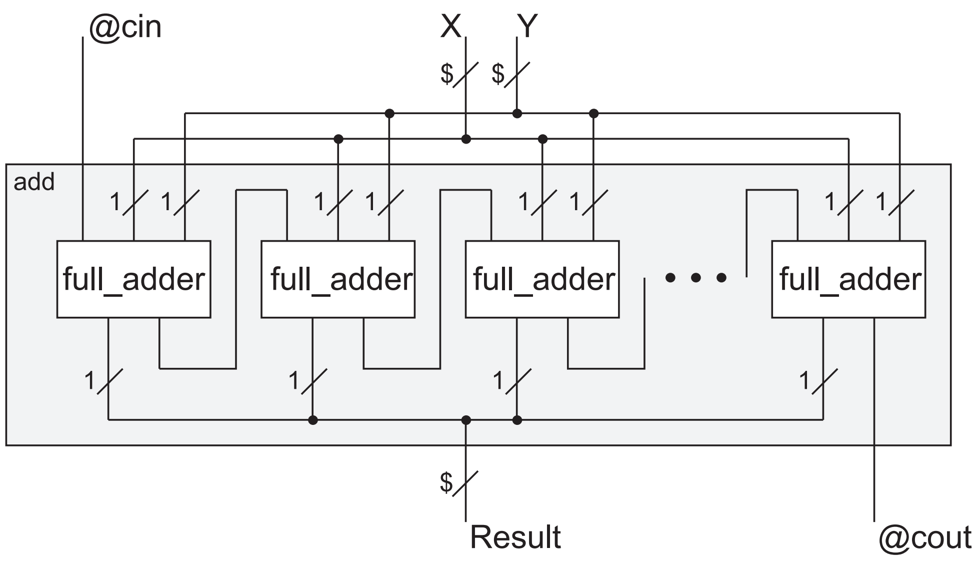


Figure 8: Structure of the parametric adder

Structural Description of a Parametric Adder

Fig. 8 shows the structure for this adder based on n interconnected full adders. The circuit adds two bit vectors X and Y and stores the result as the new vector $Result$. The Boolean variables $@cin$ and $@cout$ are the carry-in and carry-out bits.

The size of input formula and of the resulting automaton are too large for a detailed discussion in this paper, so we only present the results in terms of key data at this point. The compilation times have been measured on an Intel Centrino Duo System (2 x 2.16 GHz) with 1 GB of RAM:

Optimization	none	1st-ord. VN	2nd-ord. VN
Nodes in synt. tree	472	469	452
Depth of synt. tree	26	27	32
overall run time	11.50 sec	10.49 sec	3.47 sec
Sem. transformers	-	1	6

As we expected, first order value numbering does not contribute significatively to performance: the sharing is at the level of subcircuit types, not of fully instanced values.

The increased depth of the modified syntax tree is due to the fact that all definitions of semantics transformers are added to the top of the tree. By identifying 6 semantics transformers, the size of the tree could be reduced by 20 nodes. This does not seem too exciting at first sight; however, it has quite some impact: the overall run time of the decision process is accelerated by a factor of three.

The enormous speedup is quite surprising, since the adder's structural description already included user-defined predicates for frequently occurring constructs like the full adder and logical gates. This shows that even a carefully written formula and well structured circuits might still contain significant potential of redundancy, and therefore could benefit greatly from second-order value numbering.

6 Conclusion

We have presented second-order value numbering, a new optimization technique for suppressing redundancy, in a version tailored to the application for improving the decision procedure of jMosel, a verification tool set for monadic 2nd-order logic on strings. Our technique extends the well-known concept of value numbering to consider not merely values, but *semantics transformers* that can be efficiently pre-computed and help to avoid redundancy at a second-order level. We have illustrated the effect of this optimization for a parametric family of hardware circuits, where we observed a performance gain by a factor of 3. This result is surprising, as the design of these circuits exploits already structural similarity.

Currently we are working on a careful experimental analysis of the impact of our technique in practice using standard benchmarks and libraries. We conjecture that we will observe a growth of the improvement factor with the size of the system, i.e. a 'felt' superlinear speedup.

In a more general perspective, second-order value numbering can be regarded as a means for a specific semantic form of procedural abstraction [SHKN76, DWF$^+$07] in a similar way as value numbering (or its generalization to Value Flow graphs) is a semantic support for code motion [SKR90]. Thus besides looking for further application domains for second-order value numbering, it would also be interesting to investigate how the structural generalization of value numbering presented in [SKR90] can be raised to second-order in order to achieve a truly semantic notion of procedure abstraction for imperative programs.

Bibliography

[AWZ88] B. Alpern, M. N. Wegman, F. K. Zadeck. Detecting equality of variables in programs. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Pp. 1–11. ACM Press, New York, NY, USA, 1988.

[Bry86] R. E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8):677–691, 1986.

[Chu63] A. Church. Logic, arithmetic and automata. In *Proc. Intern. Congr. Math.* Pp. 23–35. Almqvist and Wiksells, 1963.

[CS70] J. Cocke, J. T. Schwartz. Programming Languages and their Compilers. Courant Institute of Mathematical Sciences, New York University, 1970.

[DWF$^+$07] A. Dreweke, M. Wörlein, I. Fischer, D. Schell, T. Meinl, M. Philippsen. Graph-Based Procedural Abstraction. In Society (ed.), *Proc. of the 2007 CGO*. Pp. 259–270. IEEE Computer Society, Los Alamitos, CA, USA, 2007.

[KMMG97] P. Kelb, T. Margaria, M. Mendler, C. Gsottberger. MOSEL: A Flexible Toolset for Monadic Second-Order Logic. In *Proc. TACAS'97*. Lecture Notes in Computer Science 1217, pp. 183–202. Springer Verlag, 1997.

[KMS02] N. Klarlund, A. Møller, M. Schwartzbach. MONA Implementation Secrets. *International Journal of Foundations of Computer Science*, 2002.

[Mar96] T. Margaria. Fully Automatic Verification and Error Detection for Parameterized Iterative Sequential Circuits. In *Proc. TACAS '96*. Lecture Notes in Computer Science 1055, pp. 258–277. Springer Verlag, 1996.

[SHKN76] T. Standish, D. Harriman, D. Kibler, J. Neighbors. The Irvine Program Transformation Catalogue. University of California, Irvine, 1976.

[SKR90] B. Steffen, J. Knoop, O. Rüthing. The Value Flow Graph: A Program Representation for Optimal Program Transformations. In *European Symposium on Programming*. Pp. 389–405. 1990.

[TWMS06] C. Topnik, E. Wilhelm, T. Margaria, B. Steffen. jMosel: A Stand-Alone Tool and jABC Plugin for M2L(Str). In *Model Checking Software: 13th International SPIN Workshop, Vienna (Austria)*. LNCS 3925/2006, pp. 293–298. Springer-Verlag, 2006.

Towards Theorem Proving Graph Grammars using Event-B

Leila Ribeiro[1]*, Fernando Luís Dotti[2], Simone André da Costa[3] and Fabiane Cristine Dillenburg[4]

[1] leila@inf.ufrgs.br
[4] fabiane.dillenburg@inf.ufrgs.br
Instituto de Informática
Universidade Federal do Rio Grande do Sul, Brazil

[2] fernando.dotti@pucrs.br
Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul, Brazil

[3] simone.costa@ufpel.edu.br
Instituto de Física e Matermática
Universidade Federal de Pelotas, Brazil

Abstract: Graph grammars may be used as specification technique for different kinds of systems, specially in situations in which states are complex structures that can be adequately modeled as graphs (possibly with an attribute data part) and in which the behavior involves a large amount of parallelism and can be described as reactions to stimuli that can be observed in the state of the system. The verification of properties of such systems is a difficult task due to many aspects: the systems in many situations involve an infinite number of states; states themselves are complex and large; there are a number of different computation possibilities due to the fact that rule applications may occur in parallel. There are already some approaches to verification of graph grammars based on model checking, but in these cases only finite state systems can be analyzed. Other approaches propose over- and/or under-approximations of the state-space, but in this case it is not possible to check arbitrary properties. In this work, we propose to use the Event-B formal method and its theorem proving tools to analyze graph grammars. We show that a graph grammar can be translated into an Event-B specification preserving its semantics, such that one can use several theorem provers available for Event-B (for instance, through the Rodin platform) to analyze the reachable states of the original graph grammar. The translation is based on a relational definition of graph grammars, that was shown to be equivalent to the Single-Pushout approach to graph grammars.

Keywords: Graph Grammars, Theorem Proving, Event-B

1 Introduction

Graph grammars [Ehr79, Roz97] are a formal description technique suitable for the specification of distributed and reactive systems. The basic idea of this formalism is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The operational behavior of the system is expressed via applications of these rules to graphs depicting the current states of the system. Graph grammars are appealing as specification formalism because they are formal and based on simple, but powerful, concepts to describe behavior. At the same time they also have a nice graphical layout that helps even non-theoreticians to understand a specification.

* This author is partly sponsored by CNPq/Brazil

The verification of graph grammar models through model-checking is currently supported by various approaches. Although model checking is an important analysis method, it has as disadvantage the need to build the complete state space, which can lead to the state explosion problem. Much progress has been made to deal with this difficulty, and a lot of techniques have increased the size of the systems that could be verified [CGJ$^+$01]. Baldan and König proposed [BK02] approximating the behavior of (infinite-state) graph transformation systems by a chain of finite under- or over- approximations, at a specific level of accuracy of the full unfolding [BCMR07] of the system. However, as [DHR$^+$07] emphasizes, these approaches that derive the model as approximations can result in inconclusive error reports or inconclusive verification reports.

Besides model checking, theorem proving [RV01, CW96] is another well-established approach used to analyze systems for desired properties. Theorem proving is a technique where both the system and its desired properties are expressed as formulas in some mathematical logic. A logical description defines the system, establishing a set of axioms and inference rules. The process consists of finding a proof of the required property from the axioms or intermediary lemmas of the system. In contrast to model checking, theorem proving can deal directly with infinite state spaces and it relies on techniques such as structural induction to prove over infinite domains. The use of this technique may require interaction with a human; however, the user often gains very useful perceptions into the system or the property being proved.

Each verification technique has arguments for and against its use, but we can say that model-checking and theorem proving are very complementary. Most of the existing approaches use model checkers to analyze properties of computations, that is, properties over the sequences of steps a system may engage in. Properties about reachable states are handled, if at all possible, only in very restricted ways. In this work, our main aim is to provide a means to prove structural properties of reachable graphs using the theorem proving technique.

In previous work [CR09a] we proposed a relational approach to graph grammars, providing an encoding of graphs and rules into relations. This enabled the use of first-order logic formulas to express properties of reachable states of a graph grammar. This encoding showed to be equivalent to the Single-Pushout approach to graph grammars. Verification of infinite-state systems specified as graph grammars is possible using our approach with theorem proving techniques. This approach was inspired by Courcelle's research about logic and graphs [Cou97].

Courcelle investigates in various papers [Cou94, Cou97, Cou04] the representation of graphs and hypergraphs by relational structures as well as the expressiveness of their properties by logical languages. In [Cou94] the description of graph properties and the transformation of graphs in monadic second-order logic is proposed. However, these works are not particularly interested in effectively verifying the properties of graph transformation systems (GTSs). Since theorem provers, in general, work efficiently with specifications in relational style, we extend the relational representation of graphs to graph grammar models and use such representation for the formal analysis of reactive systems through the theorem proving technique. On the other hand, other authors have investigated the analysis of GTSs based on relational logic or set theory. Baresi and Spoletini [BS06] explore the formal language Alloy to find instances and counterexamples for models and GTSs. In fact, with Alloy, they only analyze the system for a finite scope, whose size is user-defined. Strecker [Str08], aiming to verify structural properties of GTSs, proposes a formalization of graph transformations in a set-theoretic model. His goal is to obtain a language for writing graph transformation programs and reasoning about them. Nevertheless, the language has only two statements, on e to apply a rule repeatedly to a graph, and another to apply several rules in a specific order to a graph. Until now, the work just presents a glimpse of how to reason about graph transformations.

In this paper we will use Event-B to analyze properties of graph grammars. Event-B [DEP] is a state-based formal method closely related to Classical B [Abr05]. It has been successfully used in several applications, having available tool support for both model specification and analysis. There are a series of powerful theorem provers that can be used to analyze event-B specifications. Due to the similarity between event-B models and graph grammar specifications, specially concerning the rule-based behavior, in this paper we propose to translate graph grammar specifications in event-B structures, such that it is possible to use the event-B provers to demonstrate properties of a graph grammar. This translation is based on the relational definition of graph grammars.

The paper is organized as follows. Section 2 presents the relational approach of graph grammars. Section 3 briefly introduces the event B formalism. Section 4 shows how a graph grammar can be translated into an Event-B program. Section 5 contains some final remarks.

2 Relational Approach to Graph Grammars

Graph Grammars are a generalization of Chomsky grammars from strings to graphs suitable for the specification of distributed, asynchronous and concurrent systems. The basic notions behind this formalism are: states are represented by graphs and possible state changes are modeled by rules, where the left- and right-hand sides are graphs.

We use a relational and logical approach for the description of Graph Grammars: graphs and graph morphisms are described as relational structures [CR09a], that is, they are defined as tuples formed by a set and by a family of relations over this set. Proofs about the well-definedness of these definitions were detailed in [CR09b].

Definition 1 (Relational Structures) Let $\mathscr{R}$ be a finite set of relation symbols, where each $R \in \mathscr{R}$ has an associated positive integer called its arity, denoted by $\rho(R)$. An **$\mathscr{R}$-structure** is a tuple $S = \langle D_S, (R_S)_{R\in\mathscr{R}}\rangle$ such that D_S is a possible empty set called the domain of S and each R_S is a $\rho(R)$-ary relation on D_S, i.e., a subset of $D_S^{\rho(R)}$. $R(d_1,\ldots,d_n)$ holds in S if and only if $(d_1,\ldots,d_n) \in R_S$, where $d_1,\ldots,d_n \in D_S$.

A relational graph $|G|$ is a tuple composed of a set, the domain of the structure, representing all vertices and edges of $|G|$ and by two finite relations: a unary relation, *vert* $_G$, defining the set of vertices of $|G|$ and a ternary relation inc_G representing the incidence relation between vertices and edges of $|G|$. The *uniqueness edge condition* assures that the same edge doesn't connect different vertices.

Definition 2 (Relational Graph) Let $\mathscr{R}_{gr} = \{vert, inc\}$ be a set of relations, where *vert* is unary and *inc* is ternary. A **relational graph** is a $\mathscr{R}_{gr}$-structure $|G| = \langle D_G, (R_G)_{R\in\mathscr{R}_{gr}}\rangle$, where:

- $D_G = V_G \cup E_G$ is the union of sets of possible vertices and edges of $|G|$, respectively (we always assume that $V_G \cap E_G = \varnothing$);
- $vert_G \subseteq V_G$, with $vert_G(x)$ iff x is a vertex of $|G|$;
- $inc_G \subseteq E_G \times V_G \times V_G$, with $inc_G(x,y,z)$ iff x is a directed edge that links vertex y to vertex z in $|G|$.

such that the following condition is satisfied:

- **Uniqueness Edge Condition.** $\forall x,y,z,y',z'$,
  $[inc_G(x,y,z) \wedge inc_G(x,y',z') \Rightarrow y = y' \wedge z = z']$.

A relational graph morphism $|g|$ from a relational graph $|G|$ to a relational graph $|H|$ is obtained through two binary relations: one to relate vertices (g_V) and other to relate edges (g_E). The *type consistency conditions* state that if two vertices are related by g_V then the first one must be a vertex of $|G|$ and the second one a vertex of $|H|$, and if two edges are related by g_E, then the first one must be an edge of $|G|$ and the second one an edge of $|H|$. The *(morphism) commutativity condition* assures that the mapping of edges preserves the mapping of source and target vertices.

Definition 3 (Relational Graph Morphism) Let $|G| = \langle V_G \cup E_G, \{vert_G, inc_G\}\rangle$ and $|H| = \langle V_H \cup E_H, \{vert_H, inc_H\}\rangle$ be relational graphs. A **relational graph morphism** $|g|$ **from** $|G|$ **to** $|H|$ is defined by a set $|g| = \{g_V, g_E\}$ of binary relations where:

- $g_V \subseteq V_G \times V_H$ is a partial function that relates vertices of $|G|$ to vertices of $|H|$;
- $g_E \subseteq E_G \times E_H$ is a partial function that relates edges of $|G|$ to edges of $|H|$;

such that the following conditions are satisfied:

- **Type Consistency Conditions.** $\forall x, x'$,
  $[g_V(x,x')] \Rightarrow vert_G(x) \wedge vert_H(x')$; and
  $[g_E(x,x')] \Rightarrow \exists y, y', z, z' [inc_G(x,y,z) \wedge inc_H(x',y',z')]$;
- **Morphism Commutativity Condition.** $\forall x, y, z, x', y', z'$,
  $[g_E(x,x') \wedge inc_G(x,y,z) \wedge inc_H(x',y',z') \Rightarrow g_V(y,y') \wedge g_V(z,z')]$.

$|g|$ is called total/injective if relations g_V and g_E are total/injective functions.

A relational typing morphism is a relational graph morphism that has the role of typing all elements of a graph $|G|$ over a graph $|T|$.

Definition 4 (Relational Typing Morphism) Let $|G|$ and $|T|$ be relational graphs. A **relational typing morphism from** $|G|$ **over** $|T|$ is defined by a total relational graph morphism $|t^G| = \{t_V^G, t_E^G\}$ from $|G|$ to $|T|$.

A relational typed graph is defined by two relational graphs together with a relational typing morphism. A relational typed graph morphism between graphs typed over the same graph is defined by a relational graph morphism. A *(typed morphism) compatibility condition* assures that the mappings of vertices and edges preserve types.

Definition 5 (Relational Typed Graph, Relational Typed Graph Morphism) A **relational typed graph** is given by a tuple $|G^T| = \langle |G|, |t^G|, |T|\rangle$ where $|G|$ and $|T|$ are relational graphs and $|t^G| = \{t_V^G, t_E^G\}$ is a relational typing morphism from $|G|$ over $|T|$. A **relational (typed) graph morphism from** $|G^T|$ **to** $|H^T|$ is defined by a relational graph morphism $|g| = \{g_V, g_E\}$ from $|G|$ to $|H|$, such that the typed morphism compatibility condition is satisfied:

- **(Typed Morphism) Compatibility Condition.** $\forall x, x', y$,
  $[g_V(x,x') \wedge t_V^G(x,y) \Rightarrow t_V^H(x',y)]$; and
  $[g_E(x,x') \wedge t_E^G(x,y) \Rightarrow t_E^H(x',y)]$.

A relational rule specifies a possible behaviour of the system. It consists of a left-hand side $|L^T|$, describing items that must be present in a state to enable the application of the rule and a right-hand side $|R^T|$, expressing items that will be present after the application of the rule. We require that rules do not collapse vertices or edges (are injective) and do not delete vertices.

Definition 6 (Relational Rule) A **relational rule** α is given by a tuple $\langle |L^T|, |\alpha|, |R^T|\rangle$ where:

- $|L^T| = \langle |L|, |t^L|, |T|\rangle$ and $|R^T| = \langle |R|, |t^R|, |T|\rangle$ are relational typed graphs;
- $|\alpha| = \{\alpha_V, \alpha_E\}$ is an injective relational typed graph morphism from $|L^T|$ to $|R^T|$, such that α_V is a total function on the set of vertices.

A relational graph grammar is composed by a *relational type graph*, characterizing the types of vertices and edges allowed in a system, an *initial relational graph*, representing the initial state of a system and *a set of relational rules*, describing the possible state changes that can occur in a system.

Definition 7 (Relational Graph Grammar) Let $\mathscr{R}_{GG} = \{vert_T,\ inc_T,\ vert_{G0}, inc_{G0},\ t_V^{G0},\ t_E^{G0},\ (vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li}, vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})_{i\in\{1,\ldots,n\}}\}$ be a set of relation symbols. A **relational graph grammar** is a $\mathscr{R}_{GG}$-structure $|GG| = \langle D_{GG}, (r)_{r\in\mathscr{R}_{GG}}\rangle$ where

- $D_{GG} = V_{GG} \cup E_{GG}$ is the set of vertices and edges of the graph grammar, where: $V_{GG} \cap E_{GG} = \varnothing$, $V_{GG} = V_T \cup V_{G0} \cup (V_{Li} \cup V_{Ri})_{i\in\{1,\ldots,n\}}$ and $E_{GG} = E_T \cup E_{G0} \cup (E_{Li} \cup E_{Ri})_{i\in\{1,\ldots,n\}}$.
- $|T| = \langle V_T \cup E_T, \{vert_T, inc_T\}\rangle$ defines a relational graph **(the type of the grammar)**.
- $|G0^T| = \langle |G0|, |t^{G0}|, |T|\rangle$, with $|G0| = \langle V_{G0} \cup E_{G0}, \{vert_{G0}, inc_{G0}\}\rangle$ and $|t^{G0}| = \{t_V^{G0}, t_E^{G0}\}$, defines a relational typed graph **(the initial graph of the grammar)**.
- Each collection $(vert_{Li},\ inc_{Li},\ t_V^{Li},\ t_E^{Li},\ vert_{Ri}, inc_{Ri},\ t_V^{Ri},\ t_E^{Ri},\ \alpha_{i_V},\ \alpha_{i_E})$ defines a **rule**:
  - $|Li^T| = \langle |Li|, |t^{Li}|, |T|\rangle$, with $|Li| = \langle V_{Li} \cup E_{Li}, \{vert_{Li}, inc_{Li}\}\rangle$ and $|t^{Li}| = \{t_V^{Li}, t_E^{Li}\}$, defines a relational typed graph **(the left-hand side of the rule)**.
  - $|Ri^T| = \langle |Ri|, |t^{Ri}|, |T|\rangle$, with $|Ri| = \langle V_{Ri} \cup E_{Ri}, \{vert_{Ri}, inc_{Ri}\}\rangle$ and $|t^{Ri}| = \{t_V^{Ri}, t_E^{Ri}\}$, defines a relational typed graph **(the right-hand side of the rule)**.
  - $\langle |Li^T|, |\alpha_i|, |Ri^T|\rangle$, with $|\alpha_i| = \{\alpha_{i_V}, \alpha_{i_E}\}$, defines a relational rule.

Given a relational rule and a state, we say that this rule is applicable in this state if there is a match, that is, an image of the left-hand side of the rule in the state. The operational behaviour of a graph grammar is defined in terms of applications of the rules to some state graph.

Definition 8 (Relational Match) Let $\langle |L^T|, |\alpha|, |R^T|\rangle$ be a relational rule, with $|L^T| = \langle |L|, \{t_V^L, t_E^L\}, |T|\rangle$ and $|R^T| = \langle |R|, \{t_V^R, t_E^R\}, |T|\rangle$. Let $|G^T| = \langle |G|, |t^G|, |T|\rangle$ be a relational typed graph with $t^G = \{t_V^G, t_E^G\}$. A **relational match** $|m|$ **of the given rule in** $|G^T|$ is defined by a total relational typed graph morphism $|m| = \{m_V, m_E\}$ from $|L^T|$ to $|G^T|$, such that the following conditions are satisfied:

- m_E is injective;
- **Match Compatibility Condition.** $\forall x, x', y$
  $[m_V(x,x') \wedge t_V^L(x,y) \Rightarrow t_V^G(x',y)]$,
  $[m_E(x,x') \wedge t_E^L(x,y) \Rightarrow t_E^G(x',y)]$.

The application of a given rule to a match in a state essentially removes from the state all elements that are in the left-hand side of the rule that are not mapped to the right-hand side, and creates in the state all new elements of the right-hand side of the rule. The rest of the state remains unchanged.

Selected a relational rule $\langle |Li^T|, |\alpha_i|, |Ri^T|\rangle$ of a graph grammar and given a relational match $|m| = \{m_V, m_E\}$ of this rule in the initial state of the graph grammar, formulas $\theta_{vert_{G'}}$, $\theta_{inc_{G'}}$, $\theta_{t_V^{G'}}$, $\theta_{t_E^{G'}}$ described below define the resulting graph of the rule application. The elements that satisfy the stated formulas θ_{rel} are those that define the relations rel of the resulting typed graph $|G'^T|$. Table 1 presents the intuitive meaning and the equivalent notation of the formulas used in θ specifications.

$$\begin{aligned}
\theta_{vert_{G'}}(x) &= vert_{G0}(x) \ \vee\ nvert_{Ri}(x) \\
\theta_{inc_{G'}}(x,y,z) &= ninc_{G0}(x,y,z) \ \vee\ ninc_{Ri}(x,y,z). \\
\theta_{t_V^{G'}}(x,t) &= nvert_{G0}(x,t) \ \vee\ \Big[nvert_{Ri}(x) \ \wedge\ t_V^{Ri}(x,t) \Big]. \\
\theta_{t_E^{G'}}(x,t) &= nt_E^{G0}(x,t) \ \vee\ t_E^{Ri}(x,t).
\end{aligned}$$

This construction is described by a first-order definable transduction (i.e., by a tuple of first-order formulas) on relational structures associated to graph grammars. Details can be found in [CR09a].

Table 1: Formulas used in θ specifications

Formula	**Intuitive Meaning**	**Equivalent Notation**
$vert_{G0}(x)$	x is a vertex of graph $\|G0\|$.	-
$t_V^{Ri}(x,y)$	x is a vertex of $\|Ri\|$ of type y.	-
$t_E^{Ri}(x,y)$	x is an edge of graph $\|Ri\|$ of type y.	-
$vert_{Ri}(x) \wedge \nexists y\left(\alpha_{i_V}(y,x)\right)$	x is a vertex of graph $\|Ri\|$ that is not image of the rule $\|\alpha_i\|$.	$nvert_{Ri}(x)$
$inc_{G0}(x,y,z) \wedge \nexists w\left(m_E(w,x)\right)$	x is an edge of graph $\|G0\|$ with source y and target z that is not image of the match.	$ninc_{G0}(x,y,z)$
$\exists r,s\left[inc_{Ri}(x,r,s) \wedge \overline{n}(r,y) \wedge \overline{n}(s,z)\right]$	x is an edge of graph $\|Ri\|$ with source and target vertices given by binary relation $\overline{n}$.	$ninc_{Ri}(x,y,z)$
$\begin{cases} \exists v\left(\alpha_{i_V}(v,r) \wedge m_V(v,y)\right) \text{ if } r \neq y \\ \nexists v\ \alpha_{i_V}(v,r) \text{ if } r = y \end{cases}$	Vertex r is related to some different vertex y if it is image of the rule applied to some vertex v. In this case r is related with the image of the match applied to v. Vertex r is related to itself if it is not image of the rule.	$\overline{n}(r,y)$
$vert_{G0}(x) \wedge t_V^{G0}(x,t)$	x is a vertex of $\|G0\|$ of type t.	$nvert_{G0}(x,t)$
$\exists y,z\left(inc_{G0}(x,y,z)\right) \wedge \nexists w\left(m_E(w,x)\right) \wedge t_E^{G0}(x,t)$	x is an edge of graph $\|G0\|$ of type t that is not image of the match.	$nt_E^{G0}(x,t)$

3 Event-B

Event-B [DEP] is a state-based formalism closely related to Classical B [Abr05] and Action Systems [BS89].

Definition 9 (Event-B Model, Event) An Event-B Model is defined by a tuple $EBModel = (c,s,P,v,I,R_I,E)$ where c are constants and s are sets known in the model; v are the model variables[1]; $P(c,s)$ is a collection of axioms constraining c and s; $I(c,s,v)$ is a model invariant limiting the possible states of v s.t. $\exists c,s,v \cdot P(c,s) \wedge I(c,s,v)$ - i.e. P and I characterise a non-empty set of model states; $R_I(c,s,v')$ is an initialization action computing initial values for the model variables; and E is a set of model *events*.

Given states v,v' an event is a tuple $e = (H,S)$ where $H(c,s,v)$ is the guard and $S(c,s,v,v')$ is the before-after predicate that defines a relation between current and next states. We also denote an event guard by $H(v)$, the before-after predicate by $S(v,v')$ and the initialization action by $R_I(v')$.

An event-B model is assembled from two parts, a *context* which defines the triple (c,s,P) and a *machine* which defines the other elements (v,I,R_I,E).

Model correctness is demonstrated by generating and discharging a collection of proof obligations. The model *consistency* condition states that whenever an event or an initialization action is attempted, there exists a suitable new state v' such that the model invariant is maintained - $I(v')$. This is usually stated as two separate proof obligations: a feasibility $(I(v) \wedge H(v) \Rightarrow \exists v' \cdot S(v,v'))$ and an invariant satisfaction obligation $(I(v) \wedge H(v) \wedge S(v,v') \Rightarrow I(v'))$. The behaviour of an Event-B model is the transition system defined as follows.

Definition 10 (Event-B Model Behaviour) Given $EBModel = (c,s,P,v,I, R_I,E)$, its behaviour is given by a transition system $BST = (BState, BS_0, \rightarrow)$ where: $BState = \{\langle v \rangle | v \text{ is a } state\} \cup Undef$, $BS_0 = Undef$, and $\rightarrow \subseteq BState \times BState$ is the transition relation given by the rules:

$$\boxed{\text{start}}\ \frac{R_I(v') \wedge I(v')}{Undef \rightarrow \langle v' \rangle}$$

$$\boxed{\text{transition}}\ \frac{\exists (H,S) \in E \cdot I(v) \wedge H(v) \wedge S(v,v') \wedge I(v')}{\langle v \rangle \rightarrow \langle v' \rangle}$$

According to rule *start* the model is initialized to a state satisfying $R_I \wedge I$ and then, as long as there is an enabled event (rule *transition*), the model may evolve by firing an enabled event and computing the next state according to the event's before-after predicate. Events are atomic. In case there is more than one enabled event at a certain state, the choice is non-deterministic. The semantics of an Event-B model is given in the form of proof semantics, based on Dijkstra's work on weakest preconditions [Dij76].

An extensive tool support through the Rodin Platform makes Event-B especially attractive [DEP]. An integrated Eclipse-based development environment is actively developed, and open to third-party extensions in the form of Eclipse plug-ins. The main verification technique is theorem proving supported by a collection of theorem provers, but there is also some support for model checking.

4 Graph Grammars in Event-B

The behavior of an event-B model is similar to a graph grammar: there is a notion of state (given by a set of variables in event-B, and by a graph in a graph grammar) and a step is defined by an atomic operation on the current state (an event that updates variables in event-B and a rule application in a

[1] For convenience, as in [Abr05], no distinction is made between a set of variables and a state of a system.

graph grammar). Each step should preserve properties of the state. In event-B, these properties are stated as invariants. In a graph grammar, the properties that are guaranteed to be preserved are related to the graph structure (only well-formed graphs can be generated).

Now, we will present a way to model each structure of a graph grammar GG in event-B such that it is possible to use the event-B provers to demonstrate properties of a graph grammar.

Graphs: According to Def. 2, sets V_G and E_G contain all possible vertices and edge names that may appear in graphs of this relational structure. We will define these sets as:

$V_G = vert_T \cup \mathbb{N}$, where $vert_T$ is the set of names used as vertex types in GG (we assume that $vert_T \cap \mathbb{N} = \varnothing$);

$E_G = edge_T \cup \mathbb{N}$, where $edge_T$ is the set of names used as edge types in GG (we assume that $edge_T \cap \mathbb{N} = \varnothing$).

Moreover, we assume that $vert_T \cap edge_T = \varnothing$.

The type graph is defined in an event-B context as described in Figure 1, where we define all vertex and edge types as constants, as well as the incidence relation relating them. In the axioms, we define these sets explicitly (for example, axiom $axm1$ means that $vertT = \{Vertex1, Vertex2, \ldots\}$). Text after a // is a comment.

CONTEXT ctx_GG
SETS
    `vertT` // (Type Graph) Vertices
    `edgeT` // (Type Graph) Edges
CONSTANTS
    `Vertex1 Vertex2...`
    `Edge1 Edge2...`
    `incT`
AXIOMS
    axm1 : $partition(vertT, \{Vertex1\}, \{Vertex2\}, \ldots)$
    axm2 : $partition(edgeT, \{Edge1\}, \{Edge2\}, \ldots)$
    axm3 : $incT \subseteq (edgeT \times vertT \times vertT)$
    axm4 : $partition(incT, \{Edge1 \mapsto Vertex1 \mapsto Vertex1\}, \{Edge2 \mapsto Vertex1 \mapsto Vertex2\}, \ldots)$
END

Figure 1: Event-B Type Graph

Instances of vertices and edges that appear in graphs representing states will be described by natural numbers. It is not necessary to have distinct numbers for vertices and edges: a graph may have a vertex with identity 1 as well as an edge with identity 1, these elements will be different because one will be mapped to a vertex type and the other to an edge type. To be able to manipulate instances easily, we define the functions *source*, *target* and *edgeName* (see Figure 2).

A graph typed over a type graph T is modeled by a set of variables describing its set of vertices, incidence relation, and typing functions. It is possible to state the compatibility conditions of types and source and target of edges (stated in Def. 3) as invariants. However, since we will always generate well-formed graphs (the start graph is well-formed and events implement the single-pushout construction), we will skip these invariants (each invariant that is used generates proof obligations and therefore it is advisable to use only the necessary ones).

CONSTANTS
  source
  target
  edgeName
AXIOMS
  axm5 : $source \in (\mathbb{N}\times\mathbb{N}\times\mathbb{N}) \to \mathbb{N}$
  axm6 : $\forall a,b,c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow source(a \mapsto b \mapsto c) = b$
  axm7 : $target \in (\mathbb{N}\times\mathbb{N}\times\mathbb{N}) \to \mathbb{N}$
  axm8 : $\forall a,b,c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow target(a \mapsto b \mapsto c) = c$
  axm9 : $edgeName \in (\mathbb{N}\times\mathbb{N}\times\mathbb{N}) \to \mathbb{N}$
  axm10 : $\forall a,b,c \cdot a \in \mathbb{N} \land b \in \mathbb{N} \land c \in \mathbb{N} \Rightarrow edgeName(a \mapsto b \mapsto c) = a$
END

Figure 2: Auxiliary Functions

Figure 3 shows the definition of a graph G typed over T. Invariants are used to define the types of the variables (for example, tG_V is a total function from $vertG$ to $vertT$ and tG_E is a partial function from the set of natural numbers to $edgeT$). The variables $lastV$ and $lastE$ will be used to store the last number used as identity of vertex and edge, respectively (this will be necessary to create new fresh elements in the graph).

MACHINE mch_GG
SEES ctx_GG
VARIABLES
  vertG // (Graph) Vertices
  incG // (Graph) Edges
  tG_V // Typing of vertices
  tG_E // Typing of edges
  lastV
  lastE
INVARIANTS
  inv_vertG : $vertG \in \mathbb{P}(\mathbb{N})$
  inv_incG : $incG \in \mathbb{P}(\mathbb{N}\times\mathbb{N}\times\mathbb{N})$
  inv_tG_V : $tG\_V \in vertG \to vertT$
  inv_tG_E : $tG\_E \in \mathbb{N} \nrightarrow edgeT$
  inv_lastV : $lastV \in \mathbb{N}$
  inv_lastE : $lastE \in \mathbb{N}$
EVENTS
Initialisation
  begin
    act1 : $vertG := \{10\}$
    act2 : $incG := \{20 \mapsto 10 \mapsto 10\}$
    act3 : $tG\_V := \{10 \mapsto Vertex1\}$
    act4 : $tG\_E := \{20 \mapsto Edge1\}$
    act5 : $lastV := 10$
    act6 : $lastE := 20$
  end

Figure 3: Event-B Graph G

There is a special event in an event-B model that is executed before any other. This is the initialization event. In our encoding, this event will be used to create the start graph of a graph grammar. This is done by putting initial values in the variables that correspond to graph G (see Figure 3). In an event, there is no notion of order in the attributions belonging to the same event. A triple $(a,b,c) \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ is denoted by $a \mapsto b \mapsto c$ in event-B.

Rules: Left- and right-hand sides of rules are graphs, and thus will have representations as defined previously. Additionally, we have to define the partial morphism (α_V, α_E) that maps elements from the left- to the right-hand side of the rule. A simple rule structure is illustrated in Figure 4. Since rules do not change during execution, their structures will be defined as constants.

SETS

    vertL1
    edgeL1
    vertR1
    edgeR1

CONSTANTS

    v1_L1   // vertex of LHS
    e1_L1   // edge of LHS
    v1_R1   // vertex of RHS
    v2_R1   // vertex of RHS
    e1_R1   // edge of RHS
    sourceL1
    targetL1
    edgeNameL1
    incL1
    tL1_V   (Rule 1) Typing vertices of LHS
    tL1_E   (Rule 1) Typing edges of LHS
    incR1
    tR1_V   (Rule 1) Typing vertices of RHS
    tR1_E   (Rule 1) Typing edges of RHS
    alpha1V   (Rule 1) Rule morphism: mapping vertices
    alpha1E   (Rule 1) Rule morphism: mapping edges

AXIOMS

axm11 : $partition(vertL1, \{v1\_L1\})$
axm12 : $partition(edgeL1, \{e1\_L1\})$
axm13 : $incL1 \subseteq (edgeL1 \times vertL1 \times vertL1)$
axm14 : $partition(incL1, \{e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1\})$
axm15 : $tL1\_V \in vertL1 \rightarrow vertT$
axm16 : $partition(tL1\_V, \{v1\_L1 \mapsto Vertex1\})$
axm17 : $tL1\_E \in edgeL1 \rightarrow edgeT$
axm18 : $partition(tL1\_E, \{e1\_L1 \mapsto Edge1\})$
axm17 : $partition(vertR1, \{v1\_R1\}, \{v2\_R1\})$
axm18 : $partition(edgeR1, \{e1\_R1\})$
axm19 : $incR1 \subseteq (edgeR1 \times vertR1 \times vertR1)$
axm20 : $partition(incR1, \{e1\_R1 \mapsto v1\_R1 \mapsto v2\_R1\})$
axm21 : $tR1\_V \in vertR1 \rightarrow vertT$
axm22 : $partition(tR1\_V, \{v1\_R1 \mapsto Vertex1\}, \{v2\_R1 \mapsto Vertex2\})$
axm23 : $tR1\_E \in edgeR1 \rightarrow edgeT$
axm24 : $partition(tR1\_E, \{e1\_R1 \mapsto Edge2\})$
axm25 : $sourceL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow vertL1$
axm26 : $\forall a,b,c \cdot a \in edgeL1 \wedge b \in vertL1 \wedge c \in vertL1 \Rightarrow sourceL1(a \mapsto b \mapsto c) = b$
axm27 : $targetL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow vertL1$
axm28 : $\forall a,b,c \cdot a \in edgeL1 \wedge b \in vertL1 \wedge c \in vertL1 \Rightarrow targetL1(a \mapsto b \mapsto c) = c$
axm29 : $edgeNameL1 \in (edgeL1 \times vertL1 \times vertL1) \rightarrow edgeL1$
axm30 : $\forall a,b,c \cdot a \in edgeL1 \wedge b \in vertL1 \wedge c \in vertL1 \Rightarrow edgeNameL1(a \mapsto b \mapsto c) = a$
axm31 : $alpha1V \in vertL1 \rightarrow vertR1$
axm32 : $partition(alpha1V, \{v1\_L1 \mapsto v1\_R1\})$
axm33 : $alpha1E \in edgeL1 \mathrel{+\!\!\!\!\rightarrow} edgeR1$
axm34 : $alpha1E = \varnothing$

END

Figure 4: Event-B Rule Structure

The behavior of a rule is described by an event (in the example, event $rule1$ in Figure 5). Whenever there is a pair (mV, mE) that satisfies the guard conditions, the event may happen. The guard conditions assure that this pair is actually a match from the left-hand side of the rule to graph G (see Def. 8). The actions update the state graph (graph G) according to the rule. In this example one loop edge is deleted and a vertex and a new edge are created. Variables $lastV$ and $lastE$ keep track of the last number used to identify a vertex and an edge, respectively. In this example, a vertex with number $lastV+1$ is created with type $Vertex2$, and an edge with number $lastE+1$ with type $Edge2$ is also created. The source of this new edge is the image of the only vertex in the left-hand side of the rule in G and the target is the newly created vertex.

```
EVENTS
Event   $rule1 \widehat{=}$
    any
        $mV$
        $mE$
    where
        grd1 : $mV \in vertL1 \rightarrow vertG$    // total on vertices
        grd2 : $mE \in incL1 \rightarrowtail incG$    // total and injective on edges
        grd3 : $\forall v \cdot v \in vertL1 \Rightarrow tL1\_V(v) = tG\_V(mV(v))$
            // vertex type compatibility
        grd4 : $\forall e \cdot e \in incL1 \Rightarrow tL1\_E(edgeNameL1(e)) = tG\_E(edgeName(mE(e)))$
            // edge type compatibility
        grd5 : $\forall e \cdot e \in incL1 \Rightarrow mV(sourceL1(e)) = source(mE(e)) \wedge mV(targetL1(e)) = target(mE(e))$
            // source/target compatibility
    then
        act1 : $lastV := lastV + 1$
        act2 : $lastE := lastE + 1$
        act3 : $vertG := vertG \cup \{lastV + 1\}$
        act4 : $incG := \{lastE + 1 \mapsto source(mE(e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1)) \mapsto lastV + 1\} \cup (incG \setminus \{mE(e1\_L1 \mapsto$
            $v1\_L1 \mapsto v1\_L1)\})$
        act5 : $tG\_V := tG\_V \cup \{lastV + 1 \mapsto Vertex2\}$
        act6 : $tG\_E := (tG\_E \setminus \{edgeName(mE(e1\_L1 \mapsto v1\_L1 \mapsto v1\_L1)) \mapsto Edge1\}) \cup \{lastE + 1 \mapsto Edge2\}$
    end
END
```

Figure 5: Event-B Rule Event

Proving Properties: Once the start graph and all rules are represented in the event-B model, the property to be proved can be stated as an invariant. For example, we could add the invariant $card(incG) \leq 2$, meaning that no reachable graph can have more than 2 edges. For the given example, this property is true, and this can be easily proven by the Rodin platform.

5 Final Remarks

In this paper we have defined an event-B model that faithfully describes the behavior of a given graph grammar. To define this model, we used the relational definition of graph grammars, that was proven to be equivalent to the SPO approach. Now, it is possible to use the existing theorem provers for event-B to prove properties of graph grammars, for example, using the Rodin platform.

This is an initial work in using event-B to help proving properties of graph grammars. Besides implementation, case studies are necessary to evaluate and improve the proposed approach. We could also investigate to which extent the theory of refinement, that is very well-developed in event-B, could be used to validate a stepwise development based on graph grammars.

Bibliography

[Abr05] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[BCMR07] P. Baldan, A. Corradini, U. Montanari, L. Ribeiro. Unfolding semantics of graph transformation. *Inf. Comput.* 205(5):733–782, 2007.
doi:http://dx.doi.org/10.1016/j.ic.2006.11.004

[BK02] P. Baldan, B. König. Approximating the behaviour of graph transformation systems. In *Proceedings of ICGT '02 (International Conference on Graph Transformation)*. LNCS 2505, pp. 14–29. Springer, 2002.

[BS89] R.-J. Back, K. Sere. Stepwise Refinement of Action Systems. In Snepscheut (ed.), *Proceedings of the International Conference on Mathematics of Program Construction, 375th Anniversary of the Groningen University*. Pp. 115–138. Springer-Verlag, London, UK, 1989.

[BS06] L. Baresi, P. Spoletini. On the Use of Alloy to Analyze Graph Transformation Systems. In Corradini et al. (eds.), *ICGT*. LNCS 4178, pp. 306–320. Springer, 2006.

[CGJ$^+$01] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead*. Pp. 176–194. Springer-Verlag, London, UK, 2001.

[Cou94] B. Courcelle. Monadic Second-Order Definable Graph Transductions: A Survey. *Theoretical Computer Science* 126(1):53–75, 1994.

[Cou97] B. Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second-Order Logic. Pp. 313–400 in [Roz97].

[Cou04] B. Courcelle. Recognizable Sets of Graphs, Hypergraphs and Relational Structures: A Survey. In Calude et al. (eds.), *Developments in Language Theory*. LNCS 3340, pp. 1–11. Springer, 2004.

[CR09a] S. A. da Costa, L. Ribeiro. Formal Verification of Graph Grammars using Mathematical Induction. *Electronic Notes Theoretical Computer Science* 240:43–60, 2009.
doi:http://dx.doi.org/10.1016/j.entcs.2009.05.044

[CR09b] S. A. da Costa, L. Ribeiro. Relational and Logical Approach to Graph Grammars. Technical report 359, Porto Alegre: Instituto de Informática/UFRGS, 2009.

[CW96] E. M. Clarke, J. M. Wing. Formal methods: state of the art and future directions. *ACM Computing Surveys* 28(4):626–643, 1996.
doi:http://doi.acm.org/10.1145/242223.242257

[DEP] DEPLOY. Event-B and the Rodin Platform. http://www.event-b.org/ (last accessed 8 March 2009). Rodin Development is supported by European Union ICT Projects DEPLOY (2008 to 2012) and RODIN (2004 to 2007).

[DHR$^+$07] M. B. Dwyer, J. Hatcliff, R. Robby, C. S. Pasareanu, W. Visser. Formal Software Analysis Emerging Trends in Software Model Checking. In *FOSE '07: 2007 Future of Software Engineering*. Pp. 120–136. IEEE Computer Society, 2007.
doi:http://dx.doi.org/10.1109/FOSE.2007.6

[Dij76] E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.

[Ehr79] H. Ehrig. Introduction to the algebraic theory of graph grammars. In *1st International Workshop on Graph Grammars and Their Application to Computer Science and Biology*. Lecture Notes in Computer Science 73, pp. 1–69. Springer-Verlag, Germany, 1979.

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

[RV01] J. A. Robinson, A. Voronkov (eds.). *Handbook of Automated Reasoning (in 2 volumes)*. Elsevier and MIT Press, 2001.

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electronic Notes in Theoretical Computer Science* 203(1):135–148, 2008. doi:http://dx.doi.org/10.1016/j.entcs.2008.03.039

Expressiveness of graph conditions with variables

Annegret Habel[1] **and Hendrik Radke**[2*]

[1] habel@informatik.uni-oldenburg.de
[2] radke@informatik.uni-oldenburg.de
Carl v. Ossietzky Universität Oldenburg, Germany

Abstract: Graph conditions are most important for graph transformation systems and graph programs in a large variety of application areas. Nevertheless, non-local graph properties like “there exists a path”, “the graph is connected”, and “the graph is cycle-free” are not expressible by finite graph conditions. In this paper, we generalize the notion of finite graph conditions, expressively equivalent to first-order formulas on graphs, to finite HR graph conditions, i.e., finite graph conditions with variables where the variables are place holders for graphs generated by a hyperedge replacement system. We show that graphs with variables and replacement morphisms form a weak adhesive HLR category. We investigate the expressive power of HR graph conditions and show that finite HR graph conditions are more expressive than monadic second-order graph formulas.

Keywords: Graph conditions, graphs with variables, hyperedge replacement systems, monadic-second order graph formulas, weak adhesive HLR categories.

1 Introduction

Graph transformation systems have been studied extensively and applied to several areas of computer science [Roz97, EEKR99, EKMR99] and were generalized to high-level replacement (HLR) systems [EHKP91] and weak adhesive HLR systems [EEPT06b]. Graph conditions, i.e., graph constraints and application conditions, studied e.g. in [EH86, HHT96, HW95, KMP05, EEHP06, HP09], are most important for graph transformation systems and graph programs in a large variety of application areas. Graph conditions are an intuitive, graphical, yet precise formalism, well-suited for describing structural properties. Moreover, finite graph conditions and first-order graph formulas are expressively equivalent [HP09]. Unfortunately, typical graph properties like “there exists a path”, “the graph is connected”, and “the graph is cycle-free” are not expressible by first-order graph formulas [Cou90, Cou97b] and *finite* graph conditions. They only can be expressed by *infinite* graph conditions.

In this paper, we generalize the concept of graph conditions [HP09] to HR graph conditions, i.e. graph conditions with variables where the variables are place holders for graphs generated by a hyperedge replacement (HR) system. By the HR system, we obtain a finite description of a, in general, infinite set of graphs, e.g., the set of all paths. We investigate the expressive power of HR graph conditions and show that monadic second-order (MSO) graph formulas can be expressed by equivalent HR graph conditions, but also some second-order (SO) graph properties

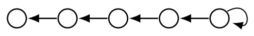
[*] This work is supported by the German Research Foundation (DFG) under grant GRK 1076/1 (Graduate School on Trustworthy Software Systems).

can be expressed by HR graph conditions.

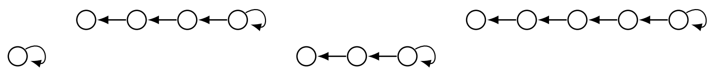
$$\begin{tikzcd}
\text{FO formulas} \arrow[r, leftrightarrow, "{[HP09]}"] \arrow[d, hook] & \text{conditions} \arrow[d, hook] \\
\text{MSO formulas} \arrow[r, "\text{this paper}"] \arrow[d, hook] & \text{HR conditions} \arrow[dl, dashed] \\
\text{SO formulas} &
\end{tikzcd}$$

The usefulness of HR conditions is illustrated by an example of a car platooning maneuver protocol.

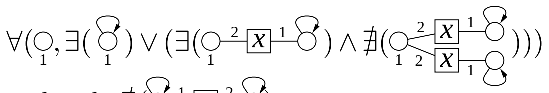
Example 1 (Car platooning) *In the following, we study "a prototypical instance of a dynamic communication system", originally taken from the California Path project [HESV91]. It represents a protocol for cars on a highway that can organize themselves into platoons, by driving close together, with the aim to conserve space and fuel. A car platoon is modeled as a directed graph where the nodes represent the cars and the direct edges the direct following relation. Additionally, the leader of a car platoon is marked by a loop.*

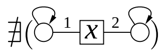


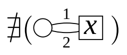
A car platooning state graph consists of zero or more car platoons.

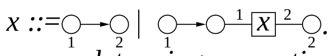


Car platooning operations like splitting a car platoon in two car platoons, or joining two car platoons into a single one can be described by graph replacement rules. When performing these operations, certain car platooning properties have to be satisfied:

(1) Every follower has a unique leader: 

(2) Leaders are not connected by a directed path: 

(3) The car platooning state graph is circle-free: 

with 

The car platooning properties are described by HR graph conditions. Bauer [Bau06] and Pennemann [Pen09] model the following relation with respect to the leader, but not the direct following relation. HR graph conditions allow to express path conditions as in the car platooning example.

The paper is organized as follows: In Section 2, we introduce graphs with variables. In Section 3, we generalize graph conditions to HR graph conditions, i.e. graph conditions with variables equipped with a hyperedge replacement (HR) system. In Section 4, we present a number of examples for HR conditions. In Section 5, we investigate the expressive power of HR conditions. A conclusion including further work is given in Section 6.

2 Graphs with variables

Graphs with variables consist of nodes, edges, and hyperedges. Edges have one source and one target and are labeled by a symbol of an alphabet; hyperedges have an arbitrary sequence of attachment nodes and are labeled by variables.

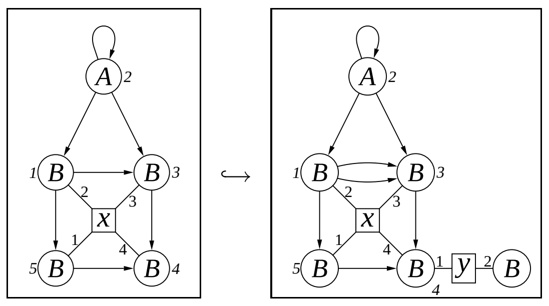
Definition 1 (Graphs with variables) Let $\mathrm{C} = \langle \mathrm{C_V}, \mathrm{C_E}, \mathrm{Var} \rangle$ be a fixed, finite label alphabet where Var is a set of variables with a mapping $\mathrm{rank}\colon \mathrm{Var} \to \mathbb{N}_0$ defining the rank of each variable. A *graph with variables* over C is a system $G = (\mathrm{V}_G, \mathrm{E}_G, \mathrm{Y}_G, \mathrm{s}_G, \mathrm{t}_G, \mathrm{att}_G, \mathrm{lv}_G, \mathrm{le}_G, \mathrm{ly}_G)$ consisting of finite sets V_G, E_G, and Y_G of *nodes* (or *vertices*), *edges*, and *hyperedges*, *source* and *target functions* $\mathrm{s}_G, \mathrm{t}_G\colon \mathrm{E}_G \to \mathrm{V}_G$, an *attachment function* $\mathrm{att}: \mathrm{Y}_G \to \mathrm{V}_G^*$, and *labeling functions* $\mathrm{lv}_G\colon \mathrm{V}_G \to \mathrm{C_V}$, $\mathrm{le}_G\colon \mathrm{E}_G \to \mathrm{C_E}$, $\mathrm{ly}\colon \mathrm{Y}_G \to \mathrm{Var}$ such that, for all $y \in \mathrm{Y}_G$, $|\mathrm{att}(y)| = \mathrm{rank}(\mathrm{ly}_G(y))$. For $y \in \mathrm{Y}_G$, $\mathrm{rank}(y) = |\mathrm{att}(y)|$ denotes the *rank* of y. For $\mathrm{Y}_G = \emptyset$, G is a *graph*. The *size* of a graph G is the number of nodes and edges, i.e., $\mathrm{size}(G) = |\mathrm{V}_G| + |\mathrm{E}_G|$. $\mathscr{G}_{\mathrm{Var}}$ denotes the set of all graphs with variables, $\mathscr{G}$ the set of all graphs, and $\mathscr{G}^n$ the set of all graphs of size $\leq n$. For $x \in \mathrm{Var}$ with $\mathrm{rank}(x) = n$, $x^\bullet$ denotes the graph with the nodes $v_1, \ldots, v_n$ and one hyperedge attached to $v_1 \ldots v_n$. A *pointed* graph with variables $\langle R, \mathrm{pin}_R \rangle$ is a graph with variables R together with a sequence $\mathrm{pin}_R = v_1 \ldots v_n$ of pairwise distinct nodes from R. We write $\mathrm{rank}(R)$ for the number n of nodes and Pin_R for the set $\{v_1, \ldots, v_n\}$.

