Pre-Proceedings

GraBaTs 2010

4th International Workshop on Graph Based Tools

September 28th 2010, University of Twente, Enschede, The Netherlands

A satellite event of

ICGT'10

Edited by:

Juan de Lara
Daniel Varro

Preface

Welcome to GraBats'10, the 4^{th} International Workshop on Graph Based Tools, colocated with the Fifth International Conference on Graph Transformation (ICGT 2010) and the seventeenth annual workshop on Software Model Checking (SPIN). This workshop aims at continuing the GRABATS series of workshops that serve as a forum for researchers and practitioners interested in the development and application of practical graph-based tools. Based upon mathematically solid underlying concepts, graphs are at the core of tools and techniques in various application areas, and dealing with practical concerns, like: Tools for model-driven development, Meta CASE tools or generators, Tools for Visual languages (UML, Domain-specific languages), Model transformation and model management tools, Visualization, animation and simulation tools, Analysis of models, transformations and programs (including verification and validation, static analysis, testing), Data Analysis and Pattern Recognition Techniques, Tool integration techniques, Software Engineering Tools, Software Evolution and Efficient algorithms (pattern matching, handling of large graph models).

In all these areas tools are developed that store, retrieve, transform and display graphs. It is the purpose of this workshop to summarize the state of the art of graph-based tool development, bring together developers of graph-based tools in different application fields and to encourage new tool development cooperations.

This year's workshop has a special emphasis on applications of graph-based tools to Model-Driven Engineering. Different tools, built around industry-driven frameworks (like Eclipse), are based on the notion of graph to perform different activities, most notably different model transformations, like animations, simulations, refactorings and model-to-model transformations.

This year we received 17 submissions, from which 12 where selected for presentation at the workshop, 4 as long presentations and 8 as short ones. In addition, the technical programme also includes an invited presentation "Methods and Tools for the Verification of Finite-State and Infinite-State Graph Transformation Systems" by Prof. Barbara König of the University Duisburg-Essen, Germany; as well as a joint invited talk with SPIN by Darren Cofer (Rockwell Collins, USA) entitled "Model Checking: Cleared for Take Off". The programme has been organized in 4 sessions: "Graph Transformation tools and applications", "Verification and analysis I", "Diagram editors, animation and visualization" and "Verification and analysis II". We hope you will find the programme interesting, a useful forum for the exchange of ideas and an incentive for your research.

We would like to thank the members of the Program Committee and the secondary reviewers for their excellent work, they are listed below. We would also like to thank the organizing committee of ICGT/SPIN 2010 for their constant support.

September 2010.
Juan de Lara, Daniel Varro.
PC chairs of GraBaTs'10.

Program Committee

- Artur Boronat, University of Leicester (UK)
- Claudia Ermel, Technical University of Berlin (Germany)
- Esther Guerra, Universidad Autónoma de Madrid (Spain)
- Ethan Jackson, Microsoft Research (USA)
- Frederic Jouault, University of Nantes (France)
- Dimitris Kolovos, University of York (UK)
- Barbara König, Universität Duisburg-Essen (Germany)
- Tihamer Levendovszky, Vanderbilt University (USA)
- Mark Minas, Universitat der Bundeswehr Munchen (Germany)
- Gabriele Taentzer, Philipps-Universitt Marburg (Germany)
- Pieter Van Gorp, Eindhoven University of Technology (the Netherlands)
- Hans Vangheluwe, Universiteit Antwerpen (Belgium)
- Gergely Varró Budapest University of Technology and Economics (Hungary) / TU Darmstadt (Germany)
- Andreas Winter, Carl von Ossietzky University, Oldenburg (Germany)
- Albert Zundorf, University of Kassel (Germany)

External Reviewers

Jörn Dreyer, Anne Etien, Ulrike Golas, Jan Jelschen, Ruben Jubeh, Christian Krause, Johannes Spohr.

Index

Session 1: Graph Transformation tools and applications.

Visual Modeling of Controlled EMF Model Transformations using Henshin..... 1
Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning (Technische Universität Berlin, Germany)

Attribute Computations in the DPoPb Graph Transformation Engine..... 15
Hanh Nhi Tran, Christian Percebois, Ali Abou Dib, Louis Feraud and Sergei Soloviev (IRIT-University of Toulouse, France)

Enabling Graph Transformations on Program Code..... 28
Michael Striewe, Moritz Balz and Michael Goedicke (Universität Duisburg-Essen)

Session 2: Verification and Analysis I.

Reachability Analysis on Timed Graph Transformation Systems..... 41
Christian Heinzemann, Julian Suck and Tobias Eckardt (University of Paderborn, Germany)

Neighbourhood Abstraction in GROOVE..... 54
Arend Rensink and Eduardo Zambon (University of Twente, The Netherlands)

Session 3: Diagram editors, animation and visualization.

Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing Techniques..... 61
Steffen Mazanek, Christian Rutetzki and Mark Minas (Universität der Bundeswehr München, Germany)

From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation..... 75
Torsten Strobl, Mark Minas, Andreas Pleuß and Arnd Vitzthum (Universität der Bundeswehr München, Germany; Lero, University of Limerick, Ireland; Technische Universität Bergakademie Freiberg, Germany)

Design of a SOM Business Process Modelling Tool based on the ADOxx Meta-modelling Platform 89
Domenik Bork and Elmar J. Sinz (Otto-Friedrich Universität Bamberg, Germany)

Visualization of Traceability Models with Domain-specific Layouting..... 102
Ábel Hegedüs, Zoltán Ujhelyi, Ákos Horváth and István Ráth (Budapest University of Technology and Economics, Hungary)

Session 4: Verification and analysis II.

Invited Talk:

Methods and Tools for the Verification of Finite-State and Infinite-State Graph Transformation Systems..... 115
Barbara König (Universität Duisburg-Essen)

Distributed Graph-Based State Space Generation..... 117
Stefan Blom, Gijs Kant and Arend Rensink (University of Twente, The Netherlands)

Applying Offline Verification of Model Transformations to Mobile Social Networks..... 130
Mark Asztalos, Péter Ekler, Laszlo Lengyel and Tihamer Levendovszky (Budapest University of Technology and Economics, Hungary)

Incremental Pattern Matching in Graph-Based State Space Exploration..... 143
Amir Hossein Ghamarian, Arash Jalali and Arend Rensink (University of Twente, The Netherlands and NetStairs.com)

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Visual Modeling of Controlled EMF Model Transformation using HENSHIN

Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning

13 pages

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

Visual Modeling of Controlled EMF Model Transformation using HENSHIN

Enrico Biermann, Claudia Ermel, Johann Schmidt and Angeline Warning

Institut für Softwaretechnik und Theoretische Informatik
Technische Universität Berlin, Germany
henshin@tfs.cs.tu-berlin.de

Abstract: The tool HENSHIN is an Eclipse plug-in supporting visual modeling and execution of rule-based EMF model transformations. This paper describes the recent extensions of HENSHIN by control structures for controlled rule applications. The control structures comprise well-known imperative structures like sequences and conditions on rule applications. Moreover, application conditions for individual rules may now be arbitrarily nested and combined by logical connectors. We present the extension of the visual EMF model transformation environment HENSHIN to edit and perform controlled EMF model transformations along an example modeling a reactive Web service-based application (personal mobility manager).

Keywords: EMF, model transformation tool, graph transformation, Henshin

1 Introduction

Transformations are key modeling artefacts in model driven development. In graph transformation approaches and tools, rules express basic transformation steps. The application of rules may be controlled implicitly like in AGG [AGG09], i.e. by a fixed strategy such as "apply rules in arbitrary order as long as possible" and by providing negative application conditions for rules. Alternatively, control strategies may be defined explicitly like in Fujaba [FNTZ00], where an activity diagram (story diagram) defines loops or conditions on rule applications. Explicit control structures raise the expressiveness of transformation systems since they provide means to regulate the transformation process without having to introduce helper structures into the rules.

In this paper, we lift implicit and explicit control structures from graph transformation to EMF model transformation and introduce an extension of our recently developed tool HENSHIN[1] by visual editors for control structures. HENSHIN is an Eclipse plug-in supporting visual modeling and execution of EMF model transformations, i.e. transformations of models conforming to a meta-model given in the EMF `Ecore` format[2]. The transformation approach we use in our tool is based on graph transformation concepts which are lifted to EMF model transformation by also taking containment relations in meta-models into account [ABJ+10].

Applying EMF model transformation rules in HENSHIN changes a model *in-place*, i.e. the model is modified directly. Note that we speak of *EMF model transformation* in a general sense,

[1] http://www.eclipse.org/modeling/emft/henshin/, originating from EMF TIGER [EMT09, BEK+06, BEL+10]
[2] Note that we use the terms *meta-model* and *model* in this paper, which are called *EMF model* and *model instance* in the EMF documentation, respectively.

comprising not only source-to-target model-to-model transformations but also model refactorings or simulation of the system's behavior[3]. The HENSHIN transformation engine provides classes that can freely be integrated into existing Java projects relying on EMF.

Figure 1 shows the basic GUI of our HENSHIN tool before the extensions presented in this paper. The tree view 1 allows the modeler to import EMF `EPackages` containing the basic meta-model(s) defining the domain of the transformation. The initial model is edited in a visual editor 2. In the rule editor 3, transformation rules can be created by editing a rule's left-hand side (LHS, the pre-condition) and right-hand side (RHS, the post-condition). The rule in Figure 1 defines an operation which adds a `Request` object and links it to existing `Departure` and a `Destination` objects. The property view 4 shows additional information for selected objects. Note that all information edited using the editors in 2, 3 and 4 can also be obtained via the tree view 1.

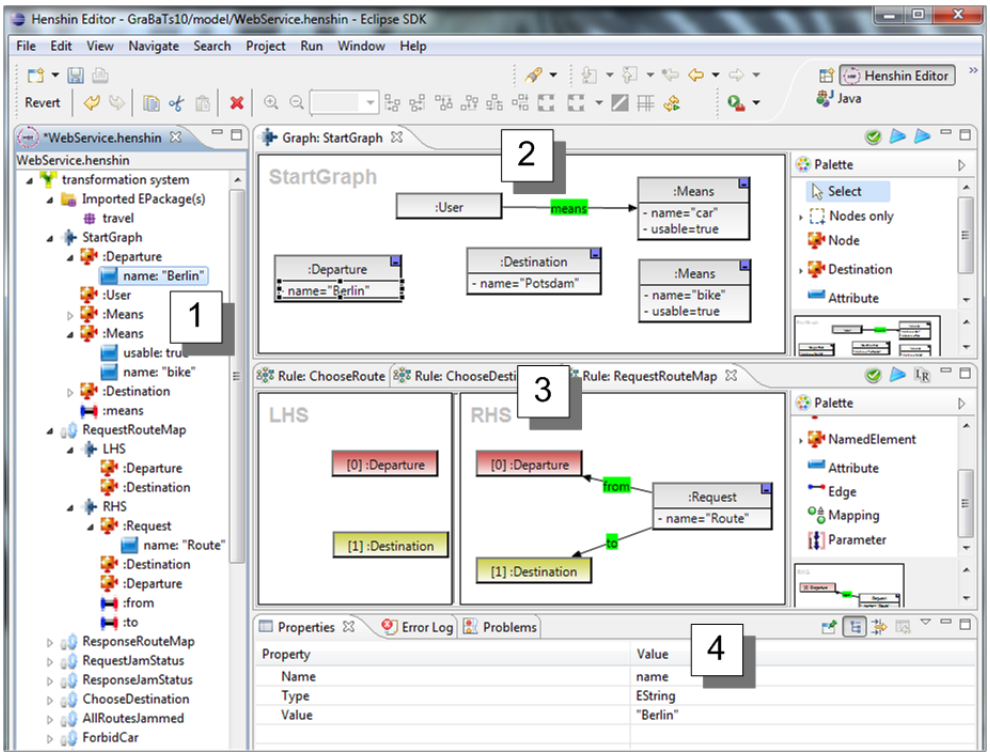


Figure 1: HENSHIN GUI with visual editors for graphs and rules.

The rule shown in 3 can now be applied to the current model in 2 leading to the transformed graph shown in Figure 2, where a `Request` object has now been created and linked to the `Departure` object named "Berlin" and the `Destination` object named "Potsdam". The layout of newly added object is computed automatically but may be adjusted by the user.

Currently there exist two implementations of the transformation engine. One is written in Java while the other translates the transformation rules to AGG [AGG09]. This is useful for

[3] like in our running example, the simulation of a personal mobility manager based on a web service

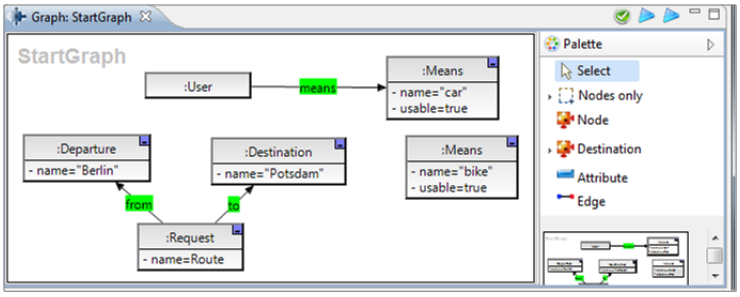


Figure 2: Transformed graph after applying rule *RequestRouteMap*

validation of consistent EMF model transformations which behave like algebraic graph transformations [BET08], e.g. to show functional behavior and correctness.

In this paper we describe the recent extension of HENSHIN supporting the use of the control structures (called HENSHIN *transformation units*), e.g. constructs for non-deterministic rule choices, rule sequences or conditional rule applications. Those constructs may be nested to define more complex control structures. Passing of model elements as parameters from one unit to another is also possible. Apart from control units defined over sets of rules, we now also support the graphical definition of application conditions for individual rules. These are application conditions in the sense of [HP09] allowing for arbitrary nesting. Several application conditions can be combined by logical connectors.