Remark 1 *The definition extends the well-known definition of graphs [Ehr79] by the concept of hyperedges in the sense of [Hab92]. Graphs with variables also may be seen as special hypergraphs where the set of hyperedges is divided into a set of edges labelled with terminal symbols (of* $\mathrm{C_E}$*) and a set of hyperedges labelled by nonterminal symbols (of* Var*).*

We extend the definition of graph morphisms to the case of graphs with variables.

Definition 2 (Graph morphisms with variables) A *(graph) morphism (with variables)* $g\colon G \to H$ consists of functions $g_\mathrm{V}\colon \mathrm{V}_G \to \mathrm{V}_H$, $g_\mathrm{E}\colon \mathrm{E}_G \to \mathrm{E}_H$, and an injective function $g_\mathrm{Y}\colon \mathrm{Y}_G \to \mathrm{Y}_H$ that preserve sources, targets, attachment nodes, and labels, that is, $\mathrm{s}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{s}_G$, $\mathrm{t}_H \circ g_\mathrm{E} = g_\mathrm{V} \circ \mathrm{t}_G$, $\mathrm{att}_H = \mathrm{att}_G$, $\mathrm{lv}_H \circ g_\mathrm{V} = \mathrm{lv}_G$, $\mathrm{le}_H \circ g_\mathrm{E} = \mathrm{le}_G$, and $\mathrm{ly}_H \circ g_\mathrm{Y} = \mathrm{ly}_G$. A morphism g is *injective* (*surjective*) if g_V, g_E, and g_Y are injective (surjective), and an *isomorphism* if it is both injective and surjective. In the latter case G and H are *isomorphic*, which is denoted by $G \cong H$. The *composition* $h \circ g$ of g with a graph morphism $h\colon H \to M$ consists of the composed functions $h_\mathrm{V} \circ g_\mathrm{V}$, $h_\mathrm{E} \circ g_\mathrm{E}$, and $h_\mathrm{Y} \circ g_\mathrm{Y}$. For a graph G, the *identity* $\mathrm{id}_G\colon G \to G$ consists of the identities $\mathrm{id}_{G\mathrm{V}}$, $\mathrm{id}_{G\mathrm{E}}$, and $\mathrm{id}_{G\mathrm{Y}}$ on G_V, G_E, and G_Y, respectively.

Example 2 *Consider the graphs G and H over the label alphabet* $\mathrm{C} = \langle \{A, B\}, \{\square\}, \mathrm{Var} \rangle$ *where the symbol* $\square$ *stands for the invisible edge label and is not drawn and* $\mathrm{Var} = \{x, y\}$ *is the set of variables of rank* 4 *and* 2*, respectively. The graph G contains five nodes with the labels A and B, respectively, seven edges with label* $\square$ *which is not drawn, and one hyperedge of rank 4 with label x. Additionally, the graph H contains a node, an edge, and a hyperedge of rank 2 with label y.*

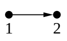


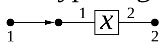
G H

The drawing of graphs with variables combines the drawing of graphs in [Ehr79] and the drawing of hyperedges in [Hab92, DHK97]: Nodes are drawn by circles carrying the node label inside, edges are drawn by arrows pointing from the source to the target node and the edge label is placed next to the arrow, and hyperedges are drawn as boxes with attachment nodes where the i-th tentacle has its number i written next to it and is attached to the i-th attachment node and the label of the hyperedge is inscribed in the box. Arbitrary graph morphisms are drawn by usual arrows "$\rightarrow$"; the use of "$\hookrightarrow$" indicates an injective graph morphism. The actual mapping of elements is conveyed by indices, if necessary.

In [PH96, Pra04], variables are substituted by arbitrary graphs. In this paper, variables are replaced by graphs generated by a hyperedge replacement system. It can be shown that satisfiability by substitution and satisfiability by replacement are expressively equivalent. By our opinion, the second satisfiability notion is the more adequate.

Definition 3 (HR systems) A *hyperedge replacement (HR) system* $\mathscr{R}$ is a finite set of replacement pairs of the form x/R where x is a variable and R a pointed graph with $\mathrm{rank}(x) = \mathrm{rank}(R)$. Given a graph G, the application of the replacement pair x/R to a hyperedge y with label x and $\mathrm{rank}(y) = \mathrm{rank}(x)$ proceeds in two steps: 1. Remove the hyperedge y from G, yielding the graph $G-\{y\}$. 2. Construct the disjoint union $(G-\{y\})+R$ and fuse the i^{th} node in $\mathrm{att}_G(y)$ with the i^{th} attachment point of R, for $i = 1,\ldots,\mathrm{rank}(y)$, yielding the graph H. Then G *directly derives* H by x/R applied to y, denoted by $G \Rightarrow_{x/R,y} H$ or $G \Rightarrow_{\mathscr{R}} H$ provided $x/R \in \mathscr{R}$. A sequence of direct derivations $G \Rightarrow_{\mathscr{R}} \ldots \Rightarrow_{\mathscr{R}} H$ is called a *derivation* from G to H, denoted by $G \Rightarrow^*_{\mathscr{R}} H$. For every variable x, $\mathscr{R}(x) = \{G \in \mathscr{G} \mid x^\bullet \Rightarrow^*_{\mathscr{R}} G\}$ denotes the set of all graphs derivable from $x^\bullet$ by $\mathscr{R}$.

Example 3 *The hyperedge replacement system $\mathscr{R}$ with the rules given in Backus-Naur form* x ::= 

| 

generates the set of all directed paths from node 1 to node 2.

Assumption 1 *In the following, let $\mathscr{R}$ be a fixed HR-system.*

Hyperedge replacement systems define replacements. A replacement morphism consists of a replacement and a graph morphism.

Definition 4 (Replacement morphisms) A *replacement* is a finite set $\rho = \{y_1/R_1, \ldots, y_n/R_n\}$ of pairs y_i/R_i where, for $i = 1,\ldots,n$, y_i is a hyperedge, $R_i \in \mathscr{R}(\mathrm{ly}(y_i))$ is a pointed graph with $\mathrm{rank}(y_i) = \mathrm{rank}(R_i)$, and $y_1,\ldots,y_n$ are pairwise distinct. The application of ρ to a graph with variables G yields the graph $\rho(G)$ obtained from G by applying $\mathrm{ly}(y_i)/R_i \in \mathscr{R}$ to y_i for $i = 1, \dots n$. $\emptyset$ denotes the *empty* replacement. A *(replacement) morphism* $\hat{g} = \langle g, repl \rangle \colon G \to H$ consists of a graph morphism $g \colon G \to H'$ and a replacement with $\rho(H') = H$. It is *injective* if g is injective. $\langle g, \emptyset \rangle$ is a *graph morphism* and $\langle \mathrm{id}_G, \emptyset \rangle$ the *identity*.

Remark 2 *For every replacement morphism* $\langle g, \rho \rangle \colon G \to H$ *with graph morphism* $g \colon G \to H'$ *and* $\rho(H') = H$*, there is a pair* $\langle \rho', g' \rangle \colon G \to H$ *with replacement* $\rho' = \{y/R \mid g(y)/R \in \rho\}$ *and a graph morphism with* $g'|_{G - \mathrm{Y}_G} = g$ *and* $g'|_{\rho'(y)} = \mathrm{id}$ *for all* $y \in \mathrm{Y}_G$*. In the following, we often use replacement morphisms consisting of a replacement and a morphism.*

$$\begin{tikzcd}
G \arrow[r, "g"] \arrow[d, Rightarrow, "\rho'"'] & H' \arrow[d, Rightarrow, "\rho"] \\
G' \arrow[r, "g'"'] & H
\end{tikzcd}$$

Remark 3 *For replacement morphisms* $\langle g_1, \rho_1 \rangle \colon G \to H$ *and* $\langle g_2, \rho_2 \rangle \colon H \to I$*, the* composition *is defined by* $\langle g_2' \circ g_1, \rho_2 \circ \rho_1^* \rangle \colon G \to I$ *where the graph morphisms and replacements are as in the figure below.*

$$\begin{tikzcd}
G \arrow[r, "g_1"] & H' \arrow[r, "g_2'"] \arrow[d, Rightarrow, "\rho_1"] & I'' \arrow[d, Rightarrow, "\rho_1^*"] \\
& H \arrow[r, "g_2"] & I' \arrow[d, Rightarrow, "\rho_2"] \\
& & I
\end{tikzcd}$$

Fact 1 *The composition of replacement morphisms is a replacement morphism.*

Notation 1 *Replacements* ρ *with* $\rho(G) = H$ *are denoted by* $\rho \colon G \Rightarrow H$*.*

In [Pra04], Ulrike Prange sketches that graphs and graph morphisms with variables based on substitution form a category and that the category with the class $\mathscr{M}$ of all injective graph morphisms is an adhesive HLR category. Similarly, one can show that graphs with variables and replacement morphisms form a category and the category with the class $\mathscr{M}$ of all injective graph morphisms is a weak adhesive HLR category.

Fact 2 (Category XGraphs) *Graphs with variables and replacement morphisms form the category* XGraphs.

Proof. Consequence of the associativity and identity of replacements and graph morphisms. □

Fact 3 (XGraphs is weak adhesive HLR) *The category* $\langle \mathrm{XGraphs}, \mathscr{M} \rangle$ *of graphs with variables with the class* $\mathscr{M}$ *of all injective graph morphisms is a weak adhesive HLR category.*

Proof. The proof is given in the Appendix B. □

3 HR conditions

Graph conditions are a well-known concept for describing graph properties in a graphical way by graphs and graph morphisms (see e.g. [EH86, HHT96, HW95, KMP05, EEHP06, HP09]). In the following, we generalize the concept to HR conditions. HR conditions are conditions in the category of graphs with variables where the variables may be replaced by graphs generated by a hyperedge replacement (HR) system. Additionally, HR conditions may contain conditions of the form $P \sqsubseteq C$ where P and C are graphs with variables with the meaning "is subgraph of". This is a counterpart to the MSO subformula $x \in X$ with the meaning "is element in".

Assumption 2 *In the following, let $\mathscr{M}'$ be the class of all injective replacement morphisms.*

Definition 5 (HR conditions) A *condition (with variables)* over a P is of the form true, $P' \sqsubseteq C$ or $\exists(a,c)$, where P', P, and C are graphs with variables, $P' \subseteq P$, $a\colon P \to C$ a morphism, and c a condition over C. Moreover, Boolean formulas over conditions over P are conditions over P. $\exists a$ abbreviates $\exists(a,\text{true})$, $\forall(a,c)$ abbreviates $\neg\exists(a,\neg c)$. A *HR condition* $\langle c,\mathscr{R}\rangle$ consists of a condition with variables c and a HR system $\mathscr{R}$. If $\mathscr{R}$ is clear from the context, we only write the condition c. A HR condition is *finite*, if every conjunction and every disjunction is finite.

$$\begin{tikzcd}
\exists(P \arrow[rr, "a"] \arrow[dr, "\hat{p}"'] \arrow[rr, phantom, "=" below, shift right=12] & & C, \arrow[dl, hook', "\hat{q}"] & c \arrow[l] \arrow[l, phantom, "\not\models" below]) \\
& G & &
\end{tikzcd}$$

Every replacement morphism *satisfies* true. A replacement morphism $\hat{p}\colon P \to G$ *satisfies* $P' \sqsubseteq C$ if $\hat{p}(P') \subseteq \hat{q}(C)$, and $\exists(a,c)$ if there exists a replacement morphism $\hat{q}$ in $\mathscr{M}'$ such that $\hat{q} \circ a = \hat{p}$ and, if c is a condition over C, $\hat{q}$ satisfies c. The satisfaction of conditions by replacement morphisms is extended to Boolean formulas over conditions in the usual way. Every graph *satisfies* true and a graph G *satisfies* the condition c, if c is a condition over $\emptyset$ and the morphism $\emptyset \to G$ satisfies c. We write $\hat{p} \models c$ [$G \models c$] to denote that all replacement morphisms $\hat{p}$ [graphs G] satisfy c. Two conditions c and c' are *equivalent*, denoted by $c \equiv c'$, if, for all replacement morphisms $\hat{p}$, $\hat{p} \models c$ iff $\hat{p} \models c'$.

Remark 4 *The definition generalizes the definitions of conditions in [HHT96, HW95, KMP05, EEHP06, HP09]. In the context of graphs, conditions are also called* constraints *and, in the context of rules, conditions are also called* application conditions. *The generalization is twofold:*

(1) *Variables in HR conditions are allowed. The variables are replaced by graphs generated by a corresponding HR system. By this generalization, several* infinite *conditions can be expressed by a* finite *HR conditions.*

(2) *Conditions of the form $P \sqsubseteq C$ are allowed. Typical examples are are $\underset{x}{\bullet} \sqsubseteq \boxed{X}$ with $X ::= \emptyset \mid \boxed{X}\ \bullet$ and $\bullet\!\!\longrightarrow\!\!\bullet \sqsubseteq \boxed{X}$ with $X ::= \emptyset \mid \boxed{X}\ \bullet\!\!\longrightarrow\!\!\bullet$. By this generalization, there is a transformation of MSO formulas into equivalent HR conditions.*

Example 4 *Consider the HR conditions*

$$\begin{array}{lcl}
\texttt{path}(1,2) & = & \exists(\underset{1}{\bullet}\ \ \underset{2}{\bullet} \to \underset{1}{\bullet}\overset{1}{-}\boxed{X}\overset{2}{\to}\underset{2}{\bullet}) \\
\texttt{connected} & = & \forall(\emptyset \to \underset{1}{\bullet}\ \ \underset{2}{\bullet}, \texttt{path}(1,2)) \\
\texttt{cyclefree} & = & \nexists(\emptyset \to \underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X}) \\
\texttt{hamiltonian} & = & \exists(\emptyset \to \underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X}, \nexists(\underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X} \to \underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X}\ \bullet))
\end{array}$$

with the HR system $x ::= \underset{1}{\bullet}\!\to\!\underset{2}{\bullet} \mid \underset{1}{\bullet}\!\to\!\bullet\overset{1}{-}\boxed{X}\overset{2}{\to}\underset{2}{\bullet}$. *A morphism with domain* $\underset{1}{\bullet}\ \ \underset{2}{\bullet}$ *satisfies the condition* $\texttt{path}(1,2)$ *iff there exists a path from the image of* 1 *to the image of* 2 *in the range. A graph satisfies* $\texttt{connected}$ *iff, for each pair of distinct nodes, there is a nonempty path, i.e., the graph is strongly connected. It satisfies* $\texttt{cyclefree}$ *iff there does not exist a cycle, i.e.,the graph is cycle-free. A graph satisfies* $\texttt{hamiltonian}$ *iff there exists a circuit and there is no additional node outside the circuit, i.e., the graph is hamiltonian.*

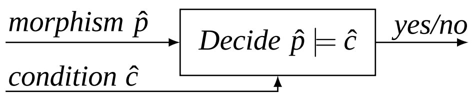
Notation 2 *For a morphism* $a\colon P \to C$ *in a condition, we just depict* C*, if* P *can be unambiguously inferred, i.e. for constraints over the empty graph* $\emptyset$ *and for conditions over some left- or right-hand side of a rule. E.g. the constraint* $\texttt{hamiltonian}$ *has the short notation* $\exists(\underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X}, \nexists(\underset{1}{\bullet}\overset{1}{\underset{2}{\leftrightarrows}}\boxed{X}\ \bullet))$.

In the following, we investigate the model checking problem for HR conditions:

Given:	A HR condition c and a graph G
Question:	$G \models c$?

For finite HR conditions, the model checking problem is decidable.

Theorem 1 (Decidability of the model checking problem) *For every finite HR condition* $\hat{c}$ *and every morphism* $\hat{p}$ *[graph* G*], it is decidable whether or not* $\hat{p} \models \hat{c}$ *[*$G \models \hat{c}$*].*



Proof. Let $\hat{c} = \langle c, \mathscr{R}\rangle$ be a finite HR condition. Without loss of generality, we may assume that $\mathscr{R}$ is monotone, i.e., for each rule $x/R \in \mathscr{R}$, $\text{size}(x^\bullet) \leq \text{size}(R)$. Otherwise, we transform $\mathscr{R}$ into an equivalent monotone HR system (see [Hab92], Theorem 1.5). Let $\hat{p} = \langle p_0, p\rangle\colon P \to G$ be a morphism. Suppose that $\text{size}(G) = n$ for some $n \geq 0$. Let Repl denote the set of all replacements.

Claim 1 *For all* $x \in \text{Var}$*,* $n \in \mathbb{N}_0$*, and* $C \in \mathscr{G}_{\text{Var}}$*, the sets*

$$\begin{array}{lcl}
\mathscr{R}^n(x) & = & \{G \in \mathscr{G}^n \mid x^\bullet \Rightarrow^*_{\mathscr{R}} G\} \\
\text{Repl}^n(C) & = & \{\rho \in \text{Repl} \mid \rho(y) \in \mathscr{R}^n(\text{ly}(y)) \textit{ for all } y \in \text{Y}_C\} \\
\mathscr{R}^n(C) & = & \{\rho \in \text{Repl}^n(C) \mid \rho(C) \leq n\}
\end{array}$$

can be constructed effectively.

Proof. For $x \in \text{Var}$ and $k \in \mathbb{N}$, define sets $\mathscr{R}^n_k(x)$ recursively as follows: $\mathscr{R}^n_1(x) = \{R \in \mathscr{G}_n \mid x/R \in \mathscr{R}\}$ and, for $k \geq 1$, $\mathscr{R}^n_{k+1}(x) = \mathscr{R}^n_k(x) \cup \{\rho(R) \in \mathscr{G}^n \mid x/R \in \mathscr{R}, \rho \in \text{Repl}^n_k(R)\}$ where $\text{Repl}^n_k(R) = \{\rho \in \text{Repl} \mid \rho(y) \in \mathscr{R}^n_k(\text{ly}(y))$ for all $y \in \text{Y}_R\}$. Since $\mathscr{R}^n_k(x) \subseteq \mathscr{R}^n_{k+1}(x) \subseteq \mathscr{G}^n$ for all $k \in \mathbb{N}$ and $\mathscr{G}^n$ is finite, there is some $l(x) \in \mathbb{N}$ such that $\mathscr{R}^n_{l(x)}(x) = \mathscr{R}^n_{l(x)+1}(x)$. This implies $\mathscr{R}^n_{l(x)}(x) =$

$\mathscr{R}^n_{l(x)+m}(x)$ for all $m \geq 1$. Now all $\mathscr{R}^n_k(x)$ up to the smallest possible $l(x)$ can be constructed effectively. Furthermore, $\mathscr{R}^n(x) = \mathscr{R}^n_{l(x)}(x)$ may be verified. $\mathscr{R}^n(x) = \bigcup_{k=1}^{\infty} \mathscr{R}^n_k$. Since the sets $\mathscr{R}^n_k(x)$ are monotonically increasing subsets of $\mathscr{G}^n$ and since $\mathscr{G}^n$ is finite, there is some $l(x) \in \mathbb{N}$ such that $\mathscr{R}^n_{l(x)}(x) \subseteq \mathscr{R}^n_{l(x)+1}(x)$. By definition, $\mathscr{R}^n_{l(x)}(x) = \mathscr{R}^n_{l(x)+1}(x) = \mathscr{R}^n_{l(x)+2}(x) = \ldots$. Thus, $\bigcup_{k=1}^{\infty} \mathscr{R}^n_k = \mathscr{R}^n_{l(x)}(x)$. The second and third statement follow directly from the first one. $\square$

For "existential" HR conditions $\hat{d} = \langle d, \mathscr{R} \rangle$ with $d = \exists(a,c)$ with $a\colon P \to C$,

$$\boxed{\hat{p} \models \langle d, \mathscr{R} \rangle \iff p \models \bigvee_{\rho \in \mathscr{R}^n(C)} \rho(d).}$$

$$\begin{array}{lll}
\hat{p} \models \hat{d} & \iff & \exists \hat{q} = \langle \rho, q \rangle \in \mathscr{M}'.\hat{q} \circ a = \hat{p} \wedge \hat{q} \models c \\
 & \iff & \exists \rho \in \mathrm{Repl}.\exists q \in \mathscr{M}.q \circ \rho(a) = p \wedge q \models \rho(c) \\
 & \iff & p \models \bigvee_{\rho \in \mathrm{Repl}} \rho(d) \\
 & \iff & p \models \bigvee_{\rho \in \mathscr{R}^n(C)} \rho(d).
\end{array}$$

Since $\mathscr{R}^n(C)$ is finite, $\bigvee_{\rho \in \mathscr{R}^n(C)} \rho(d)$ is a finite. For all existential subconditions, satisfiability can be tested and, for non-existential conditions, satisfiability can be inferred. For a graph G and a HR condition $\langle d, \mathscr{R} \rangle$ where d is a condition over $\emptyset$, $G \models \langle d, \mathscr{R} \rangle \iff \emptyset \to G \models \langle d, \mathscr{R} \rangle$. $\square$

Remark 5 *Theorem 1 makes use of the monotonicity property of HR systems. Allowing monotone replacement system instead of (monotone) HR systems one would get a corresponding decidability result.*

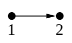
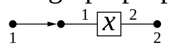
4 A classification of graph properties

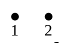
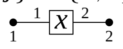
By a *graph property*, we mean a predicate on the class of graphs that is stable under isomorphism. In the following, we collect a number of graph properties known to be first order, monadic second-order, and second order, respectively, and show that most of them can be expressed by HR conditions.

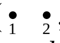
Fact 4 (classification of graph properties [Cou90]) *The following properties of a directed, labelled graph G and nodes v and w are first order (FO), monadic second-order (MSO), and second order (SO), respectively:*

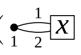
properties of a directed, labelled graph G	*FO*	*MSO*	*SO*
– *simple*	*yes*	*yes*	*yes*
– *k-regular*	*yes*	*yes*	*yes*
– *degree* $\leq k$	*yes*	*yes*	*yes*
– *has a nonempty path from v to w*	*no*	*yes*	*yes*
– *(strongly) connected*	*no*	*yes*	*yes*
– *planar*	*no*	*yes*	*yes*
– *k-colorable*	*no*	*yes*	*yes*
– *Hamiltonian*	*no*	*yes*	*yes*
– *is a tree*	*no*	*yes*	*yes*
– *is a square grid*	*no*	*yes*	*yes*
– *has an even number of nodes*	*no*	*no*	*yes*
– *has as many edges labelled a as b*	*no*	*no*	*yes*
– *has a nontrivial automorphism*	*no*	*no*	*yes*
– $\mathrm{card}(V_G)$ *belongs to a given nonrecursive set*	*no*	*no*	*no*

The following monadic second-order graph properties can be expressed by HR conditions.

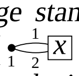
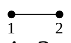
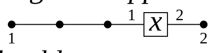
Example 5 (MSO graph properties) *For the hyperedge replacement system with the rules* $x ::=$  |  *, we have the following:*

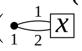
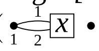
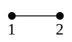
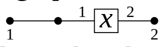
1. **Paths.** *A* nonempty path *in G is here a sequence of nodes* $(v_1, v_2, \dots, v_n)$ *with* $n \geq 2$ *such that there is an edge with source* v_i *and target* v_{i+1} *for all i and* $v_i = v_j \Rightarrow \{i, j\} = \{1, n\}$*; if* $v_1 = v_n$*, this path is a* circuit. *The HR condition* $\mathtt{path}(1,2) = \exists($  $\rightarrow$  $)$ *requires that there is a nonempty path from the image of 1 to the image of 2.*

2. **Connectedness.** *The HR condition* $\mathtt{connected} = \forall($  $, \mathtt{path}(1,2))$ *requires that, for each pair of distinct nodes, there is a nonempty path, i.e., the graph is strongly connected.*

3. **Cycle-freeness.** *The HR condition* $\mathtt{cyclefree} = \nexists($  $)$ *requires that the graph is cycle-free.*

4. **Planarity.** *By Kuratowski's Theorem (see e.g. [Eve79]) a graph is* planar *if and only if it has no subgraph homeomorphic to* $K_{3,3}$ *or* K_5. *Two graphs are* homeomorphic *if both can be obtained from the same graph by insertion of new nodes of degree 2, in edges, i.e. an edge is replaced by a path whose intermediate nodes are all new. The HR condition* $\mathtt{planar} = \nexists(K_5^*) \wedge \nexists(K_{3,3}^*)$ *where* K_5^* *and* $K_{3,3}^*$ *are obtained from the graphs* K_5 *and* $K_{3,3}$ *by replacing all edges by hyperedges with label x, respectively, requires that the graph has no subgraph homeomorphic to* K_5 *or* $K_{3,3}$*, i.e. that the graph is planar.*

5. **Coloring.** *A* coloring *of a graph is an assignment of colors to its nodes so that two adjacent nodes have the same color. A k-coloring of a graph G uses k colors. By König's characterization (see e.g. [Har69]), a graph is 2-colorable if and only if it has no odd cycles. For undirected graphs, i.e., graphs in which each undirected edge stands for two directed edges in opposite direction, the HR condition* $2\mathtt{color} = \nexists($  $)$ *with* $x ::=$  |  *requires that there are no cycles of odd length, i.e., the graph is 2-colorable.*

6. **Hamiltonicity.** *A graph is* Hamiltonian*, if there exists a Hamiltonian circuit, i.e. a simple circuit on which every node of the graph appears exactly once (see e.g. [Eve79]). For undirected graphs, the HR condition* $\texttt{hamiltonian} = \exists($  $, \nexists($  $))$ *with* $x ::=$  $|$  *requires that there exists a simple circuit in the graph and there is no additional node in the graph, i.e. every node of the graph lies on the circuit, the graph is Hamiltonian.*

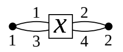
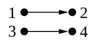
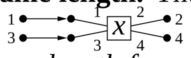
7. **Trees.** *A graph G is a* tree *if is connected, cycle-free, and has a root. A graph G has a* root *v if v is a node in G and every other node v′ in G is* reachable *from v, i.e. there is a directed path from v to v′ (see e.g. [Eve79]). For undirected graphs, the HR condition* $\texttt{tree} = \texttt{uconnected} \wedge \texttt{ucyclefree} \wedge \exists(\underset{1}{\bullet}, \forall(\underset{1}{\bullet}\ \ \underset{2}{\bullet}, \texttt{upath}(1,2)))$ *(with the undirected versions of* $\texttt{connected}$*,* $\texttt{cyclefree}$*, and* $\texttt{path}(1,2)$*) requires that the graph is connected, cycle-free, and has a root, i.e., the graph is a tree.*

The following second-order graph properties can be expressed by HR graph conditions.

Example 6 (SO graph properties)

1. **Even number of nodes.** *The HR condition* $\texttt{even} = \exists(\boxed{x}, \nexists(\boxed{x}\ \bullet))$ *with* $x ::= \ \emptyset \mid \bullet\ \bullet\ \boxed{x}$ *expresses the SO graph property "the graph has an even number of nodes".*

2. **Equal number of** a**'s and** b**'s.** *The HR condition* $\texttt{equal} = \exists(\boxed{x}, \nexists(\boxed{x}\ ⓐ) \wedge \nexists(\boxed{x}\ ⓑ))$ *with* $x ::= \ \emptyset \mid ⓐ\ ⓑ\ \boxed{x}$ *expresses the SO graph property "the graph has as many nodes labelled a as b".*

3. **Paths of same length.** *The HR condition* $\texttt{2paths}_{1,2} = \exists(\underset{1}{\bullet}\ \underset{2}{\bullet} \to$  $)$ *with* $x ::=$  $|$  *expresses the SO graph property "there exist two node-disjoint paths of same length from the image of 1 to the image of 2".*

5 Expressiveness of HR conditions

We are interested in classifying HR conditions, i.e. we want to classify the kind of graph properties that can be expressed by HR conditions. To this effect, we compare HR conditions and monadic second-order formulas on graphs [Cou90, Cou97a]. We show that there is transformations from MSO formulas into equivalent HR conditions. Vice versa, there is no such a transformation: HR graph conditions can express second-order graph properties.

Let Rel be a finite set of relation symbols. Let Var contain individual variables and relation variables of arity one. Since a relation with one argument is nothing but a set, we call these variables *set variables*. A *monadic second-order formula* over Rel is a second-order formula written with Rel and Var: the quantified and free variables are individual or set variables; there is no restriction on the arity of symbols in Rel. In order to get more readable formulas, we shall write $x \in X$ instead of $X(x)$ where X is a set variable.

Definition 6 (MSO graph formulas) Let Var be a countable set of individual and set variables. The set of all *monadic second-order (MSO) graph formulas* (over Var) is inductively defined:

For $b \in \mathrm{C}$ and $x,y,z,X \in \mathrm{Var}$, $\mathrm{l}_b(x)$, $\mathrm{inc}(x,y,z)$, $x = y$, and $x \in X$ are formulas. For formulas F, F_i ($i \in I$) and variables $x,X \in \mathrm{Var}$, $\neg F$, $\wedge_{i\in I} F_i$, $\exists x F$, and $\exists X F$ are formulas. For a formula F, $\mathrm{Free}(F)$ denotes the set of all *free* variables of F. A formula is *closed*, if $\mathrm{Free}(F) \neq \emptyset$ does not contain free variables. The *semantic* $G[\![F]\!](\sigma)$ of a formula F in a non-empty graph G under assignment $\sigma\colon \mathrm{Var} \to \mathrm{V}_G{+}\mathrm{E}_G$ is inductively defined as follows. The semantics of the formulas $\mathrm{l}_b(x)$, $\mathrm{inc}(x,y,z)$, $x = y$, $\neg F$, $\wedge_{i\in I} F_i$, and $\exists x F$ in G under σ is defined as usual (see [HP09]). $G[\![x \in X]\!](\sigma) = \mathit{true}$ iff $\sigma(x) \in \sigma(X)$ and $G[\![\exists X F]\!](\sigma) = \mathit{true}$ iff $G[\![F]\!](\sigma\{X/D\}) = \mathit{true}$ for some $D \subseteq \mathrm{V}_G$ or $D \subseteq \mathrm{E}_G$ where $\sigma\{X/D\}$ is the modified assignment with $\sigma\{X/D\}(X) = D$ and $\sigma\{X/D\}(x) = \sigma(x)$ otherwise. A graph G *satisfies* a formula F, denoted by $G \models F$, iff for all assignments $\sigma\colon \mathrm{Var} \to D_G$, $G[\![F]\!](\sigma) = \mathit{true}$.

Notation 3 *The expression $\vee_{i\in I} F_i$ abbreviates the formula $\neg \wedge_{i\in I} \neg F_i$, $F \Rightarrow G$ abbreviates $\neg F \vee G$, $\forall x F$ abbreviates $\neg\exists x \neg F$, and $\forall X F$ abbreviates $\neg\exists X \neg F$.*

Example 7 *The MSO formula $F_0(x_1,x_2) = \forall X[\{\forall y \forall z(y \in X \wedge \mathrm{edg}(y,z) \Rightarrow z \in X) \wedge \forall y(\mathrm{edg}(x_1,y) \Rightarrow y \in X)\} \Rightarrow x_2 \in X]$ [Cou97a] expresses the property "There is a nonempty path from x_1 to x_2". The formula $F_1 = (x = y) \vee F_0(x,y)$ expresses the property "$x = y$ or there is a nonempty path from x to y" and the formula $F_2 = \forall x,y[F_1(x,y)]$ expresses the property "the graph is strongly connected".*

MSO graph formulas can be transferred in equivalent finite HR graph conditions.

Theorem 2 (From MSO formulas to HR conditions) *There is a transformation $\mathrm{Cond}_{\mathscr{M}}$ from MSO formulas to HR conditions, such that, for all MSO graph formulas F and all graphs G, $G \models F \Leftrightarrow G \models \mathrm{Cond}_{\mathscr{M}}(F)$.*

Proof. Let F be a MSO formula. Without loss of generality, we may assume that F is closed and rectified, i.e. distinct quantifiers bind occurrences of distinct variables; otherwise, we build the universal closure of F and rename the variables. Since F is rectified, the variables of F can be represented by isolated nodes, edges, and hyperedges in the graphs of a constructed condition.
Let P be a graph with variables. If the set $D'_P = \mathrm{Iso}_P + \mathrm{E}_P + \mathrm{Y}_P]$, the set of all isolated nodes, edges, and hyperedges in P, is a subset of the set Var of variables, then every morphism $\hat{p}\colon P \to G$ into a non-empty graph G induces an assignment $\sigma\colon \mathrm{Var} \to D_G$ such that $\hat{p} = \sigma[D'_P]$, i.e., $\hat{p}(x) = \sigma(x)$ for each $x \in D'_P$. Vice versa, an assignment $\sigma\colon \mathrm{Var} \to D_G$ induces a mapping $D'_P \to D_G$ that may be extended to a graph morphism $\hat{p}\colon P \to G$ with $\hat{p} = \sigma[D'_P]$.

$$\begin{tikzcd}
\mathrm{Free}(F) \arrow[d, phantom, sloped, "\subseteq"] & \\
\mathrm{Var} \supseteq D'_P \arrow[r, dotted, no head] \arrow[d, Rightarrow, "\sigma"'] & P \arrow[d, "\hat{p}"] \\
D_G \arrow[r, dotted, no head] & G
\end{tikzcd}$$

The HR condition is given by $\mathrm{Cond}(F) = \mathrm{Cond}(\emptyset, F)$, where $\emptyset$ denotes the empty graph. For a MSO formula F and a graph P with $\mathrm{Free}(F) \subseteq D'_P \subseteq \mathrm{Var}$, the HR condition $\mathrm{Cond}(P,F)$ is constructed as follows: For the expressions known from first-order theory, we use the construction in [HP09]. For the atomic MSO expressions $x \in X$ and $\exists XF$, where x is an individual variable and X a set variable, we define the transformation as follows:

$$\begin{aligned}
\mathrm{Cond}(P, x \in X) &= \begin{cases} \underset{x}{\bullet} \sqsubseteq \boxed{X} & \text{if } x \in \mathrm{V}_P \\ \bullet\xrightarrow{x}\bullet \sqsubseteq \boxed{X} & \text{if } x \in \mathrm{E}_P \end{cases} \\
\mathrm{Cond}(P, \exists XF) &= \bigvee_i \langle \exists(P \to C, \mathrm{Cond}(C,F)), \mathscr{R}_i \rangle \text{ where} \\
& \quad C = P + \boxed{X}\,, \mathscr{R}_1 : X ::= \emptyset \mid \boxed{X}\,\bullet \text{ and } \mathscr{R}_2 : X ::= \emptyset \mid \boxed{X}\,\bullet\!\!\longrightarrow\!\!\bullet
\end{aligned}$$

The following claim is based on *$\mathscr{A}'$-satisfiability* obtained from the usual satisfiability by replacing all occurrences of $\mathscr{M}'$ by $\mathscr{A}'$, the class of all replacement morphisms. We write $\models_{\mathscr{A}}$ to denote $\mathscr{A}'$-satisfiability.

Claim 2 *For all rectified MSO graph formulas F, all graphs G, all assignments $\sigma\colon \mathrm{Var} \to D_G$, and all morphisms $\hat{p}\colon P \to G$ with $\sigma = \hat{p}[D'_P]$, $G[\![F]\!](\sigma) = true \Leftrightarrow \hat{p} \models_{\mathscr{A}} \mathrm{Cond}(P,F)$.*

Proof. The proof makes use of the proof of the corresponding statement for rectified FO formulas given in [HP09] and is done by structural induction.
Basis. For the atomic formulas $\mathrm{l}_b(x)$, $\mathrm{inc}(x,y,z)$, and $x = y$, the proof is as in [HP09]. For atomic formulas of the form $x \in X$, the statement follows directly from the definitions:

$$\begin{array}{lll}
 & G[\![x \in X]\!](\sigma) = true & \\
\Leftrightarrow & \sigma(x) \in \sigma(X) & \text{(Semantics of } x \in X) \\
\Leftrightarrow & \hat{p}(\underset{x}{\bullet}) \subseteq \hat{p}(\boxed{X}) & (\hat{p} = \sigma[D'_P],\ x, X \in \mathrm{Free}(x{\in}X) \in \mathrm{Free}(x{\in}X) \subseteq D'_P) \\
\Leftrightarrow & \hat{p} \models_{\mathscr{A}} \underset{x}{\bullet} \sqsubseteq \boxed{X} & \text{(Semantics of } \underset{x}{\bullet} \sqsubseteq \boxed{X}) \\
\Leftrightarrow & \hat{p} \models_{\mathscr{A}} \mathrm{Cond}(P, x \in X) & \text{(Definition of Cond)}
\end{array}$$

Hypothesis. Assume, the statement holds for rectified formulas F.
Step. For formulas of the form $\neg F$, $\wedge_{i \in I} F_i$, $\exists xF$, the proof is as in [HP09]. For formulas of the form $\exists XF$, graphs G, and assignments σ, the statement follows from the definitions and the induction hypothesis:

$$\begin{array}{lll}
 & G[\![\exists XF]\!](\sigma) = true & \\
\Leftrightarrow & \exists D \subseteq D_G . G[\![F]\!](\sigma\{X/D\}) = true & \text{(Semantics of } \exists XF) \\
\Leftrightarrow & \exists D \subseteq \mathrm{V}_G . G[\![F]\!](\sigma\{X/D\}) = true \text{ or} & \\
 & \exists D \subseteq \mathrm{E}_G . G[\![F]\!](\sigma\{X/D\}) = true & \text{(Assignment)} \\
\Leftrightarrow & \exists \hat{q}\colon C \to G \in \mathscr{A} . \hat{p} = \hat{q} \circ a \wedge \hat{q} \models_{\mathscr{A}} \mathrm{Cond}(C,F) & \text{(Hypothesis, } \hat{q} = \sigma\{X/D\}[D'_P]) \\
\Leftrightarrow & \hat{p} \models_{\mathscr{A}} \exists(P \to C, \mathrm{Cond}(C,F)) & \text{(Definition } \models_{\mathscr{A}},\ \hat{p} = \sigma[D'_P]) \\
\Leftrightarrow & \hat{p} \models_{\mathscr{A}} \mathrm{Cond}(P, \exists XF) & \text{(Definition Cond)}
\end{array}$$

where $a\colon P \to C$ with $C = P + \boxed{X}$. □

$\mathscr{A}$-satisfiability is closely related to the satisfiability of monadic second-order graph formulas. As in [HP09], there is a transformation from $\mathscr{A}'$- to $\mathscr{M}'$-satisfiability.

Claim 3 (from $\mathscr{A}'$- to $\mathscr{M}'$-satisfiability [HP09]) *There is a transformation* Msat *such that, for every condition c over $\emptyset$ and every graph G, $G \models_{\mathscr{A}'} c \Leftrightarrow G \models_{\mathscr{M}'} \mathrm{Msat}(c)$.*

Proof. Let $\langle \rho, p \rangle$ be a replacement morphism, $\langle p', \rho' \rangle$ the corresponding pair consisting of a morphism and a replacement, and $\hat{c} = \langle c, \mathscr{R} \rangle$ a HR condition. By the corresponding theorem in [HP09], $p' \models_{\mathscr{A}} c \Leftrightarrow p' \models \mathrm{Msat}(c)$. By the definition of the classes of replacement morphisms $\mathscr{A}'$ and $\mathscr{M}'$, $\langle p', \rho' \rangle \models_{\mathscr{A}'} c \Leftrightarrow \langle p', \rho' \rangle \models \mathrm{Msat}(c)$ and, for the corresponging pair $\langle p^*, \rho^* \rangle$, $\langle p^*, \rho^* \rangle \models_{\mathscr{A}'} c \Leftrightarrow \langle p^*, \rho^* \rangle \models \mathrm{Msat}(c)$. As a consequence, for every graph G, $\mathscr{G} \models_{\mathscr{A}'} c \Leftrightarrow G \models \mathrm{Msat}(c)$. $\square$

Now, for all graphs G and all closed, rectified MSO formulas F, we have:

$$\begin{array}{rcll}
G \models F & \Leftrightarrow & \forall \sigma\colon \mathrm{Var} \to D_G.G[\![F]\!](\sigma) = \mathit{true} & \text{(Definition } \models\text{)} \\
 & \Leftrightarrow & \emptyset \to G \models_{\mathscr{A}} \mathrm{Cond}(\emptyset, F) & \text{(Claim 2)} \\
 & \Leftrightarrow & G \models_{\mathscr{A}} \mathrm{Cond}(F) & \text{(Definition } \models_{\mathscr{A}}\text{, Cond)} \\
 & \Leftrightarrow & G \models \mathrm{Msat}(\mathrm{Cond}(F)) & \text{(Claim 3).}
\end{array}$$

Now, the HR condition $\mathrm{Cond}_{\mathscr{M}}(F) = \mathrm{Msat}(\mathrm{Cond}(F))$ has the wanted property. $\square$

Example 8 *The closure of the MSO graph formula*

$$F(x_1,x_2) = \forall X [\{\underbrace{\forall y \forall z (y \in X \wedge \mathrm{edg}(y,z) \Rightarrow z \in X)}_{G_1} \wedge \underbrace{\forall y' (\mathrm{edg}(x_1,y') \Rightarrow y' \in X)}_{G_2}\} \Rightarrow \underbrace{x_2 \in X}_{G_3}]$$

is transformed into the HR condition

$$\begin{array}{rl}
 & \mathrm{Cond}(\forall x_1 \forall x_2 F(x_1,x_2)) \\
= & \mathrm{Cond}(\emptyset, \forall x_1 \forall x_2 F(x_1,x_2)) \\
\equiv & \forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}, \mathrm{Cond}(\emptyset, \forall X[G_1 \wedge G_2 \Rightarrow G_3]))) \\
\equiv & \forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}, [\mathrm{Cond}(\boxed{X}, G_1 \wedge G_2 \Rightarrow G_3])) \\
= & \forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}, [\mathrm{Cond}(\boxed{X}, G_1) \wedge \mathrm{Cond}(\boxed{X}, G_2) \Rightarrow \mathrm{Cond}(\boxed{X}, G_3)]) \\
\equiv & \forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}, [\,\forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}\,\underset{y}{\bullet}\,\underset{z}{\bullet}, (\underset{y}{\bullet} \sqsubseteq \boxed{X} \wedge \exists(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}\,\underset{y}{\bullet}\!\longrightarrow\!\underset{z}{\bullet}) \Rightarrow \underset{z}{\bullet} \sqsubseteq \boxed{X}) \wedge \\
 & \forall(\underset{x_1}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}\,\underset{y'}{\bullet}, \exists(\underset{x_1}{\bullet}\!\longrightarrow\!\underset{y'}{\bullet}\,\underset{x_2}{\bullet}\,\boxed{X}\,\underset{y'}{\bullet}) \Rightarrow \underset{y'}{\bullet} \sqsubseteq \boxed{X}) \Rightarrow \underset{x_2}{\bullet} \sqsubseteq \boxed{X}\,]\,)
\end{array}$$

with $X ::= \emptyset \mid \boxed{X}\,\bullet$ *using the equivalence* $\forall(x(\forall(y,c)) \equiv \forall(y \circ x, c))$ *in [HP05].*

Inspecting the proof of Theorem 2, one may see that only rules of the form $X ::= \emptyset \mid \boxed{X}\,\bullet$ of $X ::= \emptyset \mid \boxed{X}\,\bullet\!\!\longrightarrow\!\!\bullet$ are used. A HR condition with rules of this form is called *HR0 condition*.

Corollary 1 *There is a transformation* $\mathrm{Cond}_{\mathscr{M}}$ *from MSO formulas to HR0 conditions, such that, for all MSO graph formulas F and all graphs G,* $G \models F \Leftrightarrow G \models \mathrm{Cond}_{\mathscr{M}}(F)$.

In Example 6, second-order graph properties are expressed by finite HR conditions. As a consequence, we obtain the following.

Corollary 2 *There is no transformation from finite HR conditions to equivalent MSO formulas.*

6 Conclusion

In this paper, we have generalized the notion of graph conditions to the one of HR graph conditions. The variables in the graphs can to be replaced by graphs generated by a assigned hyperedge replacement system. It is shown that there is a transformation from MSO formulas to HR conditions, but HR conditions are more powerful: they can express certain SO formulas. It remains the question whether or not there are transformations from HR0 conditions to MSO formulas and from HR conditions to SO formulas, respectively.

$$\begin{tikzcd}
\text{FO formulas} \arrow[r, leftrightarrow, "\text{[HP09]}"] \arrow[d, hook] & \text{conditions} \arrow[d, hook] \\
\text{MSO formulas} \arrow[r, "\text{this paper}"] \arrow[d, hook] & \text{HR0 conditions} \arrow[d, hook] \\
\text{SO formulas} & \text{HR conditions} \arrow[l, "???"']
\end{tikzcd}$$

Graphs with variables and all replacement morphisms form a category. Distinguishing the class of all injective replacement morphisms, we obtain a weak adhesive HLR category. As a consequence, we have

- conditions with variables,
- rules with variables as in [PH96],
- rules with application conditions based on graphs with variables.

We can adapt all results known for weak adhesive HLR categories.

Bibliography

[AHS90] J. Adámek, H. Herrlich, G. Strecker. *Abstract and Concrete Categories*. John Wiley, New York, 1990.

[Bau06] J. Bauer. *Analysis of Communication Topologies by Partner Abstraction*. PhD thesis, Universität Saarbrücken, 2006.

[Cou90] B. Courcelle. Graph Rewriting: An Algebraic and Logical Approach. In *Handbook of Theoretical Computer Science*. Volume B, pp. 193–242. Elsevier, Amsterdam, 1990.

[Cou97a] B. Courcelle. The Expression of Graph Properties and Graph Transformations in Monadic Second- Order Logic. In *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1, pp. 313–400. World Scientific, 1997.

[Cou97b] B. Courcelle. On the expression of graph properties in some fragments of monadic second-order logic. In Immerman and Kolaitis (eds.), *Descriptive complexity and finite models*. Pp. 33–62. DIMACS Series in Discrete Mathematics and Theoretical Computer Sciences, 1997.

[DHK97] F. Drewes, A. Habel, H.-J. Kreowski. Hyperedge Replacement Graph Grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 1, pp. 95–162. World Scientific, 1997.

[EEHP06] H. Ehrig, K. Ehrig, A. Habel, K.-H. Pennemann. Theory of Constraints and Application Conditions: From Graphs to High-Level Structures. *Fundamenta Informaticae* 74(1):135–166, 2006.

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

[EEPT06a] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Fundamental Theory of Typed Attributed Graph Transformation based on Adhesive HLR-Categories. *Fundamenta Informaticae* 74(1):31–61, 2006.

[EEPT06b] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. EATCS Monographs of Theoretical Computer Science. Springer, Berlin, 2006.

[EH86] H. Ehrig, A. Habel. Graph Grammars with Application Conditions. In Rozenberg and Salomaa (eds.), *The Book of L*. Pp. 87–100. Springer, Berlin, 1986.

[EHKP91] H. Ehrig, A. Habel, H.-J. Kreowski, F. Parisi-Presicce. Parallelism and Concurrency in High Level Replacement Systems. *Mathematical Structures in Computer Science* 1:361–404, 1991.

[Ehr79] H. Ehrig. Introduction to the Algebraic Theory of Graph Grammars. In *Graph-Grammars and Their Application to Computer Science and Biology*. LNCS 73, pp. 1–69. Springer, 1979.

[EKMR99] H. Ehrig, H.-J. Kreowski, U. Montanari, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation*. Volume 3: Concurrency, Parallelism, and Distribution. World Scientific, 1999.

[Eve79] S. Even. *Graph Algorithms*. Computer Science Press, Rockville, Maryland, 1979.

[Hab92] A. Habel. *Hyperedge Replacement: Grammars and Languages*. LNCS 643. Springer, Berlin, 1992.

[Har69] F. Harary. *Graph Theory*. Addison-Wesley, Reading, Mass., 1969.

[HESV91] A. Hsu, F. Eskafi, S. Sachs, P. Varaiya. The Design of Platoon Maneuver Protocols for IVHS. Technical report, Institute of Transportation Studies, University of California at Berkeley, 1991.

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

[HP05] A. Habel, K.-H. Pennemann. Nested Constraints and Application Conditions for High-Level Structures. In *Formal Methods in Software and System Modeling*. LNCS 3393, pp. 293–308. Springer, 2005.

[HP09] A. Habel, K.-H. Pennemann. Correctness of High-Level Transformation Systems Relative to Nested Conditions. *Mathematical Structures in Computer Science* 19:245–296, 2009.

[HW95] R. Heckel, A. Wagner. Ensuring Consistency of Conditional Graph Grammars—A Constructive Approach. In *SEGRAGRA '95*. ENTCS 2, pp. 95–104. 1995.

[KMP05] M. Koch, L. V. Mancini, F. Parisi-Presicce. Graph-based Specification of Access Control Policies. *Journal of Computer and System Sciences* 71:1–33, 2005.

[LS04] S. Lack, P. Sobociński. Adhesive Categories. In *Foundations of Software Science and Computation Structures (FOSSACS'04)*. LNCS 2987, pp. 273–288. Springer, 2004.

[Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Universität Oldenburg, 2009. http://formale-sprachen.informatik.uni-oldenburg.de/skript/fs-pub/diss_pennemann.pdf.

[PH96] D. Plump, A. Habel. Graph Unification and Matching. In *Graph Grammars and Their Application to Computer Science*. LNCS 1073, pp. 75–89. Springer, 1996.

[Pra04] U. Prange. Graphs with Variables as an Adhesive HLR Category. Technical report, Technische Universität Berlin, Falultät IV — Elektrotechnik und Informatik, 2004.

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

A Weak adhesive HLR categories

We recall the notions of weak adhesive HLR categories, i.e. categories based on objects of many kinds of structures which are of interest in computer science and mathematics, e.g. Petri-nets, (hyper)graphs, and algebraic specifications, together with their corresponding morphisms and with specific properties. Readers interested in the category-theoretic background of these concepts may consult e.g. [EEPT06b].

Definition 7 (Weak adhesive HLR category) A category $\mathscr{C}$ with a morphism class $\mathscr{M}$ is a *weak adhesive HLR category*, if the following properties hold:

1. *$\mathscr{M}$ is a class of monomorphisms closed under isomorphisms, composition, and decomposition*, i.e., for morphisms f and g, $f \in \mathscr{M}$, g isomorphism (or vice versa) implies $g \circ f \in \mathscr{M}$; $f, g \in \mathscr{M}$ implies $g \circ f \in \mathscr{M}$; and $g \circ f \in \mathscr{M}$, $g \in \mathscr{M}$ implies $f \in \mathscr{M}$.

2. *$\mathscr{C}$ has pushouts and pullbacks along $\mathscr{M}$-morphisms*, i.e. pushouts and pullbacks, where at least one of the given morphisms is in $\mathscr{M}$, and *$\mathscr{M}$-morphisms are closed under pushouts*

and pullbacks, i.e. given a pushout (1) as in the figure below, $m \in \mathcal{M}$ implies $n \in \mathcal{M}$ and, given a pullback (1), $n \in \mathcal{M}$ implies $m \in \mathcal{M}$.

3. *Pushouts in $\mathcal{C}$ along $\mathcal{M}$-morphisms are weak VK-squares*, i.e. for any commutative cube (2) in $\mathcal{C}$ with pushout (1) with $m \in \mathcal{M}$ and ($f \in \mathcal{M}$ or $b, c, d \in \mathcal{M}$) in the bottom and where the back faces are pullbacks, the following statement holds: the top face is a pushout iff the front faces are pullbacks.

$$\begin{tikzcd}
A \arrow[r] \arrow[d, "m"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, "n"] \\
B \arrow[r] & D
\end{tikzcd}$$

$$\begin{tikzcd}
& A' \arrow[rr] \arrow[dl] \arrow[dd] & & C' \arrow[dl] \arrow[dd, "c"] \\
B' \arrow[rr, crossing over] \arrow[dd, "b"'] & & D' & \\
& A \arrow[rr, "f"] \arrow[dl, "m"'] & & C \arrow[dl] \\
B \arrow[rr] & & D \arrow[from=uu, crossing over, "d"] &
\end{tikzcd} \qquad (2)$$

Fact 5 (Graphs is weak adhesive HLR [EEPT06b]) *The category* $\langle \text{Graphs}, \text{Inj} \rangle$ *of graphs with class* Inj *of all injective graph morphisms is a weak adhesive HLR category.*

Further examples of weak adhesive HLR categories are the categories of hypergraphs with all injective hypergraph morphisms, place-transition nets with all injective net morphisms, and algebraic specifications with all strict injective specification morphisms.

Weak adhesive HLR-categories have a number of nice properties, called HLR properties.

Fact 6 (HLR-properties [LS04, EEPT06b]) *For a weak adhesive HLR-category* $\langle \mathcal{C}, \mathcal{M} \rangle$*, the following properties hold:*

1. Pushouts along $\mathcal{M}$-morphisms are pullbacks.

2. $\mathcal{M}$ pushout-pullback decomposition. *If the diagram (1)+(2) in the figure below is a pushout, (2) a pullback,* $D \to F$ *in* $\mathcal{M}$ *and* ($A \to B$ *or* $A \to C$ *in* $\mathcal{M}$)*, then (1) and (2) are pushouts and also pullbacks.*

3. Cube pushout-pullback decomposition. *Given the commutative cube (3) in the figure below, where all morphisms in the top and the bottom are in* $\mathcal{M}$*, the top is pullback, and the front faces are pushouts, then the bottom is a pullback iff the back faces of the cube are pushouts.*

$$\begin{tikzcd}
A \arrow[r] \arrow[d] \arrow[dr, phantom, "(1)"] & C \arrow[r] \arrow[d] \arrow[dr, phantom, "(2)"] & E \arrow[d] \\
B \arrow[r] & D \arrow[r] & F
\end{tikzcd}$$

$$\begin{tikzcd}
& A' \arrow[rr] \arrow[dl] \arrow[dd] & & C' \arrow[dl] \arrow[dd] \\
B' \arrow[rr, crossing over] \arrow[dd] & & D' & \\
& A \arrow[rr] \arrow[dl] & & C \arrow[dl] \\
B \arrow[rr] & & D \arrow[from=uu, crossing over] &
\end{tikzcd} \qquad (3)$$

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

B Constructions and proofs

In this section, show that the category $\langle \text{XGraphs}, \mathscr{M} \rangle$ of graphs with variables with the class $\mathscr{M}$ of all injective graph morphisms is a weak adhesive HLR category. For this purpose, we show that $\mathscr{M}$ is a class of monomorphisms closed under isomorphisms, composition, and decomposition (Lemma 1), the category has pushouts and pullbacks along $\mathscr{M}$-morphisms (Lemma 2), and pushouts along $\mathscr{M}$-morphisms are weak VK squares (Lemma 3).

Lemma 1 *In* XGraphs*, the following properties hold:*

1. *$\mathscr{M}$-morphisms are monomorphisms.*
2. *$\mathscr{M}$-morphisms are closed under composition and decomposition.*

Proof. Straightforward □

In the following, we have to show the existence of pushouts and pullbacks along $\mathscr{M}$-morphisms. Since every morphism consists of a replacement and a graph mophisms, we first give a construction for an injective graph morphism and a replacement. The remainder construction can be done as usual for graph morphisms.

Lemma 2 (Pushouts and pullbacks along $\mathscr{M}$-morphisms) *For every morphism $m\colon A \hookrightarrow B$ in $\mathscr{M}$ and every replacement $\rho\colon A \Rightarrow C$, there is an object D, a morphism $m'\colon C \hookrightarrow D$ in $\mathscr{M}$ and a replacement $\rho'\colon B \Rightarrow D$ forming a pushout. Vice versa, for every morphism $m'\colon C \hookrightarrow D$ in $\mathscr{M}$ and every replacement $\rho'\colon B \Rightarrow D$, there is an object A, a morphism $m\colon A \hookrightarrow B$ in $\mathscr{M}$, and a replacement $\rho'\colon A \Rightarrow C$ and forming a pullback.*

$$\begin{tikzcd}
A \arrow[r, Rightarrow, "\rho"] \arrow[d, hook, "m"'] \arrow[dr, phantom, "(1)"] & C \arrow[d, hook, "m'"] \\
B \arrow[r, Rightarrow, "\rho'"'] & D
\end{tikzcd}$$

Proof. The pushout object D, m', and ρ' are constructed as follows.

- $\mathrm{V}_D = \mathrm{V}_B + (\mathrm{V}_C - \mathrm{V}_A)$
- $\mathrm{E}_D = \mathrm{E}_B + (\mathrm{E}_C - \mathrm{E}_A)$
- $\mathrm{Y}_D = \mathrm{Y}_B + (\mathrm{Y}_C - \mathrm{Y}_A) - m_{\mathrm{Y}}(\mathrm{Dom}(\rho))$
- $\mathrm{s}_D(e) =$ if $e \in \mathrm{E}_B$ then $\mathrm{s}_B(e)$ else if $e \in \mathrm{E}_C$ and $\mathrm{s}_C(e) \in \mathrm{V}_C - \mathrm{V}_A$ then $\mathrm{s}_C(e)$ else $m_{\mathrm{V}}(\mathrm{s}_C(e))$ for $e \in \mathrm{E}_D$. $\mathrm{t}_D(e)$ and $\mathrm{att}_D(y)$ are defined analogously.
- $\mathrm{lv}_D(v) =$ if $v \in \mathrm{V}_B$ then $\mathrm{lv}_B(v)$ else $\mathrm{lv}_C(v)$ for $v \in \mathrm{V}_D$. $\mathrm{le}_D(e)$ and $\mathrm{ly}_D(y)$ are defined analogously.
- $m'\colon C \to D$ is given by $\langle m'_{\mathrm{V}}, m'_{\mathrm{E}} \rangle$ where m'_{V} is defined by $\mathrm{m}'_{\mathrm{V}}(v) =$ if $v \in \mathrm{V}_A$ then $m_{\mathrm{V}}(v)$ else v for all $v \in \mathrm{V}_C$, and m'_{E} is defined analogously. $\rho'\colon B \Rightarrow D$ is defined by $\rho' = \{m(y)/R \mid y/R \in \rho\}$.

Then $m' \colon C \hookrightarrow D$ is a morphism in $\mathscr{M}$, $\rho' \colon B \Rightarrow D$ is a replacement, and the constructed diagram is a pushout. The pullback object A, m, and ρ are as follows.

- $\mathrm{V}_A = \mathrm{V}_B \cap \{m'_{\mathrm{V}}(v) \mid v \in \mathrm{V}_C\}$
- $\mathrm{E}_A = \mathrm{E}_B \cap \{m'_{\mathrm{E}}(e) \mid e \in \mathrm{E}_C\}$
- $\mathrm{Y}_A = \mathrm{Y}_B \cap \{m'_{\mathrm{Y}}(y) \mid y \in \mathrm{Y}_C\} \cup \mathrm{Y}_C$
- $\mathrm{s}_A(e) = \mathrm{s}_B(e)$ for all $e \in \mathrm{E}_A$. $\mathrm{t}_A(e)$ is defined analogously.
  $\mathrm{att}_A(y) =$ if $y \in C$ then $\mathrm{att}_C(y)$ else $\mathrm{att}_B(y)$ for all $y \in \mathrm{Y}_A$.
- $\mathrm{lv}_A(v) = \mathrm{lv}_B(v)$ for all $v \in \mathrm{V}_A$. $\mathrm{le}_A(e)$ and $\mathrm{ly}_A(y)$ are defined analogously.
- $m \colon A \to B$ is restriction of $m' \colon C \to D$ to A and $\rho \colon A \to C$ is defined by $\rho = \{y/R \mid y \in \mathrm{Y}_A, (m_{\mathrm{Y}}(y)/R) \in \rho'\}$.

Then $m \colon A \to B$ a graph morphism in $\mathscr{M}$, $\rho \colon A \Rightarrow C$ is a replacement, and the constructed diagram is a pullback. $\square$

Lemma 3 *In* XGraphs, *pushouts along $\mathscr{M}$-morphisms are weak VK squares.*

Proof. By case analysis, similar to the proof in [EEPT06a]. In the following, we will use the notation $ABCD$ to designate the square from $B \leftarrow A \to C$ to $B \to D \leftarrow C$. In the commutative cube below, let $ABCD$ be a pushout with $m \in \mathscr{M}$ and the back faces $A'AB'B$ and $A'AC'C$ be pullbacks. Assume that $f \in \mathscr{M}$ or $b,c,d \in \mathscr{M}$.

$$\begin{tikzcd}
& A' \arrow[rr,"g"] \arrow[dl,"n"'] \arrow[dd,"a"] & & C' \arrow[dl,"n^*"'] \arrow[dd,"c"] \\
B' \arrow[rr,"g^*"] \arrow[dd,"b"'] & & D' \arrow[dd,"d"] & \\
& A \arrow[rr,"f"] \arrow[dl,"m"'] & & C \arrow[dl,"m^*"] \\
B \arrow[rr,"f^*"'] & & D &
\end{tikzcd}
\qquad
\begin{tikzcd}
B' \arrow[rr,"g^*"] \arrow[dr,"u"] \arrow[dd,"b"'] & & D' \arrow[dd,"d"] \\
& B_2 \arrow[ur,"g_2^*"] \arrow[dl,"b_2"] & \\
B \arrow[rr,"f^*"'] & & D
\end{tikzcd}$$

We proceed by case distinction.
Case 1. $f \in \mathscr{M}$ and one of the back arrows, i.e. a,b,c, is in $\mathscr{M}$. By Lemma 2, all morphisms except the "vertical" morphisms a,b,c,d are in $\mathscr{M}$. Because the back sides of the cube are pullbacks, all of a,b,c are in $\mathscr{M}$. By commutativity of the cube, $d \in \mathscr{M}$. The proof proceeds as for the category of hypergraphs in [EEPT06a].
Case 2. $b,c,d \in \mathscr{M}$ and one of g,f,f^* is in $\mathscr{M}$. Analogous to Case 1.
Case 3. f is in $\mathscr{M}$ and none of the back arrows a,b,c are in $\mathscr{M}$. Because the back sides are pullbacks, $f^*,g \in \mathscr{M}$.
Case 3.1. Assume the top is a pushout. Then $g,g^* \in \mathscr{M}$. Assume a pullback object B_2 with morphisms $g_2^* \colon B_2 \to D'$ and $b_2 \colon B_2 \to B$. Then, there is a unique $u : B' \to B$ with $g^* = g_2^* u$ and $b = b_2 u$. Similar to [EEPT06a], we can show that u is bijective and therefore $B' \cong B_2$.
Case 4. $b,c,d \in \mathscr{M}$ and $g,f,f^* \notin \mathscr{M}$. The proof is analogous to Case 3. $\square$

Proof of Theorem 3. 1.$\mathscr{M}$ is a class of monomorphisms closed under isomorphisms, composition and decomposition: The class of all injective graph morphisms in XGraphs is a class of monomorphisms. It suffices, then, to show that $g \circ f$ is in $\mathscr{M}$ for morphisms $g, f \in \mathscr{M}$, and that $g \circ f \in \mathscr{M}$ implies $f \in \mathscr{M}$. Both propositions are true due to [AHS90, Proposition 7.34].
2. XGraphs has pushouts and pullbacks along $\mathscr{M}$-morphisms: For $A \hookrightarrow B$ in $\mathscr{M}$, the pushout of $B \hookleftarrow A \rightarrow E$ is constructed by splitting the morphism $A \rightarrow E$ into a replacement $\rho : A \Rightarrow C$ and a morphism $C \rightarrow E$, constructing (1) as pushout of $B \hookleftarrow A \Rightarrow C$ accoring to Lemma 2 and (2) as pushout of $D \hookleftarrow C \rightarrow E$ as usual (e.g. in the category of hypergraphs [EEPT06b]). Then (1)+(2) is a pushout.

$$\begin{tikzcd}
A \arrow[r, Rightarrow] \arrow[d, hook] \arrow[rr, bend left] \arrow[dr, phantom, "(1)"] & C \arrow[r] \arrow[d, hook] \arrow[dr, phantom, "(2)"] & E \arrow[d, hook] \\
B \arrow[r, Rightarrow] \arrow[rr, bend right] & D \arrow[r] & F
\end{tikzcd}$$

For $E \hookrightarrow F$ in $\mathscr{M}$, the pullback of $B \hookrightarrow F \leftarrow E$ is constructed by splitting the morphism $B \hookrightarrow F$ into a replacement $\rho : B \Rightarrow D$ and a morphism $D \rightarrow F$, constructing (2) as usual as pullback (e.g., as in the category of hypergraphs [EEPT06b]) and (1) as pullback according to Lemma 2. Then (1)+(2) is a pullback. □

From Stochastic Graph Transformations to Differential Equations

Mayur Bapodra and Reiko Heckel

Department of Computer Science, University of Leicester, UK
{mb294, reiko}@le.ac.uk

Abstract: In a variety of disciplines models are used to predict, measure or explain quantitative properties. Examples include the concentration of a chemical substance produced within a given period, the growth of the size of a population of individuals, the time taken to recover from a communication breakdown in a network, etc.

The models such properties arise from are often discrete and structural in nature. Adding information on the time and/or probability of any actions performed, quantitative models can be derived. In the first example above, commonly referred to as kinetic analysis of chemical reactions, a system of differential equations describing the evolution of concentrations is extracted from specifications of individual chemical reactions augmented with reaction rates.

Recently, this construction has inspired approaches based on stochastic process specification techniques aiming to extract a continuous, quantitative model of a system from a discrete, structural one. This paper describes a methodology for such an extraction based on stochastic graph transformations.

The approach is based on a variant of the construction of critical pairs and has been implemented using the AGG tool and validated for a simple reaction of unimolecular nucleophilic substitution (SN_1).

Keywords: stochastic graph transformations, chemical reactions, law of mass action, ordinary differential equations

1 Introduction

In chemistry, understanding the kinetics of reactions, i.e, the evolution of concentrations of chemical agents over time, is essential for a variety of purposes ranging from the interpretation of experiments in the lab to the planning of large scale industrial systems. Analogies between chemical reactions and rewriting have inspired approaches like CHAM [2] where chemical reactions serve as a paradigm for concurrency, or Kappa [8] focussing on modelling biological systems. In graph rewriting the modelling of chemical reactions has been studied in [7]. The idea is to present molecules as graphs (or graph expressions) and to formalise reaction equations as graph or term rewrite rules. Such an encoding offers opportunities for cross-fertilisation, transferring concepts, problems, and ideas between chemistry and computer science.

In this paper, we will exploit this in order to rephrase chemical reaction kinetics in terms of stochastic graph transformations. More precisely, we want to derive the ordinary differential equations (ODEs) that describe the evolution of concentrations of chemical species over time. While in simple cases these equations can be derived by hand using the law of mass action

[15], for large, complex reaction networks automation is required. We provide the formalisation necessary for automating the approach and conduct a feasibility study based on a prototype implementation. More generally, we hope to contribute to bridging the gap between discrete modelling techniques focusing on change of structure and continuous ones modelling quantitative changes. While focussing on chemical reactions, the technique is a general one and could be applied to other systems that can be described in a similar way.

Specifically, our approach is based on stochastic graph transformations [11] which combine rules to capture the reactive behaviour of the system with a specification of rate constants governing the speed at which the reactions occur. We have studied (and continue to do so) the problems of stochastic simulation and model checking of such specifications [13, 16]. These analysis approaches suffer from scalability issues in terms of the size of the populations considered (such as the number of starting molecules allowed in the system). Ordinary differential equations are an alternative level of abstraction less prone to this problem.

This paper is organised as follows. First, a treatment of background and related work is given, focusing on recent rule-based approaches by Faeder et. al [8] and Feret et. al [9]. This is followed by an overview of the methodology employed, including how the derivation of ODEs is enabled by well-known constructions of graph transformation theory. Then our case study of the SN_1 reaction is introduced, and the results obtained by the analysis are given, followed by a short discussion of tool support. Finally, the conclusion contains a discussion of required further work, and ways to evolve the current methodology.

2 Background and Related Work

In this section, we review a number of approaches to the derivation of ODEs from process calculi, biological modelling languages, net-based and rewriting approaches. In the first category, Cardelli [4] converts expressions in the Chemical Ground Form (CGF) process algebra (a close formalisation of chemical reaction rules) into ODEs. PEPA [14], a stochastic process algebra that incorporates rate information into communication activities, can express a reaction system as a set of interacting sequential components. These interactions can be expanded to a state representation of the reaction system, from which an activity matrix bearing a resemblance to the stoichiometric matrix can be extracted. The stoichiometric matrix describes the consumption or production of chemical species by individual reactions and is a necessary ingredient to the derivation of ODEs (see Section 3).

This matrix bears a close similarity to the incidence matrix of a Petri net, so it comes as no surprise that nets have been utilised to the same end. Each place represents a distinct chemical species (starting material, product or intermediate) in the reaction process and each transition is an elementary reaction which transforms and converts these species. Transitions are labelled with rates. In [10] it is described how a discrete Petri net can be converted into a continuous one by allowing places to have a positive real number of tokens representing the concentration of that particular chemical species in the system. Firing transitions is carried out continuously, the speed dependent on the token values at the input places. This models the fact that the speed of a reaction is dependent on the concentrations of the species required. The ODEs can be deduced from the incidence matrix for such a Petri net.

Both the process algebraic methods above and the Petri net method are limited in that the specification of reactions requires the molecules involved to be defined in their entirety. For an extremely large reaction network comprising hundreds of reactions between molecules, such a specification becomes unfeasible. This is widely known as the combinatorial explosion problem. In contrast, we follow the work of Faeder et. al [8] and Feret et. al [9] in using a rule-based approach. Here, we model reactions at the level of functional groups rather than entire molecules. Reactions are described by local context, depicting only those elements that are directly involved in the reaction [5]. In this way, a fully defined set of concrete reactions between all reacting species is reduced to a smaller set of reaction rules that can be contextualized to fit possibly numerous specific pairs of reactants.

The BioNetGen Language (BNGL) [8] is a versatile rule-based modelling language, developed with biochemical networks in mind. The language can be used to specify rules and starting reactants as terms, which can be visualised graphically. To give a continuous numerical solution for the ODEs representing the system, the BioNetGen software constructs the entire reaction network via the systematic and exhaustive application of rules to a set of seed species, and their subsequent derivatives, until no further change occurs.

In contrast to measuring the concentration of species, [9] introduces the concept of "coarse-graining", where so-called fragments are an alternative unit of dynamics. Fragments are subgraphs of species and can be envisioned as the smallest units at which the system can make distinctions on the dynamics. This reduces the ODEs necessary to describe a system to a much smaller set, as one fragment encapsulates a number of species, including their reactivity. Fragments are constructed directly from the rules according to a number of constraints. The modelling language used is Kappa, which again has an algebraic and a graphical representation.

We aim to follow these two approaches, rephrasing their constructions in terms of graph transformation theory to understand the relationship between the two fields and explore possible mutual benefits. Specifically, we attempt the derivation of ODEs via critical pair analysis of graph transformation rules. This is essentially a combination of the static analysis approach in [9] with the BNGL method in [8] which requires an enumeration of the chemical species in the system. However, where BNGL requires execution of the rules on a set of seed species (which may require numerous instances of each starting reactant), our method only requires the starting materials to be defined more generally, without considering repeated instances. The adherence to a graphical representation is particularly useful in reducing the semantic gap between the model and real chemistry [3, 18], since graphical representations potentially bear a conceptual similarity to structural formulae used in Chemistry. Atoms (or molecular species, for a higher level of abstraction) can naturally be seen as nodes in a graph, with the bonds between them represented as bi-directional edges.

We differ slightly from both the approaches in that we attempt to define a more general consideration of rate constants. In Chemistry, the reactivity of a site in a molecule (and therefore the rate constants for reactions involving that site) can be affected by additional context, possibly several positions away in the carbon chain. This additional context may not be a defining part of the functionality of that site, and therefore would not necessarily be included in the rule that captures that functionality. In both the coarse-grained and BNGL approaches, to account for this additional rate-altering context, a general rule must be expanded to include that context, so that a different rate constant can be assigned. This can be seen as an instantiation of the general rule so that it is applicable in a more specific instance. In both approaches, the instantiation must be done manually. In our approach, we automatically instantiate all general rules to apply to specific, singular molecular context so that a distinct rate constant can be applied to each of these instantiated reactions. We make the assumption that additional context always has an effect on the rate constant. Also, this instantiation may be important in systems outside the chemical and biological realms. As our aim is to avoid restricting the application domain, we incorporate this generalisation into the methodology presented here.

While the fragment approach in [9] is an innovative way of reducing the number of ODEs necessary to describe the dynamics of a system, it is more difficult to describe the dynamics of a specific chemical species since fragments are essentially patterns that encapsulate a group of species exhibiting the same behaviour. Defining dynamics in terms of classically recognisable, individual species may be more favourable to chemists or biologists that are interested in the actual population levels. A further limitation of the fragment approach is that disjoint components in the same rule cannot represent two reactive sites of the same molecule. This means that any intramolecular reactivity, although already functionally captured by another general rule, must be specified separately. This limitation arises since fragments are constructed from rules, and any such intramolecular reactivity not specified as intramolecular by a rule could potentially lead to an infinite number of fragments (see [9] for further details).

3 Stochastic graph transformation rules to ODEs

This section explains the fundamental ideas behind the methodology. We start out with a description of the standard approach to derivation of ODEs from chemical reactions and illustrate by the same example the correspondence with place-transition nets. The problem of deriving ODEs from a stochastic graph transformation system is thereby reduced to encoding it into a suitable net.

3.1 ODEs from Chemical Reactions and PT Nets

Our derivation of ODEs follows from the law of mass action. In order to state the dynamics of a chemical system in terms of ODEs, the complete set of possible reactions in the system must be known. For each reaction, we must also know how many molecules of each species are consumed and produced by that reaction. We build up what is known as a stoichiometric matrix which relates each elementary reaction to each molecular species in the system by the aggregate effect the reaction has on that species' population. Consider the following 3 elementary reactions which comprise an example reaction mechanism.

$$\begin{gathered}\mathrm{A} \xrightarrow{k_1} \mathrm{B}\\ \mathrm{B} \xrightarrow{k_2} \mathrm{A}\\ \mathrm{A}+\mathrm{B} \xrightarrow{k_3} \mathrm{C}\end{gathered}$$

The k_x values are rate constants for each of these reactions. The rate law of any elementary reaction is defined as the product of a rate constant for that reaction (e.g. k_1 for the first reaction) with the concentrations of any reactants needed in the initiation of that reaction (e.g. the concentration $[A]$ of A for the first reaction). The rate law gives us an indication of the "speed" at which that elementary reaction goes. The rate laws for the reactions above are given respectively as

$$\mathrm{k_1[A]}$$
$$\mathrm{k_2[A]}$$
$$\mathrm{k_3[A][B]}$$

To obtain an ODE for this reaction system with respect to a particular reactant we consider how each elementary reaction affects the population of that reactant. For example, A is consumed once by $\mathrm{k_1}$, produced once by $\mathrm{k_2}$, and consumed once by $\mathrm{k_3}$. To get an ODE for A, we multiply how many net molecules of A are produced by a reaction (negative values indicate net consumption) by the rate law for that reaction, and sum over all of the rate laws. For A, this gives:

$$\mathrm{d[A]/dt = k_1[A] + k_2[A] - k_3[A][B]}$$

From the procedure above, it is clear that all that is needed to derive these ODE's is an indication of what effect each reaction has on each chemical species in the system. This information is entirely contained in the stoichiometric matrix described above. The rate laws are extracted by multiplying the rate constant for each reaction by the concentrations of every molecule needed to initiate the reaction. If we do this for every elementary reaction, we can produce a rate law vector of length n, where n is the total number of elementary reactions. A multiplication of this vector and the stoichiometric matrix produces a system of ordinary differential equations:

$$d[X]/dt = S \cdot R \qquad (1)$$

where $d[X]/dt$ is the differential with respect to time, t, of a chemical species, X, in the system, S is the stoichiometrix matrix, and R is the rate law vector. Thus, the derivation of ODEs is straightforward once the stoichiometric matrix is established.

	A	**B**	**C**
$\mathbf{k_1}$	-1	1	0
$\mathbf{k_2}$	1	-1	0
$\mathbf{k_3}$	-1	-1	1

Table 1: Example stoichiometric matrix

The stoichiometric matrix for our simple example reaction mechanism is given in Table 1. This can be seen as an incidence matrix for a simple net with places A, B, C and transitions k_1, k_2, k_3, shown in Figure 1. As discussed in Section 2, a Petri net whose transitions are labelled by rate constants can be translated into a system of ODEs in much the same way as a system of reaction equations [10]. In fact, starting out from the incidence matrix of the net, the ODEs can be written out immediately following the matrix equation (1) above. It therefore suffices to encode the graph transformation system into a place transition net, which is the aim throughout the rest of this section.

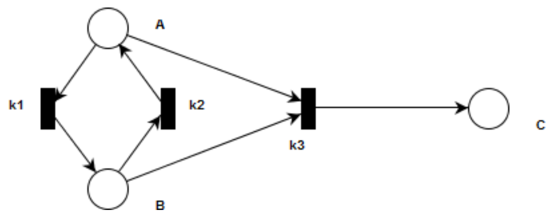


Figure 1: Place Transition Net representing reaction mechanism of Table 1

3.2 Nets From Graph Transformation Systems Modelling Chemical Reactions

A fundamental step in modeling chemical reactions is deciding how to represent molecular species. A hypergraph approach is given in [7]. Hypergraphs are explained formally in [6]. Edges in hypergraphs (known as hyperedges) can connect multiple vertices, and the connections are ordered. Atoms can be modelled as hyperedges, and we can restrict the valency of an element by specifying the minimum and maximum number of nodes each hyperedge can connect to. Bonds between atoms are then nodes connected by hyperedges. An example of a water molecule in the hyperedge approach is given in Figure 2. The ordering of connections can be used to resemble 3D configurations around a central atom. We follow a similar approach, but instead use

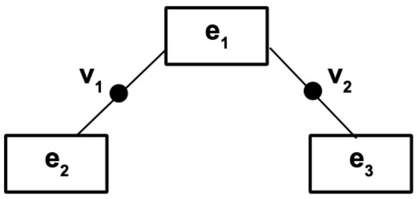


Figure 2: Hypergraph representing H_2O. e_1 is a hyperedge of type Oxygen (O), while e_2 and e_3 are of type Hydrogen (H). Nodes v_1 and v_2 represent O-H bonds. The O hyperedge is connected to 2 nodes.

bipartite graphs in which atoms (formerly hyperedges) are now represented as special types of nodes. Bonds between atoms can be represented as first an edge from the atom node to one of that atom's bonding nodes (formally nodes in the hypergraph approach). Secondly, an edge from that bonding node to another bonding node, itself connected to an atom node, represents a bond. The example for water is given in Figure 3.

A graph transformation rule, or production, is given by a span of graph morphisms and is used to specify how a graph can change. The top line of Figure 4 shows this diagrammatically. The complete diagram is known as the double pushout (DPO) construction [6]. L is the left-hand side (LHS) of the rule and specifies the preconditions for the application of a rule. K is the gluing graph and specifies which graph elements in L are unchanged by the rule (i.e. "read" only). R is the right-hand side (RHS) of the rule and specifies the postconditions in the application of the rule. Nodes and edges in L have an identity and these are mapped to nodes and edges in K and R. In order to perform a transformation on a given graph, G, there must be a match for the nodes and edges in L within G. In other words, there must be an injective morphism such that every

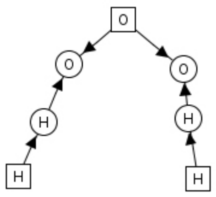


Figure 3: Graph representing H_2O. Atoms are square nodes (with their chemical symbol given by the node label), while round nodes represent bonding capacity for that element. The square O node has 2 O round nodes, hence representing that oxygen can have two bonds.

member in L is mapped to a unique member in G. There can be more than one such match, in which case one is non-deterministically chosen for the transformation. Once a match is found, the nodes and edges not in K are deleted to give D, and the new nodes and edges in R are copied into the graph, finally giving H. A graph transformation system $\mathcal{G} = (TG, R)$ consists of a type graph TG and a set of rules R.

$$\begin{tikzcd}
L \arrow[d, "m"] & K \arrow[l, "l"'] \arrow[r, "r"] \arrow[d, "k"] & R \arrow[d, "n"] \\
G & D \arrow[l, "f"] \arrow[r, "g"'] & H
\end{tikzcd}$$

Figure 4: Definition of graph transformation (DPO)

As anticipated, the approach consists in deriving from a graph transformation system the incidence matrix of a PT net which, when seen as stoichiometric matrix, gives rise to the system of ODEs required. Not all graph transformation systems can be translated into PT nets without loss of information. We first identify a class of *accountable* systems where this translation is straightforward, then show how to map a wider class of GTS into accountable ones.

Given a graph p called pattern, the *number of occurrences of p in a graph G*, denoted $\#_p(G)$, is the number of injective graph morphisms from p to G. A graph transformation system $\mathcal{G} = (TG, R)$ consisting of a type graph TG and a set of rules R is *accountable to a set of graphs P* if there exists an *accounting function* $acc : R \times P \to \mathbf{Z}$ such that for all transformations $G \overset{r}{\Longrightarrow} H$ and $p \in P$, $acc(r, p) = \#_p(H) - \#_p(G)$ is the (possibly negative) growth of the number of occurrences of p in G to H. As a consequence, the number of occurrences of each pattern in P created/destroid by each rule in R must be fixed, in particular independent of the graph and match the rule is applied to.

Given a GTS $\mathcal{G}$ accountable to P, we obtain a PT net with set of transitions R, set of places P and incidence matrix I given by $I(r, p) = acc(r, p)$. The main challenge is therefore to derive from an existing GTS one with the same rewrite relation that satisfies the accountability property. We will do so by replacing rules of the system by sets of instantiated rules, adding context to determine their effect on occurrences of patterns in P.

Rules allow us to specify general reactivity by including in the LHS of the rule only the local molecular context necessary for a reaction to occur, as stated in Section 2. While a graph transformation system describes the structural aspects of a system, adaption of this to a stochastic graph transformation system can inform us of non-functional, quantitative properties. The inherent speed at which reactions occur is one such non-functional property, which is captured in a chemical system by the rate constant. The rate constant is a direct measure of the reactivity of a certain pair of reactants via a reaction. Each GT reaction rule can be assigned a rate constant value just as we would if we were defining a reaction mechanism manually. This introduces stochasticity to the graph transformation system and indicates in basic terms how likely a transition is to occur from one state to another via a rule (see [11] for a more formal treatment of stochastic graph transformation systems).

While general reaction rules help us to concisely specify the functional behaviour of the system, they can not (generally) be assigned a rate constant and therefore can not be used in the stoichiometric matrix. Rate constants can not be assigned based on local context alone, since atoms even several positions away in a hydrocarbon chain can have some effect, however minor, on the reactivity of a site participating in a rule. There may be applications other than Chemistry where this additional context becomes important when assigning the probability of that rule being applied. Instantiation of a rule expands the rule to include the necessary context to assign a specific rate constant, and ensures each rule acts on a specific set of complete molecules only. Each general rule will therefore result in one or more instantiated rules.

Consider the construction in Figure 5. L, K and R are the span of a general reaction rule. M is a pattern graph representing a chemical species. Let us consider some possible matches in graph G for L (the LHS of the reaction rule) and M as l and m respectively. G is essentially one possible overlap, or union, of M and L. In this union, some of the graph elements of M and L may intersect.

A match for M in G simply identifies that the molecule given by M exists in G. The general reaction rule, however, by definition must force some change in graph G. As molecules in our model are constrained to always have a set number of bonds, any change must involve the initial breaking of a bond. In our graph representation, this translates to the deletion of at least one graph element i.e. node or edge. If this deleted node or edge is in the set of graph elements given by the intersection of M and L in G, application of the rule at the given matches will cause the molecule defined by the match for M in G to change. We can therefore conclude that the general reaction rule can be applied to M and consumes it. If the deleted node or edge is not in the set of graph elements given by the intersection of M and L in G, the application of the reaction rule at the specific matches given by l and m is independent of M and does not affect it. By extending the graph in the LHS, gluing graph and right-hand side (RHS) of the general rule to incorporate the molecule given by M, we can instantiate the general reaction rule to apply only to this specific molecule. The specific and singular applicability of the newly instantiated rule allows it to be assigned an elementary reaction rate constant.

In a similar fashion, we can look at the RHS of the general rule, with a match r in H, and M, with a match m in H, as per the right side of Figure 5. This time we consider which nodes and edges are newly created in R that are not in K and L. If any of these graph elements are in the intersection of M and R in H, we can conclude that the application of the general reaction rule produces the molecule in M. We would not be able to pattern match for M before applying the general reaction rule. In the same way that the general rule reducing M was instantiated, we can do the same for general rules that produce the pattern. Combining the results of the consumption and production analysis gives us a net value of how each reaction affects each species, and allows us to populate our required stoichiometric matrix.

$$\begin{tikzcd}
M \arrow[dr, "m"] & & L & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[d] \\
& G & & D \arrow[ll] \arrow[r] & H
\end{tikzcd}
\qquad
\begin{tikzcd}
L \arrow[d] & K \arrow[l] \arrow[r] \arrow[d] & R \arrow[dr, "r"'] & & M \arrow[dl, "m"] \\
G & D \arrow[l] \arrow[rr] & & H &
\end{tikzcd}$$

Figure 5: Rule instance consuming M (left) and rule instance creating M (right)

Before any instantiation can take place, we must generate the complete set of species that arise in a reaction network. We can use the same analysis as above to achieve this. Whenever a general rule is found to consume a molecule given by graph M, we can use the union of L and M to derive any new arising species. We can apply the GT general reaction rule in the forward direction to the union at the match given by the critical overlapping between the two graphs. The result can then be checked for any disjoint graphs that represent fully-formed molecular species that have not been noted as yet. By fully-formed, we mean that the minimum and maximum valency constraints for each atomic node are preserved. This ensures we do not consider invalid subgraphs that represent only partial molecules. Similarly, for sequential dependencies, examining the application of the reverse of the general rule to a union of M and R, gives us the species that produce M via this reaction rule. Any new molecules must themselves be added as new molecular pattern graphs. The critical overlappings analysis between rules and molecular pattern graphs is then repeated to see if the application of general rules to the new molecular pattern graphs produces any further new species. This procedure is iterated until no further change occurs. At this point, we can progress to the instantiation stage.

The analyses above can be implemented using critical pair analysis information. The pattern graph, M, can be seen as an identity rule. An identity rule is one in which the LHS, gluing graph and RHS of the rule are identical. It can be applied to a graph, G, if there is an injective mapping of the LHS of the rule in G. Since the application does not alter G, identity rules can be used to check for the presence of a pattern (represented by the LHS of the rule) within graphs. We can then conduct a critical pair analysis between pairs of rules to deduce possible conflicts or dependencies among them. Figure 5 depicts this for the case of conflicts (consumption of patterns). If the identity rule cannot be applied after the application of the reaction, we have a conflict. In our case, one rule is a general reaction rule, while the other is a molecular identity rule. The conflict analysis creates all possible graphs G through all possible unions of M and L. If in any of these, the intersection of the deleted nodes and edges in L (i.e. the nodes and edges in L but not in K) and the nodes and edges in M is not an empty set, we have a parallel conflict, since the application of the general rule disables the application of the identity rule. Note that we only do the check in one direction; as the identity rule does not affect the graph, we do not need to consider what would happen if the identity rule were applied first. A parallel conflict pair between a reaction rule and identity rule then informs us that that reaction rule consumes the molecule represented by the identity rule.

Similarly, the right part of Figure 5 shows the sequential dependence critical pair check. Now, we consider whether the order in which rules are applied affects the overall outcome. The sequential dependence analysis creates all possible graphs H through all possible unions of R and

M. If in any of these, the intersection of the newly created nodes and edges in *R* and the nodes and edges in *M* is not an empty set, we have sequential dependence between the reaction and identity rules. The identity rule cannot be applied before the reaction rule, enabling us to conclude that the reaction rule produces the molecule represented by the identity rule. Again we only need to perform the check in one direction.

A systematic critical pair analysis between all instantiated reaction rules and identity rules yields the stoichiometric matrix needed to derive ODEs. The methodology is better illustrated using the case study, which is described in the next section.

4 Case Study - the SN_1 reaction

In applying the methodology described in the preceding section, we start with a rule-based GTS approach and define all possible general reactivity as general reaction rules. This is followed by an instantiation procedure such that the GTS satisfies the accountability property. Although, we can then translate this GTS into a PT-net, we do this by deriving the incidence matrix for the PT-net rather than representing the full net graphically. Since the incidence matrix is a representation of the stoichiometric matrix for the reaction system, we can derive the ODEs directly from this.

SN_1 stands for unimolecular nucleophilic substitution. It involves the replacement of a good leaving group on a carbon atom (e.g. a chlorine atom) with a group that is less able to support negative charge (e.g. a hydroxyl group, OH). Unimolecular refers to the fact that the rate determining step involves only one molecule, the halocarbon. The reaction occurs readily once the Cl- group leaves the molecule. The reaction was chosen for its simplicity as a fundamental step in testing and demonstrating the basics of our methodology.

4.1 The Graph Transformation System

Figure 6 shows the type graph used for this simple reaction. The nodes in the graph are labelled with a ”*” indicating that this is a type specification. The type graph captures the necessary conditions to restrict the bonding of atoms, their valencies (the total number of bonds a particular atom is allowed to have) and any other idiosyncracies of molecular chemistry. In Figure 6, atoms are represented as square nodes, each distinct species of reactive interest having its own node type. The round nodes are atom-specific bonding nodes. A bond between atoms is represented by an edge (arrows with filled arrowheads) between two of these bonding nodes. Each bonding node is connected to only one atom node. Each atom node has an exact number of bond nodes it can connect to, which is established using cardinalities between the two types of nodes (the numbers alongside the edges). For example, an O atom can only have two bonds. In this way, the valency of each species is established.

The type graph also exhibits inheritance (arrows with unfilled arrow heads). All bonding nodes are subtypes of the generic *BondNode* type. This enables us to more easily specify the allowed edges between each pair of bonding node types. Inheritance is only used in the type graph and constraints. The type graph and constraints are mechanisms which check the validity of a start graph or a produced graph before committing the applied transformation, in much the same way that database constraints are evaluated in a transaction. The use of supertypes in rules may cause

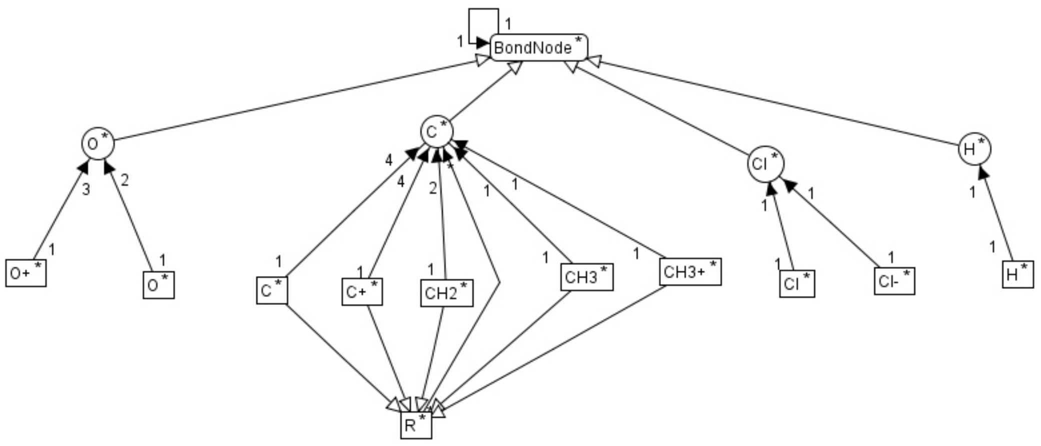


Figure 6: Type graph for SN_1 reaction, produced in AGG

unwanted overlappings during critical pair analysis, since every supertype node or edge can be substituted by its subtypes.

Note that C and C^+, for example, have the same bonding node type (C) associated with them. If a reaction rule changes the charge of an atom, the atom node must first be deleted, and then a new charged one introduced in its place, so no mapping can exist between the neutral and charged atom on the LHS and RHS of the rule respectively. By maintaining that both node types can be connected to the same bonding node, the mapping for the bonding node of the changing atom can remain constant. This allows us to indirectly track the identity of atoms even if their charge changes throughout a reaction.

While the type graph contains C, C^+, and H node types, our model abstracts the methyl group to a single CH_3 node type, with its charged version to $CH_3{}^+$. Since the critical pair analysis constructs an overlap between the graphs on the LHS and RHS of rules, a lower number of nodes in these graphs results in a lower number of overall overlappings, and a smaller number of nodes and edges in each overlapping. This in turn results in a quicker and more efficient analysis. A single CH_3 node, if expressed in terms of C atom nodes, H atom nodes, C bonding nodes, H bonding nodes and the edges between them, would consitute a total of 10 nodes. This abstraction of atomic structure into groups is carried out in Chemistry also, in the drawing of structural formulae, and is valid if and only if no reaction rules can be applied to the full atomic representation of the abstracted group. This can be statically checked using the analysis technique described in the previous section i.e. creating a molecular identity rule from the full atomic representation and analysing whether there are any critical overlappings with any of the reaction rules. If there are, any abstraction of this group which affects the presence of the critical pairs is invalid.

In addition to the type graph, constraints may be used to specify conditions on graphs (and rules) within the system. As edges are directional, a convention is needed to ensure that edges between a specified pair of atoms are always facing the same direction. To this end, it was decided that edges would go from the less electronegative atom to the more electronegative one e.g. for a bond between C and O, the edge would always go from the C bonding node to the O bonding node. A further constraint was necessary to prevent multiple incoming edges to the carbon/hydrocarbon and oxygen round bond nodes. So that all of the carbon/hydrocarbon bonding possibilities need not be enumerated for the constraint, the type graph utilises inheritance to create an R supertype for all carbon/hydrocarbon node types. The constraint then only requires a single combination, between 2 R nodes. Finally, a constraint was needed to prevent CH_3^+ from being able to bond to anything. Figure 7 shows all five of these constraints, with “not” representing the fact that graphs containing these structures should be considered invalid.

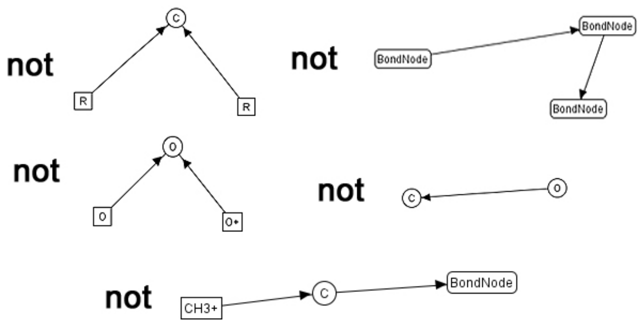


Figure 7: Additional graph constraints

Note that we have used negative constraints in our model to restrict the behaviour of bond nodes. If the overlap of M and L in the consumption evaluation described in the previous section (or M and R in the production evaluation) creates a graph G or H respectively that violates one of these negative constraints, we can deduce that the overlap is invalid and can be discarded, since the overlap is a minimal union of the two graphs. Any violation here is still present if this union were embedded into a larger graph. If the negative constraint is not violated, the overlap is maintained as valid. A positive constraint, however, specifies the required presence of a certain configuration of nodes and edges somewhere in the graph. If violated in the minimal union, it may still be fulfilled once the union is embedded into a larger graph. Positive constraints are therefore avoided in our graph representation (or if they were present, would not impact on the analysis).

An example of a graph typed over the type graph in Figure 6 is given in Figure 8. These are the starting materials for the SN_1 reaction, namely water and chloromethane.

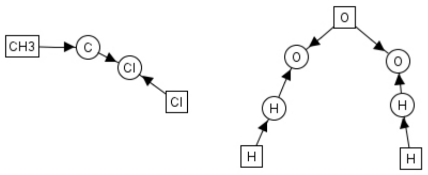


Figure 8: Graph depicting starting materials for SN_1 reaction, produced in AGG. Left: CH_3^+, Right: H_2O

The first step in our methodology is to capture all possible general rules for a reaction system. This involves the definition of reactions at the level of functional groups only i.e. local molecular context. Figure 9 depicts the general rules governing the reactivity for this small reaction mechanism. The top rule dictates that if a chloromethane molecule is found in a graph, it can be broken to form charged carbocation (CH_3^+) and Cl^- ions. The middle rule captures the attack of the positively charged carbocation by a lone pair of electrons on the O of an OH group. In the lower rule, the positively charged O^+ accepts electrons from the O^+-H bond in the presence of the Cl^- created in the the very first step of the reaction. This yields hydrochloric acid (HCl), and an alcohol product, containing an OH group. The reverse of each of these three rules were also added as potential reactions (by simply reversing the LHS and RHS of the rule). These reverse reactions may be unlikely but this will eventually be designated by a negligible rate constant for that particular reaction. For generality and a complete specification of possible activity, it is important to include them.

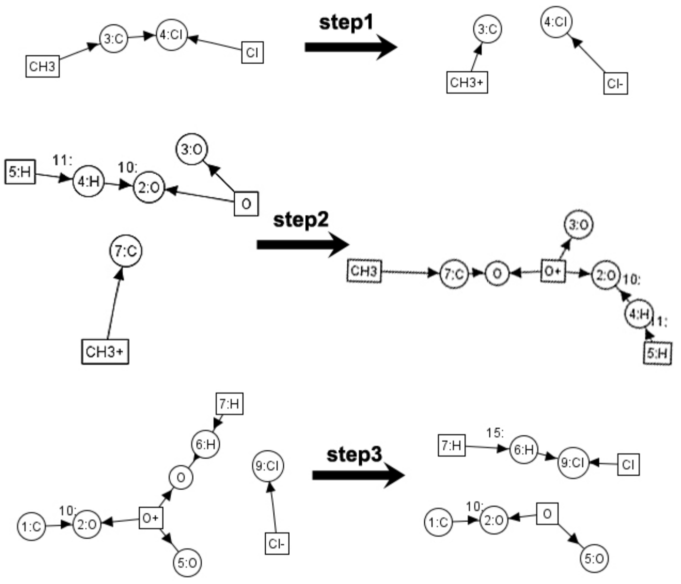


Figure 9: Steps 1 (top), 2 (middle) and 3 (bottom) of SN_1 reaction, general reaction rules

We formulate the two known starting materials in Figure 8 as two molecular identity rules, where the LHS and RHS are the same and contain only the graph of a particular molecule. To discover unknown intermediates and products in the reaction, a critical pair analysis is conducted between each general reaction rule and each molecular identity rule. This is an iterative process as described in Section 3. The results of the first iteration are given in Table 2. Each entry signifies how many of the overlappings were critical for each pair, remembering that critical pair analysis checks all possible unions of L and M for parallel conflict analysis, and R and M for sequential dependence analysis.

There are 2 critical overlappings wherever there is a conflict with H_2O. Due to the nature of our molecular representation, some of the critical overlappings returned may be spurious overlappings. One critical overlapping of the Step2-H_2O rule pair is given in Figure 10. The shaded grey area covers all of the critical nodes and edges in this overlapping. The other critical overlapping is structurally identical in that the types of the edges and nodes covered are the same. The two

	CH_3Cl		H_2O	
	PC	*SD*	*PC*	*SD*
Step1	1	0	0	0
Step-1	0	1	0	0
Step2	0	0	2	0
Step-2	0	0	0	2
Step3	0	0	0	0
Step-3	0	0	0	0

Table 2: Results of first pass critical analysis - parallel conflicts (PC) and sequential dependencies (SD)

overlappings arise, however, due to the symmetry around the critical O atom node (with mapping id 1). In real chemistry, however, these configurations have no significance to the selection of a reaction or to the outcome of the reaction. We can treat them as equivalent, formally captured by the obvious notion of isomorphism between the corresponding transformations. Therefore this entry in the table can be reduced to 1. To evaluate two overlappings for chemical equivalence, the results of applying the reaction rule to the overlappings at their respective critical matches can be compared. If the results are isomorphic, the two overlappings can be considered to be chemically equivalent, since they react in the same way to give the same products up to isomorphism. For the SN_1 reaction studied, the molecules involved were small, possessing only one reactive site per instantiated reaction, therefore in all cases these overlappings were reduced to 1. If there were two asymmetric reactive sites in a molecule for an elementary reaction step, there would be 2 chemically distinct overlappings, since application of the reaction rule would lead to alternative products. In this case, we would have discovered two possible instantiations of the same reaction rule.

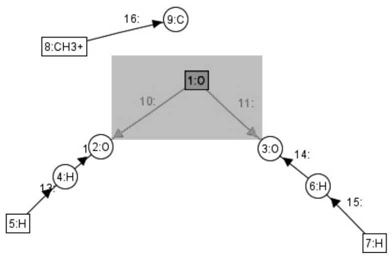


Figure 10: One critical overlapping between general reaction rule Step2, and H_2O (critical graph elements are contained within the shaded area)

New intermediates in the system can be discovered by the systematic application of the general rule to the molecules it has an impact on. Where a parallel conflict is apparent, the application of the general rule to that molecule at the match given by the critical overlapping gives us a chemical species that is produced by this reaction. Similarly, for sequential dependencies, the application of the reverse of the general rule to the molecule, gives us the species that undergoes this general reaction to give the molecule in the pattern graph in question. For example, new molecules arising from the critical overlappings in Table 2 are Cl^- and $CH_3{}^+$. We add these new molecules as molecular identity rules and repeat the critical pair analysis as these new intermediates may produce further intermediates under the rule-defined behaviour of the system. After five iterations of the critical pair analysis, the total set of chemical species for this case study can be derived as molecular identity rules. The graphs representing these intermediates are given in Figure 11, along with their chemical formulae. The result of the last iteration, of the critical pair analysis is given in Table 3, with a reduction to only chemically distinct numbers of overlappings.

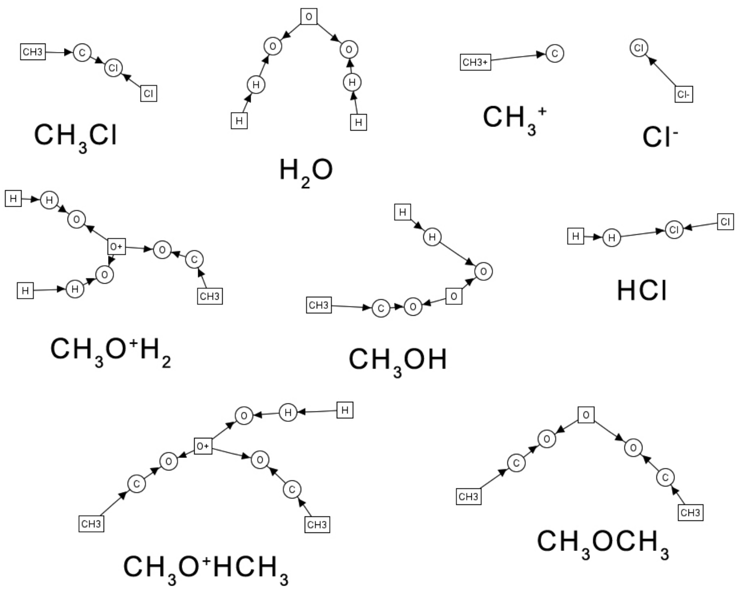


Figure 11: Complete set of chemical species for SN_1 reaction, derived through iterations of critical pair analysis

	CH_3Cl		H_2O		$CH_3{}^+$		Cl^-		$CH_3O^+H_2$		HCl		CH_3OH		$CH_3O^+HCH_3$		CH_3OCH_3	
	PC	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*	*PC*	*SD*
Step1	1	0	0	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0
Step-1	0	1	0	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0
Step2	0	0	1	0	1	0	0	0	0	1	0	0	1	1	0	1	0	0
Step-2	0	0	0	1	0	6	0	0	1	0	0	0	0	1	1	0	0	0
Step3	0	0	0	0	0	0	1	0	1	0	0	1	0	1	1	0	0	1
Step-3	0	0	0	0	0	0	0	1	0	1	1	0	1	0	0	1	1	0

Table 3: Results of iterative critical pair analysis - parallel conflicts (PC) and sequential dependencies (SD)

4.2 From Graph Transformation System to Place Transition net

To achieve the accountability condition necessary to translate our GTS to a PT net, we need to instantiate the general rules using the results of Table 3.