The paper is structured as follows: in Section 2, the basic concepts of graph and EMF transformation are reviewed. Section 3 presents our running example, the simulation of a personal mobility manager based on a web service. Modeling this example, we made extensive use of transformation units and application conditions which are introduced in Section 4 and Section 5, respectively. Section 6 provides an overview of related approaches and tools in comparison to our tool, and Section 7 concludes the paper with an outlook to future work.

2 EMF Model Transformation based on Graph Transformation

In this section, we introduce the main notions of modeling by algebraic graph transformation [EEPT06] (Subsection 2.1) and relate these notions to EMF modeling terms (Subsection 2.2).

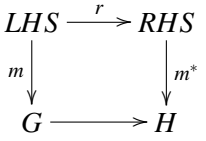
2.1 Typed Attributed Graphs and Graph Transformation

A domain-specific visual language (DSVL) is modeled by a *type graph* defining the underlying visual alphabet, i.e. the symbols (node types) and edge types which are available. Sentences or diagrams of the DSVL are given by graphs typed over (i.e. conforming to) the type graph. Node types may be attributed by attribute types.

The main idea of graph transformation is the rule-based modification of graphs where each application of a graph transformation rule leads to a new transformed graph. The core of a graph transformation rule $(LHS \xrightarrow{r} RHS)$ is a pair of graphs (LHS, RHS), called left-hand side and right-hand side, and an injective (partial) graph morphism $r : LHS \to RHS$. A graph morphism

consists of structure-preserving mappings from nodes in LHS to nodes in RHS, such that for an edge from node n_1 to node n_2 in LHS which is preserved by the rule, we have a corresponding edge from node $r(n_1)$ to $r(n_2)$ in RHS. In our approach, all graph morphisms are injective, i.e. they do not merge elements. Applying the rule $(LHS \xrightarrow{r} RHS)$ means to find a match of LHS in the source graph and to replace this matched part in the source graph by the corresponding RHS, thus transforming the source graph into the target graph (this step is called a *direct graph transformation*). Intuitively, the application of rule r to graph G via a match m from LHS to G deletes the image $m(LHS)$ from G and replaces it by a copy of the right-hand side $m^*(RHS)$. Note that a rule may only be applied if the so-called *gluing condition* is satisfied, i.e. the deletion step must not leave *dangling edges*.

Definition 1 (Graph Transformation) Let $(LHS \xrightarrow{r} RHS)$ be a typed graph transformation rule and G a typed graph with a typed graph morphism $LHS \xrightarrow{m} G$, called match.
A *direct graph transformation* $G \overset{r,m}{\Longrightarrow} H$ from G to a typed graph H via rule r, match m, and co-match m^* is shown in the diagram to the right. A sequence $G_0 \Rightarrow G_1 \Rightarrow .. \Rightarrow G_n$ of direct graph transformations is called *graph transformation*, denoted as $G_0 \overset{*}{\Rightarrow} G_n$.



A rule may be extended by input parameters, i.e. variables used to compute new attribute values for nodes in the right-hand side. When the rule is applied, the input parameters have to be bound to concrete values (either by the match or by user input).

2.2 Typed Attributed Graphs versus EMF Modeling

The Eclipse Modeling Framework EMF [EMF08] is a modeling and code generation facility for building tools and other applications based on a structured data model. Based on a meta-model, EMF provides tools and runtime support to produce a set of Java classes for the meta-model, a set of adapter classes that enable viewing and command-based editing of models conforming to the meta-model, and a basic (tree-based) editor. EMF provides the foundation for interoperability with other EMF-based tools, e.g. OCL checkers.

The conceptual similarities of modeling based on typed, attributed graphs and object-based modeling as performed by EMF are shown in Table 1.

Table 1: Mapping EMF notions to graph terminology

EMF notion	**Graph term**
EMF model	Type graph with attribution, inheritance, multiplicities. Edges can be marked as containments.
Instance model	Typed, attributed graph with containment edges
Class	Node in type graph
Object	Node in typed graph
Association	Edge in type graph (with possible multiplicities or containment mark)
Reference	Edge in typed graph that satisfies multiplicity and containment constraints.

Classes in an EMF model (i.e. the meta-model) correspond to nodes in a type graph. Associations between classes can be seen as edges in a type graph. Generalizations and multiplicity constraints of association ends can also be defined in the type graph. Objects as instantiations of classes of an EMF model are comparable to nodes in a graph which is typed by a type graph. Objects can be linked to each other by setting reference values. Such references correspond to edges in a typed attributed graph.

3 Example: Personal Mobility Manager

As running example, we specify and simulate the operational behavior of a Personal Mobility Manager (PMM), a reactive service-based application designed to satisfy requirements related to individual user mobility [LMEP08]. The aim of the system is to help the user finding an adequate route from a departure place to a destination and to propose an adequate means of transportation (either car or bike) by taking the current traffic intensity into account. We model the control flow of messages that are exchanged between the user, the PMM and corresponding Web service. To keep things simple, we do not model the actual web service here but simulate its responses by suitable variable assignments.

The modeling domain is specified as meta-model, shown in Figure 3. We have model elements for a user, his departure and destination locations, the means of transport, and requests sent to web service. A `Route` element contains a route given as response by the mobility web service, and a `JamStatus` element contains the response returned by the web service concerning the traffic on a given route.

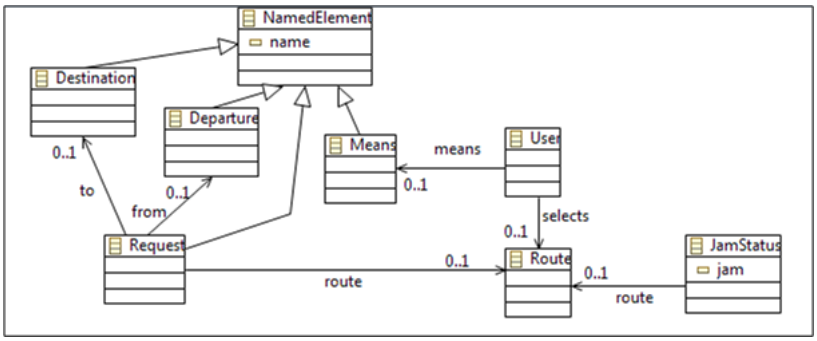


Figure 3: Meta-model for the Personal Mobility Manager

Basic PMM actions are modeled by EMF model transformation rules, shown in Figure 4.

Rule `ChooseDestination` creates a `Destination` object where the name of the destination is an input parameter; rules `RequestRouteMap` and `ResponseRouteMap` realize the creation of a route (modeled by a `Route` object) via a web service call. Having called this web service more than once, one of the returned routes is chosen by the user in rule `ChooseRoute`. For a given route, the web service is used by rules `RequestJamStatus` and `ResponseJamStatus` to get information about the current traffic situation on this route. Depending on the information obtained by the web service (and coded in the `JamStatus` node), the means of

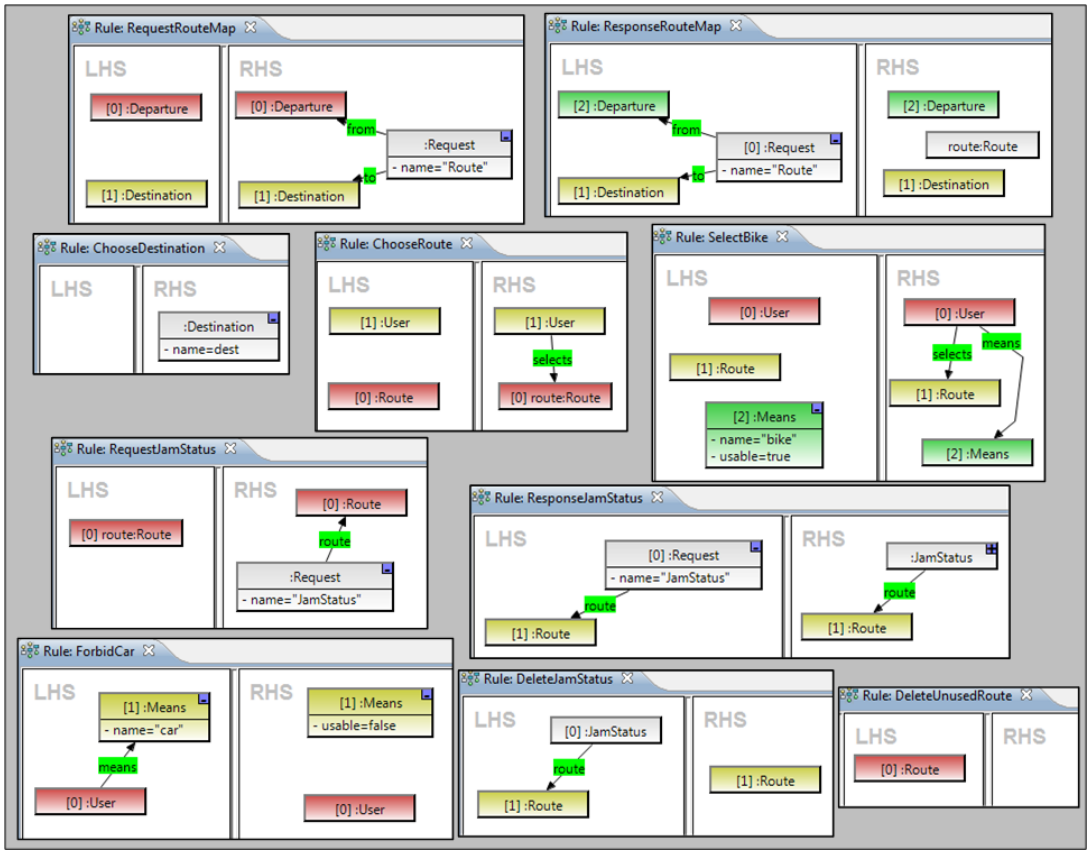


Figure 4: EMF model transformation rules for the Personal Mobility Manager

transport can be changed from the default means ”car” (as presented in the start graph in Figure 1) to the alternative means of transport ”bike”. This is realized by applying rules `ForbidCar` and `SelectBike`. At last, the information about traffic (`JamStatus` node) and possible alternative routes which have not been chosen, are deleted using rules `DeleteJamStatus` and `DeleteUnusedRoute`.

In the next section, we explain the use of HENSHIN transformation units to encapsulate and control the order of rule applications.

4 HENSHIN Transformation Units

HENSHIN transformation units may be arbitrarily nested inside each other. The most basic unit is a transformation rule. A HENSHIN transformation unit may be of type *IndependentUnit* (all subunits are applied in arbitrary order), *SequentialUnit* (all subunits are applied sequentially in a given order), *CountedUnit* (its subunit is applied a given number of times), *ConditionalUnit* (its subunits are applied depending on the evaluation of a given condition unit), and *PriorityUnit* (the applicable subunit with the highest priority is applied next). A unit is applicable (and returns

`true`) if it can be successfully executed. *PriorityUnits* and *IndependentUnits* are always applicable, while *SequentialUnits* (*CountedUnits*) are applicable only if all subunits are applicable in the given order (the given number of times). A *ConditionalUnit* is applicable if either the *then*-subunit (in case the condition is `true`) or the *else*-subunit (in case the condition is `false`) are applicable.

HENSHIN transformation units may be defined in the tree view or, alternatively, in a visual editor. The tree view shows all transformation units and their nesting hierarchy (see Figure 5). The visual editor for one unit shows the unit in a left view and one selected subunit in a right view. Unit and subunit may share parameters indicated by the coloring of the parameter fields (see Figure 5, where editors for unit `mainUnit` and unit `trafficWS` are opened in parallel). A transformation unit view shows the unit's name as header, a checkbox *Activated* which the user may select/deselect to indicate whether this unit is active (will be considered while executing), a set of parameters shown as boxes in the left column, and the names and kinds of its subunits in the right column. Arrows from (to) parameter boxes to (from) subunits indicate which parameters are input (output) of which subunit.

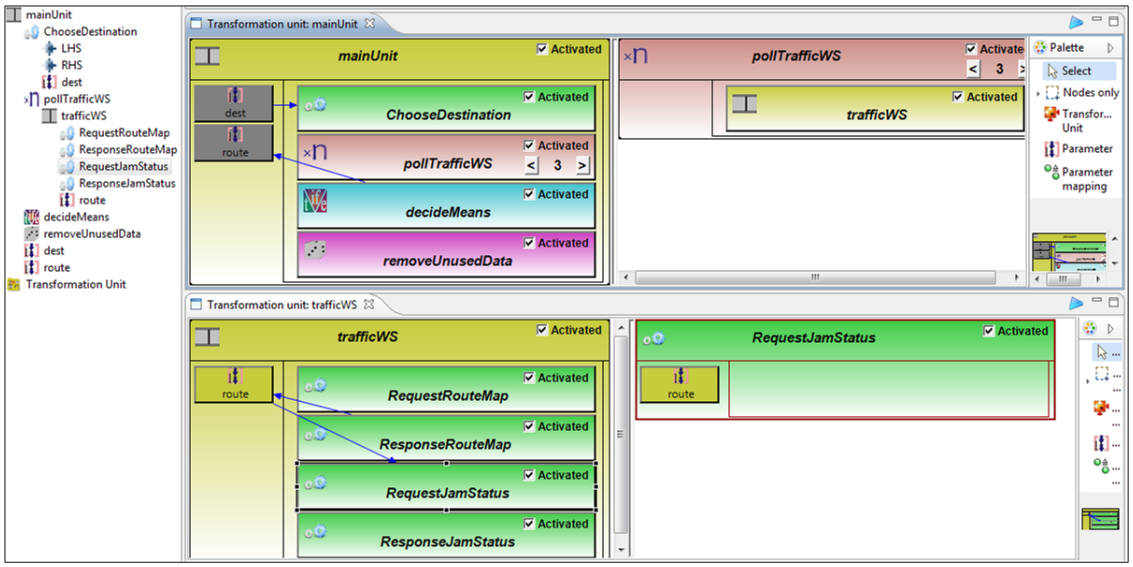


Figure 5: HENSHIN GUI with transformation unit editor

The transformation unit `mainUnit` shown in Figure 5 is the main control structure for the PMM example. It is a *SequentialUnit* (symbolized by a film strip as icon in the upper left corner) containing four subunits. This means that each subunit is applied once, in the given order from top to bottom. The first subunit, `ChooseDestination` is a transformation rule, marked by gear-wheels (see Figure 4 for the rule definition). This rule has an input parameter, the destination `dest`, a user-defined parameter. The second subunit of the main unit is a *CountedUnit* (symbolized by a "$\times n$" icon). The counter is set to 3, i.e. its subunit is applied three times. Unit `pollTrafficWS` is shown with its contents in the view to the upper right: it contains in turn a *SequentialUnit* (`trafficWS`) which controls four rules realizing the web service requests and processing the responses. The interaction of these rules within unit `trafficWS` can be seen in

the lower left view: rule `ResponseRouteMap` produces an output parameter of type `Route` which serves again as input parameter for rule `RequestJamStatus`.