An interesting general reaction in our case study is that of *Step2*. There are entries in the parallel conflicts part of the table for H_2O, $CH_3{}^+$ and CH^3OH. Closer manual examination reveals that this encapsulates two reactions; the reaction of H_2O with $CH_3{}^+$, and a secondary reaction of the methanol product, CH^3OH, with $CH_3{}^+$. This general rule can therefore be instantiated twice, as per Figure 12. This analysis can be carried out methodically for an even larger number of instantiation candidates than just the three we have here. Essentially, the overlap of M_1 with L in Figure 5 forms a graph into which a second overlap with another molecular pattern graph, M_2, can take place directly. The order in which the overlaps are constructed is inconsequential. This is possible since the type graph cardinalities ensure that molecules are always defined in their entirety. As such, there is no possible intersection of graph elements between the two separate molecules in the combined overlap, unless the two molecules are identical (in which case the overlap is meaningless and discarded).

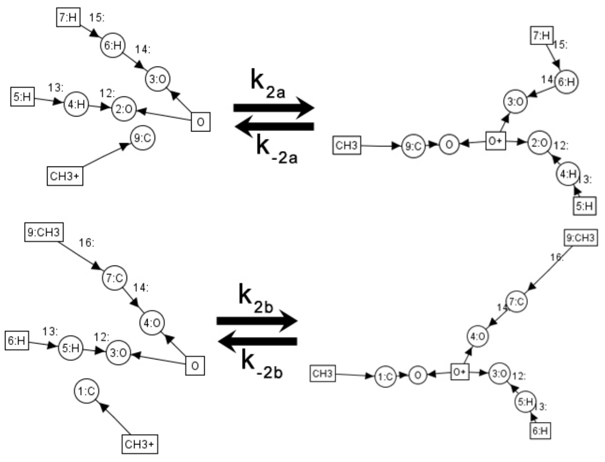


Figure 12: Instantiation of step2 for reaction with water (top) and methanol (bottom)

To identify whether a pair of molecules are indeed valid instantiation candidates, we can examine the construction in Figure 13. M_1 forms a union with L to give G_1, while M_2 forms a union with L to give G_2. G_1 and G_2 can be combined to give a final overlap, which is used to instantiate the rule fully. Since l_1 and l_2 identify the two different reactivity sites needed for the reaction, for a valid combination of G_1 and G_2, their intersection should be empty. Formally, this condition is represented as:

$$l_1^{-1}(m_1(M_1)) \cap l_2^{-1}(m_2(M_2)) = \phi$$

For each unique, valid set of instantiation species, we instantiate the general rule to include the full molecules in its context. All instantiated reaction rules are then named with rate constants, specific to the reaction they refer to. For this case study, step3 can also be instantiated twice to

$$\begin{tikzcd}
M_1 \arrow[dr, "m_1"] & & L \arrow[dl, "l_1"'] \arrow[dr, "l_2"] & & M_2 \arrow[dl, "m_2"] \\
& G_1 \arrow[dr] & & G_2 \arrow[dl] & \\
& & G & &
\end{tikzcd}$$

Figure 13: Combination of critical overlappings for instantiation of general rules

mark the deprotonation of $CH_3O^+H_2$, and of $CH_3O^+HCH_3$. Finally then, k_1 refers to the first step of the reaction (the Cl^- leaving the chloromethane molecule), k_{2a} is the attack of water on the resulting carbocation, k_{2b} the attack of methanol on the same carbocation, k_{3a} the deprotonation of the result of the water attack, and k_{3b} the deprotonation of the result of the methanol attack. The reverse reactions are represented by k_{-x} for each of these forward reactions.

With the complete set of instantiated rules and their impact on each molecular pattern graph, we can build the incidence/stoichiometric matrix for the PT net for this reaction (see Table 4) by subtracting the total parallel conflict critical overlappings (how many molecules are consumed) from the total sequential dependence critical overlappings (how many molecules are produced) for a particular instantiated reaction molecular pattern graph pair. This then gives us the impact each elementary reaction has on each chemical species. To reiterate, the reactions that the entries in the stoichiometric matrix refer to are given below:

$$\mathrm{k_1 : CH_3Cl \rightleftharpoons CH_3{}^+ + Cl^- : k_{-1}}$$
$$\mathrm{k}_{2a} : \mathrm{CH_3{}^+ + H_2O \rightleftharpoons CH_3O^+H_2 : k}_{-2a}$$
$$\mathrm{k}_{2b} : \mathrm{CH_3{}^+ + CH_3OH \rightleftharpoons CH_3O^+HCH_3 : k}_{-2b}$$
$$\mathrm{k}_{3a} : \mathrm{CH_3O^+H_2 + Cl^- \rightleftharpoons CH_3OH + HCl : k}_{-3a}$$
$$\mathrm{k}_{3b} : \mathrm{CH_3O^+HCH_3 + Cl^- \rightleftharpoons CH_3OCH_3 + HCl : k}_{-3b}$$

	CH_3Cl	H_2O	$CH_3{}^+$	HCl	Cl^-	$CH_3O^+H_2$	CH_3OH	$CH_3O^+HCH_3$	CH_3OCH_3
k_1	-1	0	1	0	1	0	0	0	0
k_{-1}	1	0	-1	0	-1	0	0	0	0
k_{2a}	0	-1	-1	0	0	1	0	0	0
k_{-2a}	0	1	1	0	0	-1	0	0	0
k_{2b}	0	0	-1	0	0	0	-1	1	0
k_{-2b}	0	0	1	0	0	0	1	-1	0
k_{3a}	0	0	0	1	-1	-1	1	0	0
k_{-3a}	0	0	0	-1	1	1	-1	0	0
k_{3b}	0	0	0	1	-1	0	0	-1	1
k_{-3b}	0	0	0	-1	1	0	0	1	-1

Table 4: Stoichiometric matrix resulting from final pass critical pair analysis and chemical equivalence analysis

4.3 ODEs from the PT net

ODE extraction from the results in Figure 4, following the procedure outlined at the beginning of Section 3 led to the ODEs shown in Figure 14. These results agree with ODEs derived manually for this simple system using the law of mass action.

```
d[CH3+]/dt =  -k-1[CH3+][Cl-] +k-2a[CH3O+H2] +k-2b[CH3O+HCH3]
              +k1[CH3Cl] -k2a[CH3+][H2O] -k2b[CH3+][CH3OH]

d[CH3Cl]/dt =  +k-1[CH3+][Cl-] -k1[CH3Cl]

d[CH3O+H2]/dt =  -k-2a[CH3O+H2] +k-3a[CH3OH][HCl] +k2a[CH3+][H2O]
                 -k3a[CH3O+H2][Cl-]

d[CH3O+HCH3]/dt =  -k-2b[CH3O+HCH3] +k-3b[CH3OCH3][HCl]
                   +k2b[CH3+][CH3OH] -k3b[CH3O+HCH3][Cl-]

d[CH3OCH3]/dt =  -k-3b[CH3OCH3][HCl] +k3b[CH3O+HCH3][Cl-]

d[CH3OH]/dt =  +k-2b[CH3O+HCH3] -k-3a[CH3OH][HCl] -k2b[CH3+][CH3OH]
               +k3a[CH3O+H2][Cl-]

d[Cl-]/dt =  -k-1[CH3+][Cl-] +k-3a[CH3OH][HCl] +k-3b[CH3OCH3][HCl]
             +k1[CH3Cl] -k3a[CH3O+H2][Cl-] -k3b[CH3O+HCH3][Cl-]

d[H2O]/dt =  +k-2a[CH3O+H2] -k2a[CH3+][H2O]

d[HCl]/dt =  -k-3a[CH3OH][HCl] -k-3b[CH3OCH3][HCl] +k3a[CH3O+H2][Cl-]
             +k3b[CH3O+HCH3][Cl-]
```

Figure 14: Result of ODE extraction for SN_1 reaction

4.4 Tool support

AGG, which can be found at [19] is a widely used graph transformation tool that was crucial to our implementation. It allows the construction of a graph grammar, namely type graph, rewriting rules, start graph and constraints. AGG also has its own critical pair analysis (CPA) engine, accessible through GUI and command line. The GUI gives a summary table similar to Table 3 and a graph for each critical overlap, specifying which graph elements are critical and their identities in each of the two rules.

AGG is open source and is implemented in Java with its own API. Adaptation of the standard implementation allowed us to schedule only relevant pairs for critical pair computation. Standard analysis would analyse every rule with every other rule. Since we only need to examine critical overlappings between reaction rules and identity rules this was a first step towards a more efficient implementation. Methods and classes from the API were also used for the Java component that performed the chemical equivalence reduction described in Section 4. CPA saves results in an XML file that can be reloaded using API methods. The ODE extraction module was also implemented with the aid of these methods. Finally, a script was created which piped the results of the CPA through the chemical equivalence program and ODE extraction program in turn. Currently, the overall program produces the ODEs as a text file, but the output could easily be changed to LaTex syntax, or a format recognized by a math solver for example. Figure 15 shows the basic architecture of the tool chain used.

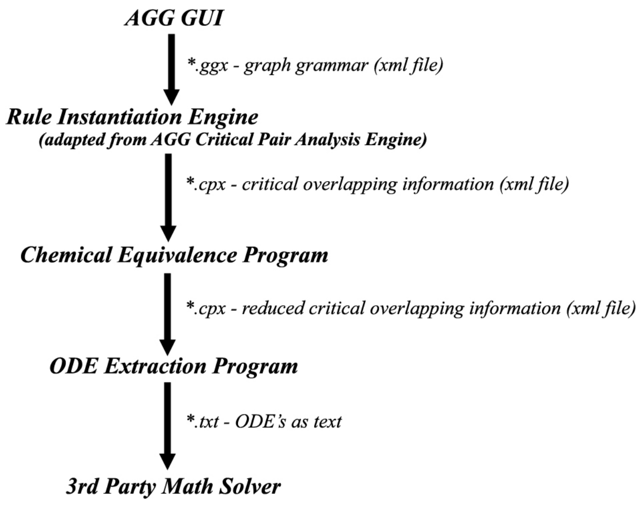

AGG GUI
*.ggx - graph grammar (xml file)
Rule Instantiation Engine
(adapted from AGG Critical Pair Analysis Engine)
*.cpx - critical overlapping information (xml file)
Chemical Equivalence Program
*.cpx - reduced critical overlapping information (xml file)
ODE Extraction Program
*.txt - ODE's as text
3rd Party Math Solver

Figure 15: Basic tool chain architecture (arrow labels show input and output file extensions)

It is worth noting that the abstraction of the CH_3 group to a single node, described in Section 4, was prompted by implementation issues for the fully detailed atomic representation. It is obvious that the critical pair analysis requires more resources the larger the rules become, since the overlap that must be constructed is larger (meaning more checks to be made for critical graph elements). Also, it is likely there will be more possible overlappings overall. Every CH_3 group represented as C, H atom nodes and C, H bonding nodes, with edges between them introduces 9 extra nodes and 9 extra edges to the graph. This early implementation, after several hours, eventually ran out of memory during CPA (assigning 1.5GB to the Java Virtual Machine), with the largest number of overlappings to check for any one pair of rules exceeding 35,000. There were also far more spurious, chemically equivalent overlappings for every critical pair, due to the extra symmetry and permutability introduced by the additional nodes and edges. In the abstracted case, this number was reduced to just over 11,000 for any pair and the instantiation round of the critical pair analysis along with critical overlapping structural equivalence testing and ODE extraction took less than 8 minutes (assigning 1.0GB to the Java Virtual Machine). While a more general model with a lower level of detail would have been an interesting representation to consider, it was equally important to use valid abstraction techniques to ensure a workable implementation.

5 Conclusion

This paper has presented a graph transformation encoding of existing rule-based approaches to defining molecular reaction mechanisms, that remains comparative to the visual approach adopted by Chemists themselves. We have shown how this approach can be used to develop a system of general reactivity, and determine fully the intermediates and products possible under this reactivity. The GTS can then be translated to a PT-net by ensuring that it satsifies the accountability condition. Through the use of static critical pair analysis, we can instantiate the GTS to represent the complete set of elementary reactions in the system. We have demonstrated how the set of ODEs that describe the overall reaction can be extracted from the incidence matrix of such a PT-net. This was demonstrated by the application of this methodology to the simple SN_1 reaction using relevant tool support. The results of our study match the ODEs we would obtain through manual derivation.

The methodology and implementation are based on existing theory of graph transformation, predominantly that of constraints, critical pairs, and isomorphism of transformations, and their compatibility with the construction of pullbacks and pushouts. However, a full formalisation of the approach and a formal statement of its assumptions and limitations is still outstanding. As it is described in this paper, the approach can be fully automated, but for the assignment of reaction rates, if

- molecules are represented by disjoint subgraphs of fixed structure, described by patterns, M
- the graph model (type graph and negative constraints) is sound and complete with respect to the molecules represented, that is, each molecule can be represented and every graph represents a legal collection of molecules

Apart from formalising this statement, future directions for research include the study of unbounded systems such as polymerisation and free radical reactions, in which elementary reactions can potentially go on forever. Currently, our methods require much manual observation and human interaction. For such reactions, it becomes imperative to be able to instantiate rules and generate molecular identity rules from newly discovered intermediates in an automated way. This is an important implementation challenge.

Other areas of research include a desire to carry out ODE extraction alongside stochastic model checking and simulation, to determine how the results compare and complement each other. Far more ambitiously, there is planned research into the automatic derivation of rate constant data based on molecular orbital composition. If this is achieved, a fully self-contained kinetic model of reactions is possible, in which ODEs and numerical rate constants can be deduced to give quantitative predictions on the progress of a reaction without the need for external data.

Bibliography

[1] Bapodra, M., (2009): *Chemical Reaction Rate Analysis Using Graph Transformations*. *BSc Computer Science with Management Project, Final Report* Available at http://www.cs.le.ac.uk/people/mb294/docs/BScFinalReport.pdf.

[2] Berry, G., Boudol, G., (1989): *The Chemical Abstract Machine*. *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages* pp. 81–94 Available at http://portal.acm.org/citation.cfm?id=96717.

[3] Benko, G., Flamm, C., Stadler, P.F., (2003): *A Graph-Based Toy Model of Chemistry*. *Journal of Chemical Information and Computer Sciences* 43, pp. 1085–1093. Available at http://pubs.acs.org/doi/abs/10.1021/ci0200570.

[4] Cardelli, L., (2008): *From Processes to ODEs by Chemistry*. *IFIP International Federation for Information Processing* 273, pp. 261–281. Available at http://www.springerlink.com/content/8g12ph2857372xm2/.

[5] Danos, V., Feret, J., Fontana, W., Harmer, R., Krivine, J., (2007): *Rule-based modelling of cellular signalling*. *18th International Conference on Concurrency Theory, Lisbon, Portugal, September, 2007, Lecture Notes in Computer Science* 4703, pp. 17-41. Available at http://www.pps.jussieu.fr/~danos/.

[6] Ehrig, H., Ehrig, K., Prange, U., Taentzer, G., (2006): *Fundamentals of Algebraic Graph Transformation*. *EATCS Monographs, Springer*

[7] Ehrig, K., Heckel, R., Lajios, G., (2006): *Molecular Analysis of Metabolic Pathways with Graph Transformation*. *International Conference on Graph Transformations 2006, Lecture Notes in Computer Science* 4178, pp. 107–121. Available at http://www.springerlink.com/content/6r637jx517320v02/.

[8] Faeder, J.R., Blinov, M.L., Hlavacek, W.S., (2009): *Rule-Based modelling of Biochemical Systems with BioNetGen*. *Methods in Molecular Biology, Systems Biology* 500, pp. 113–167. Available at http://www.springerlink.com/content/nn1v343vw2kmj00w/.

[9] Feret, J., Danos, V., Krivine, J., Harmer, R., Fontana, W., (2009): *Internal coarse-graining of molecular systems*. *Proceedings of the National Academy of Sciences of the United States of America* 106(16), pp. 6453–6458. Available at http://www.pnas.org/content/106/16/6453.

[10] Gilbert, D., Heiner, M., (2006): *From Petri Nets to Differential Equations - An Integrative Approach for Biochemical Network Analysis*. *ICATPN 2006, Lecture Notes in Computer Science* 4024, pp. 181–200. Available at http://www.springerlink.com/content/l85l6147038t012j/.

[11] Heckel, R., Lajios, G., Menge, S., (2004): *Stochastic Graph Transformation Systems*. *International Colloquium on Theoretical Aspects of Computing 2005, Lecture Notes in Computer Science* 3256, pp. 210–225. Available at http://iospress.metapress.com/content/c7ha18g96nbm7g2e/.

[12] Heckel, R., (2006): *Graph Transformation in a Nutshell*. *Electronic Notes in Theoretical Computer Science* 148, pp. 187–198.

[13] Heckel, R., (2005): *Stochastic Analysis of Graph Transformation Systems: A Case Study in P2P Networks*. *International Colloquium on Theoretical Aspects of Computing 2005, Lecture Notes in Computer Science* 3722, pp. 53–69. Available at http://www.springerlink.com/content/u54h1w273875u67r/.

[14] Hillston, J., (2005): *Fluid Flow Approximation of PEPA models*. *Proceedings of the Second International Conference on the Quantitative Evaluation of Systems, IEEE Computer Society Press* pp. 33–43. Available at http://www.dcs.ed.ac.uk/pepa/features/.

[15] Keeler, J., Wothers, P., (2008): *Chemical Structure and Reactivity, An Integrated Approach*, Oxford: Oxford University Press.

[16] Khan, A., Torrini, P., Heckel, R., (2008): *Model-based Simulation of VoIP Network Reconfigurations using Graph Transformation Systems*, *ECEASST* 16, Available at http://eceasst.cs.tu-berlin.de/index.php/eceasst/article/view/251.

[17] Palacz, W., (2008): *Practical Aspects of Graph Transformation Meta-Rules*. *Intelligent Computing in Engineering* pp. 70–77. Available at http://www.eg-ice.org/Workshops/ICE08/papers%20pdf/P002V03.pdf.

[18] Rossello, F., Valiente, G., (2005): *Graph Transformation in Molecular Biology*. *Formal Methods (Ehrig Festschrift), Lecture Notes in Computer Science* 3393, pp. 116–133. Available at http://iospress.metapress.com/content/c7ha18g96nbm7g2e/.

[19] http://tfs.cs.tu-berlin.de/agg/: *The AGG Homepage*. *Last Visited October 2008.*

Reaction Systems - a Formal Framework for Biochemical Reactions

Grzegorz Rozenberg

Leiden Center for Natural Computing
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands
rozenber@liacs.nl

Abstract

The functioning of a living cell consists of a huge number of individual reactions that interact with each other. These reactions are regulated, and the two main regulation mechanisms are facilitation/acceleration and inhibition/retardation. The interaction between individual biochemical reactions takes place through their influence on each other, and this influence happens through the two mechanisms mentioned above.

In our lecture we present a formal framework for the investigation of biochemical reactions - it is based on reaction systems. We motivate them by explicitly stating a number of assumptions/axioms that (we believe) hold for a great number of biochemical reactions - we point out that these assumptions are very different from the ones underlying traditional models of computation, such as, e.g., Petri nets. We illustrate the basic notions by both biology and computer science oriented examples. We demonstrate some basic properties of reaction systems, and demonstrate how to capture and analyze in our framework some biochemistry related notions.

The lecture is of a tutorial character and self-contained. In particular, no knowledge of biochemistry is required.

Model Transformations to Mitigate the Semantic Gap in Embedded Systems Verification

Björn Bartels, Sabine Glesner, Thomas Göthel

Software Engineering for Embedded Systems Group
www.pes.tu-berlin.de
Berlin Institute of Technology (TU Berlin)

Abstract: In this paper, we present results from the VATES project. It addresses the problem of verifying embedded software by employing a novel combination of methods that are well-established on the level of declarative models, in particular process-algebraic specifications, as well as of methods that work especially well on the level of executable code. In the VATES approach, we consider code given in an intermediate compiler representation. From this code, we (automatically) extract a model in the form of a process-algebraic system description formulated in CSP. For this low-level CSP description, we can prove that it refines a high-level CSP specification which was previously developed. To relate the LLVM code with the low-level CSP model we designed an operational semantics of LLVM. In ongoing work we investigate the extraction algorithm with respect to semantics preservation. Thereby, we are finally able to prove that given LLVM code formally conforms to its high-level CSP-based specification. In this paper we show that this approach has the potential to seamlessly integrate modeling, implementation, transformation and verification stages of embedded system development.

Keywords: (Timed) CSP, LLVM, Model Extraction, Theorem Proving

1 Introduction

Embedded systems are often employed in safety-critical areas. Their correctness is therefore extremely important in order not to endanger human lives or risk high financial losses. However, the correctness of these systems is difficult to ensure. A particular challenge is that they are highly concurrent and that non-functional properties such as the satisfaction of real-time constraints play an important role. Although there exist well-established techniques to verify abstract specifications of such systems, the verification of their actual implementations, e.g. in C++, is still an open problem.

The VATES[1] project investigates exactly these questions. It starts from the hypothesis that software in embedded systems can be characterized by certain structures (distinguished e.g. by the processes and their pattern of communication) that characterize its mode of operation and that need to be retained when transformed into executable code. We investigate these structures by taking the BOSS [MBK06] operating system as an example. BOSS is a relatively small oper-

[1] VATES=Verification and Transformation of Embedded Systems, funded by the German Research Foundation (DFG).

Figure 1: The Satellite BIRD

ating system that has been developed at the Fraunhofer Institut FIRST, Berlin. It is operational since several years within the satellite BIRD[2] (see Figure 1) of the DLR[3] designed for early fire detection. It needs to cope with high performance requirements while only featuring small resources. Such small satellites are very interesting as they have the advantage to be relatively cheap and nevertheless be very powerful at the same time. Since BOSS has been designed with the goal of verifying it in mind, it is an ideal case study for the VATES project.

We propose a novel approach that makes well-established formal verification techniques for declarative process-algebraic specifications applicable to low-level software programs. To this end, we designed an algorithm that extracts a low-level CSP model from the LLVM compiler intermediate representation [KH09]. On this basis it can be shown that a given low-level model refines a high-level model which is also given in a CSP-based formalism. The high-level specification may be investigated using our formalization of Timed CSP in the Isabelle/HOL theorem prover. It comprises a formalization of its operational semantics, several variants of bisimulations and an investigation of coalgebraic invariants which can be used to state certain liveness properties. Currently, we are using this formalization in the context of so-called parameterized systems to verify infinite-state systems. The semantic gap between the intermediate code and the low-level CSP model is closed by an investigation of the extraction algorithm. Therefore, we defined a formal operational semantics of the the Low Level Virtual Machine (LLVM) intermediate language [LA04]. The semantics is especially well-suited for the verification of embedded systems because it includes a memory model and a notion of non-determinism. Furthermore, we establish a bisimulation relation between LLVM and CSP models. With that, we can prove that a given LLVM program is a correct implementation of a given CSP model. We plan to verify that our extraction algorithm preserves the semantics of LLVM using this bisimulation.

This paper gives an overview of our results in the VATES project so far. Its remainder is structured as follows. In Section 2 we briefly introduce Timed CSP, LLVM and the theorem prover Isabelle/HOL. These constitute the main formalisms and tools that we use in our work. In Section 3 we present some of the main results of the VATES project so far: a formalization of Timed CSP in Isabelle/HOL, the idea of the extraction algorithm giving a low-level CSP model from a given LLVM program and an operational semantics of LLVM in the Isabelle/HOL theorem prover. Related work is discussed in Section 4. Finally, Section 5 concludes this paper.

[2] Bispectral Infra-Red Detection

[3] Deutsches Zentrum für Luft- und Raumfahrt (German Aerospace Center)

$$P := STOP \mid SKIP \mid a \rightarrow P \mid a : A \rightarrow P_a \mid P;P \mid P \Box P$$
$$\mid P \sqcap P \mid P \underset{A}{\|} P \mid P \setminus A \mid P \triangle P \mid P \overset{d}{\triangleright} P \mid P \triangle_d P \mid X$$

Figure 2: Syntax of Timed CSP

2 Used Formalisms and Tools

2.1 TimedCSP

The development chain, that we consider in the VATES project, starts with a specification formulated in the real-time process calculus Timed CSP. It is an extension of Hoare's CSP (Communicating Sequential Processes) [Hoa85] with timed process terms as well as timed semantics. Besides the specification and verification of reactive and concurrent systems, this also allows for the verification of timeliness. In the following, we present some of the aspects of (Timed) CSP that are most important for understanding this paper. We refer to [Sch99] for a comprehensive introduction to (Timed) CSP.

The syntax of Timed CSP is given in Figure 2. Timed CSP shares most of the operators with (untimed) CSP: STOP is a process which cannot do anything, SKIP cannot do anything except terminating indicated by the communication of the special event $\surd$, $a \rightarrow P$ can first communicate a and then behave like process P. More convenient process operators are e.g. $\Box$, $\|$ and $\setminus$ denoting *Choice*, *Parallel Composition* and *Hiding (of communication channels)*.

Timed CSP extends the CSP calculus with the timed primitives $P \overset{d}{\triangleright} Q$ (*Timeout*) and $P \triangle_d Q$ (*Timed Interrupt*). Intuitively, the meaning of a *Timeout* is that the process P can be triggered by some (external) event within d time units. If this happens, the *Timeout* is resolved in favor of P. If the time expires without P being triggered, process Q handles this situation, i.e., the *Timeout* is resolved in favor of Q. The *Timed Interrupt* construction has a similar meaning. Here, P can (successfully) *terminate* within d time units, otherwise Q is started.

There exist two main types of semantics which are typically defined in the context of CSP: The denotational (Timed) Failures semantics and the operational semantics which interprets (Timed) CSP as labeled transition system.

For CSP there exist well-established fully-automatic verification tools such as FDR [GRA05] and ProB [LF08]. FDR is well-suited for refinement checking of specifications based on the denotational semantics of CSP. ProB is well-suited to check temporal properties on CSP processes. For Timed CSP there does not exist comprehensive tool support yet. Therefore, we have formalized Timed CSP in the Isabelle/HOL theorem prover as briefly explained in Section 3.1.

2.2 LLVM

The LLVM compiler infrastructure provides a modular framework that can be easily extended by user-defined compilation passes. It also offers a diverse set of predefined analyses, i.e. points-to analysis by Steensgaard [Ste96], and optimizations that can be used out of the box. This makes LLVM a great platform for the development of source code transformation and analysis tools. The heart of the compiler infrastructure project is its intermediate representation (IR). It is a typed

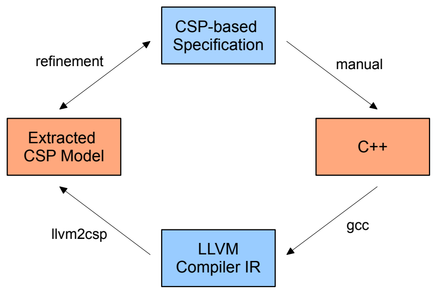


Figure 3: The VATES Proof Framework

assembler-like language [LA08], which is used internally as the basis for compiler optimizations. The LLVM framework provides gcc-based frontends for a variety of programming languages, including C++. The existence of the gcc-based front-end enables us to adapt our approach, which currently only supports C++, to a couple of other programming languages with little effort because it is source-language-independent and relies on the LLVM IR only.

2.3 Isabelle/HOL

Isabelle is a generic interactive proof assistant. It enables the formalization of mathematical models and provides tools for proving theorems that are mechanically checked. Isabelle can be instantiated with different so-called object logics. One particular instantiation of it is Isabelle/HOL [NPW02], which is based on Higher Order Logic. The main advantage of HOL is its very high expressive power. Theorem provers based on HOL require a high level of expertise but allow reasoning about models whose state space is too large (or even infinite) to be automatically checked by, say, a model checker. Unlike model checking, proving theorems in a theorem prover like Isabelle/HOL is highly interactive. Specifications have to be designed carefully to enable properties about them to be proved.

3 The VATES Approach

The context of our approach is the VATES project [GHJ07]. Its aim is to develop concepts for verifying the correctness of embedded software. Our goal is to support the verification of crucial properties on all abstraction levels of such a software system, from the abstract specification down to executable code.

The structure of our approach is given in Figure 3. We start with a high-level CSP-based specification where crucial properties are verified. To this end, we have developed a formalization of Timed CSP in the Isabelle/HOL theorem prover and the proof technique of network invariants to verify infinite-state models. This is explained in Section 3.1 in more detail.

Since we deal with embedded applications we want to allow manual code optimizations. We therefore assume that a Software developer implements this high-level specification in a high-level programming language such as C++. The high-level language can be further translated into the LLVM intermediate representation, e.g. by using the gcc compiler. We chose to relate the abstract CSP-model and the intermediate LLVM representation by automatically extracting a low-level CSP-model from it. This is done with our tool called llvm2csp which is explained in Section 3.2. To show the refinement relation between the high-level and the low-level CSP models, our formalization of Timed CSP or standard tools like FDR2 can be applied.

A crucial part in our approach is to show that the extraction algorithm of llvm2csp preserves the semantics of LLVM. To this end, we developed an operational semantics of LLVM. Due to the simplicity of intermediate languages, this is far more easily than defining a comprehensive formal semantics of e.g. C++. Furthermore, we defined a variant of bisimulation which enables to semantically compare LLVM programs and CSP models. This is presented in Section 3.3.

Altogether we are thereby able to formally relate the high-level CSP-based specification and its corresponding implementation in LLVM. Note that following the overall approach we do not have to formalize the semantics of a high-level programming language like C++ which is known to be complex task in its own. This becomes even more complicated by the introduction of concurrency. Since we want to verify the whole development chain, the formalization of some intermediate representation is inevitable in each case. So by considering *only* the intermediate language layer circumvents the complex task of formally defining the semantics of C++.

3.1 Verification with Timed CSP and Network Invariants

In a previous paper [GG09], we proposed a formalization of the operational semantics of the process calculus Timed CSP in the Isabelle/HOL theorem prover [NPW02].

3.1.1 Formalization of Timed CSP

We combined the advantages of specifying real-time systems concisely and of mechanizing correctness proofs for properties of their specifications. We transfered the coalgebraic notions of bisimulation and of invariants to Timed CSP. This allows on the one hand to relate behaviorally equivalent Timed CSP processes and on the other hand to state invariant behavior of processes. To this end, we formalized the syntax of Timed CSP as inductive datatype and the operational semantics as inductively defined set of triples. It is convenient to define (greatest) bisimulations and (greatest) invariants w.r.t. to state predicates coinductively. We showed that all the considered kinds of bisimulation (strong, weak and weak timed) fulfill the congruence property with respect to the structure of Timed CSP processes. Furthermore, we showed that coalgebraic invariants are well-suited to express certain liveness conditions and that these special invariants are closed under bisimulations. This property is useful for verification as shown in [GG09] in the context of a (rather simple) satellite system specification.

3.1.2 Parameterized Systems and Network Invariants

In ongoing work we extend this theory by so-called network invariants which are suitable to verify parameterized systems. Our motivation is BOSS's real-time scheduler which we have modeled in Timed CSP. The main structure of the model is given in Figure 4. The overall system includes the scheduler itself and an arbitrarily large number of threads.

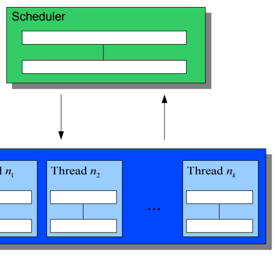


Figure 4: Structure of our Scheduler Model

The scheduler runs in parallel with arbitrarily many processes representing the threads to be managed. Each thread is characterized by a name and a priority. In order to verify the scheduler system, the specific details of the threads should have no relevance. The scheduler system should, e.g., be deadlock-free in every case, i.e., for every possible list of threads. Thus, an appropriate "induction" on the length of the list should be enough to state the system's correctness. On a certain level of abstraction, all the CSP processes representing threads are homogeneous from the scheduler's perspective as they can communicate with the scheduler (e.g., yield control), and conversely the scheduler with the threads (e.g., give control to a thread). That is why the scheduler system can be seen as a parameterized system. As a consequence, verification of parameterized systems appears to be a promising technique to verify real-time operating systems.

Parameterized systems, as considered here, have the form:

$$N_n = P_0 \odot \underbrace{P \odot \cdots \odot P}_{n}$$

where the variable n (representing a natural number) is the parameter of the system. P_0 is a control process and $P \odot \cdots \odot P$ a network of homogeneous processes. The operator $\odot$ is some kind of parallel composition, which may be equipped with hiding and renaming of communication channels. Network Invariants can be used to verify parameterized systems by dividing the infinite-state verification problem into several (ideally finite-state) verification problems such that automatic verification tools can be used. The long-term goal of our work is to combine our Isabelle/HOL formalization with automatic verification tools. In this section we focus on the presentation of the results of [WL90] concerning the inductive verification of parameterized systems as this is sufficient here. We have formalized these in the Isabelle/HOL theorem prover. We have, however, also formalized the slightly more general results of [KM89] in Isabelle.

The idea of the "appropriate induction" is formalized in network invariants: To verify whether the system N_n (regardless of the concrete parameter) implements a specification S, the main idea is to use an appropriate invariant, which overapproximates each instance such that the abstracted system is still contained in the specification S. In [WL90], it is shown how network invariants can be set up in process algebras like CCS and CSP. There, a general technique for verifying parameterized systems based on network invariants is presented: if one wishes to show that N_n fulfills a certain specification S with respect to a certain implementation relation, i.e., $N_n \leq S$, it must basically be shown (for some process Inv) that $P_0 \leq Inv$, that $Inv \odot P \leq Inv$ and that $Inv \leq S$. The crucial point is to find an appropriate invariant Inv. By induction on n, one can

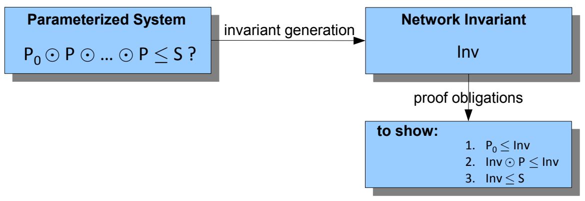


Figure 5: Framework for Verifying Parameterized Systems

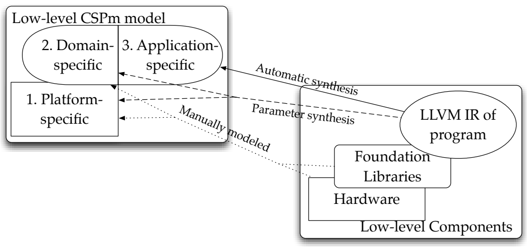


Figure 6: The three Parts of the Low-Level CSP Model

deduce that $N_n \leq Inv$. By proving that the invariant is contained in the specification, i.e., $Inv \leq S$, one can deduce that $N_n \leq S$ by transitivity of the implementation relation $\leq$. There are two main tasks that have to be performed when working with network invariants for verification. The first is finding a process which may serve as appropriate "network invariant". The second task is showing that the found process *is* indeed a network invariant. In [CGJ97], for example, a technique for finding network invariants based on network grammars is presented. This gives a more general framework than [WL90]. In [GL08], the technique of network invariants is used in the context of timed systems. The approach adopted there consists of two steps. First, a safe abstraction is performed on a given timed system. Thereby, safe means that LTL formulas are preserved under the abstraction. The abstract system is then used as a network invariant, which allows for the verification of the whole parameterized system.

The overall verification flow that we use is given in Figure 5. It is subject to future work to integrate invariant generation algorithms and automatic verification tools such as FDR2 into the existing Timed CSP formalization in Isabelle/HOL.

3.2 Relating High-Level and Low-Level Models

In this section we give a rough idea of the extraction algorithm implemented in our llvm2csp tool presented in [KH09]. Since we deal with multithreaded programs besides the behavior of individual threads, the model also needs to include information about the actual execution platform as explained in the following subsection.

```
channel read1, write1, read2, write2, read3, write3, read4, write4 : {0,1}
V1 = let V1'(v) = read1!v -> V1'(v) [] write1?x -> V1'(x)
  within read1?x -> V1 [] write1?x -> V1'(x)
V2 = V1[[read1 <- read2, write1 <- write2]]
V3 = V1[[read1 <- read3, write1 <- write3]]
V4 = V1[[read1 <- read4, write1 <- write4]]
WithHeap(P) = (P) [|{|read1, write1, ..|}|] (V1 ||| V2 ||| V3 ||| V4)
```

Figure 7: CSP$_M$ Model of the Heap.

3.2.1 Synthesizing a Low-Level CSP Model

The low-level CSP$_M$[4] model as depicted in Figure 6 contains not only processes, types and channels that are automatically generated from the LLVM IR of a program but also two predefined parts which model platform- and domain-specific parts of the system under investigation. The platform-specific part comprises the environment model and hardware details, while the domain-specific part encompasses aspects that are common to a domain of applications, e.g. system startup and scheduling, which are provided as foundation libraries that the program builds on. These two parts are mostly manually modeled but are parameterized so that they can be reused by all applications of the domain they have been designed for. Examples of such parameters are typing information for the channels and the set of thread identifiers. The third part is the application-specific one, which describes the behavior of the threads of a multithreaded program with respect to a set of given variable names, function calls and annotations [5]. Our llvm2csp tool was already successfully used to create the low-level model of the scheduler of the BOSS operating system pico-kernel, which we presented in [KBGG09]. Nevertheless we are constantly extending it by further features.

3.2.2 Design of the Low-Level CSP Model

As discussed in the previous section, the low-level CSP model is divided into three distinct parts. The domain- and platform-specific parts are manually modeled but are parameterized. The parameters and the application-specific part are synthesized from the LLVM IR of the program under consideration. Since we aim to use FDR2 for establishing the formal refinement relation between the specification and the low-level model, all models must be designed as efficiently as possible. The FDR2 manual contains a couple of rules that have to be taken into account when creating a CSP$_M$ model to achieve the best performance with FDR2.

Fig. 7 shows an efficient example of modeling a memory that stores four bit fields, constructed of parallel processes ($V1,\ldots,V4$) that model a single variable each. These four processes are structurally equal, so just one of them is modeled manually ($V1$), the others being derived from it by renaming [6]. This model of a memory is a process, which is synchronized with the application

[4] CSP$_M$ is a machine readable form of CSP extended by concepts from functional programming

[5] Annotations can be realized using so-called ghost method and ghost variables. A ghost method is a method that modifies ghost variables only, while a ghost variable is a variable that is used for verification purposes only. Ghost code is commonly compiled into the IR for verification purposes but is not part of the final binary.

[6] One of the rules mentioned before is, for example, that renaming is to be used in preference to the parameterizing of a process definition.

specific part later on using the function (*WhithHeap*). We use this concept to model the heap and the stacks of the threads. The process allows us to read an arbitrary value from uninitialized memory cells.

Our approach makes strong use of abstraction to reduce the size of the resulting low-level model in terms of reachable states. This includes abstracting the ranges of data types and abstracting away regions of code that do not transitively influence any of a given set of variables to be included in the low-level model. If, for example, concurrent accesses to a shared counter variable have to be proved race-condition-free, it is sufficient to build the model from the accesses to this shared counter and the locks protecting it.

The expressiveness of CSP_M imposes a limiting factor to formalizing the semantics of the LLVM IR. We therefore restrict ourselves to modeling facilities that are available in CSP_M. Our approach currently supports functions, function calls, conditional and unconditional branching as well as integer arithmetic. It builds on a memory model that supports integers, arrays and uninitialized values. Depending on the properties to be proved on the models, we also use the concept of error codes to detect such sources of unwanted behavior or to signal situations that were introduced by abstractions during synthesis of the model. An error code is a fresh event $a \notin \Sigma$ and is always used in the pattern $a \rightarrow STOP$. In [KH09], we use this concept to detect integer overflow that was introduced by abstraction and did not indicate a real error in the low-level model. A method on the LLVM IR level is translated into a CSP_M function by llvm2csp. The function returns sequential processes, each modeling a single IR operation. These application-specific processes end up in a domain-specific process modeling the continuation of the application, possibly including a thread switch. Further details of the application-, domain- and platform-specific models are given in [KH09].

3.3 Verification on the Intermediate Level

As explained, in our approach we relate process-algebraic system models to their implementations given in the LLVM intermediate representation. To formally account for the correctness of this relation, we formally defined the operational semantics of the LLVM intermediate language in Isabelle/HOL and establish a bisimulation relation between the implied labeled transition system (LTS) and the LTS defined by the operational semantics of the process algebraic model. Using this concept we plan to verify the LLVM2CSP extraction algorithm in future work.

3.3.1 Operational Semantics of LLVM

We model a semantic state or configuration Conf as a tuple consisting of functions S for the stack, H for the heap and M for the overall memory, as well as an instruction pointer l:

$$\mathrm{Conf} \equiv \langle S, H, M, l \rangle$$

The crucial part of the semantics is the memory model. Our model is inspired by the model presented in [BL05]. The functions S and H map non-pointer identifiers to their appropriate types and values, and pointer variables to their types and addresses. A memory state within a configuration is modeled by the tuple $M = (Addr, B, F, C)$. We define $Addr$ to be the natural numbers $\mathbb{N}$. It is however possible to replace it with types that represent memory addresses more closely. B yields the block size of a given address, while F marks a memory block as free or allocated. Finally, C returns the content stored at a certain address.

In the context of embedded (operating) systems, models are often non-deterministic. The frequent interaction of these systems with the surrounding environment, for example through a sensor that delivers some kind of information, makes the execution highly non-deterministic. Since LLVM natively offers no means to handle non-determinism, we decided to realize bounded non-determinism by annotating the source code.

Sources for non-determinism, e.g. reading from a memory location where a value delivered from a sensor is stored, need to be annotated in the code. The annotation contains the possible range of values that the sensor may return. For the formal semantics, this implies that a transition from a configuration Conf may have more than one successor configuration. For every identifier that is marked as non-deterministic, the annotations define a function $range : Identifier \rightarrow \mathscr{P}(Value)$. The successor configurations differ from the original configuration only in the value of register %dest_ident, to which the value read from the marked variable %source_ident is stored. The value is non-deterministically chosen from the set defined by the *range* function.

3.3.2 Bisimulation Relation

A labeled transition system LTS over the alphabet A is defined as a tuple (S,T), where S is a set of states and $T \subseteq S \times A \times S$ is the transition relation. A special label $\tau \in A$ is used as label for internal transitions.

We associate the set of states S with the possible configurations Conf from the previous section. As labels, we use dedicated names that relate LLVM code behavior to events in the process algebraic specification. We encode information needed for verification purposes on the process level using this technique. This includes function and system calls, as well as signals to other threads or user input. The names are annotated to the original source code.

We explain the idea by using the labels of the LTS to encode that and how a certain variable in a code snippet changes. Since we are interested in changes to variables from a set V, the only instruction that produces externally visible behavior is the `store` instruction. For the `store` instruction, we label the edge of the LTS with the value $\%dest\_ident.S(\%source\_ident)$, if %dest_ident is one of the variables to be tracked. For all other instructions except for the store instruction, we use the τ label indicating silent transitions.

The right side of Figure 9 shows the part of the labeled transition system that corresponds to the code snippet from Figure 8. Here, the set V of variables to be tracked is defined as $\{\%i, \%x\}$. First, the content of a variable %b, that was marked non-deterministic and that can range over the values 0 and 1 is loaded to the register %0 and compared with the constant 0. If the value of %b is greater than 0, the register %1, which holds the boolean value that is evaluated for the following branching condition, is set to true. In this case, the execution continues at the block labeled with `bb`, otherwise the execution continues at label `bb1`. At label `bb`, the content of the variable %c is loaded to register %2, the constant 2 is subtracted and the result is stored to variable %c. Afterwards, the content of the variable %c is stored to variable %x. Execution then continues at label `bb2`. At label `bb1`, the constant 7 is stored to variable %x and then execution proceeds at label `bb2`.

```
        store i32 0, i32* %i, align 4
        %0 = load i32* %b, align 4                      ; <i32> [#uses=1]
        %1 = icmp eq i32 %0, 1                          ; <i1> [#uses=1]
        br i1 %1, label %bb, label %bb1

   bb:                                                  ; preds = %entry
        %2 = load i32* %c, align 4                      ; <i32> [#uses=1]
        %3 = sub i32 %2, 2                              ; <i32> [#uses=1]
        store i32 %3, i32* %c, align 4
        %4 = load i32* %c, align 4                      ; <i32> [#uses=1]
        store i32 %4, i32* %x, align 4
        br label %bb2

   bb1:                                                 ; preds = %entry
        store i32 5, i32* %x, align 4
        br label %bb2
   bb2: ...
```

Figure 8: LLVM Code

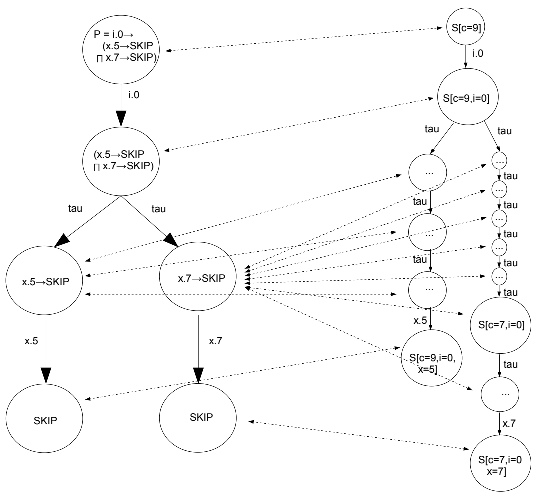


Figure 9: The Bisimulation Relation between the two LTS

Informally, the definition of weak bisimulation says that any possible transitions in a state P can be associated with a transition α in the corresponding state Q. An arbitrary number of internal τ transitions is allowed before and after the corresponding α in the evolution of Q occurs. Two processes P and Q are weakly bisimilar iff there exists a bisimulation relation R such that $(P,Q) \in R$.

In our case, T is the disjoint union of the transition systems of CSP and LLVM. Using the previously defined construction rules for the LTS representing LLVM code, we can show that the LLVM code shown in Figure 8 is bisimilar to the following process term:

$$P = \%i.0 \rightarrow (\%x.5 \ \rightarrow SKIP \sqcap \%x.7 \ \rightarrow SKIP).$$

The process specifies that a variable %i is set to 0 and afterwards it is non-deterministically chosen if the variable $\%$x is set to 5 or 7. Figure 9 shows the two labeled transition systems and sketches the bisimulation relation. Note that for every LLVM instruction that does not change any of the variables from the predefined set V, the outgoing edges representing the change in state by the instruction are labeled with τ. The nodes that are identified by the bisimulation relation are connected by the dashed arrows.

In ongoing work we formalize this notion of bisimulation using the theorem prover Isabelle/HOL in order to achieve the mechanized verification of the llvm2csp extraction algorithm presented in the previous section.

4 Related Work

The verification of low-level software systems and embedded operating systems has attracted several research projects.

Spec# [BLS04] and VCC [CDH$^+$09] are verification methodologies that translate high-level languages like C# and C to the intermediate language BoogiePL. Necessary verification information is annotated to the source code. To discharge verification conditions, the automatic theorem prover Z3 is used. In contrast to our approach, BoogiePL is not a compiler-intermediate representation but a language tailored for verification purposes. Furthermore, we focus on refinement and transformation of a high-level specification to executable code rather than direct source-code verification.

The AVACS [BDF$^+$07] project deals with the verification of complex (real-time) systems. Verification is carried out at the specification level. Transformations to and the verification of executable code are not considered within the project.

Another project concerned with the verification of a real-time operating system kernel is the VFiasco project [HT05]. Verification is carried out at source code level by defining the denotational semantics for a subset of C++. One of its results is the verification of Duff's Device, but the techniques were not applied to existing operating system components.
The L4.verified project [EKD$^+$07] uses a refinement approach that proves different layers of abstraction to be consistent. The lowest level of abstraction is given by C code, which is shown to be a refinement of a more abstract design. Neither of these approaches covers concurrency. Furthermore, the transformation to intermediate and executable code is not considered.

Closely related to our work is the verification of a real-time operating system controlling a space satellite using Timed-CSP-Z [SSC03]. Timed-CSP-Z models are translated into a petri-net-based formalism and the resulting petri-nets are analyzed. The approach focuses on deadlock detection. Neither refinement proofs nor transformations to code are within the focus of this work.

5 Conclusion and Future Work

In this paper we have summarized the main results of the VATES project so far. We briefly explained the formalization of Timed CSP in the Isabelle/HOL theorem prover and the potential of verification techniques for parameterized systems, namely network invariants, in the context of real-time operating systems. We presented the idea of the extraction algorithm which computes a CSP model for a given LLVM program. Finally, we explained an operational semantics of LLVM and how it is possible to formally relate LLVM programs and CSP models with it based on bisimulation.

We are currently working on the integration of timing behavior into our framework. Since timing analyses are already possible on the most abstract layer, we need to extend the llvm2csp tool to generate Timed CSP models. To realize this, it is necessary to augment the LLVM syntax and semantics with information about timing behavior. This includes accounting for processor-specific features like pipelining and cache management.

Since the theory underlying our approach is rather complex we plan to mechanize it entirely using the Isabelle/HOL theorem prover. Thereby we can claim that our transformations are indeed correct.

The results obtained so far are very promising. In future work, we will apply our verification approach to more complex systems, starting with the integration of further system components into the verification of the BOSS operating system.

We are convinced that our approach has the potential to enable a methodology that seamlessly integrates the modeling, implementation, transformation and verification stages of embedded real-time system development.

References

[BDF$^+$07] B. Becker, W. Damm, M. Fränzle, E. Olderog, A. Podelski, R. Wilhelm. SFB/TR 14 AVACS – Automatic Verification and Analysis of Complex Systems. *it – Information Technology* 49(2):118–126, 2007. http://it-Information-Technology.de, DOI 10.1524/itit.2007.49.2.118.

[BL05] S. Blazy, X. Leroy. Formal Verification of a Memory Model for *C*-Like Imperative Languages. In *ICFEM*. Pp. 280–299. 2005.

[BLS04] M. Barnett, K. R. M. Leino, W. Schulte. The Spec# Programming System: An Overview. In *CASSIS 2004*. Pp. 49–69. Springer, 2004.

[CDH$^+$09] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, S. Tobies. VCC: A Practical System for Verifying Concurrent C. In *Theorem Proving in Higher Order Logics. TPHOLs-09, August 2009, Munich, Germany*. Lecture Notes in Computer Science, LNCS 5674, pp. 23–42. Springer, 8 2009.
http://dx.doi.org/10.1007/978-3-642-03359-9_2

[CGJ97] E. M. Clarke, O. Grumberg, S. Jha. Verifying Parameterized Networks. *ACM Trans. Program. Lang. Syst.* 19(5):726–750, 1997.

[EKD$^+$07] K. Elphinstone, G. Klein, P. Derrin, T. Roscoe, G. Heiser. Towards a Practical, Verified Kernel. In *HOTOS'07: Proceedings of the 11th USENIX workshop on Hot topics in operating systems*. Pp. 1–6. USENIX Association, Berkeley, CA, USA, 2007.

[GG09] T. Göthel, S. Glesner. Machine-Checkable Timed CSP. In *Proc. of The First NASA Formal Methods Symposium*. Pp. 126–135. NASA Conference Publication, 2009.

[GHJ07] S. Glesner, S. Helke, S. Jähnichen. VATES: Verifying the Core of a Flying Sensor. In *Proc. Conquest 2007*. dpunkt Verlag, 2007.

[GL08] O. Grinchtein, M. Leucker. Network Invariants for Real-Time Systems. *Form. Asp. Comput.* 20(6):619–635, 2008.
doi:http://dx.doi.org/10.1007/s00165-008-0089-0

[GRA05] M. Goldsmith, B. Roscoe, P. Armstrong. Failures-Divergence Refinement - FDR2 User Manual. http://www.fsel.com/documentation/fdr2/fdr2manual.pdf, 2005.
http://www.fsel.com/documentation/fdr2/fdr2manual.pdf

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, London, 1985. ISBN: 0-131-53271-5.

[HT05] M. Hohmuth, H. Tews. The VFiasco Approach for a Verified Operating System. In *Proc. 2nd ECOOP Workshop on Programming Languages and Operating Systems*. 2005.

[KBGG09] M. Kleine, B. Bartels, T. Göthel, S. Glesner. Verifying the Implementation of an Operating System Scheduler. In *Poster Proceedings of the 3rd IEEE International Symposium on Theoretical Aspects of Software Engineering*. Tianjin, China, July 2009. In press.

[KH09] M. Kleine, S. Helke. Low Level Code Verification Based on CSP Models. In Oliveira and Woodcock (eds.), *Brazilian Symposium on Formal Methods (SBMF 2009)*. Springer, August 2009. In press.

[KM89] R. P. Kurshan, K. McMillan. A structural Induction Theorem for Processes. In *PODC '89: Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*. Pp. 239–247. ACM, New York, NY, USA, 1989.
doi:http://doi.acm.org/10.1145/72981.72998

[LA04] C. Lattner, V. Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the 2004 International Symposium on Code Generation and Optimization (CGO'04)*. Palo Alto, California, Mar 2004.

[LA08] C. Lattner, V. Adve. LLVM Language Reference Manual. http://llvm.org/docs/LangRef.html, 2008.
http://llvm.org/docs/LangRef.html

[LF08] M. Leuschel, M. Fontaine. Probing the Depths of CSP-M: A new FDR-compliant Validation Tool. *ICFEM 2008*, pp. –, 2008.

[MBK06] *Dependable Software (BOSS) for the BEESAT Pico Satellite*. Data Systems In Aerospace - DASIA 2006, May 2006, Berlin. 2006.

[NPW02] T. Nipkow, L. C. Paulson, M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. LNCS 2283. Springer, 2002.

[Sch99] S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1999.

[SSC03] A. M. Sherif, A. Sampaio, S. Cavalcante. Specification and Validation of the SACI-1 On-Board Computer Using Timed-CSP-Z and Petri Nets. In *ICATPN*. Pp. 161–180. 2003.

[Ste96] B. Steensgaard. Points-to Analysis in Almost Linear Time. In *POPL '96: Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. Pp. 32–41. ACM, New York, NY, USA, 1996.
doi:http://doi.acm.org/10.1145/237721.237727

[WL90] P. Wolper, V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. Pp. 68–80. Springer-Verlag, London, UK, 1990.

Specification and Verification of Model Transformations [*]

Frank Hermann[1], **Mathias Hülsbusch**[2], **Barbara König**[2]

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

[2] Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany,
{`mathias.huelsbusch,barbara_koenig`}`(at)uni-due.de`

Abstract: Model transformations are a key concept within model driven development and there is an enormous need for suitable formal analysis techniques for model transformations, in particular with respect to behavioural equivalence of source models and their corresponding target models.

For this reason, we discuss the general challenges that arise for the specification and verification of model transformations and present suitable formal techniques that are based on graph transformation. In this context, triple graph grammars show many benefits for the specification process, e.g. modelers can work on an intuitive level of abstraction and there are formal results for syntactical correctness, completeness and efficient execution. In order to verify model transformations with respect to behavioural equivalence we apply well-studied techniques based on the double pushout approach with borrowed context, for which the model transformations specified by triple graph transformation rules are flattened to plain (in-situ) graph transformation rules.

The potential and adequateness of the presented techniques are demonstrated by an intuitive example, for which we show the correctness of the model transformation with respect to bisimilarity of source and target models.

Keywords: Model transformation, behavioural equivalence, verification

1 Introduction

In the setting of model driven architecture (MDA), a system is implemented by first specifying an abstract model, which is subsequently refined to executable code. This is done by model transformations, which transform a source model into a more concrete target model. The Object Mangaement Group has also introduced a standard for model transformations: QVT (Query/View/Transformation). A special case is refactoring, where only the internal structure of the model or system is changed and potentially optimized. Since refactorings are expected not to modify the functional behaviour of the system, the notion of behaviour preservation is crucial: how can we specify and verify that a model keeps its original behaviour after several refactoring

[*] Research supported by DFG project Behaviour-GT.

steps? The same question is often relevant for model transformations, in order to show that the implementation matches the original specification.

In this paper we will summarize some results on the specification and verification of model transformations. As underlying modelling framework we will use graph transformation, which is well-suited to handle the graph-like structures usually arising in MDA and UML. However, many existing model transformations in practice are directly encoded as e.g. XSLT-transformations. As a first contribution of this paper we will discuss the main challenges of model transformations and present the main benefits of using graph transformation with respect to technical results and with respect to usability.

The main two aims of the paper are the following. First, we introduce recent results on triple graph grammars, a formalism that allows to specify model transformations by constructing source and target models simultaneously and recording correspondences. Second, we will describe a technique for verifying that model transformations preserve the behaviour of a model, showing that strong bisimilarity is preserved by the transformation with a variation of the so-called borrowed context technique. For this, we will derive in-situ transformation rules from triple graph grammars. Both parts of the paper are based on the same example: a model transformation translating network-like models with different types of (bidirectional and unidirectional) links.

The structure of the paper is as follows. Section 2 describes the various challenges arising in the area of model transformation. Sections 3 and 4 subsequently introduce triple graph grammars for the specification of model transformations and describe several results obtained for triple graph grammars (e.g., syntactical correctness and completeness). In Section 5 we describe how to verify the example transformation using the borrowed context technique. Finally, we will compare with related work in Section 6 and conclude (Section 7).

2 Challenges for Model Transformations

Model transformations appear in several contexts, e.g. in the various facets of model driven architecture encompassing model refinement and interoperability of system components. The involved languages can be closely related or they can be more heterogeneous, e.g. in the special case of model refactoring the source language and the target language are the same. From a general point of view, a model transformation $MT : VL_S \Rrightarrow VL_T$ between visual languages transforms models from the source language VL_S to models of the target language VL_T. Main challenges were described in [SK08] for model transformation approaches based on triple graph grammars. Here, we extend this list and also the scope and describe general challenges for model transformations.

There are two dimensions, which contain major challenges for model transformations being on the one hand functional aspects and on the other hand non-functional aspects. The first dimension of functional aspects concerns the reliability of the produced results. Depending on the concrete application of a model transformation $MT : VL_S \Rrightarrow VL_T$, the following properties may have to be ensured.

1. *Syntactical Correctness:* For each model $M_S \in VL_S$ that is transformed by MT the resulting model M_T has to be syntactically correct, i.e. $M_T \in VL_T$.

2. *Semantical Correctness:* The semantics of each model $M_S \in VL_S$ that is transformed by MT has to be preserved or reflected, respectively.

3. *Completeness:* The model transformation MT can be performed on each model $M_S \in VL_S$. Additionally, MT can be required to reach all models $M_T \in VL_T$.

4. *Functional Behaviour:* For each source model M_S the model transformation MT will always terminate and lead to the same resulting target model M_T.

The second dimension of non-functional aspects of model transformations concerns usability and applicability. Therefore, from the application point of view some of the following challenges are also main requirements.

1. *Efficiency:* Model transformations should have polynomial space and time complexity. Furthermore, there may be further time constraints that need to be respected, depending on the application domain and the intended way of use.

2. *Intuitive Specification:* The specification of model transformations can be performed based on visual patterns that describe how model fragments in a source model correspond to model fragments in a target model. The components of a model transformation can be visualized in the concrete syntax of the visual languages.

3. *Maintainability:* Extensions and modifications of a model transformation need to be easy. Side effects of local changes shall be handled and analyzed automatically.

4. *Expressiveness:* Special control conditions have to be available in order to handle more complex models, which, e.g., contain substructures with a partial ordering or hierarchies.

5. *Bidirectional model transformations:* The specification of a model transformation should provide the basis for both, a model transformation from the source to the target language and a model transformation in the inverse direction.

In the following section we present suitable techniques for the specification of model transformations based on graph transformation. These techniques provide validated and verified capabilities for a wide range of the challenges listed above.

3 Specification of Model Transformations by Triple Graph Grammars

A promising and well studied approach for the specification of model transformations is based on triple graph transformation [Sch94]. This section presents its main concepts and Sec. 4 shows its advantages from the formal and from the application point of view. The most important advantage of triple graph transformation is the combination of both, its intuitive way of specifying model transformations and its formal basis, for which correctness and completeness results are available.

Triple graphs combine three graphs - one for the source model, one for the target model and one in between, together with connecting graph morphisms for the specification of the correspondences between the elements in the source and the target model. This extension of plain graphs improves the definition of model transformations. Source models are parsed and their corresponding target models are completed without the need of deleting the source model in between. The correspondences between both models are used to guide the transformation process.

Definition 1 (Category **TripleGraphs**) Three graphs G^S, G^C, and G^T, called source, connection, and target graph, together with two graph morphisms $s_G : G^C \to G^S$ and $t_G : G^C \to G^T$ form a triple graph $G = (G^S \xleftarrow{s_G} G^C \xrightarrow{t_G} G^T)$. G is called *empty*, if G^S, G^C, and G^T are empty graphs.

A triple graph morphism $m = (m^S, m^C, m^T) : G \to H$ between two triple graphs $G = (G^S \xleftarrow{s_G} G^C \xrightarrow{t_G} G^T)$ and $H = (H^S \xleftarrow{s_H} H^C \xrightarrow{t_H} H^T)$ consists of three graph morphisms $m^S : G^S \to H^S$, $m_C : G^C \to H^C$ and $m_T : G^T \to H^T$ such that $m_S \circ s_G = s_H \circ m^C$ and $m^T \circ t_G = t_H \circ m^C$. It is injective, if morphisms m_S, m_C and m_T are injective. Triple graphs and triple graph morphisms form the category **TripleGraphs**. Given a triple graph TG, called type graph, the category **TripleGraphs**$_{TG}$ of typed triple graphs is given by the slice category **TripleGraphs**$\backslash TG$.

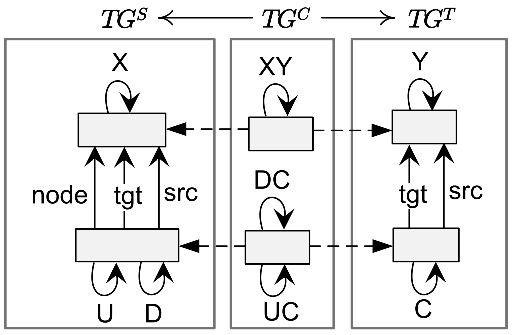


Figure 1: Triple Type Graph $TG = (TG^S \leftarrow TG^C \rightarrow TG^T)$

In the examples of this paper we consider a model transformation MT : BidiDiLang $\Rrightarrow$ UniDiLang between communication structure models. The language BidiDiLang contains models with bidirectional and undirectional links and these models are transformed to models with unidirectional connections only in the language UniDiLang. Each pair of corresponding source and target models is given by a triple graph typed over the triple type graph TG in Fig. 1, which extensively uses the concept of labeled nodes, i.e. loops of different edge types. This reduces the amount of node types, which allows us in Sec. 5.2 to verify the semantical correctness of the model transformation by transforming the triple rules into suitable in situ rules and to analyze them with respect to rules for the mixed semantics, i.e. a semantics for models that contain source and target elements at the same time.

In order to improve the intuition of triple graphs typed over TG we present the graphs by a visualization. The fill colour of the elements in the source model is light red while it is blue for the correspondence elements and yellow for elements in the target model. Furthermore, the edge types “X”, “D” (directed link), “U” (undirected link), “Y”, “C” (connection) as well as “XY”, "DC" and "UC" for the correspondence component will always be loops at different nodes and we simply write the label inside the node rectangle resp. hexagon. Note that the visualization of the type graph in Fig. 2 contains different nodes for the labels "D" and "U" as well as for "DC" and "UC". This distinction is not present in the underlying type graph in Fig. 1, but the triple rules in Fig. 4 ensure that the created models will always respect these restrictions. For instance, edges of type "src" in the source component will always start at a node labelled with "D" and end at a node labelled with "X".

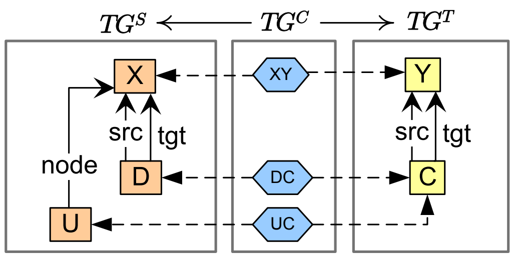


Figure 2: Visualization of the Triple Type Graph *TG*

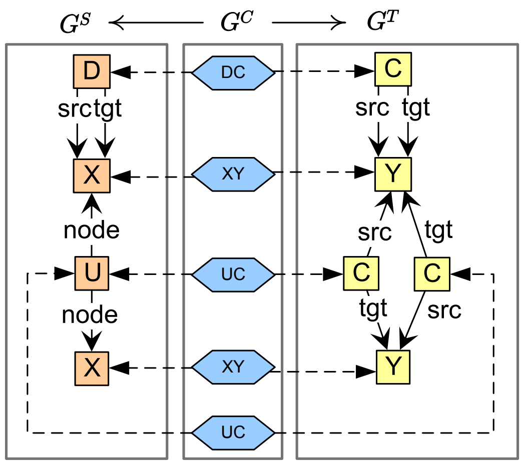


Figure 3: Triple Graph G with source model G^S and target model G^T

Example 1 (Triple graph) *The triple graph in Fig. 3 is typed over TG and shows an integrated model consisting of a source model G^S (left) and a target model G^T (right), which are connected via the correspondence nodes in the correspondence graph G^C. The source model specifies a node with label "X" having a message "m", a self referring directed link "D" and an outgoing undirected link "U". Similarly the target model contains two nodes, but labelled with "Y" and instead of one undirected link between both nodes there are two connections "C" defining possibilities for communication in both directions. The corresponding elements of both models are related by graph morphisms (indicated by dashed lines) from the correspondence graph (light blue) to the source and target componencts, respectively.*

A triple graph grammar generates a language of triple graphs, i.e. a language of integrated models consisting of models of the source and the target language and a correspondence structure in between. The triple rules of a triple graph grammar specify the synchronous creation of elements in the source component and its corresponding elements in the target component. Therefore, triple rules are non-deleting. The triple rules of a triple graph grammar are the basis for deriving the operational rules of the model transformation from models of one language into the other.

Definition 2 (Triple Graph Transformation and Triple Graph Grammar) A triple rule $tr = L \xrightarrow{tr} R$ is an injective triple graph morphisms tr from a triple graph L (left hand side) to a triple graph R (right hand side). A triple graph grammar $TGG = (TG, S, TR)$ consists of a triple graph TG (type graph), a triple graph S (start graph) and triple rules TR - both typed over TG.

Given a triple rule $tr = (tr^S, tr^C, tr^T) : L \to R$, a triple graph G and an injective triple graph morphism $m = (m^S, m^C, m^T) : L \to G$, called triple match m, a triple graph transformation step (TGT-step) $G \xRightarrow{tr,m} H$ from G to a triple graph H is given by a pushout in **TripleGraphs**. The triple graph language L of TGG is defined by $L = \{G \mid \exists$ triple graph transformation $S \Rightarrow^* G\}$.

$$\begin{tikzcd}
L \arrow[d, "tr"'] & = & (L^S \arrow[d, "m^S"'] & L^C \arrow[l, "s_L"'] \arrow[r, "t_L"] \arrow[d, "m^C"'] & TL) \arrow[d, "m^T"] \\
R & = & (R^S & R^C \arrow[l, "s_R"] \arrow[r, "t_R"'] & R^T)
\end{tikzcd}$$

Triple Rule

$$\begin{tikzcd}
L \arrow[r, "tr"] \arrow[d, "m"'] \arrow[dr, phantom, "(PO)"] & R \arrow[d, "n"] \\
G \arrow[r, "t"'] & H
\end{tikzcd}$$

Transformation Step

Model transformations based on triple graph transformation are performed by taking the source model and extending it to an integrated model, where all its corresponding elements in the correspondence and target component are completed. Thereafter, this integrated model is restricted to its target component being the result of the model transformation. For this reason, triple graph transformation rules are non-deleting. This implies that the first step in the DPO graph transformation approach [EEPT06] can be omitted, because the creation of elements is performed in the second step.

Example 2 (Triple Graph Grammar) *The triple graph grammar $TGG = (TG, \emptyset, TR)$ for the model transformation MT :* BidiDiLang $\Rrightarrow$ UniDiLang *contains the triple type graph in Fig. 2, the empty start graph and the rules TR in Fig. 4. Each rule specifies a pattern that describes how particular fragments of communication structure models shall be related. We present the rules in compact notation, i.e. the left and the right hand side of a rule are shown in one triple graph and the additional elements that occur in the right hand side only are marked by green line colour and a double plus sign.*

The rule "nodeX2nodeY" synchronously creates an "X" node in the source model and its corresponding "Y" node in the target model. Thus, in this case the left hand side of this rule is the empty triple graph, because all elements are created. The rule "directed2connection" creates directed links "D" between two "X" nodes in the source component and their corresponding connection "C" between the related "Y" nodes in the target component. Finally, the rule "undirected2connection" creates an undirected link "U" in the source component and relates it with two connections "C" for the communication in both directions between the "Y" nodes that are *already related to the "X" nodes in the source component.*

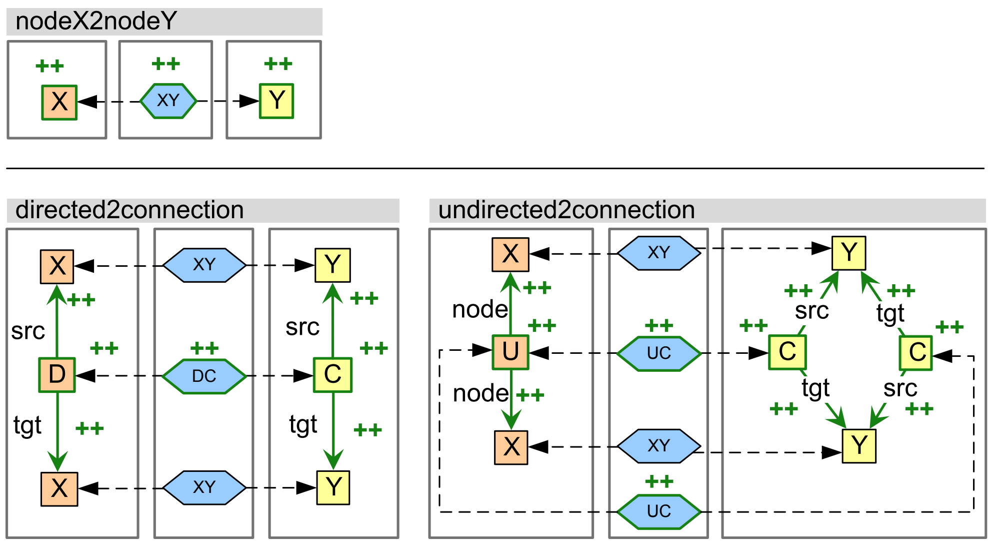


Figure 4: Triple Rules of the Triple Graph Grammar *TGG*

Based on the triple rules of a triple graph grammar the operational source and forward rules for model transformations from models of the source language to models of the target language are derived automatically [Sch94, KW07, EEE+07]. The source rules will be used to parse the given source model of a forward model transformation, which guides the forward transformation, in which the forward rules are applied. Since triple rules have a symmetric character, the backward rules for backward model transformations from models of the target to models of the source language can be derived as well.

Definition 3 (Derived Source and Forward Rule) Given a triple rule $tr = (tr^S, tr^C, tr^T) : L \to R$ the source rule $tr_S : L_S \to R_S$ is derived by extending the graph morphism $tr^S : L^S \to R^S$ with empty graphs and empty morphisms for the remaining correspondence and target components, i.e. $L_S^C = L_S^T = R_S^C = R_S^T = \emptyset$. The forward rule $tr_F = (tr_F^S, tr_F^C, tr_F^T)$ is derived by taking tr and redefining the following components: $L_F^S = R^S$, $tr_F^S = id$, and $s_{L_F} = tr^S \circ s_L$.

$$\begin{tikzcd}
L \arrow[d, "tr"'] & = (L^S \arrow[d, "tr^S"'] & L^C \arrow[l, "s_L"'] \arrow[r, "t_L"] \arrow[d, "tr^C"'] & L^T) \arrow[d, "tr^T"] \\
R & = (R^S & R^C \arrow[l, "s_R"] \arrow[r, "t_R"'] & R^T)
\end{tikzcd}$$

Triple Rule tr

$$\begin{tikzcd}
L_S \arrow[d, "tr_S"'] & = (L^S \arrow[d, "tr^S"'] & \emptyset \arrow[l] \arrow[r] \arrow[d] & \emptyset) \arrow[d] \\
R_S & = (R^S & \emptyset \arrow[l] \arrow[r] & \emptyset)
\end{tikzcd}$$

Source Rule tr_S

$$\begin{tikzcd}
L_F \arrow[d, "tr_F"'] & = (R^S \arrow[d, "id"'] & L^C \arrow[l, "tr_S \circ s_L"'] \arrow[r, "t_L"] \arrow[d, "tr^C"'] & L^T) \arrow[d, "tr^T"] \\
R_F & = (R^S & R^C \arrow[l, "s_R"] \arrow[r, "t_R"'] & R^T)
\end{tikzcd}$$

Forward Rule tr_F

Example 3 (Derived Rules) *The derived forward rules and one derived source rule of the triple graph grammar TGG in Fig. 4 are shown in Fig. 5. The source rule "nodeX2nodeY$_S$" creates a single "X" node and will be used to parse all nodes with label "X" in a given source model of a model transformation. Based on the found matches the corresponding forward rule*

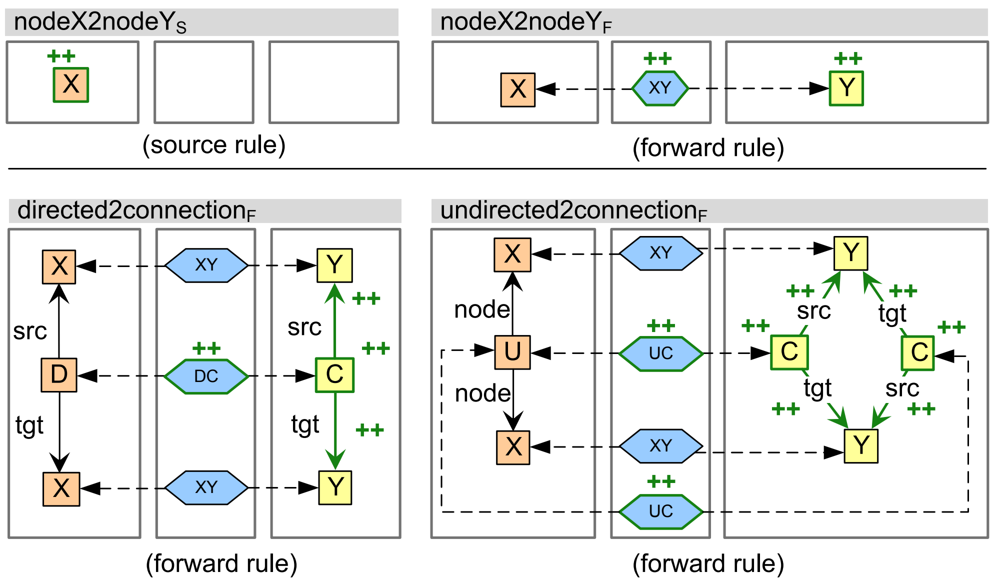


Figure 5: Some Derived Source and Forward Rules

"nodeX2nodeY$_F$" will be applied and it will insert a "Y" node in the target component for each found "X" node. Similarly, the other two forward rules specify the completion of the correspondence and target structure for communication links in the source component. Directed links "D" are transformed to directed connections "C" between the already translated and corresponding "Y" nodes. For undirected links "U" we have the case that two connections in both directions are created to complete the integrated model fragment.

As introduced in [EEE$^+$07, EHS09] model transformations can be defined based on source consistent forward transformations $G_0 \Rightarrow^* G_n$ via $(tr_{1,F}, \ldots, tr_{n,F})$, short $G_0 \xRightarrow{tr_F^*} G_n$. Source consistency intuitively means that the source model in G_0 can be parsed and all its elements are translated exactly once into corresponding fragments in the resulting target model. More precisely, source consistency of $G_0 \xRightarrow{tr_F^*} G_n$ requires that there is a source sequence $\varnothing \xRightarrow{tr_S^*} G_0$ such that the sequence $\varnothing \xRightarrow{tr_S^*} G_0 \xRightarrow{tr_F^*} G_n$ is match consistent, i.e. the S-component of each match $m_{i,F}$ of $tr_{i,F}(i = 1..n)$ is uniquely determined by the comatch $n_{i,S}$ of $tr_{i,S}$, where $tr_{i,S}$ and $tr_{i,F}$ are source and forward rules of the same triple rules tr_i. Altogether the forward sequence $G_0 \xRightarrow{tr_F^*} G_n$ is controlled by the corresponding source sequence $\varnothing \xRightarrow{tr_S^*} G_0$, which is unique in the case of match consistency.

Definition 4 (Model Transformation based on Forward Rules) A model transformation sequence $(G_S,\ G_0 \xRightarrow{tr_F^*} G_n, G_T)$ consists of a source graph G_S, a target graph G_T, and a source consistent forward TGT-sequence $G_0 \xRightarrow{tr_F^*} G_n$ with $G_S = proj_S(G_0)$ and $G_T = proj_T(G_n)$, where "$proj_X$" is the projection to the X-component of a triple graph for $X \in \{S,C,T\}$. A model transformation $MT : VL_{S0} \Rrightarrow VL_{T0}$ is defined by all model transformation sequences

$(G_S, G_0 \xRightarrow{tr_F^*} G_n, G_T)$ with $G_S \in VL_{S0}$ and $G_T \in VL_{T0}$.

Considering the source model in Fig. 3 we can construct the following source consistent forward transformation: with $G_S = G^S$: $(G^S \leftarrow \emptyset \rightarrow \emptyset) = G_0 \xRightarrow{nodeX2nodeY_F, m_1} G_1 \xRightarrow{nodeX2nodeY_F, m_2} G_2 \xRightarrow{directed2connection_F, m_3} G_3 \xRightarrow{undirected2connection_F, m_4} G_4 = (G^S \leftarrow G^C \rightarrow G^T)$ and we derive the integrated model $G = G_4$ and the target model $G_T = G^T$ as shown in Fig. 3.

4 Results for Model Transformations Based on Triple Graph Grammars

There are already many important results for model transformations based on triple graph transformation and in this section we compare the available results with respect to the listed challenges in Sec. 2.

Model transformations based on source consistent forward sequences are syntactically correct and complete with respect to the triple patterns [EEHP09], i.e. with respect to the language $VL = \{G \mid \emptyset \Rightarrow^* G \text{ in } TGG\}$ containing the integrated models generated by the triple rules. More precisely, each model transformation translates a source model into a target model, such that the integrated model that contains both models can be created by applications of the triple rules to the empty start graph. This means that both models can be synchronously created according to the triple patterns. Vice versa, a model transformation can be performed on each source model that is part of an integrated model in the generated triple language VL.

For the more formal view on these results we explicitly define the language of translatable source models VL_S and of reachable target models VL_T by $VL_S = \{G_S \mid (G_S \leftarrow G_C \rightarrow G_T) \in VL\}$ and $VL_T = \{G_T \mid (G_S \leftarrow G_C \rightarrow G_T) \in VL\}$. Based on these definitions there is the correctness and completeness result below according to Theorems 2 and 3 in [EHS09].

Theorem 1 (Syntactical Correctness) *Each model transformation sequence given by* $(G_S, G_0 \xRightarrow{tr_F^*} G_n, G_T)$*, which is based on a source consistent forward transformation sequence* $G_0 \xRightarrow{tr_F^*} G_n$ *with* $G_0 = (G_S \leftarrow \emptyset \rightarrow \emptyset)$ *and* $G_n = (G_S \leftarrow G_C \rightarrow G_T)$ *is syntactically correct, i.e.* $G_S \in VL_S$ *and* $G_T \in VL_T$*.*

Theorem 2 (Completeness) *For each* $G_S \in VL_S$ *there exists* $G_T \in VL_T$ *with a model transformation sequence* $(G_S, G_0 \xRightarrow{tr_F^*} G_n, G_T)$ *where* $G_0 \xRightarrow{tr_F^*} G_n$ *is source consistent with* $G_0 = (G_S \leftarrow \emptyset \rightarrow \emptyset)$ *and* $G_n = (G_S \leftarrow G_C \rightarrow G_T)$*.*

Concerning the non-functional properties of model transformations in the second list of challenges in Sec. 2 triple graph transformations show a very promising basis providing already most of the requested properties while the existing results above are preserved. In order to define expressive model transformations, the concept of negative application conditions (NACs) is commonly used and allows the modeler to specify complex model transformations [EHS09]. Furthermore, we have shown that information preserving bidirectional model transformations can be characterized by source consistent forward transformations based on triple graph grammars [EEE$^+$07]. Moreover, we improved the efficiency of the approach by defining an on-the-fly construction, for which termination is ensured if the source rules are creating, i.e. each triple rule creates at least one element in the source component. As a second optimization, we defined suitable conditions for parallel independence in order to perform partial order reductions [EEHP09]. Finally, model transformations based on triple graph transformations are flexible in the sense that new rules can be added without changing the existing rules whenever new structures are introduced into the visual language. For this reason, the efforts for maintainability can be kept low.

Coming back to the first list of challenges in Sec. 2 we prove in Sec. 5.2 the semantical correctness of the model transformation presented in this paper and we show how this approach can be generalized to other model transformations as well. Thus, there remains only one important challenge being the analysis of functional behaviour of model transformations. Functional behaviour of model transformations ensures unique results for any given source model. There are already several analysis techniques, which can be applied for plain graph grammars and we plan to lift them to the case with triple graphs. The techniques are based on the analysis of critical pairs in order to show confluence of the graph transformation system. The presented example in this paper shows already functional behaviour. However, for the backward direction the behaviour is not functional. Consider e.g. two “Y” nodes that are connected by two connections “C” in opposite direction. They can be transformed to one unidirectional link or to two directed links.

Summing up, triple graph transformation is an adequate and promising basis for model transformations and the existing results show already its intuitive, expressive, formally well founded and efficient character.

5 Verification of Model Transformations

5.1 The Borrowed Context Technique

In the following we will describe how to verify model transformations. Before we can even state what behaviour preservation actually means in our setting, it is necessary to introduce operational semantics, given by graph transformation rules, for both the source and the target model. These operational semantics will provide source as well as target models with labelled transition systems, where transitions correspond to the application of graph transformation rules and are of the form $G_1 \overset{\alpha}{\Rightarrow} G_2$. Note that α is the transition label, which is obtained from the applied production p via a given *map*-function, i.e., $\alpha = map(p)$. The *map*-function, assigning a global label to every rule, is necessary since we compare different operational rules. Now, behaviour preservation in our setting means that the source model and the corresponding target model are bisimilar (with respect to the labelled transitions).

We will use the borrowed context technique [EK06, RKE08], which refines a labelled transition system (or even unlabelled reaction rules) in such a way that the resulting bisimilarity is a congruence (see also [LM00]). By a congruence we mean a relation over graphs that is preserved by contextualization, i.e., by gluing with a given environment graph over a specified interface. This is a mild generalization of standard graph rewriting in that we consider “open” graphs, equipped with a suitable interface.

Note that in this section we will not work directly with triple graph grammars, although we have some preliminary ideas on the verification of model transformation based directly on triple graph grammars. Instead here we use in-situ transformation rules, where the in-situ rules are derived from the triple rules of Section 3, in order to be able to exploit the existing congruence results.

The basic idea behind the borrowed context technique is to describe the possible interactions of a part of the model with the environment, i.e., with the remaining yet unspecified rest of the model. In addition to existing labels, we add the following information to a transition: what is the (minimal) context that a graph with interface needs to evolve? More concretely we have transitions of the form

$$(J \rightarrow G) \overset{\alpha,(J\rightarrow F\leftarrow K)}{\Longrightarrow} (K \rightarrow H)$$

where the components have the following meaning: $(J \rightarrow G)$ is the original graph with interface J (given by an injective morphism from J to G) which evolves into a graph H with interface K. The label is composed of two entities: the original label $\alpha = map(p)$ stemming from the operational rule p and furthermore two injective morphisms $(J \rightarrow F \leftarrow K)$ detailing what is borrowed from the environment. The graph F represents the additional graph structure, whereas J, K are its inner and the outer interface.

We will now introduce the necessary definitions.

Definition 5 (context, cospan) A *graph with interface* is a graph morphism $J \rightarrow G$.

A *context* (also called *cospan*) consists of two injective graph morphisms $J \rightarrow F \leftarrow K$. The composition of two cospans is performed by taking the pushout.

$$\begin{tikzcd}
 & & K \arrow[dl] \arrow[dr] & & \\
J \arrow[r] \arrow[drr, dotted] & F \arrow[dr] & PO & E \arrow[dl] & M \arrow[l] \arrow[dll, dotted] \\
 & & D & &
\end{tikzcd}$$

Definition 6 (Rewriting with Borrowed Contexts) Given a graph with interface $J \rightarrow G$ and a production $p\colon L \leftarrow I \rightarrow R$, we say that $J \rightarrow G$ reduces to $K \rightarrow H$ with transition label $J \rightarrow F \leftarrow K$ 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* exists and is written as follows:

$$(J \rightarrow G) \overset{map(p),(J\rightarrow F\leftarrow K)}{\Longrightarrow} (K \rightarrow H)$$

(in words: $J \rightarrow G$ reduces to $K \rightarrow H$ with transition labels $map(p)$ and $J \rightarrow F \leftarrow K$).

$$\begin{tikzcd}
D \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & L \arrow[d] \arrow[dr, phantom, "PO"] & I \arrow[l] \arrow[r] \arrow[d] \arrow[dr, phantom, "PO"] & R \arrow[d] \\
G \arrow[r] & G^+ & C \arrow[l] \arrow[r] & H \\
J \arrow[r] \arrow[u] \arrow[ur, phantom, "PO"] & F \arrow[u] \arrow[ur, phantom, "PB"] & K \arrow[l] \arrow[u] \arrow[ur, dotted] &
\end{tikzcd}$$

After these preliminaries, we can now define the notion of bisimilation and bisimilarity with borrowed context labels. Note that under certain conditions and for closed systems this notion specializes to standard bisimilarity, which ignores the borrowed context label. This will be explained later in more detail.

Definition 7 (Bisimulation, Bisimilarity) Let $\mathscr{P}$ be a set of productions. Let $\mathscr{R}$ be a symmetric relation consisting of pairs of graphs with interfaces of the form $(J \to G, J \to G')$, also written $(J \to G)\mathscr{R}(J \to G')$.

The relation $\mathscr{R}$ is a *bisimulation* if whenever we have $(J \to G)\mathscr{R}(J \to G')$ and a transition $(J \to G) \overset{\alpha,(J \to F \leftarrow K)}{\Longrightarrow} (K \to H)$ can be derived from $\mathscr{P}$, then there exists a morphism $K \to H'$ and a transition $(J \to G') \overset{\alpha,(J \to F \leftarrow K)}{\Longrightarrow} (K \to H')$ such that $(K \to H)\mathscr{R}(K \to H')$.

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

We have shown that (strong) bisimilarity defined in transition systems with borrowed context labels is a congruence. This holds also if we enrich the labels with $\alpha = map(p)$ as described above.

Theorem 3 (Bisimilarity is a Congruence [EK06]) *Bisimilarity $\sim$ is a congruence, i.e., it is preserved by embedding into contexts as specified in Definition 5.*

5.2 Using the Borrowed Context Technique for the Verification of Model Transformations

For an in-situ model transformation within the same language, applications of the borrowed context technique are quite immediate: show for every transformation rule that the left-hand and right-hand sides L, R with interface I are bisimilar with respect to the operational rules. Then the source model must be bisimilar to the target model by the congruence result. This idea has been exploited in [RLK$^+$08] for showing behaviour preservation of refactorings.

To set up the entire machinery, we first need the operational semantics of the two languages under consideration (BiDiLang and UniDiLang). In Figures 6 and 7 we describe the dynamic evolution of a system: in both cases messages can be created and deleted at arbitary moments in time. Furthermore, in language BiDiLang the node labelled D describes a directed connection over which messages can be passed in only one direction, whereas the node labelled U describes an undirected connection allowing a movement in any direction (note that the two edges leaving the U-node have the same label and are hence undistinguishable). In the second language (UniDiLang) we have only one type of connection, working similarly to the directed connection in the first language.

Now, as announced above, in order to reuse the congruence result we are applying in-situ transformation rules (given in Figure 8) which are similar to the triple graph grammar rules given in Section 3.

Note that these in-situ rules will lead to “mixed” (or hybrid) models which incorporate components of both the source and the target model. Hence we need a joint type graph (see Figure 9) that contains node and edge types of both languages.

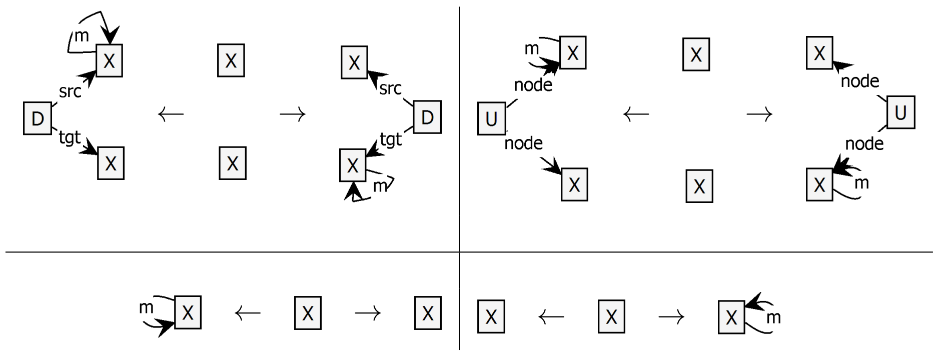


Figure 6: BiDiLang, rules of the operational semantics

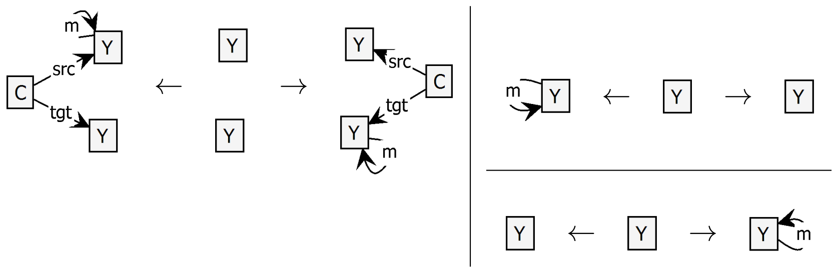


Figure 7: UniDiLang, rules of the operational semantics

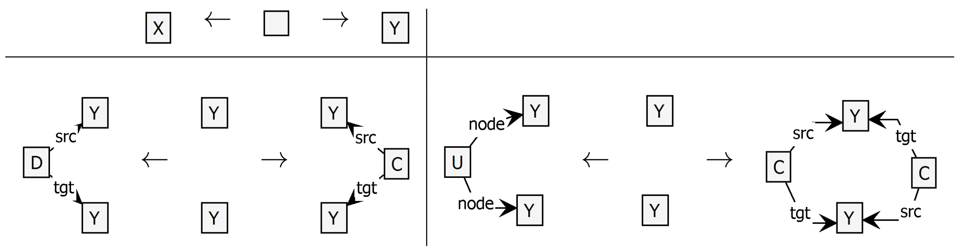


Figure 8: Rules for the in-situ model transformation

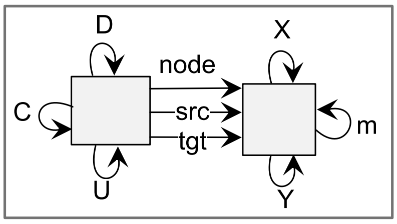


Figure 9: Combined Type Graph TG_{ST} for mixed Models

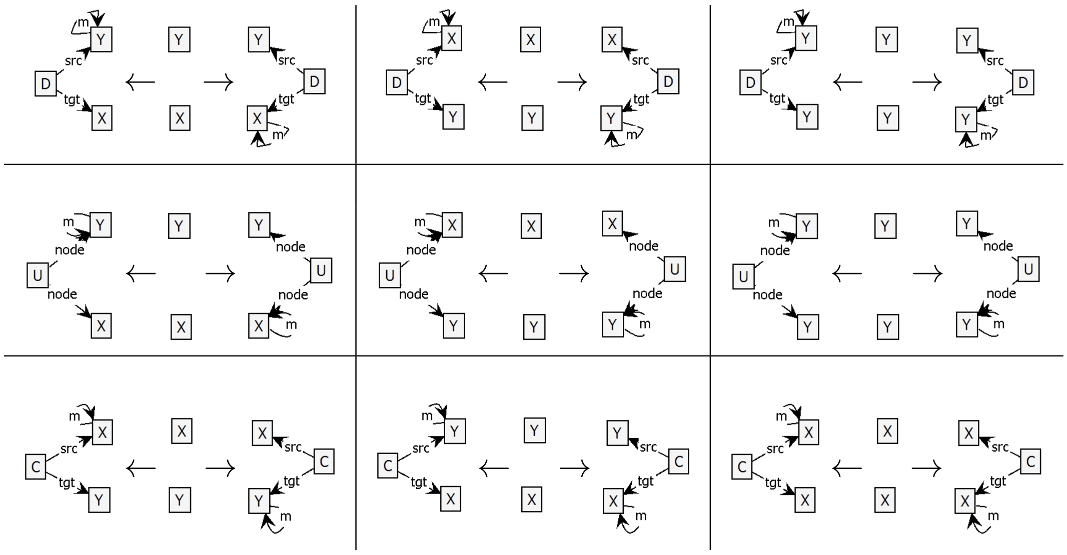


Figure 10: Additional rules of the mixed semantics

Now, since we generate mixed models but still want to exploit the congruence result, it is necessary to have an operational semantics also for those models, which has to satisfy the following conditions: (i) the mixed rules are *not* applicable to a pure source or target model; (ii) it is possible to show bisimilarity of left-hand and right-hand sides of all transformation rules. Finally, observe that our final aim is to show bisimilarity of closed graphs, i.e., of graphs with empty interface of the form $\emptyset \to G$. If the operational rules of the source and target languages have connected left-hand sides then such a graph will either borrow nothing or borrow the whole left-hand side. It can be shown that if all left-hand sides are connected, the notion of bisimilarity induced by borrowed contexts coincides with the standard one.

Hence here we use the mixed operational semantics given in Figure 10. The rules mainly describe message passing in mixed models, where a message is, for instance, passed from an X-node to a Y-node over various types of connectors.

Theorem 4 *The three rules of the in-situ model tranformation given in Figure 8 form a bisimulation relation $\mathscr{R}$, where each rule $L \leftarrow I \rightarrow R$ is split into a pair $(I \rightarrow L, I \rightarrow R)$ of the relation. Since bisimilarity is a congruence and borrowed context bisimilarity coincides with standard* *bisimilarity on source and target models, this implies that whenever a graph G_B of the source language is transformed into a graph G_U of the target language via the model transformation, then G_B is bisimilar to G_U.*

Note that in the proof we make heavy use of the up-to-context technique, which allows us to somewhat relax the requirements for bisimulation proofs given in 7. More specifically, it is enough if $K \rightarrow H$ and $K \rightarrow H'$ are in relation $\mathscr{R}$ after the removal of identical contexts. Note also that in more complex scenarios the bisimulation $\mathscr{R}$ might contain additional pairs that are not model transformation rules (see [HKR+09]).

In this fairly easy scenario one can obtain the rules of the mixed-semantics by applying the transformation rules to the (original) operational semantics of the source or target languages. In the general case, it is however currently not clear to us, how to obtain a correct set of mixed semantic rules. For small examples, the following heuristics usually gives good results:

1. Let S be the set containing all original rules of the the source and target operational semantics.
2. Choose any tranformation rule r.
3. Apply all rules in S to the left-hand side (respectively right-hand side) of r using the borrowed-context technique. This gives us several borrowed context rewriting steps.
4. If there is a matching answer with a rule in S for the right-hand side (respectively left-hand side) of transformation rule r, then do nothing.
5. If there is no such matching answer, create a new "mixed" rule, providing such a valid answer. Add this new rule to S and procede with step 2.
6. If every partial map of every rule in S has been tested for all left-hand and right-hand sides of the transformation rules, S is the mixed semantics we are looking for.

Using this heuristics one might even create a smaller set of rules for the mixed semantics in comparison to applying the transformation rules to the rules for the operational semantics in every possible way (see [HKR+09]).

6 Related Work

There are several other approaches based on triple graph transformation, e.g. using constraint-patterns [OGLE09]. While these patterns can lead to a more compact specification, there are fewer results for several of the listed challenges, e.g. the handling of termination and therefore completeness is more complex and not ensured in general.

As mentioned before, there are already suitable techniques for the analysis of functional behaviour of model transformations based on plain graph transformation systems [EEL+05]. However, plain graph transformation systems do not show some of the important benefits of triple graph transformation, as, for instance, completeness and the general notion of syntactical correctness with respect to the triple patterns specified by the intuitive triple rules. Furthermore, plain graph transformation systems are unidirectional while triple graph transformation systems automatically provide bidirectional model transformations.

The work closest to ours for showing the semantical correctness of model transformations in the sense of showing behaviour preservation for a transformation between models of different types is [GGL$^+$06]. They present a mechanised proof of semantics preservation for a transformation of automata to PLC-code, based on TGG rules. This proof faced some problems since it was not trivial to present graph transformation within Isabelle/HOL.

As opposed to model transformation between different source and target models, there has been more work on showing behaviour preservation in refactoring. The methods presented in [KCKB05, PC07, NK06, GSMD03] address behaviour preservation in model refactoring, but are in general limited to checking a certain number of models. The employment of a congruence result is also proposed in [BHE08] which uses the process algebra CSP as a semantic domain. A number of approaches to showing correctness of refactorings also focus on preserving specific aspects instead of the full semantics (see [MT04]).

7 Conclusion

In order to provide validated model transformations, which are a main component in model driven architecture (MDA), there is a strong need for formal analysis and verification. We have shown that triple graph transformation is an adequate technique providing both, an intuitive way of specification and a formal basis for which several analysis techniques as well promising execution algorithms are available. The two lists of challenges for model transformations in Sec. 2 contain many different and important aspects and depending on the concrete model transformation there may be some of them that cannot be achieved.

Even though, the presented approach in Sec. 3 based on triple graph transformation shows many capabilities and many of the listed challenges can be achieved or handled adequately, respectively. The available results discussed in Sec. 4 include for instance syntactical correctness and completeness and the specification of model transformations is performed in an intuitive and elegant way. While the general analysis of functional behaviour of a model transformation will be a part of future work we have exemplarily shown how the specified model transformation can be analyzed with respect to behaviour preservation and therefore, with respect to semantical correctness.

For this purpose we transformed in Section 5 the model transformation based on a triple graph grammar into an in-situ model transformation based on plain graph grammars. In a next step we introduced a proof technique for showing that a transformation preserves the behaviour of a model. A similar method was introduced by us in [HKR$^+$09] for a different model transformation. In [HKR$^+$09] it was even necessary to work with weak, saturated bisimilarity with negative application conditions due to the higher complexity of the case study. However, the general idea can just as well be presented and understood with the simpler case study presented in this paper.

Currently we have not yet mechanized the technique, but we have started to work on an implementation. One drawback is the fact that it is necessary to find a suitable mixed semantics, which might become quite large and unwieldy. Hence we are currently working on a more straightforward approach that combines triple graph grammars with borrowed contexts, by asking that each borrowed context step of the source model must be answered by a borrowed context step of the target model (and vice versa) in such a way that the labels can be translated into each other via the model transformation rules. However there are some remaining technical difficulties (e.g., what happens if the label can only be partially translated?) yet to be solved.

Note however that the in-situ transformation rules are not without merits: in the case of system migration, where we migrate piece by piece of an evolving system from one version to another, we might well have such mixed intermediate states which have to be handled. Think of a heterogeneous LAN, where one wants to replace the mail server, the firewall and the file server. The complete system must be in working order all the time, but in many cases the exchange of the components will not happen synchronously. In such a setting we want to show that also the hybrid models preserve the behaviour and the migration does not disrupt the correct working of the system.

Acknowledgements: We would like to thank Arend Rensink, Maria Semenyak, Christian Soltenborn and Heike Wehrheim for joint work on a case study, which gave us the ideas on which we based Section 5.

Bibliography

[BHE08] D. Bisztray, R. Heckel, H. Ehrig. Verification of Architectural Refactorings by Rule Extraction. In *FASE '08*. LNCS 4961, pp. 347–361. Springer, 2008.

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

[EEHP09] H. Ehrig, C. Ermel, F. Hermann, U. Prange. On-the-Fly Construction, Correctness and Completeness of Model Transformationsbased on Triple Graph Grammars: Long Version. In Schürr and Selic (eds.), *ACM/IEEE 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09)*. lncs 5795, pp. 241–255. Springer, 2009. To appear.
http://tfs.cs.tu-berlin.de/publikationen/Papers09/EEHP09.pdf

[EEL$^+$05] H. Ehrig, K. Ehrig, J. de Lara, G. Taentzer, D. Varró, S. Varró-Gyapay. Termination Criteria for Model Transformation. In Wermelinger and Margaria-Steffen (eds.), *Proc. Fundamental Approaches to Software Engineering (FASE)*. Lecture Notes in Computer Science 2984, pp. 214–228. Springer Verlag, 2005.
http://tfs.cs.tu-berlin.de/publikationen/Papers05/EEL+05.pdf

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

[EHS09] H. Ehrig, F. Hermann, C. Sartorius. Completeness and Correctness of Model Transformations based on Triple Graph Grammars with Negative Application Conditions. *ECEASST* 18, 2009.
http://eceasst.cs.tu-berlin.de/index.php/eceasst/issue/view/27

[EK06] H. Ehrig, B. König. Deriving Bisimulation Congruences in the DPO Approach to Graph Rewriting with Borrowed Contexts. *Mathematical Structures in Computer Science* 16(6):1133–1163, 2006.

[GGL$^+$06] H. Giese, S. Glesner, J. Leitner, W. Schäfer, R. Wagner. Towards Verified Model Transformations. In *3rd International Workshop on Model Development, Validation and Verification (MoDeVa)*. Pp. 78–93. Le Commissariat á l'Energie Atomique - CEA, Genova, Italy, 2006.

[GSMD03] P. V. Gorp, H. Stenten, T. Mens, S. Demeyer. Towards automating source-consistent UML refactorings. In *UML 2003*. LNCS 2863, pp. 144–158. Springer, 2003.

[HKR$^+$09] M. Hülsbusch, B. König, A. Rensink, M. Semenyak, C. Soltenborn, H. Wehrheim. Verifying Full Semantic Preservation of Model Transformation is Hard. Unpublished, October 2009.

[KCKB05] M. van Kempen, M. Chaudron, D. Kourie, A. Boake. Towards proving preservation of behaviour of refactoring of UML models. In *SAICSIT '05*. Pp. 252–259. 2005.

[KW07] E. Kindler, R. Wagner. Triple Graph Grammars: Concepts, Extensions, Implementations, and Application Scenarios. Technical report TR-ri-07-284, Universität Paderborn, 2007.

[LM00] J. J. Leifer, R. Milner. Deriving Bisimulation Congruences for Reactive Systems. In *Proc. of CONCUR 2000*. Pp. 243–258. Springer, 2000. LNCS 1877.

[MT04] T. Mens, T. Tourwé. A Survey of Software Refactoring. *IEEE Trans. Software Eng.* 30(2):126–139, 2004.

[NK06] A. Narayanan, G. Karsai. Towards Verifying Model Transformations. In *GT-VMT '06*. ENTCS 211, pp. 185–194. 2006.

[OGLE09] F. Orejas, E. Guerra, J. de Lara, H. Ehrig. Correctness, Completeness and Termination of Pattern-Based Model-to-Model Transformation. In Kurz et al. (eds.), *Proc. of the 3rd Int. Conf. on Algebra and Coalgebra in Computer Science (CALCO'09)*. Lecture Notes in Computer Science 5728, pp. 383–397. Springer, 2009.

[PC07] J. Pérez, Y. Crespo. Exploring a Method to Detect Behaviour-Preserving Evolution Using Graph Transformation. In *Third International ERCIM Workshop on Software Evolution*. Pp. 114–122. 2007.

[RKE08] G. Rangel, B. König, H. Ehrig. Deriving Bisimulation Congruences in the Presence of Negative Application Conditions. In Amadio (ed.), *Proc. Foundations of Software*

Science and Computational Structures (FOSSACS'08). Lecture Notes in Computer Science 4962, pp. 413–427. Springer Verlag, 2008.
doi:10.1007/978-3-540-78499-9
http://www.springerlink.com/content/e950520638346408/

[RLK+08] G. Rangel, L. Lambers, B. König, H. Ehrig, P. Baldan. Behavior Preservation in Model Refactoring using DPO Transformations with Borrowed Contexts. In *Proc. International Conference on Graph Transformation (ICGT'08)*. Lecture Notes in Computer Science 5214. Springer Verlag, Heidelberg, 2008.