The third subunit of `mainUnit`, `decideMeans`, is a *ConditionalUnit* (symbolized by an *if-then-else* icon). Clicking on its field, a detailed view of this unit is opened (see Figure 6). Here, a condition called `AllRoutesJammed` (which will be discussed in Section 5) is checked which is given as an empty rule where we check its application condition. If the condition is evaluated to `true`, the two rules `ForbidCar` and `SelectBike` in the sequential unit are applied in this order. Otherwise, rule `ChooseRoute` is applied and the parameter `route` is returned to the parent unit `mainUnit`.

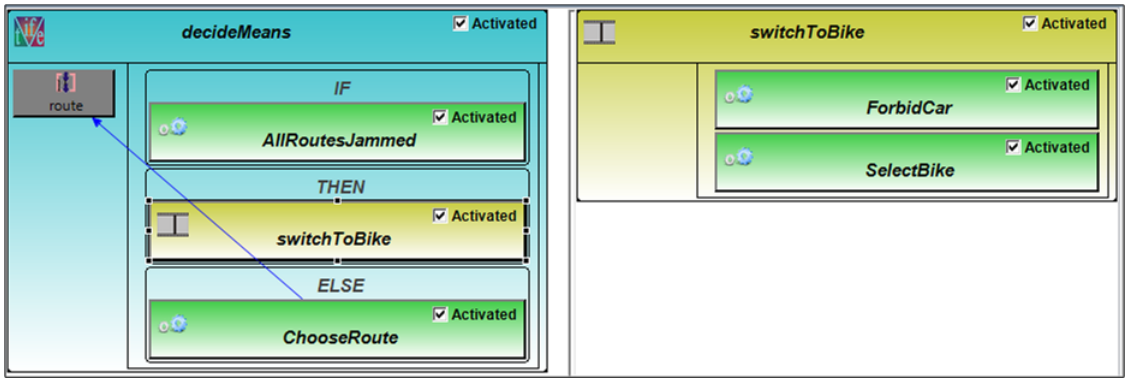


Figure 6: HENSHIN transformation unit `decideMeans`

The last child unit of `mainUnit` is the *IndependentUnit* `removeUnusedData` (with a die as icon symbol). This unit contains two rules, `DeleteJamStatus` and `DeleteUnused-Route` which perform garbage collection and are applied in arbitrary order, as long as possible.

5 Application Conditions

For graph transformation rules, well-known negative application conditions may be used that forbid to apply a rule if a certain structure is present in the graph. As a generalization, application conditions (introduced as *nested application conditions* in [HP09]) further enhance the expressiveness of graph transformations by providing a more powerful mechanism to control rule applications. While application conditions are as powerful as first order logic on graphs, we can still obtain most of the interesting results available for graph transformations *without* application conditions also for transformations *with* application conditions [EHL+10a, EHL10b] if certain additional properties hold.

Like transformation units, application conditions can be nested. Moreover, application conditions may be negated, and several application conditions may be combined by using the logical connectors AND and OR.

Definition 2 (Graph condition and application condition) A *graph condition ac* over graph G is of the form *true* or $\exists(a,c)$ where $a : P \to C$ is a graph morphism from a premise graph P to

a conclusion graph C, and c is a condition over C. Moreover, Boolean formulas over conditions over P yield conditions over P, i.e. $\neg c$ and $\wedge_{j\in J} c_j$ are (Boolean) conditions over P where J is an index set and c, $(c_j)_{j\in J}$ are conditions over P. Additionally, $\exists a$ abbreviates $\exists(a, true)$, $\forall(a,c)$ abbreviates $\neg\exists(a,\neg c)$, *false* abbreviates $\neg true$, $\vee_{j\in J} c_j$ abbreviates $\neg \wedge_{j\in J} \neg c_j$, and $c \implies d$ abbreviates $\neg c \vee d$. A graph condition ac is called *application condition* of rule $r : L \to R$ if ac is a graph condition over L; an application condition of the form $\neg\exists a$ is usually called *negative application condition*.

A condition is satisfied by a morphism into a graph if the required structure exists, which can be verified by the existence of suitable morphisms.

Definition 3 (Satisfaction of conditions) Given a graph condition ac, a morphism $p : P \to G$ satisfies ac, written $p \models ac$, if $ac = true$. A morphism $p : P \to G$ satisfies condition $ac = \exists(a,c)$ if there is an injective graph morphism $q : C \to G$ such that $q \circ a = p$ and q satisfies c. The satisfaction of conditions by graphs and morphisms is extended to Boolean conditions in the usual way. A rule $L \to R$ is applicable only if the application condition ac is satisfied for its match $m : L \to G$, i.e. if $m \models ac$.

Let us consider once more the *ConditionalUnit* `decideMeans` from our PMM example (see Figure 6). Here, the condition `AllRoutesJammed` is expressed by an empty rule[4] with a nested application condition, shown in Figure 7.

In view 1 of Figure 7, the empty rule is shown together with the outermost condition graph (condition over *LHS*). In the tree view of 1, it can be seen that we require $\neg\exists$*Route*, i.e. a morphism from graph `Route` (consisting of a single `Route` node) into the host graph must not exist for the rule to be applicable. Since this application condition is nested, we require a further condition for the `Route` graph, formulated as disjunction (OR-construct) over two more conditions: $(\neg\exists\, HasNoJamStatus \vee \exists\, IsFree)$. This formula can be seen in the tree view of 2, as well as in the corresponding visual hierarchical view where the formula is depicted as an OR block with two compartments. Clicking on one of the two parts of the disjunction in the visual view (or on one of the two OR branches in the tree view) opens the next level, either for the formula $\neg\exists$*HasNoJamStatus* in 3 or for the formula $\exists$*IsFree* in 4. Here, we have arrived at the basic level of graph morphisms. The complete nested application condition means that the empty rule is applicable (returns *true*) if there exists no route that has either no `JamStatus` node or that has a `JamStatus` node with attribute `jam=false`. Recall that in this case (all routes are jammed) unit `decideMeans` (see Figure 6) applies rule `switchToBike`, otherwise a route is chosen for the car as transport means.

6 Related Work

There are a number of model transformation engines which can modify models in EMF format such as ATL [JK05], EWL [KPPR07], Tefkat [LS05], VIATRA2 [VB07], MOMENT [Bor07].

[4] Note that we allow arbitrary transformation units as conditions in *ConditionalUnits*. While this may lead to side effects if a unit different from the empty rule is used, the conceptual advantage is that components of HENSHIN transformation units always are transformation units in turn.

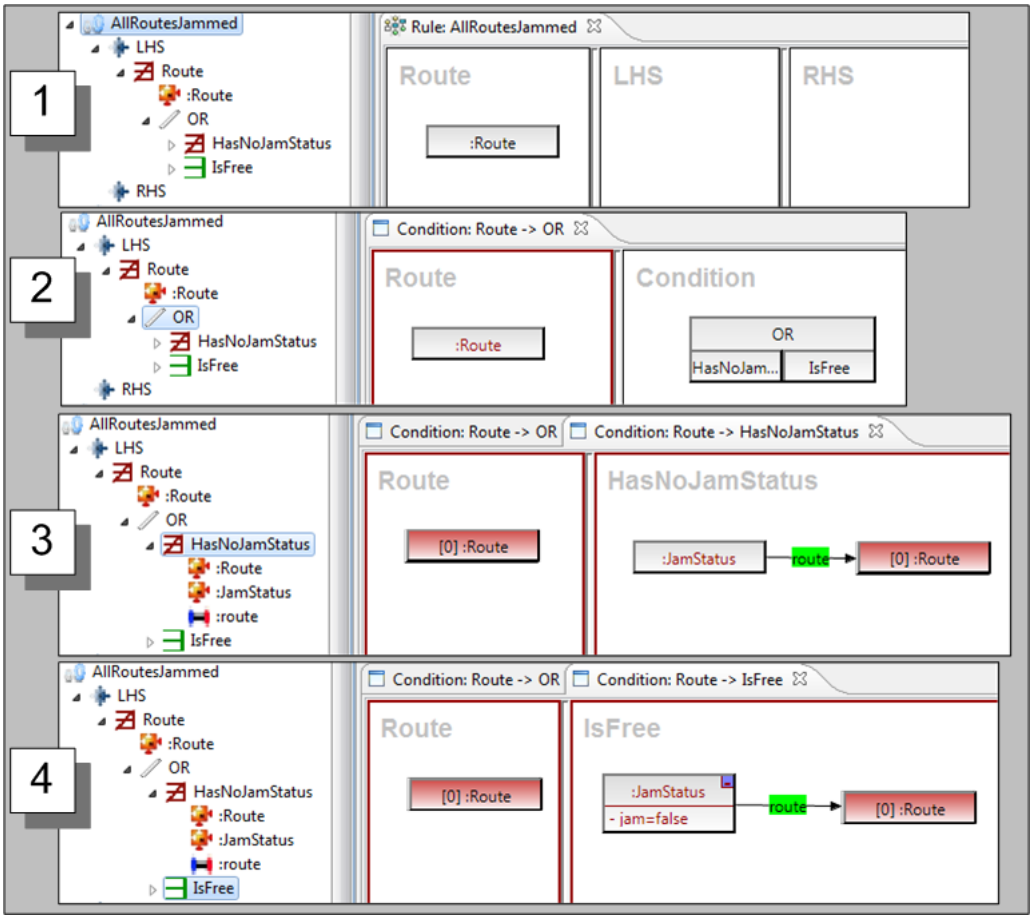


Figure 7: Empty rule `AllRoutesJammed` with application condition

For ATL, a formal semantics based on Maude has been introduced recently [TV10]. Formal semantics defined in Maude for MOMENT and for ATL might be exploited for analyzing EMF model transformations. None of these tool environments supports visual editing of control structures.

Graph transformation tools like PROGRES [SWZ99], AGG [AGG09], FuJaBa [FNTZ00] and MoTMoT [FOT10] feature visual editors which also support the definition of control structures, e.g. by story diagrams in FuJaBa, which were extended by implicit control in [MV08]. The tool GrGen.NET [GK08] also supports the arbitrary nesting of application conditions but is based on a textual specification language. MoTMoT (Model driven, Template based, Model Transformer) is a compiler from visual model transformations to repository manipulation code. The compiler takes models conforming to a UML profile for Story Driven Modeling as input and outputs Java Metadata Interface (JMI) code. Control structures are expressed by activity diagrams. Since the MoTMoT code generator is built using AndroMDA, adding support for other repository platforms (like EMF) is possible in principle and consists of adding a new set of code templates.

To the best of our knowledge, none of the existing EMF model transformation approaches

(based on graph transformation or not) support confluence and termination analysis of EMF model transformation rules yet. Here, the HENSHIN approach and tool environment serves as a bridge to make well-established tool features and formal techniques for graph transformation available for model-driven development based on EMF.

7 Conclusion

In this paper, we presented two extensions for supporting controlled EMF transformations in our EMF transformation environment HENSHIN. The first extension supports the visual definition of HENSHIN transformation units which may be hierarchically nested (the basic unit being a rule) and which restrict the possible rule application sequences in a suitable way. The second extension concerns the definition of application conditions for transformation rules. Such conditions may be nested as well, and they may be combined by logical connectors such as AND and OR. We illustrated the usage of the extended HENSHIN environment by a simulation example of a personal mobility manager (PMM). Apart from the PMM example, HENSHIN has been applied also for larger case studies, e.g. for model refactorings [ABJ+10] and model-to-model transformations such as the *Ecore2Genmodel* case study of the Transformation Tool Contest 2010 [BE10][5].

The extended HENSHIN environment provides visual views for all control structures and conditions supporting zooming into deeper nesting levels. Thus, the visualization is independent of the complexity of the (nested) control structures, as only two levels are shown at a time. Both tree view editing and visual editing is supported at all levels. For editing formulas within application conditions, from the user's perspective an additional textual view of a complete formula might be desirable [GP96]. The integration of such a textual formula view in HENSHIN is work in progress.

A special kind of transformation units in HENSHIN are *AmalgamationUnits*, which are useful to specify *forall*-operations on recurring model patterns. An *AmalgamationUnit* is a multi-rule scheme containing the model pattern and a fixed kernel rule part. An amalgamated rule, induced by such a scheme, is a kind of parallel rule synchronized at the kernel rule part. Its application modifies all recurring instances of the model pattern in one step. The development of a visual editor within HENSHIN for *AmalgamationUnits* is work in progress.

Furthermore, on the theoretical side we aim to lift confluence and termination analysis results from the rule level to the level of transformation units.