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

[SK08] A. Schürr, F. Klar. 15 Years of Triple Graph Grammars. In *Proc. Int. Conf. on Graph Transformation (ICGT 2008)*. Pp. 411–425. 2008.
doi:10.1007/978-3-540-87405-8_28

Stepping from Graph Transformation Units to Model Transformation Units

Hans-Jörg Kreowski[1], **Sabine Kuske**[2], **Caroline von Totth**[3]*

[1] kreo@informatik.uni-bremen.de
[2] kuske@informatik.uni-bremen.de
[3] caro@informatik.uni-bremen.de
Department of Computer Science
University of Bremen, Germany

Abstract: Graph transformation units are rule-based entities that allow to transform source graphs into target graphs via sets of graph transformation rules according to a control condition. The graphs and rules are taken from an underlying graph transformation approach. Graph transformation units specify model transformations whenever the transformed graphs represent models. This paper is based on the observation that in general models are not always suitably represented as single graphs, but they may be specified as the composition of a variety of different formal structures such as sets, tuples, graphs, etc. which should be transformed by compositions of different types of rules and operations instead of single graph transformation rules. Consequently, in this paper, graph transformation units are generalized to model transformation units that allow to transform such kind of composed models in a rule-based and controlled way. Moreover, two compositions of model transformation units are presented.

Keywords: graph transformation, model transformation, transformation units, model transformation units

1 Introduction

Computers are devices that can be used to solve all kinds of data-processing problems – at least in principle. The problems to be solved come from economy, production, administration, science, education, entertainment, and many other areas. There is quite a gap between the problems as one has to face them in reality and the solutions one has to provide so that they run on a computer. Therefore, computerization is concerned with bridging this gap by transforming a problem into a solution. Many efforts in computer science contribute to this need of transformation. First of all, compilers are devices that transform programs in a high-level language into programs in a low-level language where the latter are nearer and more adapted to the computer than the former. The possibility and success of compilers have fed the dream of transforming descriptions of data-processing problems automatically or at least systematically into solutions that are given by

* The authors would like to acknowledge that their research is partially supported by the Collaborative Research Centre 637 (Autonomous Cooperating Logistic Processes – A Paradigm Shift and Its Limitations) funded by the German Research Foundation (DFG).

smoothly running programs. In recent years, the term model transformation has become popular for this idea.

In this paper, graph transformation units are generalized to model transformation units as rule-based devices to model model transformations in a compositional framework. Our approach has three sources of inspiration:

1. Following the ideas of model-driven architecture (MDA; cf., e.g., [Fra03]), the aim of model transformation is to transform platform-independent models (PIMs), which allow to describe problems adequately, into platform-specific models (PSMs), which run properly and smoothly on a computer. As a typical description of the PIMs, one may use UML diagrams while PSMs are often just programs in some common higher-level language like Java or C++. A significant model transformation language within the framework of MDA is the QVT standard of the OMG [OMG08].

2. One encounters quite an amazing number of model transformations in theoretical computer science – in formal language theory as well as in automata theory in particular. These areas provide a wealth of transformations between various types of grammars and automata like, for example, the transformation of nondeterministic finite automata into deterministic ones or of pushdown automata into context-free grammars (or the other way round) or of arbitrary Chomsky grammars into the Pentonen normal form (to give a less known example).

3. Graph transformation units (cf., e.g., [KKS97, KK99, KKR08]) are rule-based devices to model binary relations between initial and terminal graphs. If the initial graphs are interpreted as input models and the terminal graphs as output models, then such a unit embodies a model transformation. The transformation of UML sequence diagrams into UML collaboration diagrams in [CHK04] and the transformation of well-structured flow diagrams into *while*-programs in [KHK06] are examples of this kind. This observation supports the idea to use graph transformation units as building blocks for the modeling of model transformations.

While the models in the MDA context are often diagrammatic or textual, the examples of theoretical computer science show that models may also be tuples with components being sets of something. Accordingly, graphs as well as tuples, sequences, and sets of models are introduced as models in Section 3, while Section 2 provides the necessary mathematical preliminaries. The basic steps of model transformation are defined in Section 4 by actions that are applied componentwise to tuples of models and consist of rules in case of graph components and of data type operations in all other cases. Based on models and actions, the notion of a model transformation unit is introduced in Section 5 providing the descriptions of input, working and output models, a set of actions, and a control condition to regulate the use of actions. The semantics of such a unit is a transformation of input models into output models. In Section 6, the sequential and parallel compositions of model transformation units are studied. In this way, complex model transformations can be built up from simple ones in a modular way. While we discuss related work in Section 7, the paper ends with some concluding remarks. As a running example, the transformation of right-linear grammars into finite state automata is developed in several stages.

2 Preliminaries

In this section, we recall the notion of a graph rule base providing a class of graphs, a class of rules and a rule application operator. In the following sections graphs are used as basic visual models and rules are used for their elementary transformations. Besides graphs, we use identifiers, truth values, and non-negative integers as smallest atomic models. Moreover, cartesian products, free monoids, and powersets are recalled because these constructions will be used to build up composite models in the next section.

2.1 Graph Rule Bases

A *graph rule base* $B = (\mathscr{G}, \mathscr{R}, \Longrightarrow)$ consists of a class of graphs $\mathscr{G}$, a class of rules $\mathscr{R}$, and a rule application operator $\Longrightarrow$ with $\underset{r}{\Longrightarrow} \subseteq \mathscr{G} \times \mathscr{G}$ for every $r \in \mathscr{R}$. The rule application operator is used in infix notation, i.e, $(G,H) \in \underset{r}{\Longrightarrow}$ is denoted by $G \underset{r}{\Longrightarrow} H$.

2.2 Graph Classes

There are many different kinds of graph classes, two of which are explored here further: the class of directed edge-labelled graphs and the class of finite state graphs, the latter being a subclass of the former.

Directed edge-labelled graphs. The class of directed, edge-labelled graphs with individual, possibly multiple edges is defined as follows. Let Σ be a set of labels. A *graph* over Σ is a system $G = (V, E, s, t, l)$ where V is a set of *nodes*, E is a set of *edges*, $s, t \colon E \to V$ are mappings assigning a *source* $s(e)$ and a *target* $t(e)$ to every edge in E, and $l \colon E \to \Sigma$ is a mapping assigning a label to every edge in E. An edge e with $s(e) = t(e)$ is also called a *loop*. For a node $v \in V$ the number of edges which have v as source is denoted by $\mathit{outdegree}(v)$ and the number of edges that point to v is the *indegree* of v. An edge e with label x is called an x-pointer if $\mathit{indegree}(s(e)) = 0$ and $\mathit{outdegree}(s(e)) = 1$. The components V, E, s, t, and l of G are also denoted by V_G, E_G, s_G, t_G, and l_G, respectively. The set of all graphs over Σ is denoted by $\mathscr{G}_\Sigma$.

For graphs $G, H \in \mathscr{G}_\Sigma$, a *graph morphism* $g \colon G \to H$ is a pair of mappings $g_V \colon V_G \to V_H$ and $g_E \colon E_G \to E_H$ that are structure-preserving, i.e., $g_V(s_G(e)) = s_H(g_E(e))$, $g_V(t_G(e)) = t_H(g_E(e))$, and $l_H(g_E(e)) = l_G(e)$ for all $e \in E_G$.

If the mappings g_V and g_E are inclusions, then G is called a *subgraph* of H, denoted by $G \subseteq H$. For a graph morphism $g \colon G \to H$, the image of G in H is called a *match* of G in H, i.e., the match of G with respect to the morphism g is the subgraph $g(G) \subseteq H$.

Finite state graphs. One particular subclass of $\mathscr{G}_\Sigma$ are finite state graphs and finite state graphs with word transitions. More concretely, let I be some input alphabet such that $\mathrm{I}^* \uplus \{\mathit{start}, \mathit{final}\} \subseteq \Sigma$ [1]. Then the graph in Figure 1 represents a finite state graph with word transitions over $I = \{a, b, c\}$, where the edges labelled with $w \in I^*$ represent transitions, and the sources and targets of the transitions represent states. The start state is indicated with a *start*-pointer and every final

[1] Given sets X and $Y, X \uplus Y$ denotes the disjoint union of X and Y.

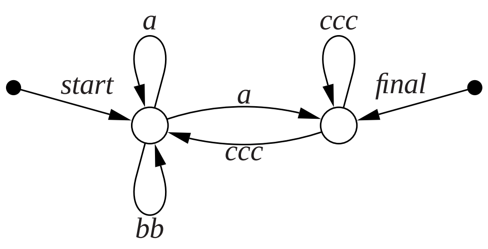


Figure 1: A finite state graph with word transitions

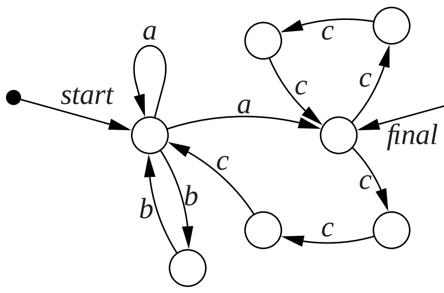


Figure 2: A finite state graph

state with a *final*-pointer. States are depicted as unfilled circles whereas all other nodes are shown as small filled circles. Figure 2 shows a finite state graph where each transition is labelled with a symbol from I.

2.3 Rules

To be able to transform graphs, rules are applied to the graphs yielding graphs again. One rule class that can be used to transform graphs in $\mathscr{G}_\Sigma$ is defined as follows. A *rule* $r = (L \supseteq K \subseteq R)$ consists of three graphs $L, K, R \in \mathscr{G}_\Sigma$ such that K is a subgraph of L and R. The components L, K, and R of r are called *left-hand side*, *gluing graph*, and *right-hand side*, respectively. A rule may be depicted as $L \rightarrow R$ if K is clear from the context (the numbered nodes form the common gluing graph).

An example of a rule is given in Figure 3. The left-hand side of this rule consists of two nodes v_1 and v_2 and an edge from v_1 to v_2 that is labelled with a word xyu from some alphabet I^* where

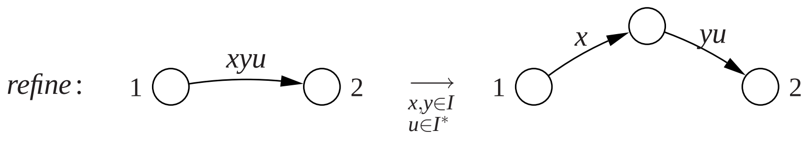


Figure 3: The graph transformation rule *refine*

x and y are symbols of I. The gluing graph consists of the two nodes v_1 and v_2; the right-hand side is obtained from the gluing graph by inserting a new node v_{new} and two new edges e_1 and e_2 where e_1 points from v_1 to v_{new} and is labelled with x, and e_2 points from v_{new} to v_2 and is labelled with yu.

2.4 Rule Application

The application of a graph transformation rule to a graph G consists of replacing a match of the left-hand side in G by the right-hand side in such a way that the match of the gluing graph is kept. Hence, the application of $r = (L \supseteq K \subseteq R)$ to a graph $G = (V, E, s, t, l)$ consists of the following three steps.

1. A match $g(L)$ of L in G is chosen.
2. Now the nodes of $g_V(V_L - V_K)$ are removed, and the edges of $g_E(E_L - E_K)$ as well as the edges incident to removed nodes are removed yielding the *intermediate graph* $Z \subseteq G$.
3. Afterwards the right-hand side R is added to Z by gluing Z with R in $g(K)$ yielding the graph $H = Z \uplus (R - K)$ with $V_H = V_Z \uplus (V_R - V_K)$ and $E_H = E_Z \uplus (E_R - E_K)$. The edges of Z keep their labels, sources, and targets so that $Z \subseteq H$. The edges of R keep their labels; they also keep their sources and targets provided that those belong to $V_R - V_K$. Otherwise, $s_H(e) = g(s_R(e))$ for $e \in E_R - E_K$ with $s_R(e) \in V_K$, and $t_H(e) = g(t_R(e))$ for $e \in E_R - E_K$ with $t_R(e) \in V_K$.

The application of a rule r to a graph G is denoted by $G \underset{r}{\Longrightarrow} H$, where H is the graph resulting from the application of r to G. A rule application is called a *direct derivation*.

If the rule *refine* in Figure 3 is applied to a finite state graph, it splits a word transition labelled with a word w of length at least two into two consecutive transitions, the first of which takes the first symbol of w, while the second one gets labelled with the remainder of w. In particular, if *refine* is applied as long as possible to the finite state graph in Figure 1, one gets the finite state graph in Figure 2.

2.5 Example for a Graph Rule Base

Directed edge-labelled graphs over Σ together with the rule class of 2.3 and the rule application operator given in 2.4 form a rule base which will be used in all examples of this paper.

2.6 Further Basic Types

In addition to graph rule bases, we assume a set of identifiers *ID*, the set of truth values BOOL $= \{TRUE, FALSE\}$, and the set of non-negative numbers $\mathbb{N}$. All these sets are equipped with the usual predicates and operations, i.e. the arithmetic operations like $+, -, \cdot, \leq, =$, etc. for $\mathbb{N}$, the Boolean operations like $\wedge, \vee, \neg, \rightarrow$, etc. for BOOL, and the equality predicate $=$ for *ID*.

All involved sets may be subject to the following three constructions that yield sets again:

1. the cartesian product $X_1 \times \cdots \times X_k$ for sets $X_1, \ldots, X_k, k \in \mathbb{N}$;
2. the free monoid X^* for a set X;
3. the powerset $set(X)$ for a set X that contains all subsets of X.

Furthermore we assume that the usual operations of these data types are available, like the projections in the case of the product, concatenation and other string-processing operations in the case of X^* and the usual operations and predicates on sets like $\cup, \cap, \in, \subseteq$, etc.

3 Models and Model Types

Many models used in computer science are of a graphical, diagrammatic, and visual nature, and they can be represented as graphs in an adequate way in most cases. Moreover, further types of elementary models such as numbers, values, or identifiers may be useful in addition to graphs. And models may not occur only as singular items, but also as tuples or as some other collections of models like sequences and sets. To cover this, we define models and their types in a recursive way.

Definition 1 (models and their types) Models together with their types are recursively defined as follows:

1. Let Y be a class of graphs $\mathcal{G}$, ID, BOOL, or $\mathbb{N}$. Then each $y \in Y$ is a *model of type* Y.
2. If m_i is a model of type T_i for $i = 1, \ldots, k$ for some $k \in \mathbb{N}$, then the k-tuple $(m_1, \ldots, m_k)$ is a *model of type* $T_1 \times \cdots \times T_k$.
3. If m_i is a model of type T for $i = 1, \ldots, k$ for some $k \in \mathbb{N}$, then the sequence $m_1 \cdots m_k$ is a *model of type* T^*.
4. If m is a set of models of type T, then m is a *model of type* $set(T)$.

Note that in this way every model gets a type which is a set of models, but can serve as a name on the syntactic level as well. To stress the semantic level we may write $\mathfrak{M}(T)$ for T.

Point 1 makes sure that all graphs and – in this way – all diagrams with graph representations are models. Besides graphs, truth values, numbers and identifiers become available as elementary models. Point 2 allows one to consider a $k-$tuple of models as a model and makes k models simultaneously available in this way. Point 3 and Point 4 also make many models of the same type available at the same time. While Point 3 provides them as a sequence, Point 4 collects them as a set.

The types of models as introduced above may be considered as free because they are based on the free constructions product, free monoid, and power set. But in many cases, it may not be reasonable to transform all models of a free type without any further restriction. For example, a Chomsky grammar $G = (N, T, P, S)$ is not just a quadruple of type $set(ID) \times set(ID) \times set(ID^* \times ID^*) \times ID$, but N and T should be finite and disjoint, S should be a nonterminal, and a pair $(u, v) \in P$ should consist of two strings of terminals and nonterminals rather than of arbitrary identifiers. To make such restrictions possible, we introduce constrained types.

Definition 2 (constrained model types) Let T be a model type.

1. Then $\mathscr{X}(T)$ is a class of *constraints* if each $x \in \mathscr{X}(T)$ specifies a set of models of type T, i.e. $SEM(x) \subseteq \mathfrak{M}(T)$.
2. For $x \in \mathscr{X}(T)$, $\langle T \ with \ x \rangle$ is called a *constrained model type*. The models of this type are the models of $SEM(x)$, denoted by $\mathfrak{M}(\langle T \ with \ x \rangle)$.

The definition is used in a recursive way considering the free model types and the constrained model types both as model types. Consequently, one can build types of the form $\langle\langle T \ with \ x\rangle \ with \ y\rangle$ with iterated constraints.

Examples for Constraints

1. For the model type $\mathscr{G}$, constraints x with $SEM(x) \subseteq \mathscr{G}$ are called graph class expressions in the framework of graph transformation units and are extensively used there to specify initial and terminal graphs. Examples of graph class expressions are the following.

   (a) Single graphs $Z \in \mathscr{G}$ with $SEM(Z) = \{Z\}$ are useful as start graphs of graph grammars.

   (b) For $\mathscr{G} = \mathscr{G}_\Sigma$ with $\Sigma \subseteq ID$, a subset $X \subseteq \Sigma$ describes $SEM(X) = \mathscr{G}_X$ which may serve as terminal labels.

   (c) For $\mathscr{G} = \mathscr{G}_\Sigma$ and $X \subseteq \Sigma$, the expression $pointers(X)$ specifies all graphs in $\mathscr{G}_\Sigma$ in which all edges labelled with some $x \in X$ are pointers (cf. 2.2).

   (d) For $\mathscr{G} = \mathscr{G}_\Sigma$ and $X \subseteq \Sigma$, the expression $one(X)$ specifies all graphs in which for each $x \in X$ there occurs exactly one x-labelled edge, i.e., $|\{e \in E_G \mid l_G(e) = x\}| = 1$ for each $x \in X$.

2. Logical formulas are further typical examples for constraints. They may involve model variables and the usual predicates and operations of the basic and free types:

   (a) Boolean operations in case of BOOL like $\neg, \wedge, \vee, \rightarrow$;

   (b) arithmetic operations and predicates on $\mathbb{N}$ like $+$, $\cdot$, mod, =, $\leq$;

   (c) string operations and predicates on X^* for some set X, like concatenation, transposition, equality;

   (d) set operations and predicates like $\cup$, $\cap$, $\uplus$, =, $\subseteq$, $\in$.

   Consider, for example, a model (x,y,X,Y,m,n,u,v,G,H) of type $ID \times ID \times set(ID) \times set(ID) \times \mathbb{N} \times \mathbb{N} \times ID^* \times ID^* \times \mathscr{G}_\Sigma \times \mathscr{G}_\Sigma$. Then one may add the following constraints: $x = y$, $x \in X$, $y \in Y$, $X \cap Y = \emptyset$, $m \leq n$, $length(u) \geq n$, $uv \neq vu$, $u = vtranspos(v)$, $G \subseteq H$, $G \in \mathscr{G}_X$. Clearly, all the constraints may be combined by Boolean operations.

3. Another frequently used constraint for graphs and sets is the requirement of finiteness indicated by the constant model class expression *finiteness*. Instead of $\langle \mathscr{G}_\Sigma \ with \ finiteness\rangle$ we may write $fin(\mathscr{G}_\Sigma)$, and $finset(ID)$ instead of $\langle set(ID) \ with \ finiteness\rangle$.

Examples for Constrained Model Types

1. Finite state automata with word transitions can be defined as a constrained model type, i.e. a *finite state automaton* $fsa = (I, G)$ is a pair of type $\langle set(ID) \times \mathscr{G}_\Sigma \textit{ with } (\Delta \subseteq \Sigma) \wedge (G \in (\Delta \cap pointers(\{start, final\}) \cap one(\{start\})))\rangle$ where $\Delta = I^* \uplus \{start, final\}$. The constraint means that every state graph G is labelled over $I^* \uplus \{start, final\}$, *final*- and *start*-edges are pointers, and there is exactly one *start*-pointer. In the following, the constrained model type of finite state automata with word transitions is denoted by FSA^*. The type of finite state automata the transitions of which are labelled only with single symbols from I, can be defined as the finite state automata in FSA^*, but where in the constraint I^* is replaced by I, i.e., $\Delta = I \uplus \{start, final\}$. The type of all finite state automata with single-symbol transitions is denoted by FSA.

2. Chomsky grammars can be introduced in the framework above as models nearly in the same way as they are defined in the literature.

   A *Chomsky grammar* $G = (N, T, P, S)$ is a quadruple of type $set(ID) \times set(ID) \times set(ID^* \times ID^*) \times ID$ *with* finite N and T, $N \cap T = \emptyset$, $S \in N$, and $(u, v) \in P$ implies $u, v \in (N \cup T)^*$, $u \notin T^*$. G is *right-linear* if, in addition, $(u, v) \in P$ implies $u \in N$ and $v \in (T^+N) \cup T^*$.

   More formally, the constraint of an arbitrary Chomsky grammar is *with* $N, T \in finset(ID) \wedge N \cap T = \emptyset \wedge S \in N \wedge ((u, v) \in P \rightarrow (u, v \in (N \cup T)^* \wedge u \notin T^*))$. And in case of right-linear grammars one must add $((u, v) \in P \rightarrow (u \in N \wedge v \in T^+N \cup \{\varepsilon\}))$ where ε denotes the empty string. The type of right-linear grammars will be denoted by RLG. For explicit use below we mention here also the type $RLG \times \mathscr{G}_\Sigma$ which will be used for transforming right-linear grammars into finite state automata.

4 Actions and Model Transformation Processes

In this section, the dynamic part of model transformations is introduced. The basic notion is that of an action that describes an elementary step of model transformations. Then the iteration of such steps provides more complex transformations.

Each model m can be identified with the 1-tuple (m) so that one may consider tuples of models only without loss of generality. Given such a tuple $(m_1, \ldots, m_k)$, an action is also a k-tuple $a = (a_1, \ldots, a_k)$ of component operations where, for $i = 1, \ldots, k$, a_i specifies how m_i is processed by the action. If m_i is a graph, then a_i is a rule to be applied to m_i. If m_i is an identifier or truth value, then a_i may replace it by another identifier or the negated truth value respectively. If m_i is a number, string or set, then a_i may operate on it yielding a modified number, string or set respectively. Moreover, we employ the void action $a_i = -$ meaning that m_i remains unchanged. If the component actions are performed, then a new tuple $(m'_1, \ldots, m'_k)$ of models is obtained. This is made precise in the following definition.

Definition 3 (actions) Let $T_1 \times \cdots \times T_k$ be a model type.

1. Then an *action* $a = (a_1, \ldots, a_k)$ is a k-tuple such that one of the following holds for $i = 1, \ldots, k$:

   (a) $a_i = -$,
   (b) $a_i \in \mathscr{R}$ provided that $T_i \subseteq \mathscr{G}$,
   (c) $a_i = \mathit{rename}$ provided that $T_i \subseteq \mathit{ID}$ where *rename* is some mapping on T_i,
   (d) $a_i = \neg$ provided that $T_i = \mathrm{BOOL}$ where $\neg$ is the negation with $\neg \mathit{TRUE} = \mathit{FALSE}$ and $\neg \mathit{FALSE} = \mathit{TRUE}$,
   (e) a_i is a term of operations with a distinguished variable of type $\mathbb{N}$ and which evaluates to $\mathbb{N}$ provided that $T_i = \mathbb{N}$.
   (f) the same as (e) replacing $\mathbb{N}$ by T^* and $\mathit{set}(T)$ for some type T,
   (g) recursively, a_i is an action provided that T_i is a product type with more than one component.

2. Let $m = (m_1, \ldots, m_k) \in \mathfrak{M}(T_1 \times \cdots \times T_k)$. Then the action $(a_1, \ldots, a_k)$ may be performed on m yielding $m' = (m'_1, \ldots, m'_k) \in \mathfrak{M}(T_1 \times \cdots \times T_k)$ denoted by $m \underset{a}{\Longrightarrow} m'$ if the following holds for $i = 1, \ldots, k$:

   (a) $m'_i = m_i$ if $a_i = -$;
   (b) $m_i \underset{a_i}{\Longrightarrow} m'_i$ if $a_i \in \mathscr{R}$;
   (c) $m'_i = a_i(m_i)$ if $T_i \in \{\mathrm{BOOL}, \mathbb{N}, T^*, \mathit{set}(T)\}$, or $T_i \subseteq \mathit{ID}$;
   (d) $m_i \underset{a_i}{\Longrightarrow} m'_i$ if a_i is an action.

3. Let A be a set of actions. Then a *model transformation process* is a sequence of performed actions $m = m_0 \underset{a_1}{\Longrightarrow} m_1 \underset{a_2}{\Longrightarrow} \cdots \underset{a_n}{\Longrightarrow} m_n = m'$ with the *action sequence* $a_1 \cdots a_n \in A^*$. Such a process may be denoted by $m \overset{n}{\underset{A}{\Longrightarrow}} m'$ or $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ if the omitted details do not matter. The set of model transformation processes over A is denoted by $\mathit{MTP}(A)$.

Examples for Actions

Let (N, T, P, S, G) be an arbitrary model of type $\mathit{RLG} \times \mathscr{G}_\Sigma$.

1. An action that removes a nonterminal symbol X from the first component of the right-linear grammar (N, T, P, S) and then inserts a state labelled with X in the graph component can be defined as $(\mathit{remove}(X), -, -, -, \mathit{node}(X))$, where $\mathit{remove}(X)$ removes X from N and $\mathit{node}(X)$ is the graph transformation rule depicted in Figure 4.

2. An action that removes a rule with a non-empty right-hand side from the right-linear grammar while inserting a corresponding transition in the graph that contains a state for every nonterminal of the rule can be defined as $(-, -, \mathit{remove}((X, uY)), -, \mathit{edge}(X, u, Y))$; the graph transformation rule $\mathit{edge}(X, u, Y)$ is given in Figure 5.

Model transformation processes are nondeterministic on two levels. On one hand, the rule applications in graph model components are nondeterministic whereas all the other component

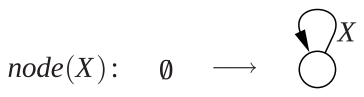


Figure 4: Graph transformation rule $node(X)$

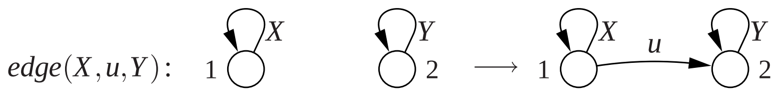


Figure 5: Graph transformation rule $edge(X,u,Y)$

actions are functional. On the other hand, the sequential composition of performed actions are only regulated by the requirement that a succesive action performs on the model yielded by the preceeding action, but there may be many actions that can process a current model. Sometimes, nondeterminism is desired, convenient, or unavoidable. But, in other cases, one would like to avoid nondeterminism or cut it down at least. This can be achieved by choosing rules and actions in such a way that only one or a few of them can be applied and performed. But the rules and actions may become quite complicated. Another possibility is extra regulation which can be provided by control conditions.

Definition 4 (control conditions) Let A be a set of actions. Then $\mathscr{C}$ is a class of *control conditions* if $SEM(c) \subseteq MTP(\mathscr{C})$ for every $c \in \mathscr{C}$.

Examples for Control Conditions

In the area of graph transformation, control conditions are frequently used for rules instead of actions. But many kinds of control conditions are easily carried over from rules to actions.

1. A typical kind of control conditions are regular expressions over A. Each regular expression r specifies a regular language $L(r)$. A model transformation process $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ belongs to $SEM(r)$ if and only if its action sequence belongs to $L(r)$. In the following, the operators concatenation, union, and Kleene star on languages will be denoted on the level of regular expressions as a semicolon, a plus and a star, respectively.

2. Another kind of control condition is a priority given by a partial reflexive and transitive relation $\leq$ on A where $a \geq a'$, but $a' \not\geq a$ means that a has higher priority than a'. A model transformation process belongs to $SEM(\leq)$ if and only if each performed action $m_{i-1} \underset{a_i}{\Longrightarrow} m_i$ has highest priority meaning that there is no $m_{i-1} \underset{a}{\Longrightarrow} \overline{m}$ with $a \geq a_i$ but $a_i \not\geq a$.

3. For any action a, the control condition $a!$ requires to apply a as long as possible. Hence, $m \overset{*}{\underset{A}{\Longrightarrow}} m'$ is in $SEM(a!)$ if the application sequence is in $\{a\}^*$ and there is no m'' such that $m' \underset{a}{\Longrightarrow} m''$. This condition can be combined with regular expressions in a straightforward way. For example, the expression $a_1!;a_2!$ requires to apply first a_1 as long as possible and then a_2 as long as possible.

5 Model Transformation Units

The previous sections provide all the ingredients that are needed to introduce model transformation units as devices to specify model transformations. Such a unit consists of the type of mdels to be transformed, of the actions to be performed, and of the control condition that regulates the transformation process. Moreover, the types of input and output models are specified, including their relation to the type of working models. The reasons to separate input, output and working models is that input and output may have different types and that it may be convenient to use further component models for intermediate processing.

Definition 5 (model transformation unit)

1. A *model transformation unit* is a system $mtu = (ITD, OTD, WT, A, C)$ where WT is a product type $T_1 \times \cdots \times T_l$ called *working type*, ITD is the *input type declaration* which consists of the constrained product type $\langle I_1 \times \cdots \times I_k \; with \; x\rangle$ and a mapping $initial\colon [k] \to set[l]$ such that $initial(i) \cap initial(j) = \emptyset$ for $i \neq j$ and $I_i = T_j$ for $i = 1, \ldots, k$ and $j \in initial(i)$, OTD is the *output type declaration* which consists of a constrained product type $\langle O_1 \times \cdots \times O_n \; with \; y\rangle$ and an injective mapping $terminal\colon [n] \to [l]$ with $O_i = T_{terminal(i)}$ for $i = 1, \ldots, n$, A is the *set of actions* with respect to the working type and C is the *control condition*. The type $I = \langle I_1 \times \cdots \times I_k \; with \; x\rangle$ is called *input type* of *mtu* and the mapping *initial initialization*. The type $O = \langle O_1 \times \cdots \times O_k \; with \; y\rangle$ is called *output type* of *mtu* and the mapping *terminal terminalization*.

2. The model transformation modeled by the model transformation unit *mtu* is a mapping $SEM(mtu)\colon \mathfrak{M}(\langle I_1 \times \cdots \times I_k \; with \; x\rangle) \to set(\mathfrak{M}(\langle O_1 \times \cdots \times O_n \; with \; y\rangle))$ which is defined as follows by $m' = (m'_1, \ldots, m'_n) \in SEM(mtu)(m_1, \ldots, m_k)$ for every $m = (m_1, \ldots, m_k) \in \mathfrak{M}(\langle I_1 \times \cdots \times I_k \; with \; x\rangle)$ if and only if the following holds:

   There are working models $\overline{m} = (\overline{m}_1, \ldots, \overline{m}_l)$, $\overline{m}' = (\overline{m}'_1, \ldots, \overline{m}'_l) \in \mathfrak{M}(T_1 \times \cdots \times T_l)$ such that

   (a) $\overline{m}_j = \begin{cases} m_i & \text{for} \quad i = 1, \ldots, k \text{ and } j \in initial(i) \\ init(T_j) & \text{for} \quad j \not\in initial([k]) = \bigcup\limits_{i \in [k]} initial(i) \end{cases},$

   (b) $\overline{m} \overset{*}{\underset{A}{\Longrightarrow}} \overline{m}' \in SEM(C)$,

   (c) $m'_i = \overline{m}'_j$ for $i = 1, \ldots, n$ and $terminal(i) = j$,

   (d) $m' \in SEM(y)$.

   The initial model $init(T_i)$ in (a) may be chosen in some appropriate way, like 0 for $T_i = \mathbb{N}$, the empty string ε for $T_i = T^*$, the empty set $\emptyset$ for $\mathrm{T}_i = set(T)$ or *FALSE* for $\mathrm{T}_i = \mathrm{BOOL}$.

In other words, an input model m of type $\langle I_1 \times \cdots \times I_k \; with \; x\rangle$ is first of all extended to a working model $\overline{m}$ of type $T_1 \times \cdots \times T_l$ by taking the components of m as components of $\overline{m}$ according to the mapping *initial* and by initializing the components of m that are not covered by *initial* by the initial models of the respective component types. Then $\overline{m}$ is transformed into

$\overline{m}'$ performing the given actions such that the model transformation process satisfies the control condition. Finally, the components of m' are selected as components of $\overline{m}'$ due to the mapping *terminal*. The type requirements of *terminal* together with point (d) make sure that the output model m' is of type $\langle O_1 \times \cdots \times O_n \textit{ with } y\rangle$.

In examples *initial* will be represented in the form $i \mapsto j_1, \ldots, j_i$ if $\mathit{initial}(i) = \{j_1, \ldots, j_i\}$ and *terminal* in the form $i \mapsto j$ for $\mathit{terminal}(i) = j$.

Remark 1 Given a model transformation unit *mtu* with input type $I = \langle I_1 \times \cdots \times I_k \textit{ with } x\rangle$ and output type $O = \langle O_1 \times \cdots \times O_n \textit{ with } y\rangle$, *mtu* can be graphically represented by

$$\begin{tikzcd} {} \arrow[r, "I"] & \boxed{\mathit{mtu}} \arrow[r, "O"] & {} \end{tikzcd}$$

emphasizing that *mtu* specifies a transformation of input model into output models.

Examples for Model Transformation Units

A model transformation unit that transforms right-linear Chomsky grammars into finite state automata is given in Figure 6. The components of this model transformation unit $\mathit{RLG2FSA}^*$ are the following:

$\mathit{RLG2FSA}^*$	
input:	RLG & $1 \mapsto 1, 2 \mapsto 2, 3 \mapsto 3, 4 \mapsto 4$
add:	$5\colon \mathscr{G}_\Sigma$ & $\mathit{init}(5) = \emptyset$ for $\Sigma = (N \cup T)^* \uplus \{\mathit{start}, \mathit{final}\}$
actions:	$a_1 = (\mathit{remove}(X), -, -, -, \mathit{node}(X))$ for $X \in N$
	$a_2 = (-, -, \mathit{remove}((X, \varepsilon)), -, \mathit{final}(X))$ for $X \in N$
	$a_3 = (-, -, \mathit{remove}((X, uY)), -, \mathit{edge}(X, uY))$ for $X, Y \in N, u \in I^+$
	$a_4 = (-, -, -, -, \mathit{start}(S))$
	$a_5 = (-, -, -, -, \mathit{remove\_loop}(X))$ for $X \in N$
cond:	$a_1!; a_2!; a_3!; a_4; a_5!$
output:	FSA^* & $1 \mapsto 2, 2 \mapsto 5$

Figure 6: The model transformation unit $\mathit{RLG2FSA}^*$ transforms right-linear Chomsky grammars (*RLG*) into finite state automata with word transitions (FSA^*)

- A model of the working type is a quintuple where the first four components of the working type correspond to the four types of a right-linear grammar; the last component is equal to $\mathscr{G}_\Sigma$ and serves to build up the finite state graph. It is initialized with the empty graph $\emptyset$. The alphabet Σ must equal $(N \cup T)^* \uplus \{\mathit{start}, \mathit{final}\}$ where N are the nonterminal symbols and T the terminal symbols of the input grammar, and *start* and *final* will serve to label the start and final states of the finite state graph respectively.

- The input type declaration is composed of the constrained model type for right-linear grammars and the initialization $initial : [4] \to set([5])$ with $initial(i) = \{i\}$ for $i = 1, \dots, 4$. This means that the four components of the input type are the first four components of the working type. Hence, the four components of every input model are used as the first four components in the model the model transformation unit starts working with.

- The output type declaration consists of the constrained model type FSA^* and the terminalization *terminal* with $terminal(1) = 2$ and $terminal(2) = 5$. Hence, every output model of the unit is the pair consisting of the second and the last component of the model the unit ends working with, provided that the type of this pair equals FSA^*.

- The set of actions of $RLG2FSA^*$ consists of five actions, each of which contains among other operations a graph transformation rule depicted in Figures 4, 5 and 7.

  1. The first action $a_1 = (remove(X), -, -, -, node(X))$ serves to generate a state in the graph for each nonterminal of the input grammar. More concretely, every application of this action generates a state with name X while removing the nonterminal X from the set of nonterminals.
  2. The second action $a_2 = (-, -, remove((X, \varepsilon)), -, final(X))$ inserts final pointers at all final states of the graph, while removing the corresponding rules from the grammar.
  3. The third action $a_3 = (-, -, remove((X, uY)), -, edge(X, u, Y))$ serves to generate transitions from those rules of the grammar that have a nonterminal in their right-hand side. Every application of a_3 removes such a rule from the rule set in the third component at the same time that a corresponding transition in the graph is generated.
  4. Action $a_4 = (-, -, -,, start(S))$ inserts the start pointer at the state S if S is the start symbol of the grammar.
  5. Finally, the last action $a_5 = (-, -, -, -, remove\_loop(X))$ for $X \in N$ serves to remove all state names in order to obtain a finite state graph.

- The control condition $a_1!; a_2!; a_3!; a_4; a_5!$ requires that at first all states be generated. This is achieved by applying a_1 as long as possible. The application of a_2 as long as possible inserts for every rule with the empty word as right-hand side a *final*-pointer while removing this rule. Then a_3 requires to insert a transition for every remaining rule. Then the start state is inserted by a_4 and afterwards all state names are removed by applying a_5 as long as possible.

If the input model of $RLG2FSA^*$ is the right-linear grammar $(\{S, A\}, \{a, b, c\}, P, S)$ with $P = \{(S, aSa), (S, aA), (S, bbS), (A, cccA), (A, \varepsilon)\}$, the ouput model is $(\{a, b, c\}, G)$ where G is the finite state graph with word transitions in Figure 1.

Finite state graphs with word transitions can be transformed into finite state graphs with symbol transitions by the model transformation unit FSA^*2FSA given in Figure 8.

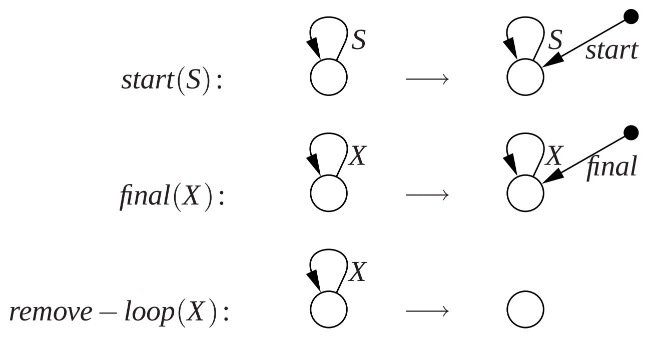


Figure 7: Graph transformation rules for the actions of model transformation unit *RLG2FSA**

FSA**2FSA*

input:	FSA^* & $1 \mapsto 1, 2 \mapsto 2$
actions:	$a = (-, refine)$
cond:	$a!$
output:	FSA & $1 \mapsto 2, 2 \mapsto 2$

Figure 8: The model transformation unit *FSA***2FSA* transforms finite state automata with word transitions (*FSA**) into finite state automata (*FSA*)

The input type declaration consists of the constrained model type *FSA** of finite state automata with word transitions and the initialization *initial* that maps the two components of every input model to the first two components of the working type. The working type of the unit is equal to $set(ID) \times \mathscr{G}_\Sigma$ (it is assumed that $\Sigma \subseteq ID$); the output type declaration consists of the model type *FSA* for finite state automata and the terminalization *terminal*, which is the identity in this case. The only action *a* applies the rule *refine* of Figure 3 to the graph component of the current model, while the control condition requires to apply the action *a* as long as possible. If the input model of *FSA***2FSA* is equal to the state automaton $(\{a,b,c\},G)$ where G is the finite state graph of Figure 1, the output is equal to $(\{a,b,c\},G')$ where G' is the finite state graph in Figure 2.

6 Sequential and Parallel Composition

Model transformation units can be used as building blocks for more complex model transformation constructions obtained by sequential and parallel composition. This leads to the notion of model transformation expressions on the syntactic level. Semantically, the sequential composition of model transformations is just the usual one of relations. And the parallel composition uses the fact that all models are considered as tuples of some product types so that the product of such types yields again models of some product type.

Definition 6 (compositional expressions)

1. The set $\mathscr{C}\mathscr{X}$ of compositional expressions is defined recursively:

   (a) model transformation units are in $\mathscr{C}\mathscr{X}$,

   (b) $cx_1, \dots, cx_k \in \mathscr{C}\mathscr{X}$ implies $cx_1; \dots; cx_k \in \mathscr{C}\mathscr{X}$ (sequential composition),

   (c) $cx_1, \dots, cx_k \in \mathscr{C}\mathscr{X}$ implies $cx_1 \parallel \dots \parallel cx_k \in \mathscr{C}\mathscr{X}$ (parallel composition).

2. The semantic relation of a compositional expression $cx \in \mathscr{C}\mathscr{X}$ is defined according to its syntactic structure:

   (a) If $cx = mtu$ for some model transformation unit, then $SEM(cx) = SEM(mtu)$.

   (b) If $cx_1; \dots; cx_k$ for some model transformation units cx_i with $i = 1, \dots, k$, then $SEM(cx_1; \dots; cx_k) = SEM(cx_1) \circ \dots \circ SEM(cx_k)$, and

   (c) $(m'_1, \dots, m'_k) \in SEM(cx_1 \parallel \dots \parallel cx_k)(m_1, \dots, m_k)$ if and only if $m'_i \in SEM(cx_i)$ for $i = 1, \dots, k$.

Examples

The sequential composition $RLG2FSA^*; FSA^*2FSA$ of the model transformation units in Section 5 transforms right-linear grammars into finite state automata so that the language generated by the input grammar is recognized by the automaton.

The formal language theory offers many examples of sequential compositions of model transformations like the transformation of right-linear grammars into finite state automata followed by their transformation into deterministic automata followed by the transformation of the latter into minimal automata.

A typical example of a parallel composition is given by the acception processes of two finite state automata that run simultaneously. If they try to accept the same input strings, this parallel composition simulates the product automaton that accepts the intersection of the two accepted regular languages.

To make the definition of compositional expressions more transparent, one may assign an input type and an output type to each compositional expression. Then the relational semantics of an expression turns out to be a relation between input and output types.

Definition 7 (input and output types) The input type in and the output type out of a compositional expression $cx \in \mathscr{C}\mathscr{X}$ is recursively defined.

1. If $cx = mtu$ for some model transformation unit with input type I and output type O, then $\mathtt{in}(mtu) = I$, $\mathtt{out}(mtu) = O$,

2. If $cx = cx_1; \dots; cx_k$ for some model transformation units cx_i with $i = 1, \dots, k$, then $\mathtt{in}(cx_1; \dots; cx_k) = \mathtt{in}(cx_1)$ and $\mathtt{out}(cx_1; \dots; cx_k) = \mathtt{out}(cx_k)$,

3. If $cx = cx_1 \parallel \ldots \parallel cx_k$ for some model transformation units cx_i with $i = 1, \ldots, k$, then $\mathsf{in}(cx_1 \parallel \ldots \parallel cx_k) = \mathsf{in}(cx_1) \parallel \ldots \parallel \mathsf{in}(cx_k)$ and $\mathsf{out}(cx_1 \parallel \ldots \parallel cx_k) = \mathsf{out}(cx_1) \parallel \ldots \parallel \mathsf{out}(cx_k)$, where the parallel composition of model types is defined as follows

   (a) $(T \parallel T') = (T \times T')$ provided that T and T' are free,

   (b) $(\langle T \; with \; x \rangle) \parallel (\langle T' \; with \; x' \rangle) = T \parallel \langle T' \; with \; x \wedge x' \rangle$,

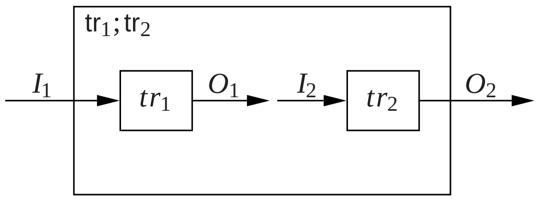
   (c) $T \parallel (\langle T' \; with \; x' \rangle) = \langle (T \parallel T') \; with \; x' \rangle$ provided that T is free, and

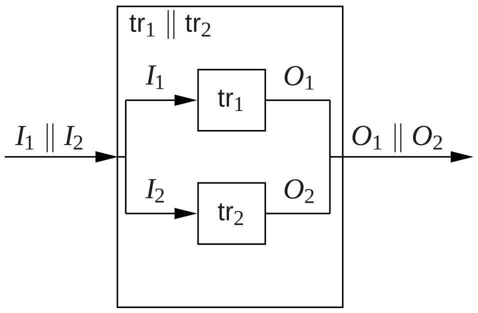
   (d) $(\langle T \; with \; x \rangle) \parallel T' = \langle (T \parallel T') \; with \; x \rangle$ provided that T' is free.

Due to these definitions, it is easy to see that compositional expressions describe transformations from input models to output models.

Observation: $SEM(cx)(m) \in set(\mathfrak{M}(\mathsf{out}(cx)))$ for all $m \in \mathfrak{M}(\mathsf{in}(cx))$.

The compositions can be quite intuitively depicted:





The sequential and parallel compositions on the level of model transformation expressions have the disadvantage that their results cannot be subject to further constraints. This is particularly problematic with respect to the parallel composition because the composed units run in parallel, but without any interaction. This is quite all right provided that the components are meant to run independently of each other. But in many cases of parallel composition one intends that the components exchange information or process some data simultaneously. Such interrelations and interactions could be achieved by adding further constraints. This requires either to extend the notion of constraints to the level of model transformation expressions or to flatten such expressions into model transformation units. The latter is done in the following.

6.1 Sequential Composition

Let $mtu_1 = (ITD_i, OTD_i, WT_i, A_i, C_i)$ for $i = 1,2$ be two model transformation units with input types $I_i = \langle I_{i,1} \times \cdots \times I_{i,k_i} \; with \; x_i \rangle$ and output types $O_i = \langle O_{i,1} \times \cdots \times I_{i,n_i} \; with \; y_i \rangle$. By definition of the semantics of the sequential composition $mtu_1;mtu_2$, the following holds: $m'' = (m''_1, \ldots, m''_{n_2}) \in SEM(mtu_1;mtu_2)(m)$ for $m = (m_1, \ldots, m_{k_1}) \in \mathfrak{M}(I_1)$ if and only if there is an m' with $m' \in SEM(mtu_1)(m)$ and $m'' \in SEM(mtu_2)(m')$. This means in particular that $m' \in \mathfrak{M}(O_1) \cap \mathfrak{M}(I_2)$ and therefore $n_1 = k_2$. To avoid too much technical trouble, we assume in addition that $WT = WT_1 = O_1 \times \cdots \times O_{n_1} = I_1 \times \cdots \times I_{k_2} = WT_2$. Then the sequential composition of mtu_1 and mtu_2 can be simulated by the model transformation unit

$$mtu(mtu_1;mtu_2) = (ITD_1, OTD_2, WT, A_1 \cup A_2, C(C_1, C_2, y_1, x_2, A_1, A_2))$$

where the control condition is chosen in such a way that a model transformation process $\overline{m} \overset{*}{\underset{A_1 \cup A_2}{\Longrightarrow}} \overline{m}''$ is accepted if and only if it decomposes into $\overline{m} \overset{*}{\underset{A_1}{\Longrightarrow}} \overline{m}' \overset{*}{\underset{A_2}{\Longrightarrow}} \overline{m}''$ with the following properties:

1. $\overline{m} \overset{*}{\underset{A_1}{\Longrightarrow}} \overline{m}'$ is accepted by C_1,
2. $\overline{m}' \in SEM(y_1) \cap SEM(x_1)$,
3. $\overline{m}' \overset{*}{\underset{A_2}{\Longrightarrow}} \overline{m}''$ is accepted by C_2.

Such a control condition may have the form of a transition system:

$$\begin{tikzcd}
{} \arrow[r] & s_0 \arrow[r, bend left, "{A_1^*, C_1}"] & s_1 \arrow[r, bend left, "{-, y_1 \wedge x_1}"] & s_2 \arrow[r, bend left, "{A_2^*, C_2}"] & s_3 \arrow[r] & {}
\end{tikzcd}$$

requiring that at the beginning the actions of A_1 are iterated regarding C_1, that the result must obey y_1 and x_2 and that finally actions of A_2 are iterated regarding C_2.

It is not difficult to show that the following correctness result holds.

Observation: $SEM(mtu_1;mtu_2) = SEM(mtu(mtu_1;mtu_2))$.

6.2 Parallel Composition

Let $mtu_i = (ITD_i, OTD_i, WT_i, A_i, C_i)$ for $i = 1,2$ be two model transformation units each with input type $I_i = \langle I_{i,1} \times \cdots \times I_{i,k_i} \; with \; x_i \rangle$ and initialization $initial_i : [k_i] \longrightarrow set[l_i]$ as well as output type $O_i = \langle O_{i,1} \times \cdots \times O_{i,n_i} \; with \; y_i \rangle$ and terminalization $terminal : [n_i] \longrightarrow [l_i]$. Then the parallel composition of mtu_1 and mtu_2 can be simulated by the model transformation unit

$$mtu(mtu_1 \parallel mtu_2) = (ITD, OTD, WT_1 \times WT_2, A, C)$$

where

- *ITD* consists of the input type $I_1 \times I_2$ and the initialization $initial_i : [k_1+k_2] \longrightarrow set[l_1+l_2]$ with $initial(i) = initial_1(i)$ for $i \in [k_1]$ and $initial(i) = l_1 + initial_2(i-k_1)$ for $i = k_1 + 1, \ldots, k_1+k_2$,
- *OTD* consists of the output type $O_1 \times O_2$ and the terminalization $terminal\colon [n_1+n_2] \longrightarrow [l_1+l_2]$ with $terminal(i) = terminal_1(i)$ for $i \in [n_1]$ and $terminal(i) = l_1 + terminal_2(i-n_1)$ for $i = n_1+1, \ldots, n_1+n_2$,
- $A = A_1{}' \times A_2{}'$ with $A_1{}' = A_1 \cup \{-\}^{l_1}$ and $A_2{}' = A_2 \cup \{-\}^{l_2}$, and
- the control condition C is chosen in such a way that a model transformation process $(\overline{m}_1, \overline{m}_2) \overset{*}{\underset{A}{\Longrightarrow}} (\overline{m}_1{}', \overline{m}_2{}')$ is accepted if and only if it decomposes into $\overline{m}_1 \overset{*}{\underset{A_1{}'}{\Longrightarrow}} \overline{m}_1{}'$ and $\overline{m}_2 \overset{*}{\underset{A_2{}'}{\Longrightarrow}} \overline{m}_2{}'$ so that the former is accepted by C_1 and the latter by C_2 after removal of the void steps given by the performance of the void actions $(-, \ldots, -)$.

The construction relies on the cartesian product of types and actions. Because the working type components 1 to l_2 become the components l_1+1 to l_1+l_2, the initialization and terminalization must be adapted accordingly. The actions of mtu_1 and mtu_2 are extended by the void action $(-, \ldots, -)$ with l_1 and l_2 components respectively. This is necessary because the actions of mtu_1 and mtu_2 may run in parallel, but the model transformation processes are of different lengths in general so that they cannot run fully simultaneously.

It is again not difficult to show the following correctness result.

Observation: $SEM(mtu_1 \parallel mtu_2) = SEM(mtu(mtu_1 \parallel mtu_2))$.

7 Related Work

In this section we briefly describe a selection of related work concerning model transformation. Since there exists quite an amount of publications we restrict ourselves to papers that are concerned with model transformations in the context of graph transformation. Moreover, we also mention some work that is concerned with the composition of model transformation definitions.

Model transformations based on graph transformation. One approach to define model transformations is by triple grammars [Sch94, KS06, SK08]. Each rule of a triple grammar can be easily transformed into a forward rule, a source rule, and a backward rule. The source rules are used to generate source models that – represented as graph triples – have the form $(S, \emptyset, \emptyset)$ where S represents the source model. The forward rules are used to produce target models from source models. These target models – represented as graph triples – have the form (S, C, T). Each application of a forward rule determines a subgraph of S of the rule application. The backward rules are used to transform a target model $(\emptyset, \emptyset, T)$ to a source model (S, C, T). In [EEE$^+$07] it is shown that any source consistent model transformation based on triple grammars is backward information preserving. This means that the target model (generated by the forward rules of the grammar) can be transformed into the source model via the backward rules of the grammar. Roughly spoken, a model transformation MT is source consistent if there is a transformation that generates the source model from $(\emptyset, \emptyset, \emptyset)$ and that completely determines the matches in the source model of the forward rules applied in MT. In [EE08] models are graphs equipped with a semantics given as a set of simulation rules, and a model transformation is composed of generating first an integrated model by graph transformation rules and restricting it then to the target model. It is shown under which conditions semantical correctness and completeness of model transformations are achieved. In [Küs06] an approach to model transformation is presented that uses transformation units based on typed attributed graph transformation. It provides criteria for syntactic correctness as well as for termination and confluence.

Examples of model transformation tools based on graph transformation are VIATRA2 [VB07], GReAT [BNvBK06] and ATOM^3 [dLVA04]. VIATRA2 integrates graph transformation and abstract state machines. Basically, model transformation steps are captured by graph transformation rules whereas abstract state machines control the order of rule application. GReAT mainly consists of a pattern specification language, a transformation rule language and a control flow language. The graph transformation rules of GReAT include for example input and output interfaces where the former can receive graph objects from previous rules and the latter can send graph objects to another rule. ATOM^3 focuses on modeling complex systems composed of various formalisms and allows to transform them into a single common formalism based on graph transformation. In [dLT04], ATOM^3 is combined with AGG for validation purposes.

In general, the mentioned publications on model transformation with graph transformation are very close to our approach – they are however restricted to transform mainly graphs, not tuples of graphs, sets or sequences as proposed in this paper.

Composition of model transformations. In the literature one can find two main types of composition techniques for model transformation definitions: external and internal composition. The first one chains model transformations sequentially whereas the second composes the rules of a set of model transformation definitions into one transformation definition. In this sense the compositions presented in Subsections 6.1 and 6.2 can be considered as internal compositions.

In [Wag08] the composition of model transformation definitions via superimposition is described, which is a feature of the ATLAS Transformation Language [JK05]. Superimposition of modules is an internal composition technique where models can be superimposed on top of each other yielding a module that contains the union of all transformation rules. In [YCWD09] the authors consider composition of model transformation definitions that transform high-level models into low-level models by defining a correspondence model that specifies the relations between the high-level meta models. The low-level correspondence model is automatically generated so that the low-level models can be composed homogeneously. In this way, new concerns can be added to existing model transformation definitions. In [CM08] two approaches for reusing model transformation definitions are proposed. The first one is called factorization and it allows to extract common parts of model transformation definitions obtaining in this way a base transformation definition which can be reused. The second concerns composition of transformation definitions which have compatible source metamodels but different target metamodels. Metamodels are related via small new metamodels and the transformations are integrated via an integration transformation definition that locates and connects the join points (without knowing the rules but some kind of trace information) by using so-called refinement rules. One approach towards composition of model transformations based on graph transformation is studied in [BHE09] where models are typed graphs that are mapped to semantic domains. The authors define spatial compositionality of semantic mappings which roughly spoken means that the semantics of a model is equal to the semantics that is obtained by embedding the semantics of a piece of the model into some context. It is assumed that the semantic mappings are graph transformation systems with a functional behavior and it is shown under which conditions they behave compositionally. In [KKS07] a first approach towards structured model transformation is proposed that allows package import, package merge and generalization according to a standardized packaging concept of the UML. In particular, the authors extend triple graph grammars by the mentioned concepts.

8 Conclusion

In this paper, we have introduced the notion of model transformation units as a generalization of graph transformation units. Models are tuples of graphs and other data structures like strings, sets, numbers, etc. Models of this kind cover graphical models like UML diagrams as well as set-theoretic models like grammars and automata. They are transformed componentwise by rule applications in the cases of graphs and by applications of data type operations in the other cases. Besides a set of such actions, a model transformation unit provides descriptions of input, output, and working models as well as a control condition to regulate the use of actions. Semantically, a transformation of input models into output models is specified. Moreover, we have studied sequential and parallel compositions of model transformation units as means to build up complex transformations from simple ones.

Although the considerations in this paper seem to be promising, more work is needed to underpin the significance of this novel approach, including the following points.

1. As pointed out in Section 4, the introduced kind of model transformation is nondeterministic. Therefore, sufficient conditions are of interest that guarantee the completeness of model transformations on one hand and their functionality on the other hand if these properties are desired.

2. Concerning our running example, it is known from the literature that a right-linear grammar generates the same language as is recognized by the finite state automaton resulting from the transformation. One intention of our approach is to support such correctness proofs. Therefore, notions of correctness and an appropriate proof theory must be studied in the future.

3. An interesting question in this respect is whether and how these correctness notions are compatible with the sequential and parallel compositions so that the correctness of the components yields the correctness of the composed model transformation.

4. Further explicit and detailed examples are needed to illustrate all introduced concepts more convincingly.

Bibliography

[BHE09] Dénes Bisztray, Reiko Heckel, and Hartmut Ehrig. Compositionality of model transformations. *Electr. Notes Theor. Comput. Sci.*, 236:5–19, 2009.

[BNvBK06] Daniel Balasubramanian, Anantha Narayanan, Christopher P. van Buskirk, and Gabor Karsai. The graph rewriting and transformation language: GReAT. *ECEASST*, 1, 2006.

[CHK04] Björn Cordes, Karsten Hölscher, and Hans-Jörg Kreowski. UML interaction diagrams: Correct translation of sequence diagrams into collaboration diagrams. In John l. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE 2003)*, volume 3062 of *Lecture Notes in Computer Science*, pages 275–291, 2004.

[CM08] Jesús Sánchez Cuadrado and Jesús García Molina. Approaches for model transformation reuse: Factorization and composition. In Vallecillo et al. [VGP08], pages 168–182.

[dLT04] Juan de Lara and Gabriele Taentzer. Automated model transformation and its validation using AToM^3 and AGG. In Alan F. Blackwell, Kim Marriott, and Atsushi Shimojima, editors, *Diagrammatic Representation and Inference*, volume 2980 of *Lecture Notes in Computer Science*, pages 182–198. Springer, 2004.

[dLVA04] Juan de Lara, Hans Vangheluwe, and Manuel Alfonseca. Meta-modelling and graph grammars for multi-paradigm modelling in AToM^3. *Software and System Modeling*, 3(3):194–209, 2004.

[EE08] Hartmut Ehrig and Claudia Ermel. Semantical correctness and completeness of model transformations using graph and rule transformation. In Ehrig et al. [EHRT08], pages 194–210.

[EEE$^+$07] Hartmut Ehrig, Karsten Ehrig, Claudia Ermel, Frank Hermann, and Gabriele Taentzer. Information preserving bidirectional model transformations. In Matthew B. Dwyer and Antónia Lopes, editors, *FASE*, volume 4422 of *Lecture Notes in Computer Science*, pages 72–86. Springer, 2007.

[EHRT08] Hartmut Ehrig, Reiko Heckel, Grzegorz Rozenberg, and Gabriele Taentzer, editors. *Graph Transformations, 4th International Conference, ICGT 2008, Leicester, United Kingdom, September 7-13, 2008. Proceedings*, volume 5214 of *Lecture Notes in Computer Science*. Springer, 2008.

[Fra03] David S. Frankel. *Model Driven Architecture. Applying MDA to Enterprise Computing*. Wiley, Indianapolis, Indiana, 2003.

[JK05] Frédéric Jouault and Ivan Kurtev. Transforming models with ATL. In Jean-Michel Bruel, editor, *Satellite Events at the MoDELS 2005 Conference*, volume 3844 of *Lecture Notes in Computer Science*, pages 128–138. Springer, 2005.

[KHK06] Hans-Jörg Kreowski, Karsten Hölscher, and Peter Knirsch. Semantics of visual models in a rule-based setting. In R. Heckel, editor, *Proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)*, volume 148 of *Electronic Notes in Theoretical Computer Science*, pages 75–88. Elsevier Science, 2006.

[KK99] Hans-Jörg Kreowski and Sabine Kuske. Graph transformation units with interleaving semantics. *Formal Aspects of Computing*, 11(6):690–723, 1999.

[KKR08] Hans-Jörg Kreowski, Sabine Kuske, and Grzegorz Rozenberg. Graph transformation units – an overview. In P. Degano, R. De Nicola, and J. Meseguer, editors, *Concurrency, Graphs and Models*, volume 5065 of *Lecture Notes in Computer Science*, pages 57–75. Springer, 2008.

[KKS97] Hans-Jörg Kreowski, Sabine Kuske, and Andy Schürr. Nested graph transformation units. *International Journal on Software Engineering and Knowledge Engineering*, 7(4):479–502, 1997.

[KKS07] Felix Klar, Alexander Königs, and Andy Schürr. Model transformation in the large. In Ivica Crnkovic and Antonia Bertolino, editors, *ESEC/SIGSOFT FSE*, pages 285–294. ACM, 2007.

[KS06] Alexander Königs and Andy Schürr. Tool integration with triple graph grammars - a survey. *Electr. Notes Theor. Comput. Sci.*, 148(1):113–150, 2006.

[Küs06] Jochen Malte Küster. Definition and validation of model transformations. *Software and System Modeling*, 5(3):233–259, 2006.

[OMG08] OMG. Meta object facility (MOF) 2.0 query/view/transformation (QVT). http://www.omg.org/spec/QVT/, 2008.

[Sch94] Andy Schürr. Specification of graph translators with triple graph grammars. In Ernst W. Mayr, Gunther Schmidt, and Gottfried Tinhofer, editors, *Graph-Theoretic Concepts in Computer Science*, volume 903 of *Lecture Notes in Computer Science*, pages 151–163. Springer, 1994.

[SK08] Andy Schürr and Felix Klar. 15 years of triple graph grammars. In Ehrig et al. [EHRT08], pages 411–425.

[VB07] Dániel Varró and András Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming*, 68(3):187–207, 2007.

[VGP08] Antonio Vallecillo, Jeff Gray, and Alfonso Pierantonio, editors. *Theory and Practice of Model Transformations, First International Conference, ICMT 2008, Zürich, Switzerland, July 1-2, 2008, Proceedings*, volume 5063 of *Lecture Notes in Computer Science*. Springer, 2008.

[Wag08] Dennis Wagelaar. Composition techniques for rule-based model transformation languages. In Vallecillo et al. [VGP08], pages 152–167.

[YCWD09] Andrés Yie, Rubby Casallas, Dennis Wagelaar, and Dirk Deridder. An approach for evolving transformation chains. In Andy Schürr and Bran Selic, editors, *MoDELS*, volume 5795 of *Lecture Notes in Computer Science*, pages 551–555. Springer, 2009.

Test-driven Language Derivation with Graph Transformation-Based Dynamic Meta Modeling

Gregor Engels and Christian Soltenborn

University of Paderborn
{engels,christian}@uni-paderborn.de

Abstract: Deriving a new language L_B from an already existing one L_A is a typical task in domain-specific language engineering. Here, besides adjusting L_A's syntax, the language engineer has to modify the semantics of L_A to derive L_B's semantics. Particularly, in case of *behavioral* modeling languages, this is a difficult and error-prone task, as changing the behavior of language elements or adding behavior for new elements might have undesired *side effects*.

Therefore, we propose a *test-driven language derivation process*. In a first step, the language engineer creates example models containing the changed or newly added elements in different contexts. For each of these models, the language engineer also precisely describes the expected behavior. In a second step, each example model and its description of behavior is transformed into an *executable test case*. Finally, these test cases are used when deriving the actual semantics of L_B - at any time, the language engineer can run the tests to verify whether the changes he performed on L_A's semantics indeed produce the desired behavior.

In this paper, we illustrate the approach using our graph transformation-based semantics specification technique *Dynamic Meta Modeling*. This is once more an example where the graph transformation approach shows its strengths and appropriateness to support software engineering tasks as, e.g., model transformations, software specifications, or tool development.

Keywords: language engineering, semantics, testing, DMM, graph transformation

1 Introduction

In today's world of software engineering, *domain-specific modeling languages* (DSLs) have become an important tool. A DSL is a language which has been created for the sake of being used in a certain, usually narrow domain. The language elements are abstractions of the domain's important concepts. As a result, DSLs are usually intuitive to understand and therefore well-suited as a base for the communication with the stakeholder's domain experts.

Moreover, the intuitive understandability of well-designed DSLs also results in models of a higher quality: Abstraction is always a difficult task, also for modelers. In case of DSLs, a (hopefully large) part of the necessary abstraction process has been performed at language creation time and therefore does not need to be performed by the modeler again, who can concentrate on modeling the actual business logic.

However, defining a DSL from scratch is not an easy task. The language engineer does not only need to specify the abstract and concrete syntax of the language, but also its semantics. The latter can be quite difficult, especially for languages describing behavior.

As a result, in case a language exists which has similar properties as the envisioned one, it would be desirable to reuse that language as a starting base. In this way, the language engineer can rely on an existing, proven language core, and can concentrate on performing the modifications needed for the target DSL.

In this paper, we describe a scenario of language derivation in the context of *Dynamic Meta Modeling* (DMM), a graph transformation-based semantics specification technique developed at our research group [EHS99, Hau05]. The idea is to enhance an existing language with domain-specific concepts, and to add the semantics of the new concepts to the already existing DMM semantics specification.

DMM aims at two seemingly contrary goals: DMM specifications shall be *easily understandable* and *formal*. The only prerequisite for the usage of DMM is that the language's abstract syntax is defined by means of a *metamodel*. DMM rules are (annotated) UML object diagrams, i.e., instances of the language's metamodel. As a result, due to the visual, familiar notion of DMM rules, language engineers familiar with that metamodel can easily read DMM specifications.

Behind the hoods, DMM rules are *graph transformation rules* [Roz97]. In a nutshell, this means that a DMM rule manipulates graphs. The idea is that graphs as well as rules are *typed* over the language's metamodel, and that DMM rules are used to describe changes on instances of that metamodel (and therefore behavior).

Finally, given a certain language's instance (i.e., a model) and a DMM specification, a transition system can be computed, where states are states of execution of the model, and transitions are applications of DMM rules. The transition system reflects the complete behavior of the model; it can then e.g. be analyzed with model checking techniques [ESW07]. A more detailed introduction to DMM will be given in Sect. 2.

Now, given a language equipped with a DMM specification, and provided that that language is suited as base for deriving a new language by enhancement, the language engineer has to add DMM rules to the existing DMM ruleset which define the new language element's semantics. An often occuring problem in object-oriented scenarios is that some behavior defined in the context of a certain type shall not be applied in the context of a subtype, i.e., behavior has to be *overridden* by new behavior. In [EFS09], we have introduced a notion of *rule overriding* exactly suited for this situation, which we will use to cope with that problem.

Finally, the goal must be to perform all language changes such that the semantics of the source language is not hampered, and that the new parts of the semantics correctly reflect the intentions of the language engineer.

Since a DMM specification is formal, the first and obvious idea is to formalize that notion of correctness by means of requirements the specification shall fulfill, and then to *prove* that this is indeed the case. However, the experiences from software development seem to imply that proving the correctness of a reasonable complex system is often just not feasible; therefore, the most important technique in software quality assurance is *testing*.

In [SE09] we have suggested a pragmatic approach to help creating high-quality semantics, which is inspired by the well-known approach of *Test-Driven Development* [Bec02]. It is moti-

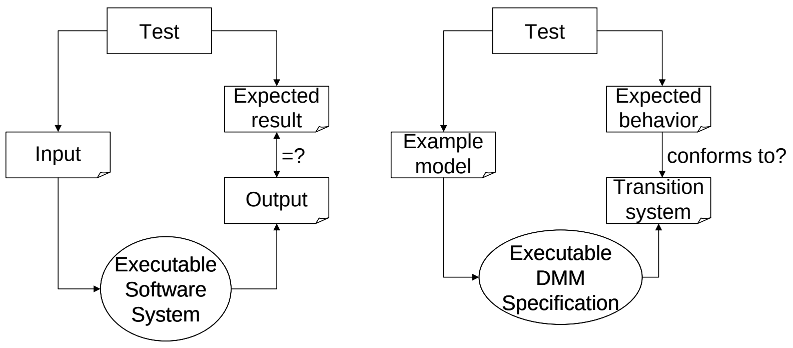


Figure 1: Comparison of testing of software systems (left) and semantics specifications (right); the test subject is depicted as an oval.

vated by the fact that a semantics specification basically follows the Input-Process-Output (IPO) model, where a certain model can be seen as the input, and the semantics of that model is the output (e.g., represented as a transition system).

Figure 1 shows our approach and its relation to the testing of software systems. In case of testing software systems, a test case consists of some input for the system and the system's expected result. The test succeeds if the actual output of the system is equal to the expected result.

In contrast to that, we want to test a DMM specification. Therefore, a test consists of an example model and its expected behavior. From that model and the DMM specification, a transition system can be computed which represents the model's behavior. The test succeeds if the actual behavior conforms to the expected behavior, i.e., if the expected behavior (and *only* the expected behavior) is contained in the transition system.

In this paper, we show how to apply the approach of test-driven semantics specification within the scenario of language derivation. For this, we will discuss a small example of language enhancement: We will enhance the language of UML Activities. While doing this, we will point out some problems, and we will show how DMM rule overriding can help to solve these problems, and how a test-driven approach can be used that the new language's semantics has a certain quality.

Structure of Paper In the next section, we will give an introduction to our semantics specification technique Dynamic Meta Modeling. Based on that, in Sect. 3 we will introduce a small example of language modification, discuss side effects of that modification, introduce our approach of test-driven semantics specification, and discuss how that approach can be used as a "safety net" against such side effects when performing language derivation. Section 4 will point out some work related to ours, and Sect. 5 will conclude and discuss our future plans.

2 Dynamic Meta Modeling

We have argued in Sect. 1 that DMM specifications are formal, but also easily understandable. This is an advantage to many other formalisms, which can only be used by experts of that formal-

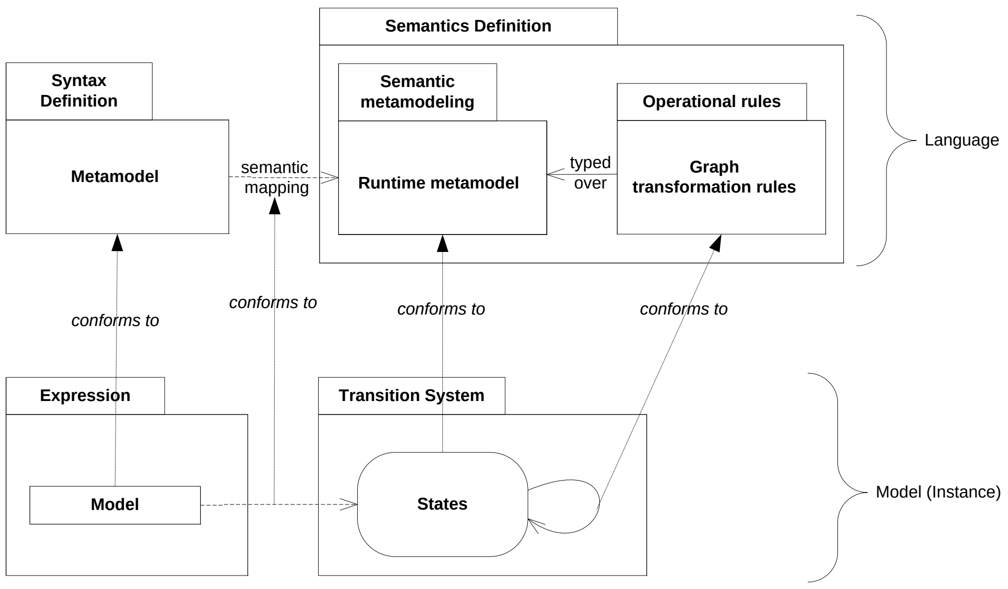


Figure 2: Overview of the DMM approach

ism. For instance, the π-calculus [MPW92] is a powerful formalism for semantics specification, but the average language user can not be expected to understand a π-calculus specification, let alone use it to specify the semantics of a language.

DMM aims at delivering semantics specifications which indeed can be understood by such users. It does that by providing a visual language for semantics specification. Additionally, a DMM specification is based on the metamodel of the according language, allowing users who are familiar with that metamodel to easily read a DMM specification.

In a nutshell, a DMM specification is created by first extending the language's metamodel with concepts needed to express states of execution; the enhanced metamodel is called *runtime metamodel*. Then, the behavior is defined by creating operational rules which modify instances of that runtime metamodel. An overview of DMM is provided as Fig. 2.

Since our goal will be to enhance the language of UML Activities, let us investigate the language's semantics specification as an example: the metamodel provided by the OMG [Obj09] only contains syntactic information, i.e., it describes the set of valid UML Activities. The language's dynamic semantics is specified using natural language: for instance, the UML specification document states that "the semantics of Activities is based on token flow". However, the language's metamodel does not contain the concept of token.

Therefore, the runtime metamodel adds that concept: A class `Token` is introduced such that instances of that class are associated to the language elements they are located at (e.g., `Actions`). As a consequence, an instance of the runtime metamodel describes a state of execution of an Activity by having `Token` objects sitting at particular elements. Figure 3 shows an excerpt of the runtime metamodel for UML Activities. The runtime class representing the concept of tokens is depicted in bold. Its `location` associations to the `ActivityNode` and `ActivityEdge` classes allow to model a concrete state of execution of an Activity: If a token

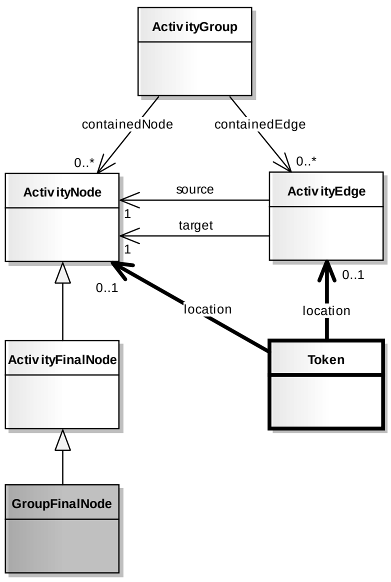


Figure 3: DMM runtime metamodel for UML Activities.

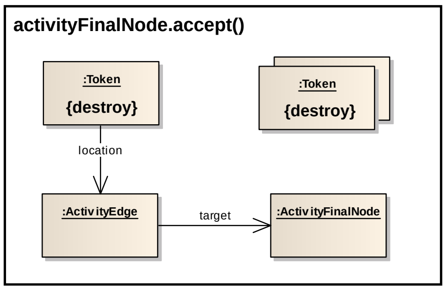


Figure 4: The original DMM rule describing the `ActivityFinalNode`'s semantics.

is sitting on a particular `Action`, the model contains a `location` link from the token to the object representing that `Action`.

Now, the operational rules come into play; a DMM rule is depicted as Fig. 4. Its semantics is as follows: The rule can be applied if an incoming `ActivityEdge` of an `ActivityFinalNode` carries at least one token. If this is the case, the rule is applied: all tokens flowing through the Activity are deleted (no matter where they are located), bringing the execution of the whole Activity to an immediate end.

The underlying formalism of DMM are *graph transformations*. Using the GROOVE toolset [Ren04], DMM specifications give rise to transition systems which describe the complete behavior of the according models. The start state of such a transition system is a model (in our case, an instance of the runtime metamodel, where e.g. `InitialNodes` are already equipped with a token). Now, every rule of the DMM specification is checked for applicability; if a rule can be applied, the application will lead to a new (and different) state (where e.g. the location of tokens has changed); the resulting transition is labeled with the applied rule. For every newly derived state, the process starts over again until no new states are found.[1]

A transition system computed in that way can then be analyzed using model checking techniques. The properties to be verified need to be formulated over the applications of rules. For instance, if we want to know if a certain `Action` can ever be executed, we need to check if the transition system contains a transition which is labeled with the rule corresponding to the `Action`'s execution.

More concretely, as there is only one generic, parameterized rule defining the semantics of Actions, the rule's parameter has to be the name of this Action. For example, if we want to know whether the Action with name "A" is ever executed, we have to check whether the transition system representing the model's behavior contains a transition labeled action.start("A").

Now that we have gotten a precise idea of DMM, we are ready to perform our language enhancement in the next section.

3 Test-driven Language Derivation

In the last section, we have seen how DMM semantics specifications work in general, and we have investigated a small part of a semantics specification for UML Activities. Our next step will be to enhance the Activity language by adding a language element, and to define that element's semantics by means of an additional DMM rule in Sect. 3.1. We will discuss possible problems caused by that language modification.

To cope with such problems, we will then introduce the reader to our approach of *test-driven semantics specification* [SE09] in Sect. 3.2. Based on that, we will show in Sect. 3.3 how to reuse the concepts of that approach when deriving a new language from an existing one: In Sect. 3.4, we will fix the semantics specification (partly) by using rule overriding, and we will use test cases as guidance. Finally, Sect. 3.5 shows the tooling involved into the whole process.

3.1 Example: Enhancing UML Activities

UML Activities are a powerful behavioral language which can be used in all kinds of domains, from specifying low-level algorithms to defining high-level business processes. However, no language can contain all elements used within all kinds of contexts. As a result, the need for domain-specific languages arises.

For instance, consider the usage of UML Activities for business process modeling. The Activity language contains two language elements dedicated for the termination of (parts of) an Activity: The `ActivityFinalNode` terminates the execution of the complete Activity, whereas the `FlowFinalNode` can be used to terminate single flows of execution (e.g., in the case of concurrency).

However, the need might arise for more flexible ways to terminate flows of execution. As an example, we want to add a `GroupFinalNode` which—if it consumes a token—brings all execution within the node's `ActivityGroup` to an immediate end (but does not affect flows

[1] DMM specifications might give rise to an infinite transition system; in this case, standard techniques from model checking such as *bounded model checking* can be applied.

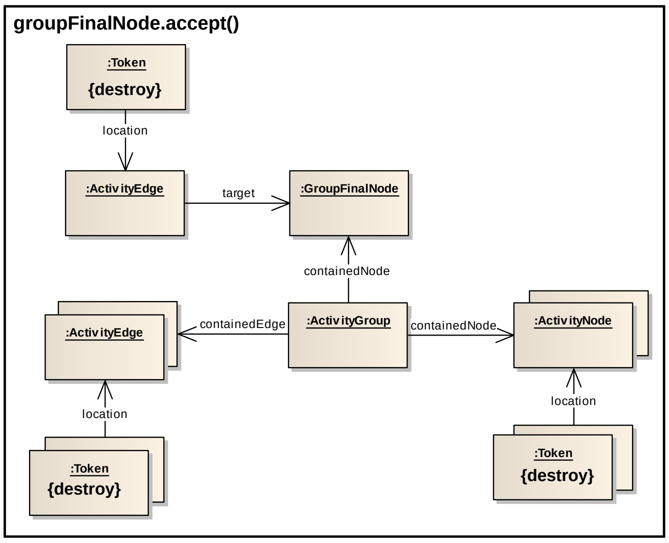


Figure 5: The modified DMM rule describing the `ActivityFinalNode`'s new behavior.

of execution outside the `ActivityGroup`).

The first step is to enhance the language's syntax, i.e., to add the `GroupFinalNode` to the language's metamodel. The integration of the new metaclass can be performed in different ways. Since the `GroupFinalNode`'s behavior is pretty similar to the behavior of the `ActivityFinalNode` (which consumes all tokens flowing in the Activity as soon as it receives a token; it has been depicted as Fig. 4 on page 5), and since the language engineer wants to reuse the concrete syntax of that node, he decides to add the element as a subclass of the `ActivityFinalNode`'s metaclass (as depicted in Fig. 3).

The language engineer also has a definition of the `GroupFinalNode`'s behavior in mind. An according DMM rule is depicted as Fig. 5. However, as we will see later, the performed language modification causes a couple of problems.

Now, we suggest to deal with such problems by performing modifications of semantics specifications in a *test-driven* way. In a nutshell, the language engineer will first create example models of the modified language. Such an example model contains one or more of the modified language elements in a certain context which should be related to the modifications which have been performed. For instance, the modification we have described above implies that the language engineer creates an example model which shows the new behavior of the `GroupFinalNode`.

Additionally, the language engineer will describe the expected behavior of that model in a precise, semi-formal way (more on that in the next section). Finally, executable test cases are generated from the example models and their behavior descriptions. The language engineer can now perform the modifications of the semantics specification against these test cases: If the tests all pass, he can take this as a sign that the modifications have been performed correctly. Otherwise, the failing test cases will hopefully point him to the problematic modifications he performed.

Since the idea to perform test-driven language derivation is based on our idea of test-driven

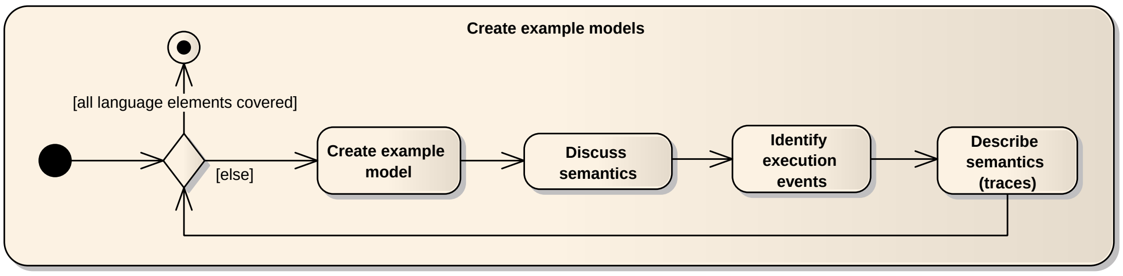

Create example models
[all language elements covered]
[else]
Create example model
Discuss semantics
Identify execution events
Describe semantics (traces)

Figure 6: Create example models

semantics specification [SE09], we want to shed light on that approach in the next section.

3.2 Test-driven Semantics Specification

We have already seen in the introduction that a semantics specification follows the Input-Process-Output model in some sense: a model serves as input, and the semantics of that particular model is the output. In this section, we want to explain this idea in more detail.

"Test-driven semantics specification" means that the semantics of a language is developed against existing test cases: As soon as all test cases succeed, the semantics specification is finished (and we have some hope that it indeed has an appropriate quality). In our scenario of test-driven semantics specification, the input specified by a test case is an *example model*, i.e., a model which demonstrates certain behavioral properties of some language elements. Sect. 3.2.1 will deal with the creation of example models and the description of their expected behavior.

Section 3.2.2 will then show how to automatically transform each example model and its description of behavior into an executable test case. Finally, Sect. 3.2.3 will point out how to perform the actual semantics specification against the test cases.

3.2.1 Creating Example Models

The starting point is the abstract syntax of the language under consideration. It defines all language elements and their relations with each other. In the case of DMM, the abstract syntax must be given as a metamodel. Based on the abstract syntax, the example models should be created step by step, systematically going from the most basic to more complex language constructs[2].

Then, for each example model the expected behavior needs to be identified, and to be described in a semiformal way. This is done using so-called *traces of execution events*. An *execution event* in our sense is some interesting event happening during the execution of a model. For instance, in the example below (which is in the context of UML Activities), we will use execution events corresponding to the execution of a particular `Action`.

Such execution events can then be composed to *traces*. A trace is a sequence of execution events which occur when a particular model is being executed. It describes one possible way of executing a particular model. The process of creating example models is depicted as Fig. 6.

[2] The example models can of course be created using the language's concrete syntax

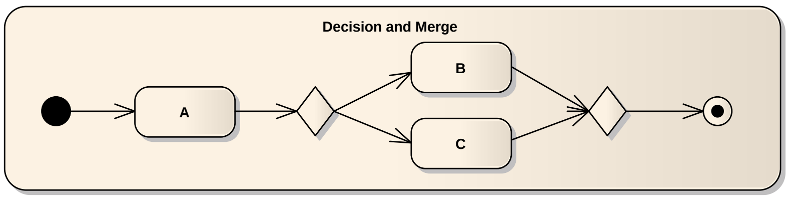


Figure 7: Example Activity containing a simple `DecisionNode`/`MergeNode` structure

Let us illustrate the above with a simple example, which is depicted as Fig. 7. Its purpose is to demonstrate the semantics of the `DecisionNode` and `MergeNode`. This example is interesting because of the fact that it allows for more than one possible execution: a token flowing through the Activity will—as soon as it has passed `Action` "A"—be routed either to `Action` "B" *or* to `Action` "C".

Obviously, the interesting execution events which occur when that model is executed are the executions of the contained `Actions`. As a result, we identify the event ActionExecutes(Name) which refers to the execution of an Action with name Name.

We have already seen above that due to the involved DecisionNode, there are two ways to execute the model. Therefore, we will describe the model's behavior by two traces of execution events:

ActionExecutes("A") ActionExecutes("B")

and

ActionExecutes("A") ActionExecutes("C")

We decided to reduce the semantics of Activities to the possible orders of execution of `Actions`, since the `Actions` are the places where the actual work will be performed. However, it would also be possible to use more fine-grained traces like InitialNode() ActionExecutes("A") DecisionNode() ActionExecutes("B") MergeNode() ActivityFinalNode().

With these traces, we have already finished the description of our example model's behavior. We can now turn to the transformation into an executable test case in the next section.

3.2.2 Deriving Test Cases

In this section, we want to investigate how to automatically verify that an example model indeed behaves as we expect it to. This is done in two steps: First, we formalize the traces of execution events of our example model by translating them into a notion of *temporal logic*. Second, we use a model checker to verify whether the transition system raising from the example model and our semantics specification contains exactly the expected behavior (and nothing else). To make our test cases executable, the described process is triggered by a small Java framework we have implemented on top of JUnit [GB].

Let us start with translating the traces of execution events into temporal logic. The idea of the translation is as follows: We want to express that the transition system contains a path starting from the start state such that all execution events occur on that path in the desired order, and that no other execution events occur in between.

We have seen in Sect. 2 that DMM specifications give rise to a transition system where each transition is labeled with the DMM rule creating that transition. As a result, we have to map our execution events to the according DMM rules. Having done that, the translation is pretty straight-forward: From each execution event e, an expression $\mathbf{EF}(r_e)$ is generated, where r_e is the DMM rule corresponding to event e. Such an expression is true iff there **E**xists a path such that **F**inally, rule r_e occurs as a label of one of the transitions.

These expressions are then nested to express the sequence of events to occur: For instance, the sequence e_1e_2 is translated into the CTL formula $\mathbf{EF}(r_{e_1} \wedge \mathbf{EF}(r_{e_2}))$, expressing that there must be some occurence of r_{e_1}, and from that point on, there must be an occurence of r_{e_2}.

The fact that there must not be any events in between e_1 and e_2 is represented by using a CTL **U**ntil expression which makes sure that no unexpected rules occur until the next desired rule occurs. We do not show this construction here; the interested reader is pointed to [SE09].

Having computed our CTL formulas $f_1, \dots, f_n$ (one for each trace of execution of the example model's behavior), we can check whether all these traces are indeed contained in the resulting transition system. This is done by using a model checker to verify whether the formulas hold on our transition system; if this is the case, the behavior is contained as expected.

Finally, we need to make sure that the transition system *only* contains the expected behavior. This is verified by a final CTL formula which reads as follows: $\mathbf{AF}(f_1 \vee \dots \vee f_n)$. It makes sure that on **A**ll paths, one of the expected sequences of events takes places.

If all CTL formulas as described above hold for our transition system, we can be sure that for our example model, the semantics specification produces exactly the desired behavior. If the model checker finds out that one of the expressions does not hold, the resulting counter example will be helpful when fixing the errors of our specification.

3.2.3 Specifying the Semantics

The actual semantics specification can then be performed against the test cases we got from the last step. We start with specifying the semantics of the most simple language elements of our language. As soon as our specification contains the semantics description of all elements contained in one of our example models, that model and its description of behavior are transformed into an executable test case, which can then be verified against the current state of the semantics specification as described in the last section.

Now, if the derived test case succeeds, we can continue with specifying the more complex elements' semantics, until finally all language elements are covered. Otherwise, we need to fix the specification until the test case succeeds.

Note that all test cases are executed within every iteration of the process described above; this is to prevent regression errors (i.e., destroying the behavior of an already specified element by specifying the semantics of a still unspecified element). The whole process is depicted as Fig. 8. Now that we have seen how to create a semantics specification in a test-driven way, we can transfer the concepts used into our scenario of deriving a new language from an existing one.

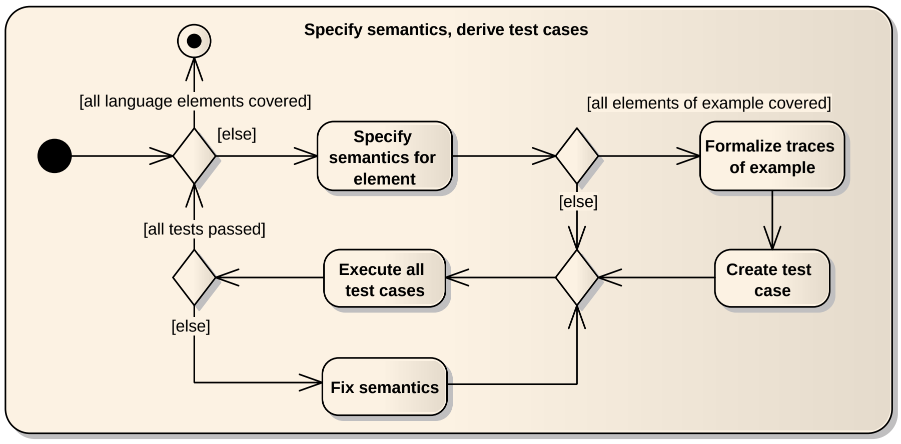

Specify semantics, derive test cases
[all language elements covered]
[else]
Specify semantics for element
[all elements of example covered]
Formalize traces of example
[else]
[all tests passed]
Execute all test cases
Create test case
[else]
Fix semantics

Figure 8: Specify semantics, create test cases from example models

3.3 Test-driven Derivation Process

Recall the language modifications we have in mind: we want to add a language element `GroupFinalNode` with the purpose of terminating the execution of a particular `ActivityGroup`. To reach this goal, we have added the according metaclass to the metamodel by subclassing an existing one.

Now, in a test-driven setting, our next step consists of defining a test case against which we can then specify the language element's behavior, i.e., an example model.[3] How does such an example model for our language extension look like?

There is one major requirement: The example model needs to demonstrate the behavior of interest. In our case, this means that we need a UML Activity containing our `GroupFinalNode`, and the Activity's structure should be such that the `GroupFinalNode`'s existence indeed has an impact.

Despite that, the example model should be as simple as possible. This has one major advantage: In case the test case derived from our example model fails at a later stage, it will be easier to figure out the cause of the failure, since less language elements can be involved.

Figure 9 shows an example model which suits our needs. Obviously, it is very simple. Additionally, it demonstrates the behavior of our new language element. To explain this, assume for a moment that the Activity does not contain the `ActivityGroup`, and that the `GroupFinalNode` is a simple `ActivityFinalNode` as the one above. Then, the semantics would be as follows: since the whole Activity's execution is terminated as soon as a token arrives at one of the `ActivityFinalNodes`, a possible behavior would be that just the execution of *A* (or *B*) takes place. In this case, one of the tokens has flown all the way down to the upper (lower) `ActivityFinalNode` before the execution of *B* (*A*) has even started (i.e., the other token is still sitting on the `ActivityEdge` in front of that `Action`). The situation is different in the

[3] Of course, there should usually be more than one example model. For instance, in our case the example model demonstrates that the `GroupFinalNode` deletes a token within its `ActivityGroup` only, but does it really destroy *all* tokens within that group?

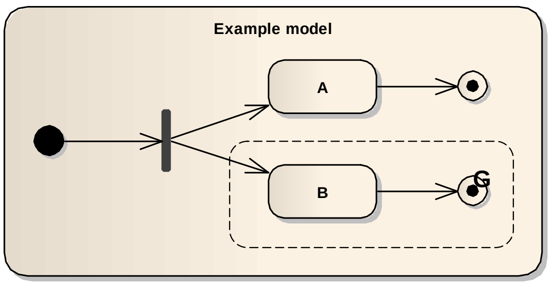


Figure 9: Example model demonstrating the behavior of the `GroupfinalNode`

case of our real example model containing the `ActivityGroup` and the `GroupFinalNode`: Since the group "encapsulates" the effect of the `GroupFinalNode`, it will always be the case that A is executed (however, B might not be executed as discussed above). We will appreciate this fact by creating our traces of execution events accordingly: There will be no trace where A is not executed.

Now, being equipped with an example model (or a set of example models) and its expected behavior, we can continue with specifying the according behavior. This is done by adding the rule we have seen as Fig. 5 to the DMM ruleset.

We are now ready to execute our test case. As it turns out, the test fails. This is due to an error we made: We did not take into account that the left-hand's graph of the rule activityFinalNode.accept() (see Fig. 4 on page 4) basically[4] is a subgraph of the new rule's left-hand graph. As a result, that rule matches whenever the new rule matches. The consequence is that the transition system resulting from the example model still contains traces where A is not executed.

This is a problem as described earlier: In our situation, it does not suffice to just add the DMM rule describing the new element's behavior. In addition, the modeler has to make sure that the old behavior does not take place in the context of the new language element `GroupFinalNode`. The next section will show how to cope with this problem using DMM rule overriding.

3.4 Using DMM Rule Overriding

We have seen above that the enhancement of our semantics specification is not finished yet: We need to prevent the `ActivityFinalNode`'s behavior from being applied in the context of the new element `GroupFinalNode`. One way to do this would be to change rule activityFinalNode.accept() such that it only matches if the `ActivityFinalNode` is not contained in an `ActivityGroup` (by adding an according negative application condition to that rule). This would indeed fix our failing test, since the `ActivityFinalNode`'s behavior would not take place any more in that situation.

Fortunately, the language's semantics has been developed in a test-driven way. In this case, our already existing test cases will hopefully tell us whether we have broken any existing behavior with our changes. And this is indeed the case: Obviously, rule activityFinalNode.accept() does not match any more in case its `ActivityFinalNode` belongs to an `ActivityGroup` (this

[4] The only difference is the typing of the FinalNodes, but that doesn't affect the matching here.

is exactly the change we have performed above). However, the semantics of the `Activity-FinalNode` stays the same, no matter whether it belongs to an `ActivityGroup` or not. In other words: An `ActivityFinalNode` which does belong to an `ActivityGroup` shall still delete all tokens flowing through the Activity, no matter where they are located. This does not happen, since rule activityFinalNode.accept() does not match any more in such a situation. As a consequence, every test case in which an `ActivityFinalNode` belongs to an `ActivityGroup` will now fail, pointing us to the fact that our language modification has had some side effects.

However, the problem is easily fixable using a more sophisticated DMM construct: Rule overriding [EFS09]. Before we start to explain that construct, let us first look into DMM rules more deeply.

Every DMM rule has a so-called *context node*, and a rule is defined in the context of that node. Since every node in a DMM rule has a type, the context node implies a type for the rule itself: the context node's type can be seen to *own* the behavior described by the rule (just as a method is owned by the class it is defined in). Note that this concept strengthens the similarity of DMM and object-oriented programming languages and therefore increases the understandability of DMM specifications.

In the rule activityFinalNode.accept(), the context node is the node typed `ActivityFinal-Node`; as expected, the context node of the new rule groupFinalNode.accept() is the node typed `GroupFinalNode`. As a result, the new rule has two interesting properties: first, as we have seen above, the rule's left-hand graph is a subgraph of the left-hand graph of rule activityFinalNode.accept() (this caused our problem at the beginning), and second, the context node's type of rule groupFinalNode.accept() is a subtype of the context node's type of rule activityFinalNode.accept().

The described situation is exactly the prerequisite for using DMM rule overriding. The idea is as follows: Given two rules r_1, r_2 such that the two properties mentioned above hold, rule r_2 can *override* rule r_1 (in our example, activityFinalNode.accept() would be r_1, and groupFinalNode.accept() would be r_2).

If a rule is overridden, its matching is affected: An overridden rule r_1 only matches a host graph if there is a morphism from the rule's left-hand graph into the host graph *and* if there is no overriding rule r_2.[5] It is easy to see that this indeed fixes our problem from above: Letting groupFinalNode.accept() override rule activityFinalNode.accept() prevents the former rule from matching in the context of our new type `GroupFinalNode`, therefore leading to the desired behavior.

3.5 The Tool Chain

It remains to shed some light on the tools involved in the process of modifying an existing language (which are depicted as Fig. 10). In the DMM chapter, we have already mentioned that the Groove toolset [Ren04] is used to compute the transition system used for analysis of a model's behavior. However, DMM supports quite sophisticated language features which are not directly supported by the notion of graph transformation rules Groove supports. For instance,

[5] Note that in [EFS09], we have in fact defined two notions of rule overriding; in [EFS09], the notion used within this paper is called *complete overriding*.

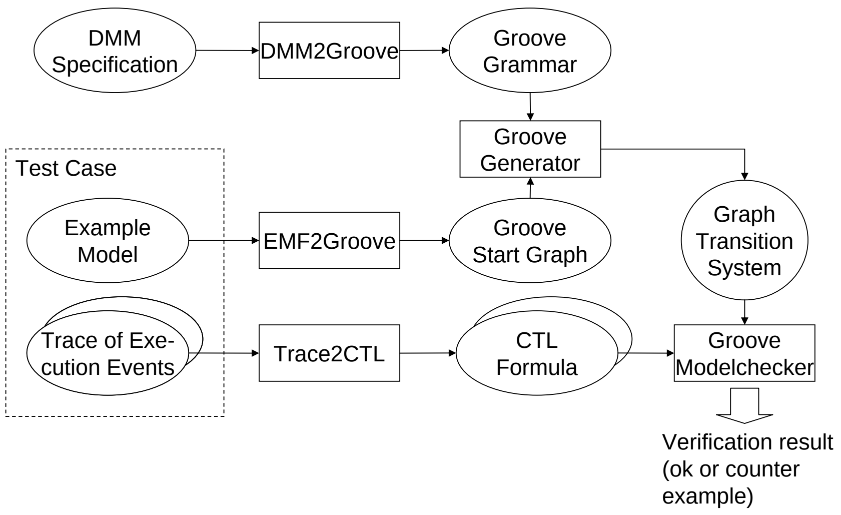


Figure 10: The DMM tool chain

DMM rules can explicitly invoke other DMM rules, which is not supported by common graph transformation tools.

Therefore, an own visual language for DMM specifications has been developed using Eclipse-based frameworks such as EMF [SBPM08], GMF [Ecl09a], and UML2 [Ecl09b] – part of the latter project is an EMF implementation of the UML metamodel. As expected, the syntax of the DMM language is defined by means of a metamodel. The DMM component DMM2Groove is responsible for translating an instance of the DMM metamodel into a valid Groove grammar ready to be executed on a proper graph state.

Additionally, the DMM tooling is capable of handling arbitrary EMF models. The according DMM component EMF2Groove takes such an EMF model (e.g., a concrete UML Activity as instance of the UML2 metamodel mentioned above) and transforms it into a Groove graph. In the next step, both the Groove grammar and the start graph are fed into the Groove Generator, which is then used to compute a transition system representing the complete model's behavior.

We have seen earlier in this section that a DMM test case not only consists of an example model, but also of a set of traces of execution events, each describing one possible execution of the example model. The DMM component Trace2CTL is responsible for generating CTL formulas from these traces (please refer to [SE09] for the details of the generation).

Finally, the transition system and the generated CTL formulas serve as input for the Groove model checker, which checks whether the formulas hold for the transition system (and therefore for the model from which the start graph had been generated). The model checker will either report that a formula holds, or it will provide a counter example showing under which circumstances the CTL property is violated; that counter example can then be used to understand why the CTL property is violated, and should therefore be very helpful when fixing the semantics specification.

4 Related Work

The work most closely related to ours probably is the work by Sadilek et.al. [Sad08]. His goal is to quickly prototype DSLs. The comparable scenario is as follows: A language's semantics might first be specified using a formal language, e.g. *Abstract State Machines*, for the sake of proving properties of the DSL's semantics. Later on, a second, more efficient semantics specification might be created which shall be *semantically equivalent* to the first one. Since both semantics specifications allow for DSL instances to be executed, the language engineer can now create test models of the DSL, execute them and compare the resulting execution traces. The main difference to our approach is that Sadilek uses tests to compare two semantics specifications, whereas we use them to convince ourselves that the semantics specification indeed produces the behavior the language engineer had in mind.

In the area of language engineering, several approaches for defining DSLs exist. For instance, MetaCase provides MetaEdit [SLTM91], Microsoft provides the DSL Tools as part of MS Visual Studio [CJKW07], and the Eclipse foundation provides the Graphical ModelingFramework [Ecl09a]; all these approaches aim at an easy creation of visual languages. openArchitectureWare [HVEK07] provides a set of tools which allow for the easy creation of textual languages, including powerful editor support.

To our knowledge, all the above approaches focus on defining DSLs from scratch. Additionally, they do not focus at all on the definition of behavioral semantics: the approaches provide support for code generation, but they do not provide a means to systematically create high-quality code generators; the generation is pretty much done ad-hoc.

Quite some work exists on typed graph transformations, on which we defined our notion of rule overriding. For instance, in [LBE$^+$07], de Lara et.al. show how to integrate attributed graph transformations with node type inheritance, therefore allowing for the formulation of *abstract* graph transformation rules (i.e., rules which contain abstract nodes). The resulting specifications tend to be more compact, since a rule containing abstract nodes might replace several rules which would otherwise have to be defined for each of the concrete subtypes. The resulting formalism does not provide support for refinement of rules (and is therefore comparable with the expressiveness of the current state of DMM).

In [TR05], Taentzer et.al. show how to formulate structural properties of type graphs with inheritance using graph constraints, and they provide a translation into standard graphs. In contrast to our work, they concentrate on structure, whereas our approach modifies the behavior of rules participating in an *overrides* relation.

5 Conclusion

In this paper, we have shown how test-driven approaches from software engineering can be reused in the field of language derivation. For this, we have first introduced our semantics specification technique Dynamic Meta Modeling, and we have explained how graph transformations serve as the backing formalism of DMM. Based on that, we have introduced a simple example of language enhancement in Sect. 3, and we have discussed the problem of overriding existing behavior.

We have then shown how to perform language derivation in a test-driven way, following the approach of test-driven semantics specification. We have also shown how rule overriding can be used to override already existing DMM graph transformation rules, and we have demonstrated all that by fixing the flaws we (intentionally) introduced within our language modification example.

Overall, we have presented an example of applying the well-established formalism of graph transformations [EEKR99] in a typical software engineering scenario. Currently, we are investigating different notions from software engineering for the sake of using them within our test-driven semantics specification approach, the most important one being *test coverage criteria*: for instance, a minimum such criterion is that over all test cases, every DMM rule has been applied at least once, but more complex coverage criteria are worth investigating.

Additionally, our focus is on developing tool support for test-driven language engineering. Our tool support has the following goals:

- Easy specification and execution of test cases.
- Back-propagation of the model checker's counter example in case a test case failed.
- Visual debugging and execution of test cases.

Finally, two of our students are working on complex semantics specifications for behavioral diagrams of the UML, using test-driven semantics specification. In the course of their work, they will also have to modify the DMM specification of an existing language. While performing their work, the students will make use of the existing DMM tooling; we plan to learn from their efforts about further needed tooling.

Bibliography

[Bec02] K. Beck. *Test-Driven Development by Example*. Addison-Wesley Longman, Amsterdam, The Netherlands, 2002.

[CJKW07] S. Cook, G. Jones, S. Kent, A. Wills. *Domain-Specific Development with Visual Studio DSL Tools*. Addison-Wesley Professional, 2007.

[Ecl09a] Eclipse Foundation. Graphical Modeling Framework. http://www.eclipse.org/modeling/gmf/, 2009. online, accessed 5-5-2009.

[Ecl09b] Eclipse Foundation. UML2 Project. http://www.eclipse.org/uml2/, 2009. online, accessed 5-5-2009.

[EEKR99] H. Ehrig, G. Engels, H.-J. Kreowski, G. Rozenberg (eds.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 2: Applications, Languages, and Tools*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1999.

[EFS09] G. Engels, D. Fisseler, C. Soltenborn. Improving Reusability of Dynamic Meta Modeling Specifications with Rule Overriding. In R. DeLine (ed.), *Proceedings of the 2009 IEEE Symposium on Visual Languages and Human-Centric Computing*

(VL/HCC 2009), Corvallis, Oregon (USA). Pp. 39–46. IEEE Computer Society, Piscataway, NJ (USA), 2009.

[EHS99] G. Engels, R. Heckel, S. Sauer. Dynamic Meta Modelling: A Graphical Approach to Operational Semantics. In *Proceedings of the workshop on Rigorous Modeling and Analysis with the UML: Challenges and Limitations (satellite event of the Conference on Onject-Oriented Programming, Systems, Languages, and Applications (OOPSLA 1999)), Denver, CO (USA)*. 1999.

[ESW07] G. Engels, C. Soltenborn, H. Wehrheim. Analysis of UML Activities using Dynamic Meta Modeling. In Bosangue and Johnsen (eds.), *Proceedings of the FMOODS 2007 Conference*. LNCS 4468, pp. 76–90. Springer, 2007.

[GB] E. Gamma, K. Beck. JUnit Homepage. http://www.junit.org/. online, accessed 1-2-2010.

[Hau05] J. H. Hausmann. *Dynamic Meta Modeling*. PhD thesis, University of Paderborn, 2005.