References

[ABJ+10] T. Arendt, E. Biermann, S. Jurack, C. Krause, G. Taentzer. Henshin: Advanced Concepts and Tools for In-Place EMF Model Transformations. In *Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'10)*. 2010. To appear.

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

[5] On the TTC webpage http://planet-research20.org/ttc2010/ a Share demo of HENSHIN can be found as well.

[BE10] E. Biermann, C. Ermel. Modeling the "Ecore to GenModel" Transformation with EMF Henshin. In *Proc. Transformation Tool Contest 2010 (TTC'10)*. 2010. http://planet-research20.org/ttc2010/.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. Graphical Definition of In-Place Transformations in the Eclipse Modeling Framework. In *Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'06)*. LNCS 4199, pp. 425–439. Springer, 2006.

[BEL+10] E. Biermann, C. Ermel, L. Lambers, U. Prange, G. Taentzer. Introduction to AGG and EMF Tiger by Modeling a Conference Scheduling System. *Int. Journal on Software Tools for Technology Transfer* 12(3-4):245–261, 2010. http://www.springerlink.com/content/p4n1g45627852743/

[BET08] E. Biermann, C. Ermel, G. Taentzer. Precise Semantics of EMF Model Transformations by Graph Transformation. In Czarnecki (ed.), *Proc. Int. Conf. on Model Driven Engineering Languages and Systems (MoDELS'08)*. LNCS 5301, pp. 53–67. Springer, 2008. http://tfs.cs.tu-berlin.de/publikationen/Papers08/BET08.pdf

[Bor07] A. Boronat. *MOMENT: A Formal Framework for Model Management*. PhD thesis, Universitat Politècnica de València, 2007.

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

[EHL+10a] H. Ehrig, A. Habel, L. Lambers, F. Orejas, U. Golas. Local Confluence for Rules with Nested Application Conditions. In *Proc. Int. Conf. on Graph Transformation*. 2010. To appear. http://tfs.cs.tu-berlin.de/publikationen/Papers10/EHL+10.pdf

[EHL10b] H. Ehrig, A. Habel, L. Lambers. Parallelism and Concurrency Theorems for Rules with Nested Application Conditions. *Electronic Communications of the EASST* 26, 2010. http://journal.ub.tu-berlin.de/index.php/eceasst/issue/view/36

[EMF08] Eclipse Consortium. Eclipse Modeling Framework (EMF) – Version 2.4. 2008. http://www.eclipse.org/emf.

[EMT09] TFS-Group, TU Berlin. EMF Tiger. 2009. http://tfs.cs.tu-berlin.de/emftrans.

[FNTZ00] T. Fischer, J. Niere, L. Torunski, A. Zündorf. Story Diagrams: A new Graph Rewrite Language based on the Unified Modeling Language. In *Proc. Int. Workshop on Theory and Application of Graph Transformation (TAGT)*. LNCS 1764, pp. 296–309. Springer, Berlin, 2000.

[FOT10] FOTS-Group, University of Antwerp. MoTMoT: Model driven, Template based, Model Transformer. 2010. http://www.fots.ua.ac.be/motmot/index.php.

[GK08] R. Geiß, M. Kroll. GrGen.NET: A Fast, Expressive, and General Purpose Graph Rewrite Tool. In Schürr et al. (eds.), *Proc. 3rd Intl. Workshop on Applications*

of Graph Transformation with Industrial Relevance (AGTIVE'07). LNCS 5088. Springer, 2008. http://www.springerlink.com/content/r27386x52l85/

[GP96] T. R. G. Green, M. Petre. Usability Analysis of Visual Programming Environments: a Cognitive Dimensions Framework. *Journal of Visual Languages and Computing* 7(2):131–174, 1996.

[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.

[JK05] F. Jouault, I. Kurtev. Transforming Models with ATL. In *MoDELS Satellite Events*. LNCS 3844, pp. 128–138. Springer, Berlin, 2005.

[KPPR07] D. Kolovos, R. Paige, F. Polack, L. Rose. Update Transformations in the Small with Epsilon Wizard Language. *Journal of Object Technology* 6(9):53–69, 2007. http://www.jot.fm/issues/issues_2007_9/paper3

[LMEP08] L. Lambers, L. Mariani, H. Ehrig, M. Pezze. A Formal Framework for Developing Adaptable Service-Based Applications. In *Proc. Fundamental Approaches to Software Engineering (FASE'08)*. LNCS 4961, pp. 392–406. Springer, 2008. http://www.springerlink.com/content/y24k478ww2212259/

[LS05] M. Lawley, J. Steel. Practical Declarative Model Transformation with Tefkat. In *MoDELS Satellite Events*. LNCS 3844, pp. 139–150. Springer, Berlin, 2005.

[MV08] B. Meyers, P. Van Gorp. Towards a Hybrid Transformation Language: Implicit and Explicit Rule Scheduling in Story Diagrams. In *Proc. of the 6th Int. Fujaba Days*. 2008.

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

[TV10] J. Troya, A. Vallecillo. Towards a Rewriting Logic Semantics for ATL. In *Proc. Int. Conf. on Model Transformation (ICMT'10)*. LNCS 6142, pp. 230–244. Springer, 2010. http://www.springerlink.com/content/a494202j7440q2h0/

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

Attribute Computations in the DPoPb Graph Transformation Engine

Hanh Nhi Tran, Christian Percebois, Ali Abou Dib, Louis Féraud, Sergei Soloviev

IRIT, University of Toulouse
118 Route de Narbonne, F-31062 TOULOUSE CEDEX 9
{tran, percebois, aboudib, feraud, soloviev}@irit.fr

Abstract: One of the challenges of attributed graph rewriting systems concerns the implementation of attribute computations. Most of the existing systems adopt the standard algebraic approach where graphs are attributed using sigma-algebras. However, for the sake of efficiency considerations and convenient uses, these systems do not generally implement the whole attribute computations but rely on programs written in a host language. In previous works we introduced the Double Pushout Pullback (DPoPb) framework which integrates attributed graph rewriting and computation on attributes in a unified categorical approach. This paper discusses the DPoPb's theoretical and practical advantages when using inductive types and lambda-calculus. We also present an implementation of the DPoPb system in the Haskell language which thoroughly covers the semantics of this graph rewriting system.

Keywords: attributed graph rewriting, attribute computation, algebraic graph transformation, Haskell language.

1.Introduction

The last decade shows a great interest in graph rewriting in MDE [15] as a model transformation technique. For these applications, it is important to use attributed graphs. There are several works on attributed graph transformations (see *e.g.* [11][12][13][3][8]) mostly based on the algebraic data types to implement attribute computations. In the algebraic approach, attributes are given as values within some Σ-algebras. Therefore, information is directly integrated in the graph structure by creating a new "attribute node" for each value of an algebraic sort. This approach is theoretically sound but shows some limits on the expressiveness of attribute computation and is especially difficult to be completely implemented. Thus most of the existing systems resting on the standard algebraic approach hardly respect the theoretical foundation. Consequently, the attributed graphs rewriting in these systems is validated theoretically but not practically.

In [5] and [6] we introduced DPoPb, a unified categorical model of attributed graph transformations using inductive types for attribute values and lambda-terms for computations. Keeping the same conceptual scheme as in the DPO constructions, the goal of DPoPb is to put the attribute computation to work in a more uniform way by staying within the same theory for implementing computations. We argue in this paper that the inductive types and lambda-calculus make attribute computations in DPoPb more expressive than in the DPO-standard model [1] and facilitate the implementation of attributed graph rewriting system. This claim is justified by an implementation of DPoPb engine in Haskell [2] which conforms entirely the theoretical model hence allows validating theoretically and practically the DPoPb attributed graphs rewriting.

The paper is organized as follows. In Section 2 we analyse the differences between the theoretical solutions for attribute computations in the DPoPb and the standard algebraic approach represented by the HLR framework [3]. Section 3 discusses the implementation of HLR and DPoPb graph rewriting systems. In Section 4, we sketch out the development of the DPoPb prototype in Haskell to validate the theoretical foundation. Finally, in the last section we discuss some current and future works to improve the DPoPb approach.

2.Attribute computations in the DPO and DPoPb approaches

In this section we compare the solutions for attribute computations in the HLR approach [3] with the DPoPb's one. We first outline attribution computations in each approach, and then we use an example to show their differences.

2.1. Attribute Computation in HLR framework

HLR framework [3] is representative for the algebraic approach attributed graph rewriting. This work is now very well-known so we just give an outline of the approach for attribute computation.

In order to model attributed graphs with attributes for nodes and edges, HLR extend the classical notion of graphs to E-graphs. An E-graph G has two different kinds of nodes, implementing the graph and data nodes, and three kinds of edges, the usual graph edges and special edges used for the node and edge attribution. An E-graph morphism f_G is defined then as a classical graph morphism. Let $DSIG = (S_D,OP_D)$ be a data signature with attribute value sorts $S'_D \in S_D$. An attributed graph $AG = (G,D)$ consists of an E-graph G together with a DSIG-algebra D such that $\uplus_{s \in S'_D} D_s = V_D$ where V_D is the set of data nodes of graph G. For two attributed graphs $AG^1 = (G^1,D^1)$ and $AG^2 = (G^2,D^2)$, an attributed graph morphism $f : AG^1 \rightarrow AG^2$ is a pair $f = (f_G, f_D)$ with an E-graph morphism $f_G : G^1 \rightarrow G^2$ and an algebra homomorphism $f_D : D^1 \rightarrow D^2$ such that the square in Fig. 1 commutes for all $s \in S'_D$, where the vertical arrows are inclusions.

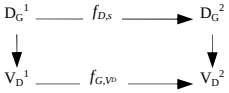
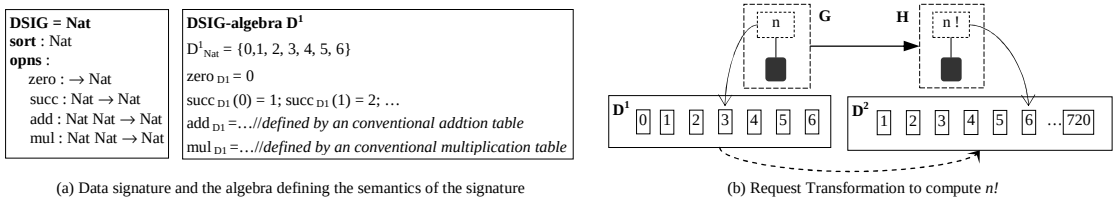


Fig. 1. The algebra homomorphism f_D in the attributed graph morphism f: $AG^1 \rightarrow AG^2$

Given a data signature DSIG as depicted, attributed graphs and attributed graph morphisms form the category *AGraphs* and graph rewrites can be realized by constructing the pushout of the category using the double pushout or simple pushout approach. In this section we consider the double pushout approach in HLR. A transformation rule $p: L \leftarrow K \rightarrow R$ is given by three attributed graphs (with variables) K, L, and R and two morphisms $l: K \rightarrow L$ and $r: K \rightarrow R$ which have to be injective on the graph structure and isomorphic over the $\sum$-algebra. In order to describe computations on the attributes, terms containing variables are to be used; *e.g.*, in the graph R, an attribute $x + y$ can be found if in the graph K the variables x and y are present.

To show how to combine graph transformations with computations on attributes in the HLR framework, we rest on an example for computing the value of *n!*. In this example, we use the signature *Nat* which defines the operators *zero, succ, add* and *mul* on the sort *Nat*. We can define two algebras D^1 and D^2 associated to the signature *Nat* to give two different semantics for *Nat*. For the illustration purpose, in this example, we use D^1 and D^2 which are different only on the carrier sets as shown in Fig. 2b. We can define then the attributed graph type $AG^1 = (G,D^1)$ to represent the graphs which can be attributed by the values from 0 to 6 and the attributed graph type $AG^2 = (G,D^2)$ to represent the graphs attributed by the values from 0 to 720.



(a) Data signature and the algebra defining the semantics of the signature

(b) Request Transformation to compute *n!*

Fig. 2. Calculating *n!* by graph rewriting and attribute computations in HLR

Given a value $n \leq 6$, let us suppose we compute the value of *n!* by graph rewriting in the system supporting the attributed graphs defined with the above *Nat* signature. We use the graph *G* of type AG^1 composed of one node having the attribute $n \in D^1_{Nat}$. The computation's result will be stored in the graph *H* of type AG^2 having also one node attributed by $n \in D^2_{Nat}$. Fig. 2b describes the needed transformation from *G* to *H*. However, with the signature *Nat* we cannot realize such a transformation because the factorial operator *!* is not defined. Consequently, we must decompose the computation of *n!* into many transformation steps, using the four rules represented in Fig. 3.

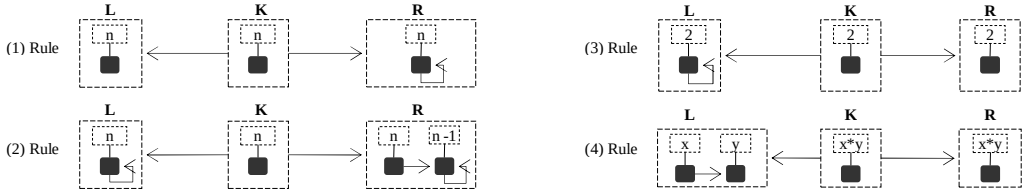


Fig. 3. Rules used for calculating *n!* in HLR

The first rule is applied once on the initial graph to prepare the list of number analyzing *n!*. Rule 2 is for delegating the computation of *n-1!* to a new node. This rule is applied until number 2 is reached. When the rule 2 is completed, a chain of nodes is created with the final node in still self-referred. To stop the delegation, the rule 3 is applied once to obtain a simple chain of number from *n* to 2. Rule 4 now is applicable: it takes the numbers of the last two nodes and multiplies them, then stores the result in the first of these two nodes, and deletes the very last one. Applying rule 4 as long as possible to obtain the result represented by only one node left, containing the computed result for *n!*. We can see that the computation of *n!* in this example with HLR system requires four transformation rules and *2n-3* steps.

For each application of a rule, the attributes of graph *L* must be updated to produce the graph *R*. In the literature, two main different approaches have been defined in order to specify attribute-value changes: relabeling attribute-nodes [11][26] and reconnecting attribute-value nodes [12][8]. In the relabeling issue, no built-in data types on labels are encoded and programs are in general based on rule scheme labelled with terms over algebras. On the opposite, in the reconnecting mode, a new edge is added each time the graph must reference a new attribute value. The example in Fig. 2b is illustrated with reconnecting scheme.

2.2. Attribute computation in the DPoPb approach

The DPoPb framework relies on the classical DPO approach for the structural part and uses type theory with inductive types for attribute computations. The precise definition of attributed graphs in the DPoPb approach can be found in [4][5][6]. Below we will give the essential information necessary to explain how attributes are implemented and computed in the DPoPb framework.

DPoPb Attributed Graphs and Attributed Graph Morphisms

In the DPoPb approach, an attributed graph is defined with two parts: a graph structure composed of labeled nodes and edges and a set of attributes associated to edges or nodes. DPoPb uses type theory to code attributed graphs: finite types to describe the structure of graphs and general inductive types to define data types and computations.

A morphism between two attributed graphs G and H is a 3-level structure morphism $f: G \rightarrow H$ defined by two components: the first component specifies the structural part of the morphism and the second one represents the attribute part of the morphism.

– The structural part, denoted f_S, a graph morphism from G to H, is the first level.
– The attribute part has two levels.
  - A relational level, denoted f_R. It includes the multirelation R between the attributes of H and G. For each attribute b of type B of the vertex s of G, a partition of the set $R_{\{s,b\}}$ of the H's attributes necessary to compute b. Each element of the partition (a subset of $R_{\{s,b\}}$) together with b defines a tree, and the set of all trees of f_R is called the forest of the morphism.
  - To each tree described above is associated a computation function represented by a lambda-term t in a way such that if the leaves of a tree are the attributes $a_1,\ldots, a_n$ of the types $A_1,\ldots, A_n$ respectively and its root is the attribute b of type B, then the type of the term is $A_1 \rightarrow \ldots \rightarrow A_n \rightarrow B$. The conditions to be satisfied is that $t(a_1,\ldots, a_n) = b$.

Usually the equality is the ordinary reduction-based equality of lambda-terms. Two morphisms f, g: $G \rightarrow H$ are considered as equal if all components on all levels are equal.

Fig. 4 shows an example of the DPoPb formalism representing attributed graph and attributed graph morphisms. In this example, we have a forest R composed of three trees T_1, T_2 and T_3 in which T_1 = *({7,8}, 15, λx y . x + y)*, T_2= *({"Good"}, 4, λs . length s)* and T_3 = *({"Good","Luck"}, "GoodLuck", λs1 s2 . s1 ++ s2).*

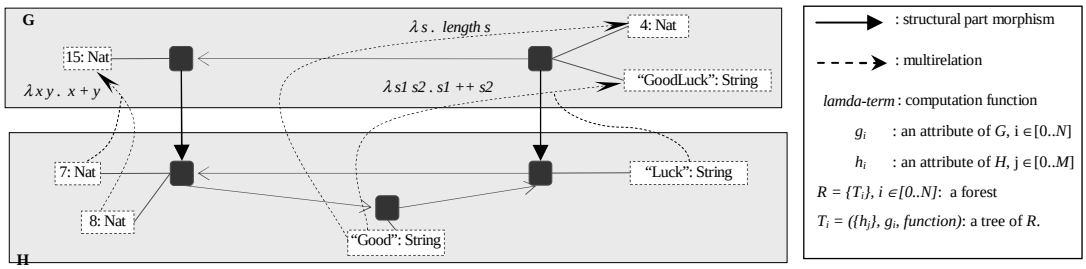


Fig. 4 An attributed morphism having a computation forest composed of three trees

The above definition of morphisms in DPoPb requires some comments on the reverse direction for the attribute relations and the role of partitions and associated trees in the attribute part of the morphism.

In our framework which uses the double pushout approach to rewrite graphs, the arrows for attribute parts are reversed with respect to the arrows of the structural parts. This reversal permits us to have a pseudo-pullback (pushout in dual category) to organize the computations with attributes. The main idea of changing the orientation of the arrows for the functions is to allow that the attributes in graph L can be stored in the graph K and then the attributes in graph R can easily "go and pick up" any value of attributes in the graph K. Because graph K is the intersection of graphs L and R, it contains only the common attributes of L and R. To preserve

information, we may need several computation functions converging to the same target. The use of multirelations follows naturally from the same assumptions. We assume that all lambda-terms (containing, probably, free variables) are defined in the same context that remains fixed (in principle, the context may be infinite). Thanks to this mechanism, several attributes in graph *L* can be computed into one attribute in graph *K*, and several attributes in graph *R* can share the same value of attributes in *K*.

Attributes computations in DPoPb

Now we take the same example presented in Section 2 on the computation of *n!* to illustrate the attribute computation in DPoPb. InDPoPb to calculate *n!* we need only one rule shown in Fig.5.

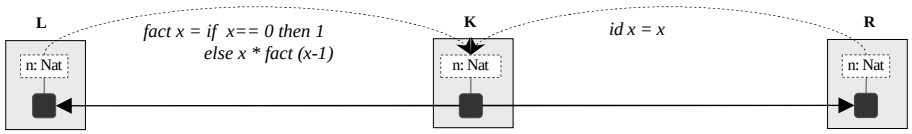


Fig. 5. Calculating *n!* by graph rewriting and attribute computation in DPoPb

The number *n* for which *n!* will be computed is specified as an attribute of type *Nat* associated to the unique structural node of the graph *L*. The computation of *n!* is realized by a graph rewrite which preserves the structure graph. So we just discuss here the computation to perform on attributes. The attribute part of the morphism $l: K \rightarrow L$ specified at two levels: a relation connecting the attribute *Nat* in *L* to the attribute *Nat* in *K* and a lambda-term associated to this relation to define the computation function realized during the graph rewriting. To simplify programs of Fig. 5, we replaced the lambda-term by the definition of a function *fact* which computes the factorial value for its parameter *x*. On the right-hand, the attribute part of the morphism $r: K \rightarrow R$ specified with a relation connecting the attribute *Nat* in *R* to the attribute *Nat* in *K* and an *id* function that allow copy the value of *Nat* in *K* to *Nat* in *R*. Given an initial graph *G* with a concrete value of *n*, the lambda-term of *l* will be applied to *n* to produce the result *n!*. This computation is performed by the β-reduction mechanism which substitutes the effective value of *n* for the formal variable *x* in the term. So the computation of *n!* in this example with DPoPb system requires only one transformation rule and one realizing step.

As seen in the illustrating example of this section, in certain cases, the systems based on ∑-algebras cannot represent directly complex computations on attributes if the operators used in the computation are not defined in the supporting ∑-algebras. In such cases, users have to decompose the computation into several rules (e.g. as illustrated in Fig. 2).

In DPoPb, computations are based on lambda-calculus [25], a formal model for computations. Let us recall that in this system, we can express computations with lambda-terms which allow representing the terms (*s*), the function abstractions (λs . *t*) and the function applications (*t s*). Lambda-terms then can be evaluated by the simple and powerful β-reduction mechanism based on substitutions. This reduction mechanism is semantically defined and can be easily implemented by a computer program. Thanks to this generic model of computation, the DPoPb approach can allow a more natural and straightforward way to represent complex computations defined only by abstraction and application of functions (e.g. as shown in Fig. 5).

3. Implementation of HLR and DPoPb

In this section, we compare the potential implementations of HLR and DPoPb with respect to their computational models. First we analyse the requested effort to implement exactly the foundation models of each approach. Then we discuss the solutions used by some significant systems to implement the HLR framework, as well as our solution for implementing the DPoPb framework. Discussions show the distance between the implementation and the formal model in each approach and some important side-effects raised in the tools resting on HLR.

3.1. Underlying mechanism for the transformation engine

In the HLR approach [3], attribute values are defined by separate data nodes which are elements of some algebras. When attributed graphs and graph morphisms are considered over Σ-algebras, operations and constants defining the algebra must be always present and thus previously defined. Of course, this is practically impossible because it is very cumbersome to implement large graphs and unattainable for infinite graphs. In a tool implementation this problem could be solved by including the attribute values of the algebra graph that are directly reachable from the structural part of the graph. Consequently, most of the systems based on the approach HLR use an object-oriented programming language to implement attribute computations. Concretely, pre-conditions, post-conditions and actions are mainly expressed in an object-oriented programming language: Java for AGG [1] and Fujaba [21], C++ for GReAT [20], and Python for AToM^3 [10]. Besides this popular solution, actions changing the models are sometimes coupled to the rule selection process as in VIATRA [17] which supports ASMs or in VMTS [18][19] where UML-like models are manipulated owing to stereotyped activity diagrams, XSLT and (Imperative) OCL.

Let's consider a system using an external language to express actions as well as conditions on attributes, for instance AGG tool. In AGG, graphs are attributed by Java objects which can be instances of Java classes from libraries like JDK or from user-defined classes. The main difference with the formal system is the use of Java classes and expressions instead of algebraic specifications and terms. Thanks to some interoperability with the host language, the obtained system is a general purpose graph transformation tool covering a large variety of applications including graph transformation. However, classes of the underlying object-oriented language whose semantics is not covered by the formal foundation belong to applications as well.

A guiding principle of DPoPb is to propose a close relationship between the formal ground and its underlying engine. The lambda-calculus which formalizes the algorithmic notion of a function is proposed as a model of attribute computations. Existing graph nodes describing attributes thus can be reused and updated thanks to lambda-terms implementing the attribute computations. Lambda-terms can be easily expressed in a functional programming which is based also on lambda-calculus. Such an implementation preserves the semantics of the formal model, and provides static strong typing, polymorphism, higher-order functions and lazy evaluation for the graph rewriting system. For implementing the DPoPb prototype, we chose the Haskell language and benefit all of these advantages of the lambda-calculus paradigm.

3.2. Types declarations of attributed graphs

In HLR graph transformation tools, an attribute is often declared as a variable in a conventional programming language. For instance, in AGG, an attribute is implemented by a Java variable which can be assigned to any value conforming to its type. Because users can use any Java

acceptable expressions to compute attributes' value, the Java type system defines the type system of the graph rewriting. This issue is not specific to AGG; it exists also in other known graph transformation systems such as Fujaba, GReAT, AToM^3... A strongly typed language such as Java is considered useful to reinforce the security of programs by preventing programmers from making freely mistakes. In fact, this statement is not true in certain cases. For example, a class in Java is perhaps a wrong subtype of its superclass. In order to be a subtype, the methods of the subclass must satisfy the superclass' specifications. This relation cannot be checked at compile-time, so it is possible to create a subclass that is not a subtype [22]. Hence the type system used by the graph transformation scheme where each attribute has a name, a type and a value can be unsecure.

For instance, let us consider the two following classes *Integer* and *MyInteger* in Java. *MyInteger* looks like an *Integer* by adding an attribute which specifies a name *s* for the value *v*:

```
public class Integer {
        private int v ;
        public Integer (int v) {…}
        public boolean equals (Integer i) {...}
}

public class MyInteger {
        private int v ;
        private String s ;
        public MyInteger (int v, String s) {…}
        public boolean equals (MyInteger i) {…}
}
```

MyInteger is not a subtype of *Integer*. To insure subtyping, we need that *MyInteger* must have a stronger specification than *Integer*. This is not the case because the type of the parameter of the *equals* method of *MyInteger* should be at most as strict as in the supertype. Using the Java's *extends* relation between an *Integer* and *MyInteger* is also not appropriate with respect to subtyping. This is mainly because a C++ or Java class defines at the same time attributes (state) and methods (behaviour). Subclasses are not subtypes. Consequently, an *Integer* object cannot be dynamically substituted by a *MyInteger* one during the rewriting process.

In contrast to these systems, the DPoPb can avoid such kinds of problem on the type system. In Haskell, a safe polymorphic type system is supported by a powerful type inference algorithm. A type specification is separated from its methods (functions). A class specifies the operations that the types must support. It's a template for types. A type is said to be an instance of a class if it supports these operations. For instance, here is the (incomplete) *Eq* class from the Standard Prelude defining the == (equals) and /= (different) functions.

```
class Eq a where                        -- a is an instance of Eq
(==), (/=) :: a -> a -> Bool            -- if a implements == and /=
x /= y   =   not (x == y)

data MyInteger = MyInteger {v :: Integer, s :: String}

instance Eq MyInteger where
   (MyInteger v1 s1) == (MyInteger v2 s2)    =   (v1 == v2) && (s1 == s2)
```

This code defines *MyInteger* as a data type which wraps an *Integer* presenting the value *v* and a *String s* representing the name of the value. This type is then considered as an instance of *Eq*. The three definitions (class, data type and instance) are completely separated and there is no rule about how they are grouped.

We think that this separation is more secure than actual attribute declarations in an object-oriented host language because well-typed lambda-terms are always well-behaved with respect to reduction. In addition, all the types associated with a function definition can be checked at

compile-time, and inferred automatically. To take a full advantage of the typed lambda-calculus, an attractive perspective of DPoPb is to define type checking rules between the types of the computation functions and the types of the attributes of the attributed graph.

3.3. Attributes computations

Loading compiled codes

As previously stated, several transformation systems rely upon an underlying language for the specification of textual constraints and attribute updates. These definitions have to be compiled and provided to the graph transformation machinery in the form of a dynamic library, which is loaded at runtime. Within the transformation environment, it is quite easy to propose a special attribute editor that pops up when a graph object is selected for attribution. However, the user cannot directly access to the code of the function dealing with these attributes. As the function is considered as a black box, round trips between the host language and the graph rewriting tool are necessary to finalize the computation code. Each round trip is translated by a compilation process in the host language.