[HVEK07] A. Haase, M. Völter, S. Efftinge, B. Kolb. Introduction to openArchitectureWare 4.1.2. MDD Tool Implementers Forum (Part of the TOOLS 2007 conference, Zürich), 2007.

[LBE$^+$07] J. de Lara, R. Bardohl, H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. Attributed Graph Transformation with Node Type Inheritance. *Theoretical Computer Science* 376(3):139–163, 2007.

[MPW92] R. Milner, J. Parrow, D. Walker. A Calculus of Mobile Processes, I. *Information and Computation* 100(1):1–40, 1992.

[Obj09] Object Management Group. OMG Unified Modeling Language (OMG UML) – Superstructure, Version 2.2. http://www.omg.org/docs/formal/09-02-02.pdf, 2 2009.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *AGTIVE 2003 – Revised Selected and Invited Papers*. LNCS 3062, pp. 479–485. Springer, 2004.

[Roz97] G. Rozenberg (ed.). *Handbook of Graph Grammars and Computing by Graph Transformation, Vol. 1: Foundations*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1997.

[Sad08] D. A. Sadilek. Prototyping Domain-Specific Language Semantics. In *Companion to the 23rd ACM SIGPLAN conference on Object-oriented Programming Systems Languages and Applications*. ACM, New York, 2008.

[SBPM08] D. Steinberg, F. Budinsky, M. Paternostro, E. Merks. *EMF: Eclipse Modeling Framework, Second Edition*. Addison-Wesley Professional, 2008.

[SE09] C. Soltenborn, G. Engels. Towards Test-Driven Semantics Specification. In A. Schürr (ed.), *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems (MoDELS 2009), Denver, Colorado (USA)*. Pp. 378–392. Springer, Berlin/Heidelberg, 2009.

[SLTM91] K. Smolander, K. Lyytinen, V.-P. Tahvanainen, P. Marttiin. MetaEdit: a Flexible Graphical Environment for Methodology Modelling. In *Proceedings of the third international conference on Advanced information systems engineering (CAiSE 91)*. Pp. 168–193. Springer-Verlag New York, Inc., New York, NY, USA, 1991.

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

From Separate Formal Specifications to Certified Integrated Visual Modelling Techniques and Environments Position Statement

Hartmut Ehrig
TFS-Group
Technische Universität Berlin

Abstract: In this position statement we discuss the state of the art and role of formal specification and modelling techniques in different periods with special focus on the work of the TFS-group at TU-Berlin. In the past (1970 – 1990) single formal specification techniques have been developed with little impact on practical software development. In the present (1990 – 2010) integrated and visual modelling techniques have gained more and more importance. For the future (2010 – 2020) we try to sketch the idea of a **Certified Integrated Visual Modelling Technique and Environment** based on an integration of graph theory, graph transformation and Petri net theory, short **Dynamic Graph and Net Theory.**

Keywords: Software system modelling**.** formal specification, modelling techniques, visual modelling , TFS – group

1 Introduction

This paper is a position statement for the panel discussion at the International Colloquium on Graph and Model Transformation at TU-Berlin, February 11-12, 2010. Past, presence and future of formal aspects of system modelling with special focus on the work of the TFS (Theoretische Informatik : Formale Spezifikation) –group at TU-Berlin are discussed in the following three sections :

- Formal Software Specification Techniques in the Past (1970 – 1990)
- Formal Software System Modelling in the Present (1990 – 2010)
- Future Aspects of Formal Software System Modelling (2010 – 2020)

2 Formal Software Specification Techniques in the Past (1970 – 1990)

In the beginning software development was just programming of algorithms and data types, especially implementation of numerical algorithms in FORTRAN and ALGOL. This means mathematical modelling in programming was mainly numerical mathematics, although mathematical notions of algorithms had been developed already in mathematics based on Turing machines, recursive functions, and λ-calculus. These ideas were picked up in theoretical computer science leading to the areas of algorithms and complexity theory.

The concept of abstract data types – on the other hand – was developed in the early 1970ies by computer scientists like Parnas and Hoare influenced by the debacles of large software systems in the late 1960ies. At that time it was not at all clear whether abstract data types and software systems in general could be modeled by mathematical concepts. On the contrary, the vast majority of practical computer scientists at that time was convinced that mathematical methods may be useful for numerical analysis of algorithms, but not at all for software system development.

In addition to automata theory, formal languages, algorithms, and complexity the main challenge for theoretical computer science was to find mathematical models for programming language semantics, abstract data types, and concurrency of processes. **The ultimate goal was to define correctness of software systems w.r.t. given formal requirements and to develop compositional correctness and proof techniques**.

First steps in this direction in the 1970ies were the development of operational and denotational semantics of programming languages by pioneers like Scott, Stratchey, and Nivat. The concepts of Petri nets and process calculi have been defined by other pioneers like Petri, Milner, Hoare, Rozenberg, and Montanari in order to model concurrent and distributed processes.

The main contributions of the TFS-group in this period were :

1. **A categorical approach to automata theory in order to unify different kinds of deterministic, partial, stochastic, nondeterministic, and topological automata [EKKK74].**
2. **Development of the double pushout (DPO) approach for graph transformation – together with Schneider and Rosen- in order to define graph languages and to model operational semantics [Ehr79].**
3. **The fundamentals of algebraic specification of data types and software modules in the initial algebra approach [EM85/90], based on the pioneering work of Goguen, Thatcher, Wagner, and Wright [ADJ75].**

In order to show how theory and practice of software development can influence each other the TFS- and SWT-group at TU-Berlin, chaired by H. Ehrig and Ch. Floyd respectively, created and organized together with M.Nivat (Paris) and J.Thatcher (Yorktown Heights, USA) the new conference series TAPSOFT (Theory and Practice of Software Development). It was held with great success at TU-Berlin in 1985 bringing together several pioneers from theoretical and practical computer science and about 500 researchers for both areas. TAPSOFT was continued successfully as biannual conference until 1997 at different prominent locations all over Europe. It certainly helped to bridge the gap between theory and practice in software development.

Summarizing, the first steps of formal software system specification in the past have been difficult, but at least several promising formal specification techniques for data types, modules, and concurrent processes have been developed.

3 Formal Software System Modelling in the Present (1990 – 2010)

Starting in the 1990ies formal methods, especially formal specification techniques, were supported by several basic research actions and projects launched by the European Community (EC). The TFS-group was especially involved in COMPASS, on algebraic specification techniques, and in COMPUGRAPH, GETGRATS, APPLIGRAPH, and SEGRAVIS on graph transformation techniques.

Graph grammars and transformation systems had been developed in the past mainly from the language and the process specification point of view [Roz97], but data types had to be handled separately by algebraic specification techniques. This situation was typical in the past and there was a need for integrated techniques simultaneously taking care of data types and processes for software systems. Typical examples of integrated specification techniques studied by the TFS-group have been LOTOS, High-Level Nets, and Attributed Graph Grammars (AGG) integrating algebraic specifications with CCS, Petri nets, and graph transformation systems respectively.

This need for integration was supported by the priority program SoftSpez (Integration of Software Specification Techniques for Applications in Engineering) of the German Research Council (DFG). This was initiated by W. Brauer, M. Broy, H. Ehrig (coordinator),H.-J. Kreowski, H. Reichel and H. Weber from computer science, and E. Schnieder and E.Westkämper from engineering [Ehr et al 2004].

But the need for integration was not only supported by integrated techniques, but also by the integration of different views, most prominently supported by UML [UML 2.0]. Today UML is an international standard for object oriented software development and also a well-known visual modelling technique, which is widely used in practice. The semantics of UML, however, is mainly informal and various attempts have been made to provide a formal semantics for specific UML diagram techniques. In the last decade it turned out that typed attributed graph transformation is an intuitive and powerful visual modelling technique. Moreover it has a precise formal semantics and a rich mathematical theory [EEPT06], which supports correctness for visual modelling of software systems. In addition to visual modelling also model transformation becomes more and more important for efficient construction, correctness , and consistency of software system modelling. In fact, graph transformation techniques are also most suitable for model transformations between visual and visual or textual modelling languages. Syntactical and semantical correctness can be supported by the theory of algebraic graph transformation [EEPT06] and simulation and analysis techniques by the tool AGG [AGG] developed by the TFS-group. Especially the concept of triple graph grammars introduced by A. Schürr [Sch94] is very useful for model transformation and integration and a promising formal construction and semantics developed in [EEHP09].

A most prominent forum, where the results of formal software system modelling have been presented, are the ETAPS conferences. The European Joint Conferences on Theory and Practice of Software (ETAPS) were created in 1998 as annual conferences, combining especially the well-established biannual conferences TAPSOFT and ESOP. After the first ETAPS conferences in Lisbon 1998 and Amsterdam 1999 ETAPS was continued with great success at TU Berlin organized by H.Ehrig, S. Jähnichen, B. Mahr (general chair) and P.Pepper, based on the good experience with TAPSOFT in 1985. Today ETAPS is the most prominent joint conference combining theory and practice of software in Europe, supported by the European associations EATCS, EAPLS, and EASST. Moreover ETAPS is well-established on the international level .Especially graph transformations were supported by the international graph grammar workshops from 1978 until 1998 and the international conference on graph transformation (ICGT) since 2002 in Barcelona, Rome, Natal, Leicester, and coming up in Twente and Bremen.

Summarizing the situation today, formal software system modelling – at least for small systems - is well-accepted in theory and practice, especially visual modelling using graph transformation techniques. Software system modelling is mainly based on integration and visualization of models with model transformations supported by various construction, analysis, and verification techniques.

4 Future Aspects of Formal Software System Modelling (2010 – 2020)

The ultimate goals for the semantical challenges in the past are mainly valid today, but we have done important steps to realize them at least for small systems already. Let us rephrase the goals for the future as follows :

Formal software system modelling should support the modular development and integration of correct software systems in the large, where construction, correction and verification are based on visual models and model transformations using compositional semantics and proof techniques.

In section 3 we have discussed that some of these aspects are realized for small systems already today. Of course, it is important to make sure that these techniques scale up for large systems. But what is missing in addition to realize the ultimate goals in the future? Let us point out the following two aspects concerning theory and practice of graph transformation systems :

1. **Dynamic Graph and Net Theory**
2. **Certified Integrated Visual Modelling Techniques and Environments**

The idea of dynamic graph and net theory is an integration of mathematical graph theory and optimization of algorithms in the sense of the MATHEON research center at TU-Berlin with the theory of graph transformations and Petri nets in theoretical computer science. Part of this integration has been done by the TFS-group leading to the concept of reconfigurable and higher-order Petri nets. This allows a dynamic interaction of rule based modification of the Petri net structure with the well-known token game [PEHP08] on one hand and nets and rules as token on the other hand. However, the interaction of graphs algorithms with rule based transformations of the underlying graphs is certainly promising for the evolution of algorithms in a changing environment, but only little work has been done in this direction up to now.

The idea of a certified integrated visual modelling technique and environment is an integration of visual modelling techniques inspired by UML with advanced graph and model transformation techniques. These techniques should be supported by powerful analysis, model checking, and theorem proving techniques. First steps for such an integration is first of all the PhD-thesis of L. Lambers [Lam09] concerning certification of rule based modelling supported by the AGG-system [AGG]. Other important steps are efficient and correct model transformations based on triple graph grammars [EEHP09], and the work of A. Habel and K.-H. Pennemann [HP08] on correctness and verification of nested graph constraints for graph transformations.

References

[ADJ75] ADJ-Group(Goguen,Thatcher,Wagner,and Wright): Abstract Data Types as Initial Algebras and the Correctness of Data Representation.Proc.Conf. on Computer Graphics, Pattern Recognition and Data Structures, 1975

[AGG] Attributed Graph Grammar Tool AGG : http://tfs.cs.tu-berlin.de/agg/

[Ehr79] Ehrig,H. : Introduction to the Algebraic Theory of Graph Grammars (A Survey), Springer LNCS 73 (1979), 1- 69

[Ehr et al 04] Ehrig,H. et. al. (Editors) : Integration of Software Specification Techniques for Applications in Engineering. Final Report DFG Priority Program SoftSpez, Springer LNCS 3147 (2004)

[EEHP09] Ehrig,H.,Ermel,C.,Hermann,F.,Prange,P.: On the Fly Construction, Correctness and Completeness of Model Transformations based on Triple Graph Grammars, Proc. MODELS'09

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

[EKKK74] Ehrig,H.,Kiermeier,K.-D.,Kreowski,H.-J.,Kuehnel,W. : Universal Theory of Automata, Teubner 1974

[EM85/90] Ehrig,H.,Mahr,B. : Fundamentals of Algebraic Specification 1 and 2, EATCS Monographs, Springer 1985/90

[HP08] Habel,A.,Pennemann,K.-H. : Correctness of High-Level Transformation Systems Relative to Nested Conditions, MSCS 19 (2008), 245-296

[Lam09] Lambers,L. : Certification of Rule Based Modelling by Graph Transformation, PhD-thesis, TU-Berlin 2009

[PEHP08] Prange,U.,Ehrig,H.,Hoffmann,K.,Padberg,J. : Transformations in Reconfigurable Place/Transition Nets, Springer LNCS 5065 (2008), 96-113

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

[UML2.0] Unified Modelling Language UML 2.0 : http://www.omg.org/uml/ , 2003

Position Paper: Formal Methods and Agile Development

Michael Löwe

FHDW Hannover

Abstract: Modern software development is agile. It accepts that software systems undergo a lot of changes due to changes in the application context and base technology in their life cycle. Thus, most of the activities in the development process are redesign steps. Even requirements are not stable. They change in time as the context of the system changes. There is no time for complex correctness proofs of the implementation wrt. the requirements. Automatic (regression) testing has proved to be sufficient for correct system behaviour. Therefore the agile developer does not learn and apply formal methods himself. In order to be agile, however, he relies on tools for automatic refactoring of the system or of certain parts of it. These tools are able to change the system structure without changing its behaviour. We argue in this paper that, in order to build such tools, further research in the area of formal system modelling is urgently needed.

Keywords: Agile Software Development, Graph Transformation

1 Introduction

There have been two major trends in software engineering for the last decade:

1. Raising the level of abstraction for software systems design (vertical development) and

2. Providing (more sophisticated) methods for agile development (horizontal development).

Catch words like ”Model-driven Development”, ”Service-Oriented Architectures”, and ”Business Process Modelling” are connected to the first trend. The second trend is characterized by concepts like ”Software Refactoring”, ”Test-First”, ”Extreme Programming” or ”Dynamic Systems Development”.

In the first area, formal methods, especially graph transformations, have provided precise semantics for model specifications and transformation concepts from abstract to concrete system descriptions including correctness notions for static as well as dynamic models (i. e. data structures and process models respectively). There is little to improve here. The level of abstraction that is provided to the standard programmer today by software development environments, modern design and programming languages and especially by program generation tools can hardly be increased. And the mapping of abstract levels to concrete machine oriented levels can be performed almost automatically and without any interference of the designer. Vertical development form a very abstract level to the concrete level of execution is a high-level compilation process nowadays. Research in formal methods has done its job here. Educating the designers such that they can handle the abstractions is the challenge today.

In the second area, formal methods have not been applied that much, yet. At first glance, agility and formal preciseness do not go together well. We argue in this position paper against this first impression and show that there is great potential for graph transformation techniques in agile contexts.

2 Agile development

The agenda for agile development is provided by the ”Manifesto for Agile Software Devolpment” by Kent Beck et al.:

> We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:
>
> Individuals and interactions over processes and tools
> Working software over comprehensive documentation
> Customer collaboration over contract negotiation
> Responding to change over following a plan
>
> That is, while there is value in the items on the right, we value the items on the left more.

Agile development accepts that nothing is stable in software development. Requirements might change dramatically if, for example, customers use first rapid system prototypes. They learn what they want by using what the thought they have wanted. Due to these rapid changes, there is no time for an orderly formal development process that enforces correctness proofs of the implemented system wrt. the requirements. If there are such proofs, not only the software has to be refactorized frequently but also these proofs have to be rewritten over and over again. Thus, formal methods do not seem to be applicable in agile contexts. And agile developers are not very likely to appreciate education in these techniques.

But there is a different level, where formal methods can support agile processes. The rapid redesign of software systems is not chaotic. It is a continuous process that introduces, changes or removes system structure, mostly without changing the external behaviour of the system. Hence, what is needed is a catalogue of evolution patterns that improve the system's structure to a certain extend and preserve system semantics (incl. proof). The application of these patterns needs to be automated by a tool (like for example refactorizations in eclipse) and delivered to the agile developer.

Practical applicability, however, requires that we do not restrict ourselves to the level of static and dynamic models only. Since agile development aims at quick system development and early production with the system under development, we have to take into account that the models are populated. This means that there is (typically giga-bytes of) data typed in the static model and (lots of) running processes typed in the dynamic model. Thus, evolution patterns have to provide canonically induced and correct migrations on the instance level as well. Therefore, formal methods that support agile development shall provide

1. Suitable models for ”populated” systems (model and instance),

2. Formal concepts for model refactorisations and induced instance migrations,

3. Notions of correctness for such refactorisations/migrations, and

4. A catalogue of practically useful and correct patterns.

The existing body of concepts and results within the research area of ”Graph Transformation” seems to be a good starting point for this programme.

3 Formal Model for Systems: Model and Instance

Agile software development modifies *complete running systems*. It is not only the information, the operation, or the process model that is changed by refactorisations. This change also comprises at least the current system state. This state is made up by all the data that is accessible by the system (usually in a database) and the current point (or points in the case of multi-threading) of execution. Therefore, suitable formal models must be able to specify system models together with system states. A formal model for instance for object-oriented concepts must comprise the class model, the specification of the operations and methods, the currently existing object world, and the current execution context, i. e. the already sent but not yet executed messages and their execution order.

If we include explicite process models (for example specified in the Business Process Modelling Notation BPMN) into our framework, the state can get even more complex. Having the process model at hand, the current state not only comprises information about the current execution context but also the process history that has led to the current state. Additionally, the indeterministic future of the process (starting at the current point of execution) can be thought of as part of the current state.

The model and the state cannot be considered separately. The state is always determined by the model which is usually expressed by a typing relation between state items and elements in the model. For a formal framework of agile development this typing relation is central, since model changes must lead to minimal state restructurings that allow correct retypings.

In the context of graph transformation, a suitable model for the typing relation can be given by a morphism from the state graph into the model graph.

4 Refactorisations and Induced Migrations

Agile development demands automatic refactorizations of whole systems (models and states). If state migrations have to be calculated or performed manually (or by time consuming batch jobs), development becomes slow and looses its agility. Since the state continuously changes in a running system, the only way to initiate general changes is to change the model (which is constant while the state is changing). Therefore state migrations shall (1) be uniquely induced by model changes and (2) must be executable without any interactions of the developer.

It depends on the type of system that is developed whether it can be switched off during migration. Real-time embedded systems in critical applications for example cannot never be switched off. And service orientation requires minimal down-time also for modern information systems. Thus, a framework for agile development must provide some means for *migration on demand*: The state is not changed completely, it is changed step by step as the execution wrt. the new model proceeds and requires retyped state structures. This mechanism requires (i) model versioning, (ii) coexistence of different models within the running system, and (iii) (partial) typings of the same state into different models.

In the context of graph transformation, model changes can be expressed by simple graph transformation rules and their application. The canonical extension of the model change to the existing state requires some kind of universal quantification (perform the model change for *all* instances), which is not a standard mechanism in many approaches to graph transformations.

5 Correctness of Migrations

A formal framework for agile development can provide proof methods by which tool designers can show that their migrations do change the system structure but not its observable behaviour. Such proofs are valuable since the tool user can rely on the correctness of the transformation without knowing the formal languages in which the proof was formulated. The basis for such proof methods is formal semantics for complete systems. (The semantics depends on the chosen notion of state!) Here well-known notions from for example algebraic specification (observable equivalence) or process algebra (bisimulation) can be reused. If a transformation cannot be proven generally correct for all system states but only for a certain class of states, appropriate tool support shall be provided that checks the required properties of the state.

Graph transformation techniques for proving invariants of the generated graph language can support the efforts towards such proof methods.

6 Catalog of and Tool Support for Correct Evolution Patterns

All the work that has been sketched in the previous sections has one aim, namely a catalog of (partially) correct evolution patterns and its implementation within a software development environment or some software generation tool and - if migration on demand is realized - the runtime environment of the execution language. This catalog shall - amongst others - comprise patterns for the

- Introduction of new structure
- Removal of unused structure
- Introduction and removal of abstractions (observer, composite, state, etc.)
- Introduction (and removal) of structural indirection (adapter, proxy, visitor, etc.)
- Introduction (and removal) of operational indirection (command, event, etc.)
- Introduction (and removal) of transaction support
- Introduction (and removal) of locking strategies
- Introduction (and removal) of versioning and historization

- Introduction and removal of parallelism
- Decomposition of process steps
- Merging of process steps
- Introduction (and removal) of process alternatives
- Introduction (and removal) of remote communication and distribution structure

The documentation of the patterns can be provided as some sort of graph transformation rules.

7 Conclusion

In this position paper, we have argued that formal system modelling and transformation can support agile software development. It provides urgently needed concepts and tools for the consistent and correct transformation of complete and running systems. While object-oriented modelling and programming has become a quasi-standard in the software community, the approaches, languages, and methods in the research area of graph transformation are still very different. In order to produce some remarkable effect on the application domain of agile development (and other application areas), some standardization towards *the graph transformation framework and development environment* is needed.

Position Statement
Models in Software and Systems Development[1]

Bernd Mahr

Technische Universität Berlin

Abstract: The development of software and systems is, by its very nature, highly depending on models. But models also play a role in the software and system's design, where they represent the constraining standards as well as the choices of ideas and perspectives applied in the systems modelling, implementation and technology. The model of model-being, developed by the author, is briefly explained and is then used to discuss model interconnections resulting from *model compositions* and *metamodel applications*. It is claimed that the analysis of model interconnections, prerequisite or underlying software and systems design, can provide new insights into the designs architecture and may lead to new kinds of development tools.

Keywords: software, design, model, model-being, model interconnections

1 Introduction

The choice of design in software and systems development is naturally constrained by mainly four factors: first, by the requirements and expectations on the properties and features of the intended systems future application and use, second, by the norms and standards to be met, third, by the quality and limitations of resources available for the intended systems modelling, implementation and technical realization, and fourth, by the reality of the social and technical environments, in which the intended system is being embedded when applied. However, the systems future applications, its modelling, implementation, technical realization as well as its environments are at the time, before the system is being developed, not directly accessible. At that time these constraining factors can only be identified and addressed by means of prospective models. And it is not only the fact that these constraints in the development process are to be mediated by models, it is also the likeliness of change, which affects the execution of the development task: it is not unusual that expectations and requirements on a systems application and use are being modified before the system is delivered; it is also common experience that resources for its modelling, implementation and technical realization

[1] This paper builds on and repeats some of the content of Bernd Mahr: *Information Science and the Logic of Models*, Journal of Software and Systems Modelling (2009) 8, Springer Verlag 2009, p. 365-383, (See also the German original: Bernd Mahr: *Die Informatik und die Logik der Modelle*, Informatik Spektrum, 32, 3, 2009, pp. 228 – 249)

do not stay stable while the system is being developed; and it is certain that the environments in which the system is being embedded, will not be in a constant state over time[2]. All models involved in the task of development are therefore to some degree unpredictable in regard to their future adequacy and trustworthiness. And there is yet another difficulty in systems development: when the system is being modelled and implemented, all matters of its functionality and design are to be expressed as features and properties of the system itself, which is to say that the models which capture requirements and expectations of the systems application and use and of all the other constraining factors in the development task, have to be encoded as features and properties of the system.

It would, however, be wrong to conclude from these observations, that the development of software and systems are impossible tasks. There are mainly two reasons why their development, despite of these difficulties, has a good chance of success: first, the fact that in practice systems behaviour is rarely judged on a predefined and completely rigorous basis. Users usually accept to adapt their expectations, activities and patterns of use to what the system is able to do, at least to a certain degree. And second, more than 60 years of experience in theory and practice of software development have lead to techniques and tools, which enable architects and developers to cope with the consequences of mediation and change. In the widest sense, these techniques and tools concern means of coding, abstraction and coordination, all being based on models and modelling techniques. It is, however, surprising that we have little knowledge about the principal conditions for something to be a model and about the activities constituent for model use, namely the activities implied by modelling and model application. And we can hardly say what in general counts as a good model and what does not. These deficiencies may not be severe in modelling disciplines with a high level of standardisation, but they are definitely present with the general notions of model and modelling, and are also found in the fields of information and conceptual modelling[3], and in the many disciplines of computer based modelling.

Having observed this, the question comes up of what benefit it might be to gain deeper knowledge about the notion of model in general, namely in view of the tasks of software and systems development. And one might also ask, if there is a chance at all to clear the general notion of model, which is widely assumed to be indefinable. To respond to the second question first, there is much more to know about models and modelling than there is known today. But to acquire this knowledge one has to give up some of the epistemological customs of explanation: when thinking about models one can no longer avoid to constructively treating their subject and context dependency; and in order to seek for an answer to the question of *what is a model* in ontology or some formal theory, one has to rephrase this question as to *what justifies the judgement that a given object is a model*, and answer it in the realm of logic; and finally one has to focus on the structure of contextual relationships characteristic for models rather than to try to find the kind of similarity which relates an original with its model[4]. To respond to the second question, it has first of all to be observed that it is generally

[2] If application and use of a system makes a difference, which is what is intended by its provision, it will change its environment.

[3] Bernhard Thalheim: *Entity Relationship Modeling – Foundations of Database Technology*, Berlin und Heidelberg: Springer, 2000.

[4] Herbert Stachowiak: *Allgemeine Modelltheorie*, Wien / New York: Springer, 1973.

impossible to restrict the notion of model to only certain familiar types or to ask for computer science owned notions of model and modelling. Software systems are applied in almost any field of science, engineering and daily life, and the design of software has to deal with the modelling cultures and disciplines in all these fields. It is therefore unavoidable to either know about the different conceptions of model and modelling in these fields, or to develop a general conception of universal applicability. It is claimed here that a model of model-being can be conceptualized, which not only covers all known conceptions of models as particular fragments or specializations, but which can at the same time be used as a most sensitive tool for analysis and design, not only in the case of modelling in general, but also in individual situations of model use.[5] It is the potential of this analytical tool which is promising to also yield deeper insights into the structure of model interrelations of software. And from these insights, one can expect, it will be possible to derive techniques and tools for the tasks of software and systems development.

2 The epistemic pattern of model-being

The question of *what justifies a judgement that a given object is a model*, presupposes that the model-being of this object is the conclusion of a judgement for which there are grounds. If it is possible to phrase general conditions which are necessary and sufficient for such a judgement to be *acceptable*, one might take these conditions as an argument form, which, if properly instantiated, justifies the judgement of model-being in individual cases. The argument form of model-being is then seen to be the logic of models in general, while its instantiations determine the logic of individual models. Since judgements are actions of a subject, the model-being of an individual object is relativised by the subject dependency of its judgement. And since the argument form of model-being yields necessary and sufficient conditions for acceptability and not for defined or objective truth, the notion of model, conceptualized in terms of this argument form, is relativised by the subject dependency of accepting.

The conditions implied by the argument form for model-being can not be inherent properties of the object judged to being a model. This is obvious from the fact that any object can be acceptably judged to being a model if it is only positioned into a proper context, for example into a context of production in which it takes the role of a prototype. And because there is no object which is a model by necessity, since it is always possible to position an object into a context in which there is no meaning of models at all, one has to conclude that the conditions implied by the argument form of model-being are context dependent.

If an object is judged to being a model, it is necessarily conceived of as a model by the judging subject. Assuming in general that to conceive of an object means nothing else but to identify

[5] The model of model-being, for which this strong claim is made, has been developed in the last decade in interdisciplinary studies and projects by the author and his co-workers. See for example Bernd Mahr: *Modellieren. Beobachtungen und Gedanken zur Geschichte des Modellbegriffs*, in: Sybille Krämer, Horst Bredekamp (ed.): *Bild-Schrift-Zahl*, München: Fink, 2004, p. 59-86; Bernd Mahr: *Ein Modell des Modellseins. Ein Beitrag zur Aufklärung des Modellbegriffs*, in Ulrich Dirks, Eberhard Knobloch (ed.): *Modelle*, Frankfurt am Main: Peter Lang, 2008, p. 187-218; Reinhard Wendler: *Die Rolle der Modelle in Werk- und Erkenntnisprozessen*, Dissertation am Kunstgeschichtlichen Seminar der Philosophischen Fakultät III der Humboldt-Universität zu Berlin, Juli 2008; and (Mahr, 2009).

the objects involvement in context relationships[6], it seems justified to defining the argument form for model-being as to being a complex of context relationship types. The pattern of these relationship types is then taken to characterise the situations, in which an object has the role of a model. This diagram depicts this pattern[7].

$$\begin{tikzcd}
 & \overset{\text{cargo}}{\chi} \arrow[dr] & \\
A \arrow[ur] & G \arrow[l] \arrow[r] \arrow[u, "\text{transports}"] \arrow[d, "\text{is / as}"] & B \\
 & \underset{\text{model}}{\mu} \arrow[ul, "\text{of}"] \arrow[ur, "\text{for}"'] &
\end{tikzcd}$$

The diagram defines the argument form of model-being and is composed of epistemic object and object relationship types. It is therefore also called the *epistemic pattern of model-being*. The term *epistemic* is used here to indicate that objects and relationships of an instantiated argument form have existence only as intentional objects, i.e. as being conceived of by the judging subject. Objects of type A, G and B may be conceived of to be concrete or abstract, but the objects of type μ and χ as well as all relationships are necessarily abstract, i.e. their existence is independent from space and time.

The intended meaning of the epistemic pattern of model-being, which guides the instantiation of its object and relationship types, is the following:

1. In the context of this pattern an object of type G is called *model object* and an object of type μ is called *model*. An object of type G is not by its identity a model. It has to be distinguished from the model as which it is seen, because different model objects can

[6] This assumption forms the basis of the *model of conception* developed by the author. The first thoughts on this model have been described in Bernd Mahr: *Gegenstand und Kontext – Eine Theorie der Auffassung*, in: K. Eyferth, B. Mahr, R. Posner, F. Wysotzki (Ed.): *Prinzipien der Kontextualisierung*, KIT Report 141, TU Berlin, 1997, p. 101 - 119. A set-theoretical study of the model is given Tina Wieczorek: *On Foundational Frames for Formal Modelling – Sets, epsilon-sets and a model of conception*, Aachen: Shaker, 2009. For a philosophical justification of the model see Bernd Mahr: *Intentionality and modelling of conception*, 2009, to appear.

[7] For justification of this pattern see (Mahr, 2009) and Bernd Mahr: *Ein Modell des Modellseins. Ein Beitrag zur Aufklärung des Modellbegriffs*, in Ulrich Dirks, Eberhard Knobloch (ed.): *Modelle*, Frankfurt am Main: Peter Lang, 2008, p. 187-218. See also Bernd Mahr: *Modellieren. Beobachtungen und Gedanken zur Geschichte des Modellbegriffs*, in: Sybille Krämer, Horst Bredekamp (ed.): *Bild-Schrift-Zahl*, München: Fink, 2004, p. 59-86.

.

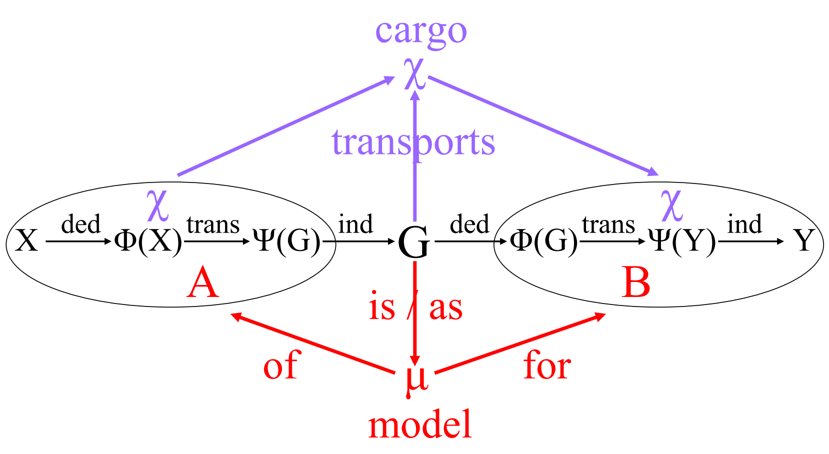
represent the same model. The fact that an object is seen *as a model* assigns it the role of a model object and determines the relationship between it and the model it represents. Instance: The model, depicted as a particular class diagram, which is its model object, can have different other diagrammatic representations as model objects.

2. A model is always a model of something, the type of which is here denoted by A. And the fact that a model is seen to be a *model of* something, determines the relationship between the model and that of which it is a model. Instance: the model which has the above mentioned class diagram as its model object, is a model of the universities library system.
3. Composing the relationships *as-a-model* and *model-of* yields the fact that the model object is seen to be a model of A. This fact determines the relationship between a model object and that of which it is a model. Instance: the above mentioned class diagram is a model of the universities library system.
4. A model is always a model for something, the type of which is here denoted by B. And the fact that a model is seen to be a *model for* something, determines the relationship between the model and that for which it is a model. Instance: the model which has the above mentioned class diagram as its model object, is a model for the object architecture of the planned implementation of the universities library system.
5. Composing the relationships *as-a-model* and *model-for* yields the fact that the model object is seen to be a model for something. Instance: the above mentioned class diagram is a model for the planned implementation of the universities library system.
6. In the context of this pattern an object of type χ is called *cargo*. For a model to make sense there must be something, named *its cargo*, which carries over from that of which a model object is a model, to that, for which it is a model. The cargo of a model is seen to be *transported* by the model object *from* that of which it is a model *to* that for which it is a model. This fact determines the three relationships between objects of type M and χ, A and χ, and χ and B. Instance: the above mentioned class diagram transports a structure of components, component relationships and component owned operations, identified in the universities library system to its implementation.

Looking more carefully at situations of model-being, it becomes apparent that the relationship types between A and G and between G and B show the same sequential structure. In combination they form the following sequence of action types:

1. an *observation* on an *initial object* of type X, resulting *observed facts* of type Φ(X),
2. a *transformation* transforming the facts of type Φ(X) to *requirements of type* Ψ(G) imposed on a model object of type G,
3. a *realization* of the requirements of type Ψ(G) by a *model object* of type G,
4. an *observation* on the *model object*, resulting *observed facts* of type Φ(G),
5. a *transformation* transforming facts of type Φ(G) to *requirements* of type Ψ(Y) on a *terminal object* of type Y,
6. a *realization* of the requirements of type Ψ(Y) by the *terminal object* of type Y.

An object of type A may now be of type X, Φ(X), or Ψ(G), and an object of type B may now be of type Φ(G), Ψ(Y), or Y. We can then speak of a *model of* an object, *of* an observation, or *of* requirements, and of a *model for* accordingly.



In any real circumstance in which a compliant judgement of model-being is made, the epistemic pattern of model-being is instantiated by the judging subject. The subject[8] identifies objects which instantiate the types X, Φ(X), Ψ(G), Φ(G), Ψ(Y), and Y, and it identifies object relationships which result from the two observations, say α and β, the two transformations, say σ and τ, and the two realizations, say π and ρ[9], thereby producing a sequence of actions (which is to be distinguished from the presupposed sequence of action types):

$$X - \alpha - \Phi(X) - \sigma - \Psi(G) - \pi - G - \beta - \Phi(G) - \tau - \Psi(Y) - \rho - Y$$

The judgement of model-being is then acceptable, if this sequence of actions is an adequate view on a defined or a true situation of model-being.

Assuming that all objects with object types in this sequence are being replaced by their logical theories[10], one observes that observations are *deductions*, and that realizations are *inductions*. A model object is therefore at the same time the result of an induction and the source of a deduction. This dual nature of model objects can be seen as one of the most typical characteristics of objects conceived of as models.

[8] The judging subject may be a human individual, but it may also be a group of individuals, a community or, even more abstract, a culture. On the other hand, a judging subject may also be a machine or a mathematical definition. Generally speaking, the judging subject is something by the authority of which the judgement in question is affirmed.

[9] These identifications need not necessarily be conscious. They constitute part of the context in which the model object takes its role as a model. The objects and object relationships identified also need not to have existence independent from the judging subject, as they may just be thought of or generated in the moment of judgement. Their identification is assumed to be valid as long as the judgement is valid in the subject's eye. Examples show that in practice this may be a fragment of a second or, in the other extreme, if the subject is not an individual, last for centuries, or even longer.

[10] The theory of an object is the set of all sentences which are true of this object. A theory is therefore language dependent. For reasons of simplicity one might assume that the choice of this language is determined as part of the subject's judgement and also that it is the same for all objects involved.

The epistemic pattern of model being is not a formal definition of what a model is. Such a definition would make sense in certain modelling disciplines, like the disciplines of set formation or graphs, but it would hardly meet the needs of model use in most of the sciences, engineering and daily life directly. If a definition was to be given it would have to follow the structure provided by the pattern, and would have to define object and object relationship types as well as their possible instantiations as mathematical entities in some foundational theory, like category theory or the theory of sets. A mathematical definition of models would replace the judging subject by formal conditions and would have to explicitly determine criteria for relationships to be observations, transformations and realizations. Namely for transformations these criteria would either be restrictive, like the notion of homomorphism[11], or they would be very general and therefore be weak in its expressiveness and determination. But the question of what the epistemic pattern conceptually is, if it is not a definition, can nevertheless be answered: it is a model itself, i.e. something which is to be judged on the basis of the same argument form just like any other model. Its epistemological justification is therefore not different to that of a mathematical definition.

3 Complexes of interconnected models

Anything constructed can be questioned for *what* it is, *how* it was constructed, and *by what means*. In the case of software, the question '*what*' asks for the systems architecture, its functionality and its technical realization; the question '*by what means*' asks for the techniques and tools applied in its modelling, implementation and installation; and the question '*how*' asks for the concepts and models underlying its design. Though the distinction between the last two questions is not exclusive, since also models are tools, and tools like formalisms and languages are representations of descriptive models, the question '*how*' is here intended to ask for the conceptual basis on which the system was built as a solution and on the ideas and perspectives taken when this solution was found. The choices of ideas and perspectives are the choices of models applied in the systems modelling, implementation and technology. For example, access to a set of stored data can be implemented as an array, a list, a record, a stack, a tree, or the like; a system which integrates distributed patient data in a hospital and makes these data available in a net, can be implemented as an information system, a communication system, an open distributed system, a service, an agent network or a peer to peer system; and a component in a component based system can be realized as an object, a module, or a service. At the end, prerequisite or underlying any software and system is a complex of interconnected models conceptualizing the ideas and perspectives which together determine its design. These models are metamodels of the systems design which is finally implemented.

Generally, a judgement of model-being is never isolated. It is always embedded into a network of intentional relations which determine its meaning[12]. Examples show that the contextual environment of a model object is typically not only the complex of objects and object relationships, which the pattern of its model-being indicates, but that this environment also

[11] Stachowiak, 1973, p. 140 – 159.
[12] John R. Searle: *Intentionality – An Essay in the Philosophy of Mind*, Cambridge: Cambridge University Press, 2004, p. 20 – 21 and 26.

includes other models which are interconnected with the model at hand in particular ways. There are at least two elementary types of model interconnections, namely *model compositions* and *metamodel applications*. Model compositions capture situations in which, for example, a model object is the initial object of another action sequence, a terminal object is the initial object of another action sequence, a model object is related to more than one initial object, observation or requirement, and the like. And metamodel applications capture situations in which items in an action sequence are constrained by other models in that the constrained item result from the application of these other models.

In a formal framework it is possible to define types of complexes of interconnected models in a systematic way, using the sequence of action types in the pattern of model-being.

$$X - \alpha - \Phi(X) - \sigma - \Psi(G) - \pi - G - \beta - \Phi(G) - \tau - \Psi(Y) - \rho - Y$$

The following are just examples for which true situations of model-being can be given:

$$X–\alpha–\Phi(X)–\sigma–\Psi(G)–\pi–G–\beta–\Phi(G)–\tau–\Psi(Y)–\rho–Y–\alpha'–\Phi(Y)–\sigma'–\Psi(Z)–\pi'–Z$$

depicting the situation of a terminal object which is a model object itself,

$$\{(X_i–\alpha_i–\Phi(X_i)–\sigma_i–\Psi(G)_i–\pi_i) \mid i \in I\}–G–\beta–\Phi(G)–\tau–\Psi(Y)–\rho–Y$$

depicting the situation in which a model object G is a model having a set of initial objects

$$X–\alpha–\Phi(X)–\sigma–\Psi(G)–\pi–G–[X'–\alpha'–\Phi(X')–\sigma'–\Psi(G')–\pi'–G'–\beta'–\Phi(G')–\tau'–\Psi(\beta)–\rho'–\beta]–\Phi(G)–\tau–\Psi(Y)–\rho–Y$$

depicting the situation in which the observation β on the model object G is obtained by applying a metamodel with model object G'.

If prerequisite or underlying any software or system there is a complex of interconnected models conceptualizing the ideas and perspectives which together determine its design, this complex may be seen to being the software or system's *model architecture*. The model architecture of software and systems is most likely a new source of knowledge about the software and systems design, the conceptualization and identification of which can provide new insights and lead to new kinds of tools for development.

Why Model Transformations Should be Based on Algebraic Graph Transformation Concepts

Gabriele Taentzer

Philipps-Universität Marburg
Germany

Abstract: Model transformations are key activities in model-driven development (MDD). A number of model transformation approaches have been emerged for different purposes and with different backgrounds. This paper is a plea for the use of algebraic graph transformation concepts to specify and verify model transformations in MDD.

Keywords: model transformation, graph transformation

1 Introduction

Model transformations play a central role in model-driven software development. They are used to e.g. refactor models, to translate them to intermediate models, and to generate code. We distinguish endogenous transformations taking place within one model language from exogenous ones which are translations between model languages [CH06].

Model-to-model transformations are usually distinguished from model-to-text transformations. (Compare also Eclipse modeling projects at [EMP].) Why does this distinction exists? While model-to-model transformation approaches such as , QVT, ATL [EMP] and graph transformation-based approaches [SNZ08] like GrGen, MOFLON, Tiger, and VMTS, transform models based on their underlying syntax structure only, model-to-text transformations performed by tools such as Jet and Velocity, are usually mixed approaches. The abstract syntax of an input model is transformed to some text in concrete syntax, often program text. These approaches are based on templates considered as "clozes" where gaps are filled with information coming from the input model. I.e. these transformations are often performed in a weakly structured and untyped manner. No guarantees are given that that resulting text is syntactically correct. In contrast, model-to-model transformations offer the chance to transform valid syntax structures again to valid ones.

Models are often considered to be visual, although this is not an inherent property of models. It is very natural to consider the underlying structure of a visual model as graph. In case of the Eclipse Modeling Framework [EMF] which has developed to a quasi-standard modeling technology, the underlying structure of an EMF model can be considered as graph with a spanning tree (or spanning forest, i.e. several spanning trees) exposed by containment relations.

2 Specification of model transformations by algebraic graph transformations

Several specification paradigms have been applied to model-to-model transformations such as object-oriented, rule-based, constraint-based, and imperative concepts. See [CH06] for an overview on various model transformation approaches following these paradigms purely or in combination. In the following, we highlight some features of the rule-based definition of model transformations using algebraic graph transformation concepts.

Considering the transformation of visual models, graph transformation seems to be a natural choice to manipulate their underlying graph structures. A well-defined approach such as the algebraic graph transformation [EEPT06] guarantees that the resulting model has again a graph as underlying structure and thus, is structure consistent. EMF model transformations can be defined as a special kind of graph transformations not destroying the spanning tree (forest) property. The algebraic graph transformation concepts also cover typing concepts. Similarly to object-oriented inheritance concepts, node type inheritance is offered [EEPT06]. Typed algebraic graph transformations have been shown to always lead to well-typed transformation results.

While graph transformations are strong in pattern matching and replacement using it for e.g. restructuring of class structures, collections of flexible size such as the set of all features belonging to a class, can be handled less obviously. In the algebraic context, we use amalgamated graph transformation where a kernel rule is applied exactly once and multi-rules being super rules of the kernel rule are applied as often as possible. All their applications overlap in the kernel rule application. Usually, each multi-rule application covers one collection element. As shown in [BET09], this concept can also be extended to EMF model transformation in a straight forward way. Ordered collections such as parameter lists of operations can also be treated by amalgamated transformations where the ordering is expressed by edge attributes. However, edge attributes have to be kept consistent with structure manipulations by the developer.

The usual double-pushout approach to graph transformation [EEPT06] can be interpreted as a kind of in-place transformation where new parts are directly integrated into the existing graph. In addition, it is also allowed to delete existing graph parts from the given graph. To keep track with graph manipulations, the formal definition distinguishes an original graph from a resulting one. A partial graph morphism in between precisely defines the relation between both graphs. In contrast to in-place transformation, triple graph grammars (TGGs) [KS06] distinguish three graphs, namely a source graph, a target graph, and a correspondence graph which is mapped to each of the other two for establishing a correspondence relation between source and target graphs. TGGs are useful to specify exogenous model transformations. Recently, Ehrig et.al. started to build up a theory on TGGs formalized by algebraic graph transformation concepts where round-trip transformations are in the focus. (Compare e.g. [EP08].)

3 Verification of model transformations

Since model transformations are reused heavily in MDD, they should be of high quality. It is common practice to extensively test model transformations. Their verification is still in its infancy. Which properties are interesting to be verified?

As pointed out in the previous section, it is basic to each model transformation that its results are structure and type consistent. Transformation approaches such as algebraic graph transformation guarantee this property already automatically by definition without any additional verification effort.

Furthermore, model transformations shall terminate. This property cannot always be shown, since model transformation approaches are usually Turing-complete. Thus, we are confronted with the halting problem in general. But there are approaches to develop sufficient termination conditions for model transformation system, especially for algebraic graph transformation systems (see e.g. [EEPT06]).

Another general property of model transformations to be discussed is the uniqueness of their results, i.e. given an input model the transformation result should be unique up to isomorphism. Actually, this strict property is often not needed. If e.g. a code generator provided two different programs from one and the same input model, we would expect that they both are semantically equivalent wrt. the input model. (I.e. both programs exhibit the same observable behavior when being executed.) Since semantical equivalence is often difficult to show, proving the uniqueness of transformation results is, although more strict, more practicable. Algebraic graph transformation offers a rich theory to show that transformation systems are confluent, i.e. yield unique results only. This theory is based on a result for general rewrite systems which states that a rewrite system which is locally confluent and terminating, is confluent in general. Local confluence can be shown based on critical pairs, an approach which has been lifted from term rewriting systems to graph transformation systems. A critical pair shows a conflicting situation in a minimal context. If all critical pairs can be shown to be confluent, the complete transformation system is locally confluent.

Additional to these pretty general properties, transformation results also need to be elements of target languages. This means that they have to be syntactically and semantically correct. The syntax of modeling languages is usually given by meta models [MOF], i.e. a resulting model has to be type consistent and has to obey all additional well-formedness rules of the meta model. As already pointed out, algebraic graph transformations guarantee type consistent results. Well-formedness rules which can be expressed by graph constraints [HP09] can be used as post conditions to check transformation results. Furthermore, there is a technique to translate graph constraints to application conditions of transformation rules, i.e. to translate them to pre conditions of transformations. That way algebraic graph transformations provide us with an automatic and efficient procedure to check for syntactical correctness.

Semantical correctness of transformation results is most difficult to verify. We distinguish the static and dynamic analysis of transformation results. Static analysis can be performed similarly to the check for syntactical correctness. Additional constraints are specified and verified as sketched above. Further recent approaches apply model checking to graph transformations e.g. [RSV04] or to use a theorem prover [Str08]. To analyze the dynamic semantics of transformation results we assume that dynamic semantics definitions exist for the source and the target language. Assuming that an operational semantics is given by an algebraic graph transformation system for both, source and target languages, and the source semantic rules have been translated to corresponding target semantic rules, then Ehrig et.al have shown in [EE08] that an execution step of the source model corresponds to an execution step of the target model.

4 Outlook

Model transformations form an interesting research field with a lot of new research problems to be solved. In the following, we spot on a selection of topics where the application of graph transformation concepts seems to be very promising:

The application of algebraic graph transformation concepts to EMF model transformation as presented in [BET08] shows that formal concepts can be well applied in a practical setting. Furthermore, we showed the nice property that resulting EMF models are structure and type consistent by construction. In addition, the rich theory of algebraic graph transformation can be easily adapted to this kind of EMF model transformations leading to interesting verification techniques for EMF model transformations.

As stated above, distinct transformation approaches have been emerged for model-to-model and model-to-text transformations. We can observe a recent trend where model-to-model transformation approaches such as ATL are also applied to model-to-text transformations. This means that transformation results, being texts in this case, are computed on the basis of abstract syntax structures. Approaches like JaMoPP [HJSW09] for example, define meta models for Java and provide model parser and printer. Thus having a meta model for a textual language, model-to-model transformation approaches and especially graph transformation approaches can be applied guaranteeing well-structured and well-typed transformation results. Moreover, even model-to-text transformations can be verified according to interesting properties. It is up to future work to test and probably improve the efficiency of model-to-model transformation implementations compared to model-to-text transformations.

Last but not least, we can observe that the deployment of model transformation techniques in software engineering increases and more complex forms of model transformations are needed where not only one input and one output model are considered but a number of models are involved. Example scenarios are coherent refactorings of several heterogenous models and/or code, code generation from several interrelated models such as those in the Graphical Modeling Framework [GMF], yielding a number of code files, model weaving, etc. Algebraic graph transformation seems to be a promising formal basis also for such networks of model transformations, since it is based on category theory which provides us with rigorous structuring concepts.

Bibliography

[BET08] E. Biermann, C. Ermel, G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Czarnecki et al. (eds.), *Model Driven Engineering Languages and Systems, 11th International Conference, MoDELS 2008, Toulouse, France, September 28 - October 3, 2008. Proceedings*. Lecture Notes in Computer Science 5301, pp. 53–67. Springer, 2008.

[BET09] E. Biermann, C. Ermel, G. Taentzer. Lifting Parallel Graph Transformation Concepts to Model Transformation based on the Eclipse Modeling Framework. In Drewes et al. (eds.), *Manipulation of Graphs, Algebras and Pictures: Essays Dedicated to Hans-Jörg Kreowski on the Occasion of His 60th Birthday*. 2009. Available online at http://www.informatik.uni-bremen.de/~hof/hjk60-festschrift_d.html.

[CH06] K. Czarnecki, S. Helsen. Feature-based survey of model transformation approaches. *IBM Systems Journal* 45(3):621–646, 2006.

[EE08] H. Ehrig, C. Ermel. Semantical Correctness and Completeness of Model Transformations using Graph and Rule Transformation. In Ehrig et al. (eds.), *Proc. International Conference on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 194–210. Springer Verlag, Heidelberg, 2008.

[EEPT06] H. Ehrig, K. Ehrig, U. Prange, G. Taentzer. *Fundamentals of Algebraic Graph Transformation*. Monographs in Theoretical Computer Science. An EATCS Series. Springer, 2006.

[EMF] EMF. Eclipse Modeling Framework. http://www.eclipse.com/emf.

[EMP] EMP. Eclipse Modeling Project. http://www.eclipse.org/modeling/.

[EP08] H. Ehrig, U. Prange. Formal Analysis of Model Transformations Based on Triple Graph Rules with Kernels. In Ehrig et al. (eds.), *Proc. International Conference on Graph Transformation (ICGT'08)*. LNCS 5214, pp. 178–193. Springer Verlag, Heidelberg, 2008.

[GMF] GMF. Graphical Modeling Framework. http://www.eclipse.com/gmf.

[HJSW09] F. Heidenreich, J. Johannes, M. Seifert, C. Wende. JaMoPP: The Java Model Parser and Printer. Technical report TUD-FI09-10, Technical University of Dresden, Institut für Software- und Multimediatechnik, 2009. Technical Report.

[HP09] A. Habel, K.-H. Pennemann. Correctness of high-level transformation systems relative to nested conditions. *Mathematical Structures in Computer Science* 19:1 – 52, 2009.

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

[MOF] MOF. Meta Object Facility (MOF) Core. URL: http://www.omg.org/spec/MOF.

[RSV04] A. Rensink, Schmidt, D. Varró. Model Checking Graph Transformations: A Comparison of Two Approaches. In Ehrig et al. (eds.), *International Conference on Graph Transformations (ICGT)*. Lecture Notes in Computer Science 3256, pp. 226–241. Springer Verlag, Berlin, 2004.

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

[Str08] M. Strecker. Modeling and Verifying Graph Transformations in Proof Assistants. *Electron. Notes Theor. Comput. Sci.* 203(1):135–148, 2008. doi:http://dx.doi.org/10.1016/j.entcs.2008.03.039

Author Index