In DPoPb, the use of a unified formalism based on type theory for manipulating attributes enables a reliable environment so that both structural and attribute manipulations are handled in the same framework. The β-reduction mechanism used to evaluate lambda-terms can be easily implemented. Implementation which allows compiling and dynamically evaluating attribute computations is then possible. In Section 4, we will show how this capability is instantiated for the DPoPb's implementation.

Lazy evaluation

Another relevant feature for attribute computations is about lazy evaluation. Using this technique, no expression is evaluated until its value is needed and no shared expression is evaluated more than once. Lazy functions, also called non-strict, only evaluate their arguments when needed. On the opposite, C functions and Java methods are strict and evaluate their arguments in an eager mode. Lazy evaluation makes it possible for functions to manipule infinite data structures. This interesting feature enables us to describe an object without being tied to one particular application of that object.

For comparison purposes, let us consider an infinite list of integers starting from a given value. Such a lazy list can be represented in Java as a process [23] which returns objects either forever, or until no more are left[1]:

```
public interface Process {
        public Object nextElement () throws NoSuchElement;
}
public class NumFromProcess  implements Process {
        private int upto;
        public NumFromProcess (int n) {
                this.upto = n;
        }

        public Object nextElement () {
                return new Integer (this.upto ++);
        }
}
```

In Haskell language, for the same construction, we simply define:

[1] Java codes are extracted from [23]

```
numFrom n = n : numFrom (n + 1)
```

Extracting a finite list from *NumFromProcess* implies to manage exception handlings because we don't have a method to explicitly test for the presence of the next element. This test is assumed by throwing a *NoSuchElement* exception when the *nextElement* method is invoked. For instance, the Java following class *SingleProcess* computes a *Process* producing only one object:

```
public class SingleProcess  implements Process {
        private Object item;
        public SingleProcess (Object item) {
                this.item = item;
        }
        public Object nextElement () throws NoSuchElement {
                if (item == null) throw  new NoSuchElement ();
                else {
                        Object temp = item;
                        item = null;
                        return temp;
                }
        }
}
```

In addition, the *Process* interface defines a lazy list that is consumed as fast as it is produced and a shared expression is evaluated more than once. If previous elements need to be saved, then the programmer must add classes to store computed values in a structured data type.

In comparison, this mechanism is intrinsically supported by Haskell thanks to lazy evaluation. The following Haskell function *f* extract with *take* a list containing the successives values of the factorial computations, starting from 1! until *n*!. This is done by first building the infinite list *nats* and then applying the *fact* function to the obtained *nats* list. With the same technique, we build the infinite list *facts* of factorials. In this code, the *map* function is a higher-order function which goes through every element of a list and applies a function given by its first argument: + in the case of *nats* and *fact* for the list of factorials.

```
f n = take n facts
nats = 1 : map (+1) nats
```

```
facts = 1 : map (fact) nats
fact 0 = 1
fact n = n * fact (n -1)
```

With respect to functional programming languages, Java lacks some conciseness. Some libraries have been proposed to implement the lazy-evaluation mechanism for object-oriented environments. For instance Lambda4J [24] provides lazy lists and associated operations. More recently, LazyJ [23] extends Java's type system with lazy types. Besides expressiveness, a major challenge with lazy evaluation concerns sharing computation results. In all relevant functional language implementations, terms are represented as a graph. In the future, we would like to establish mappings between rewriting such terms and rewriting terms in an attributed graph.

4. The DPoPb prototype

To validate the theoretical model DPoPb, we have developed a prototype in Haskell language. In the first time, the goal of this prototype is to construct the basic DPoPb categorical concepts for attributed graphs rewriting when focusing on the implementation of attribute computation. Fig. 6 displays the architecture of our prototype.

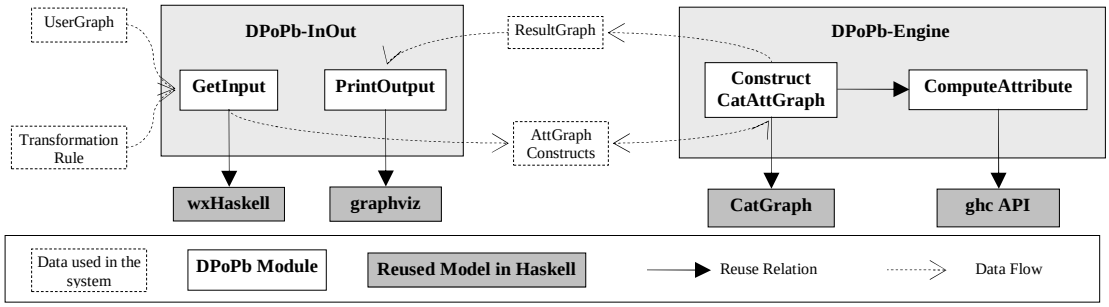


Fig. 6. Architecture of the DPoPb prototype

DPoPb prototype is a general purpose graph rewriting system composed of two modules *DPoPb-InOut* and *DPoPb-Engine*. The module *DPoPb-InOut* provides an interface for the prototype. It allows users to specify transformation rules and initial graphs (via the sub-module *GetInput*) as well as visualize the result of the transformation (by the sub-module *PrintOutput*). At the current stage of development, we base on the Haskell predefined modules *wxHaskell* and *graphviz* (defined in the Hackage Database [16]) for the graphical user interface and the graph visualization respectively.

The *DPoPb-Engine* module is the kernel of our prototype. It contains two sub-modules: *ConstructCatAttGraph* and *ComputeAttribute*. The sub-module *ConstructCatAttGraph* implements the main concepts of DPoPb including the colimits of the category of DPoPb attributed graphs (*CatAttGraph*) as well as the graph rewriting based on the approach double pushout. The sub-module *ComputeAttribute* supplies the utilities functions concerning attribute computations during the rewriting (*e.g.* the composition the attribute part of attributed graph morphims which is needed in the construction of *CatAttGraph* pushout; the dynamic evaluation of lamda-expressions representing attribute computations). In *ConstructCatAttGraph*, when constructing the structural part of the colimits we reused *CatGraph*, the implementation of Schneider [7] defining the colimits of the category of graphs. We also reuse some functions in the API of Glasslow Haskell Compiler to support the dynamic specification and evaluation of attribute computations.

Our main contributions in the development of the prototype concern solutions for the following theoretical and technical questions:

How to implement the theoretical concepts of DPoPb?

The mathematical model in [4] provided a formal framework for the category of attributed graphs *CatAttGraph* but it is not straightforward to map those categorical concepts into computational constructs. Thus, we had to define the constructive data structures and algorithms for storing graphs and graphs morphisms; for constructing the coproduct, coequalizer, pushout complement and pushout of the category *CatAttGraph*. The difficulty of this task resides in defining the attribute part of the graphs and graphs morphisms in the construction of each colimit such that its categorical properties are satisfied.

How to evaluate users-defined attribute computations at runtime?

We want to allow users to define their graphs together with attribute computations as non-compiled functions (written in Haskell for example). The challenge here is that at runtime the rewrite engine must enable the generation of Haskell codes of user-defined functions and integrate these codes into the engine in order to evaluate them in the rewrite process. Thus we

need to support the meta-programming at runtime. To support this flexibility, we relied on the Glasgow Haskell Compiler (GHC) which proposes the necessary API functions to compile and evaluate Haskell functions. We used Haskell module hint 0.3.2 [16] wrapping those GHC functions to invoke the GHC compiler at runtime.

How to update graphs during the rewriting?

The double pushout rewriting process necessitates an update of the transformation rules when the content of an initial graph is given. Using an imperative language, such an update is not difficult. However, the update implemented in an imperative language is undefined semantically and then out of control. Haskell is a pure functional language that does not allow side-effects. Hence we must ensure that computations with side-effects for the update of attributes during the rewrite process will be encapsulated to respect the functional style of the program. For this purpose, we base on monads [14]. We defined a monad transformer (*State transformer*) that enables hiding underlying machinery for updating graphs during the DPoPb process. The main interest here is that we can allow the update operations during the rewrite term process without losing the advantages of the functional paradigm and the Haskell type system.
We now show in Fig. 7 some screenshots of our prototype.

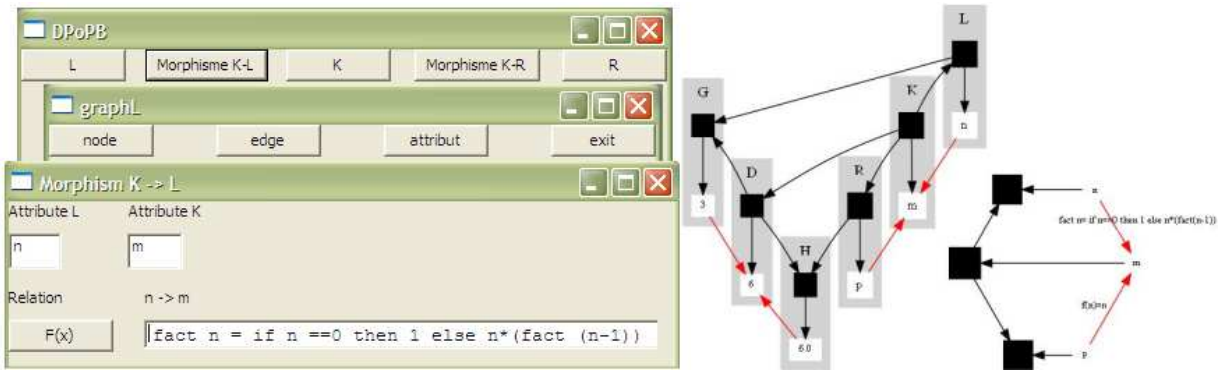


Fig. 7. DPoPb prototype's screenshots

In the left-hand are the widgets used to receive inputs including the transformation rules and the initial graph. Actually we do not implement an algorithm to find a match, so the initial graph *G* is took as an instance of graph *L* defined with concrete attributes' values. The frame *Morphism* $K \rightarrow L$ shows how the attribute part of the morphism is specified with the attribute relations and the computation functions. The input information will be encoded in the internal data structure and manipulated by the DPoPb engine module to construct the pushout complement graph *D* and the pushout graph *H*. To visualize DPoPb graphs, currently we rely on the GraphViz system [28]. The DPoPb internal data structures are thus translated to the graph descriptions in the DOT language (by using the Haskell module *graphviz* [16]) which can be displayed with GraphViz as shown in the right-hand of the figure. On the right side of this part, we display the graph representing the rules $L \leftarrow K \rightarrow R$. The attributes *n* of *L* and *p* of *R* are connected to the attribute *m* of *K* by the function factorial and the function identity respectively. The graph on the left side shows the pushout complement *D* and the pushout *H* constructed by applying the rule $L \leftarrow K \rightarrow R$ on the concrete graph *G*. Since the value of *G*'s attribute is 3, the value of *D*'s attribute is 6 - factorial of 3. The value of *H*'s attribute is the copy of *D*'s attribute, thus it is also 6.

5. Conclusion

In the HLR framework, attributed graph structures are given by algebras over a specific signature where the structure part and the attribute part are separated from each other. If this solution is theoretically acceptable, it is not very efficient and cannot be easily implemented for a general purpose graph transformation system. Consequently, users have to program and compile computation functions separately within a companion programming tool before integrating their functions into transformation rules.

Contrary to HLR approach, our approach proposes a single formalism that integrates the rewrite of structural parts of graphs with attribute computations. This solution rests on category theory and type theory thus doesn't entail a semantic gap between the theoretical model and its implementation. This advantage has been validated by an ongoing–developed prototype of the DPoPb system implemented in the Haskell language.

We have identified two directions for future researches. The first one concerns proving properties of transformations. As we want keeping trace of evaluation or verification of the correctness of attributes' computations during the transformation, we plan to import transformations supported by our DPoPb tool into the Isabelle/HOL proof assistant via Haskabelle [26] in order to specify and prove relevant properties the transformations have to satisfy. Another direction deals with reasoning on programs transformations. It relies on the use of functional programming languages for programming applications based on rewriting attributed graphs. As these languages promote a more abstract style of programming and support higher-level constructions, we have in mind simplification of programs transformation and to cope with them as functional programs.

References

[1] AGG Homepage, http://tfs.cs.tu-berlin.de/agg/

[2] Haskell Homepage, http://www.haskell.org/

[3] Hartmut Ehrig, Ulrike Prange, and Gabriele Taentzer. Fundamental theory for typed attributed graph transformation. In Hartmut Ehrig, Gregor Engels, Francesco Parisi-Presicce, and Grzegorz Rozenberg, editors, ICGT, volume 3256 of LNCS, pp. 161-177, Springer, 2004.

[4] Maxime Rebout. Une approche catégorique unifée pour la réecriture de graphes attribuées. PhD thesis, Université Paul Sabatier, 2008.

[5] Maxime Rebout, Louis Féraud, and Sergei Soloviev. A unifed categorical approach for attributed graph rewriting. In E.Hirsch, A.Razborov, A.Semenov, and A.Slissenko, editors, CSR, volume 5010 of LNCS, pp. 398-409, Springer, 2008.

[6] Maxime Rebout, Louis Féraud, Lionel Marie-Magdeleine, Sergei Soloviev. Computations in Graph Rewriting: Inductive types and Pullbacks in DPO Approach. In IFIP TC2 Central and East European Conference on Software Engineering Techniques (CEE-SET 2009), Krakow, Pologne, 2009.

[7] Hans Jurgen Schneider. Implementing the Categorical Approach to Graph Transformations with Haskell. In An Introduction to the Categorical Approach, Draft March 7, 2007.

[8] Harmen Kastenberg. Towards Attributed Graphs in Groove: Work in Progress. Electr. Notes Theor. Comput. Sci. 154(2), pp. 47-54, 2006.

[9] Schurr, A. Introduction to PROGRES, an Attribute Graph Grammar Based Specification Language. Proc. WG89. LNCS 411, pp. 151-165, Springer, 1990.

[10] de Lara, J. and Vangheluwe, H. AToM3: A Tool for Multi-Formalism Modeling and Meta-Modelling. Proc. FASE'02, LNCS 2306, pp. 174-188, Springer, 2002.

[11] Lowe, M., Korff, M., Wagner, A. An Algebraic Framework for the Transformation of Attributed Graphs. In Term Graph Rewriting: Theory and Practice, John Wiley and Sons Ltd. (1993), pp. 181-1993.
[12] Heckel, R., Küster, J., Taentzer, G. Confluence of Typed Attributed Graph Transformation with Constraints. In Proc. ICGT 2002, Volume 2505 of LNCS, Springer (2002), pp. 161-176.
[13] Berthold, M., Fischer, I., Koch, M. Attributed Graph Transformation with Partial Attribution, Technical Report 2000-2, 2000.
[14] P. Wadler. Monads for Functional Programming. In *Advanced Functional Programming*, Springer Verlag, LNCS 925, 1995.
[15] Bézivin, J. On the Unification Power of Models. Software and System Modeling (SoSym) 4(2):171-188, 2005.
[16] Hackage Database http://hackage.haskell.org/packages/hackage.html
[17] Varro, D., Pataricza, A. Generic and meta-transformations for model transformation engineering. In Baar, T., Strohmeier, A., Moreira, A., Mellor, S., eds., Proc. UML 2004, 7th International Conference on the Unified Modeling Language, Lisbon, Portugal, Springer (2004), pp. 290-304.
[18] VMTS Web Site, http://avalon.aut.bme.hu/_tihamer/research/vmts
[19] Levendovszky T., Lengyel L., Mezei G., Charaf H. A Systematic Approach to Metamodeling Environments and Model Transformation Systems. In VMTS, 2nd International Workshop on Graph Based Tools (GraBaTs); workshop at ICGT.
[20] Daniel Balasubramanian, Anantha Narayanan, Chris vanBuskirk, and Gabor Karsai. The Graph Rewriting and Transformation Language: GReAT. Proceedings of the Third International Workshop on Graph Based Tools (GraBaTs 2006); workshop at ICGT.
[21] Robert Wagner. Developing Model Transformations with Fujaba. In Proc. of the 4th International Fujaba Days 2006, Bayreuth, Germany (Holger Giese and Bernhard Westfechtel, eds.), vol. tr-ri-06-275 of Technical Report, pp. 79-82, University of Paderborn, September 2006.
[22] Sophia Drossopoulou, Susan Eisenbach. Java is Type Safe – Probably. European Conference on Object Oriented Programming, 1997.
[23] Dekker, Anthony H. Lazy functional programming in Java. SIGPLAN Not. Volume 41, Number 3, pp. 30-39, 2006.
[24] Lambda4J Web Site. http://www.nongnu.org/lambda4j/
[25] Henk Barendregt, Erik Barendsen. Introduction to Lambda Calculus, 1994.
[26] Plump, D. and S. Steinert. Towards Graph Programs for Graph Algorithms. In ICGT, LNCS 3256, pp. 128-143, Springer, 2004.
[27] Haskabelle website : http://www.cl.cam.ac.uk/research/hvg/isabelle/haskabelle.html
[28] GraphViz website : http://www.graphviz.org

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Enabling Graph Transformations on Program Code

Michael Striewe, Moritz Balz, and Michael Goedicke

12 pages

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

Enabling Graph Transformations on Program Code

Michael Striewe, Moritz Balz, and Michael Goedicke

Paluno – The Ruhr Institute for Software Technology
University of Duisburg-Essen, Essen, Germany
{michael.striewe, moritz.balz, michael.goedicke}@s3.uni-due.de

Abstract: Although the internal representation of program code in parsers and compilers is the abstract syntax tree and thus a graph, tools for handling program code as an explicit graph are rare. This contribution introduces a tool that generates abstract syntax graphs out of Java program code. Code can be read and stored as a graph, and code can be manipulated by the application of graph transformations. We show by examples how this can be used for low-level analysis and manipulation as well as for code interpretation at different levels of abstraction with formal models.

Keywords: Abstract Syntax Graph, Code Manipulation, Model Transformation, Graph Pattern Matching

1 Introduction

Program code is usually represented in a textual notation. Consequently, any manipulation on it can be expressed in terms of creating, replacing, moving, or deleting strings. However, the internal representation of program code as used by parsers and compilers is the notation of an abstract syntax tree, which is based on the type graph given by the grammar of the respective programming language. Reverse engineering and refactoring tools usually traverse this data structure for their manipulations. Since traversing is only one limited possibility to handle a graph, it is appealing to make this graph structures *explicitly* available for a general-purpose graph transformation engine. By this means any kind of code manipulation can be expressed as graph transformation rules. This is of interest whenever program code is to be analyzed, manipulated, or transformed according to well-defined rules.

In this contribution we present approaches that enable the use of graph transformation techniques for program code written in the programming language Java. Code can be read and stored as a graph, and code can be manipulated by the application of graph transformations. This contribution is organized as follows: Section 2 describes JAVA2GGX, which is the actual tool that makes Java syntax graphs available to a graph transformation engine. Sections 3 and 4 discuss low-level and high-level transformations of program code based on examples. The remaining two sections provide a brief overview about related work and draw conclusions.

2 The Basics: JAVA2GGX

In order to make Java source code available for graph transformation techniques, the explicit generation of syntax graphs is necessary. Our graph-based tools for manipulating and analyzing

```
public class SampleClass {
  public static void main(String[] args) {
    new AnotherClass(1).talk();
  }
}
```

Listing 1: Source code of SampleClass.java used as example in this section.

```
public class AnotherClass {
  public AnotherClass(int i) { }

  public void talk() {
    System.out.println("Hello world!");
  }
}
```

Listing 2: Source code of AnotherClass.java used as example in this section.

Java code are based on a component named JAVA2GGX, which in turn is based on AGG [AGG] and its file format GGX. In our case, JAVA2GGX uses the AGG programming interface (version 1.6.4) as an underlying engine for all graph transformation activities.

Abstract syntax trees are generated by a parser while processing source code. They reflect the structural elements of the source code using the grammar of a certain programming language. JAVA2GGX is capable of parsing Java files to their syntax tree according to the Java Language Specification for Java 6. Each structural element is represented by a node which is of a certain type and which possibly has attributes. Each connection between nodes is made up by directed arcs which are also typed and may have attributes.

Although an abstract syntax tree reflects all structural properties of source code, it is not always comfortable to work with a plain abstract syntax tree. For this reason JAVA2GGX automatically performs some modifications during parsing. These modifications introduce both new attributes and new arcs which implies that the result is no longer a tree but a graph. All modifications are explained in detail in the following.

An example of source code and the extracted graph is shown in listings 1 and 2 and figure 1. The layout has been done manually in AGG. Names for node types are identical to the class names used inside the parser for representing the abstract syntax tree. Arcs in blue color are additional arcs introduced by JAVA2GGX and thus not part of the parsed abstract syntax tree. They will not be considered if code is generated or rewritten from a graph.

2.1 Simplified Naming

According to the Java language specification, names are assigned to elements by referring to an element of type `SimpleName` or `QualifiedName`. In the syntax tree this is represented by outgoing arcs of type `name` towards nodes of type `SimpleName` or `QualifiedName` respectively, which contain an attribute for the name. While this structure is reasonable for language design, it adds an overhead of nodes and arcs to the graph. For this reason the graph is

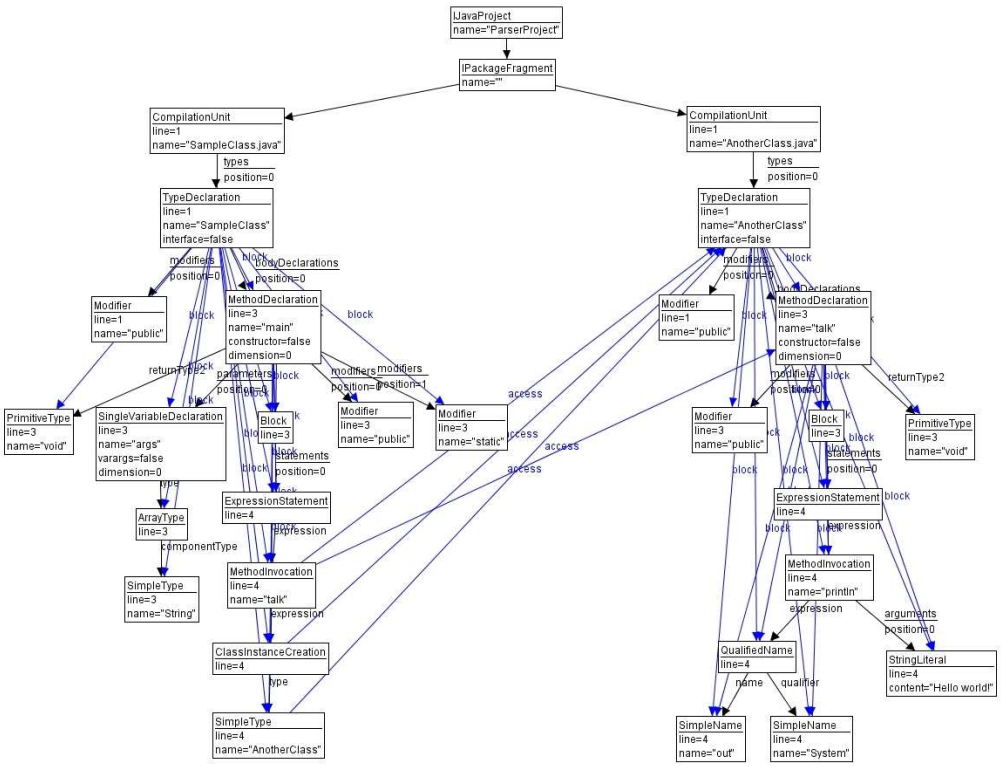


Figure 1: Sample graph generated by JAVA2GGX from the source code shown in listings 1 and 2.

simplified by assigning name attributes to several types of nodes directly, thus omitting the extra nodes for this purpose. In detail, this applies to the following structures:

- Nodes of type `QualifiedName` are omitted if they own the name property for nodes of type `ImportDeclaration` or `PackageDeclaration`. Instead, an attribute `name` is given to these nodes directly.

- Nodes of type `SimpleName` are omitted if they own the name property for nodes of type `MethodDeclaration`, `MethodInvocation`, `PackageDeclaration`, `SimpleType`, `SingleVariableDeclaration`, `TypeDeclaration` or `VariableDeclarationFragment`. Instead, these nodes are given an attribute `name` directly.

2.2 Line Numbers

Since the parser used inside JAVA2GGX provides information about the line number of each structural element, this information is included into each node. Thus each node owns an additional attribute containing the line number that is the origin of this element inside the respective source code file.

2.3 Blocks

Some Java constructs like methods or loops have a so-called "body" that is a block of statements. More precise, elements that span out a block have either outgoing arcs of type `bodyDeclarations` to each declaration statement inside the block or an outgoing arc of type `body` to an explicit `Block` node. By following arcs reverse to their direction until reaching arcs of the types named above, a path to the roots of blocks for each element can be found.

However, AGG does not support expressing paths in rules. In order to enable rules that consider block structures, JAVA2GGX introduces additional arcs of type `block`. They are drawn from a node s to a node t if there is a path from s to t where the first arc on the path is of type `body` or `bodyDeclarations`.

The example in figure 1 has five elements spanning out a block: Two type declarations and three method declarations. All elements inside these blocks are connected to their root elements by an additional blue arc of type `block`. Note that class modifiers are not part of the block and hence not connected by an additional arc. The same applies to method modifiers: They are not connected to the method declaration node by a blue arc, but belong to the elements inside the block of the surrounding type and hence have an incoming arc from there. Finally, elements that belong to the block of a method have two incoming arcs of type `block`, one from the method declaration and one from the surrounding type declaration.

2.4 Access Arcs

During parsing, names are resolved to references between syntactical elements. The parser assigns a unique identifier to each declaration of a type, method, or variable. Consequently, each call to a type, method, or variable uses this unique identifier as reference. This way the parser detects missing or duplicate declarations as well as global variables hidden by local declarations. JAVA2GGX displays these references between elements by introducing additional arcs of type `access`. There are several situations in which these arcs are drawn:

- A node of type `ClassInstanceCreation` is considered as a source of an additional arc that points to a node of type `MethodDeclaration` (which has its attribute `constructor` set to `true`) if the class instance creation uses a constructor explicitly given in the source code.

- A node of type `MethodInvocation` is considered a source of up to three additional arcs that point to a node of type `MethodDeclaration`, its parenting `TypeDeclaration`, and the related `PackageDeclaration`. The latter are only drawn if they do not point to parents of the `MethodInvocation`. However, a reference from `MethodInvocation` to `MethodDeclaration` is always drawn, thus possibly indicating a recursive method. In addition, the parenting `TypeDeclaration` of the `MethodInvocation` is also considered a source of another arc of type `access` pointing to the parenting `TypeDeclaration` of the `MethodDeclaration`. Thus types referencing each other can be identified without considering all their declared methods.

- A node of type `SimpleName` is considered a source of up to three additional arcs. Since nodes of type `SimpleName` can occur in different situations, three cases have to be distinguished:
  - If the name refers to a type, two arcs may point to the `TypeDeclaration` and the related `PackageDeclaration`, but are only drawn if they do not point to parents of the `SimpleName` itself.
  - If the name refers to a class member variable, one arc points to the related `SingleVariableDeclaration`. Another arc points to the parenting `TypeDeclaration`. A third arc points to the related `PackageDeclaration` if it is not the same as the one related to the `SimpleName` itself. In addition, the parenting `TypeDeclaration` of the `SimpleName` is also considered a source of another arc of type `access` pointing to the parenting `TypeDeclaration` of the `SingleVariableDeclaration`.
  - If the name refers to a local variable of a method, a single arc points to the related `SingleVariableDeclaration` or `VariableDeclarationFragment`.
  - If the name refers to an enumeration constant, a single arc points to the related `EnumConstantDeclaration`.
- A node of type `SimpleType` is considered a source of an arc pointing to the related `TypeDeclaration`. Another arc points to the related `PackageDeclaration` if it is not the same as the one related to the `SimpleType` itself. In addition, the parenting `TypeDeclaration` of the `SimpleType` is also considered a source of another arc of type `access` also pointing to the `TypeDeclaration`.

In figure 1 one can see five blue arcs of type `access`. All are pointing from the left subgraph to the right subgraph. The arc most easy to understand is the one from `methodInvocation` on the left to `methodDeclaration` on the right. Obviously the method "talk()" called in line 4 of SampleClass.java is the one declared in line 6 of AnotherClass.java, which is expressed by this arc. As explained above, a node of type `MethodInvocation` may be source of up to three arcs. Another one is visible in the example, pointing towards a node of type `TypeDeclaration`. As one can see, this node represents the type in which the method "talk()" is declared.

Another arc of type `access` leads from a node of type `ClassInstanceCreation` to a node of type `MethodDeclaration`. In this case the target is the constructor used for this class instance creation as declared in line 3 of AnotherClass.java. The type used in this class instance creation is attached to the node of type `ClassInstanceCreation` as another node of type `SimpleType`. This node also has an outgoing arc pointing to the node of type `TypeDeclaration` introduced above. In this case the arc means that in line 4 of SampleClass.java a class is instantiated which is declared in line 3 of AnotherClass.java.

Up to here, all arcs started at nodes representing a certain statement with a direct reference to a method or type. As explained above, parenting nodes are also considered by JAVA2GGX. Hence a fifth arc of type `access` exists in figure 1. It leads from the node of type `TypeDeclaration` on the left to another node of same type on the right. It is introduced since some elements in

the subtree on the left side access either the node on the right directly or access elements from its subtree. Hence by only looking at this arc you can already see that the type "SampleClass" accesses "AnotherClass" in some way.

2.5 Termination Conditions

In loops of type "for", "while", and "do ... while" there is an expression providing an explicit termination condition. In the AST nodes of type `ForStatement`, `WhileStatement`, and `DoStatement` have an outgoing arc of type `expression` leading to the root of the subtree representing the termination condition. To gain explicit information about nodes used in this subtree, additional arcs of type `termination` are introduced by JAVA2GGX during parsing for all nodes inside this subtree including the root itself. Thus, whenever there is a path from a node s of type `ForStatement`, `WhileStatement`, or `DoStatement` to another node t with the first arc of the path being of type `expression`, there will be an additional blue arc of type `termination` leading from s to t.

3 Low-level Transformations and Analysis

As the abstract syntax graph is a detailed representation of software, it allows for transformations and analysis on the level of single source code elements and small patterns of few elements. We refer to these as low-level transformations and analysis in the following, in contrast to high-level transformations and analysis related to abstract specifications which we will discuss in section 4.

The goal of low-level analysis of program code as discussed in this paper is to determine whether certain simple constructs expressed by patterns are present or absent in the given source code. This allows to check for style and programming conventions, e.g. searching for empty `catch` clauses or unused method parameters. Since this is known as "static analysis" of source code, it is no particular feature of JAVA2GGX, but can also be done with many other typical tools for static code analysis like CHECKSTYLE [Che] or PMD [PMD]. These tools also use the abstract syntax graph. However, they do not handle graphs as an explicit data structure, but traverse the AST programmatically. Contrary, patterns to be searched by JAVA2GGX have to be defined as graph transformation rules instead of relying on programming, thus taking advantage of the descriptive rules. Since these rules cannot be implemented straightforward, one of the following strategies is to be used:

- Optimistic rules assume that a given piece of code is correct unless an undesirable structure is present. In this case, the erroneous structure is placed on the left hand side (LHS) of the rule. The right hand side (RHS) contains the same structure since changes to the code are not intended. An additional "error node" is added there, which is a node of a fixed type that does not appear in normal syntax graphs and that contains a message describing the detected error. So whenever the erroneous structure is present in a solution, this rule matches and inserts the error node into the graph.

- Pessimistic rules assume that a given piece of code is wrong unless a certain structure is present. The LHS of a rule is empty (so that it is always applicable) and correct structures

are added as negative application conditions, preventing the rule from being applied if they are found in the graph. The RHS contains only an appropriate error node.

While this seems to be much more complicated on the first glance, it allows for direct combination of pattern search and program manipulation. With JAVA2GGX, spotted undesirable patterns can be changed based on transformation rules directly, or pattern search can be performed by the application of alignment rules. These rules may adjust syntactical divergences before rules for searching special patterns are used. This is interesting in contexts where not one large software project is analyzed, but where many similar but different projects are searched, i.e. in e-learning and automated assessment [KG06, KG08]. In addition, the explicit use of error nodes as markers allows to store results of a search inside the graph data structure and reuse them later on with more complex rules. If source code is to be generated out of manipulated syntax graphs, any additional nodes and arcs introduced by JAVA2GGX are ignored and only nodes belonging to an abstract syntax tree are used to produce the textual representation.

4 High-level Transformations and Analysis

So far we considered analysis and manipulation of Java code at the level of programming language constructs. While this is helpful in the cases introduced so far, source code is often related to abstract specifications like formal models. Transformations for them are already used in model-driven software development [BEK+06, JKS06]. However, an issue considered in this context is that program code is usually derived from specifications unidirectionally with code generation [HT06] only. The reading and writing access to Java code for graph transformations allows for more sophisticated approaches of integrating model specifications and program code since the code can fully participate in such transformations.

4.1 Approach

The reason for fact that (graph) transformations considering source code are usually unidirectional is that generated code does no longer contain the semantics of abstract models. Source code can thus only be integrated in bidirectional transformations if it follows certain rules that make an interpretation possible based on unambiguous rules. To overcome the gap between abstract specifications of software and implementations that are partly created manually instead of being completely generated out of models, we proposed the approach of *embedded models* [BSG10]. An embedded models defines a program code pattern that carries the abstract syntax of a formal model. The pattern code is then not used as meta data, but is interpreted and executed at run time by an execution framework that realizes the execution semantics of the underlying formal model. The pattern code is connected to arbitrary other program code by means of interface code used for data exchange and initiation of business logic execution.

The architecture of embedded models is illustrated in figure 2: The program code pattern is integrated in other program code by invocations of interface code and provision of entry points used by the execution framework. At the same time, transformations exist that use the embedded model syntax to connect the program code to abstract specifications. By this means the source code can be integrated in the chain of transformations at the model level.

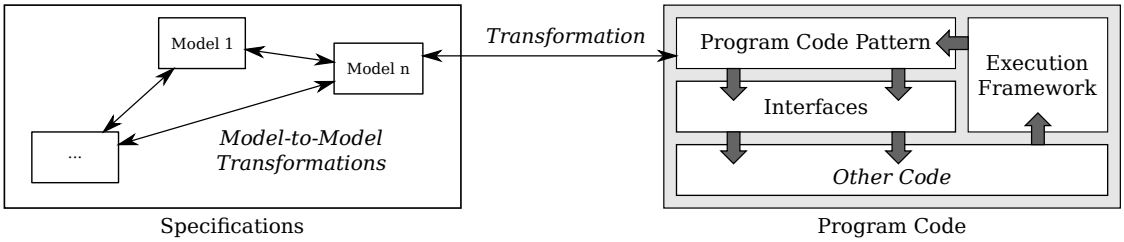


Figure 2: The elements of an embedded model definition. The program code following the pattern can be interpreted unambiguously so that it can be integrated in model transformations across different abstraction levels.

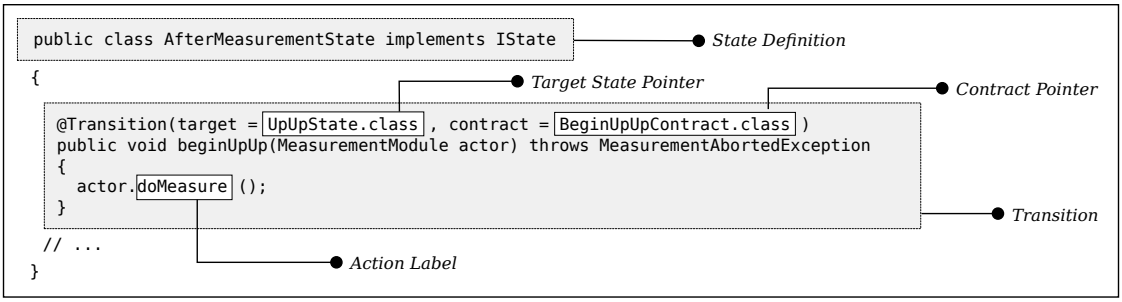
One of the embedded models developed so far considers state machine models. Basically, these models are constituted by a set of classes implementing states and transition contracts, while transitions are implemented as methods. Figure 3 explains some details: The class at the top implements a marker interface `IState`, so it is a state class whose unique name represents the state's name. Its methods are marked as transitions by a Java meta data annotation `@Transition` whose attributes refer to the target state and a contract class (bottom of figure 3) containing guards and updates. An interface type referred to as "actor" is passed to transition methods. Its methods are interpreted as action labels which are called when the transition fires.

Guards and updates are implemented as two methods in a contract class, both evaluating boolean expressions. The method `checkCondition` acts as a guard, deciding whether the related transition may fire or not. It uses the current variable values of the state machine for this decision. The method `validate` acts as an update validator, indicating whether variables match the expected value or value range after actions have been performed. Thus it compares the current values with the values from the point in time before the transition fired. Both methods access a "variables" type which is a facade type representing the variables constituting the state space of the state machine.

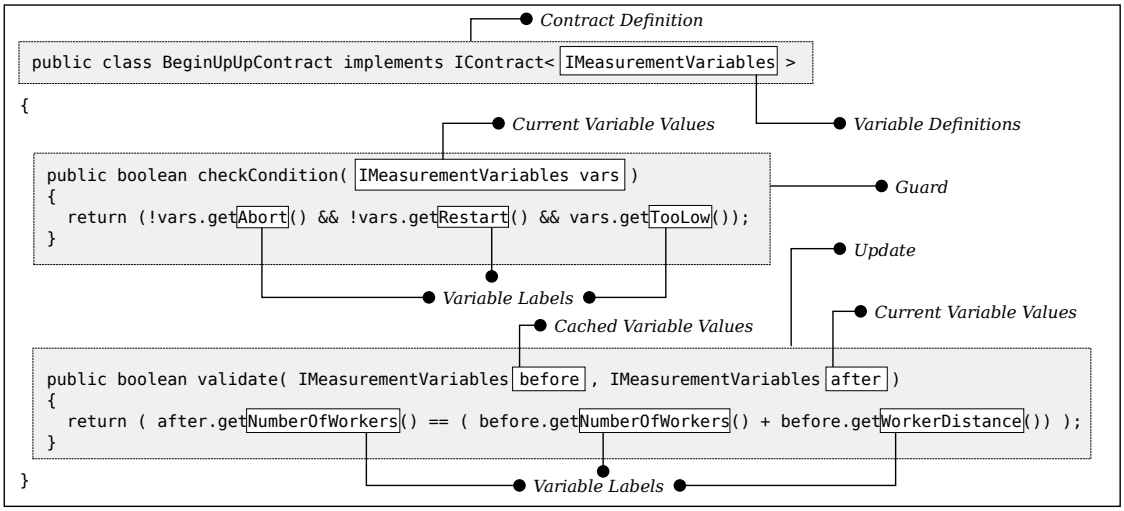
4.2 Program Evolution by Model Transformation

The use of embedded models allows to consider program code at higher levels of abstraction and thus to apply high-level transformations, too. As stated above, state machine models are only one example for embedded models. Thus we will now discuss an example in which we use both state machine models as well as process models [BG09]. At the level of models a state machine can be transformed into a process model automatically while losing only just a few features of state machines that cannot be expressed in process models. Since embedded models rely on static structures in program code and JAVA2GGX allows to transform these program code structures into a graph, model transformation rules that are able to transform a state machine model into a process model or vice versa can be rewritten in order to transform program code with an embedded state machine into program code with an embedded process model or vice versa respectively. For a detailed description please refer to our previous publication [GSB09].

The actual set of graph transformation rules used to implement the evolution from state machine models to process models consists of 21 rules. At first, two rules are concerned with



State and Transition Definition in Source Code



Contract Definition in Source Code

Figure 3: A state definition with an outgoing transition and its contract. The first method of the contract checks a pre-condition with the current variable values, while the second method checks a post-condition by comparing the current values to previous values.

converting states to decision nodes and transitions to activity nodes. One of these rules – changing states to process nodes and creating activity nodes – is shown in figure 4 in a simplified manner. Due to the use of embedded models, elements to be moved can easily be identified by their annotations on the left hand side of the rule and thus reassembled on the right hand side. Similarities between state machines and process models allow to reuse larger parts of existing program code, e.g. complete method bodies.

After nodes have been converted or created, a set of six rules applies changes for cleaning up the graph, like reordering imports or removing unnecessary annotations. These rules are not necessary for a valid embedded process model, but they remove unused nodes, thus helping to keep the graph small. Two additional rules remove contracts not longer needed as well as useless decision nodes that have been introduced by rules applied earlier. Afterwards, the set of nodes is complete, so three rules can be applied for marking start nodes, end nodes, and merge nodes. An additional rule is necessary for splitting up activity nodes since an embedded state machine allows to have more than one action label in transitions, but only one activity per activity node is

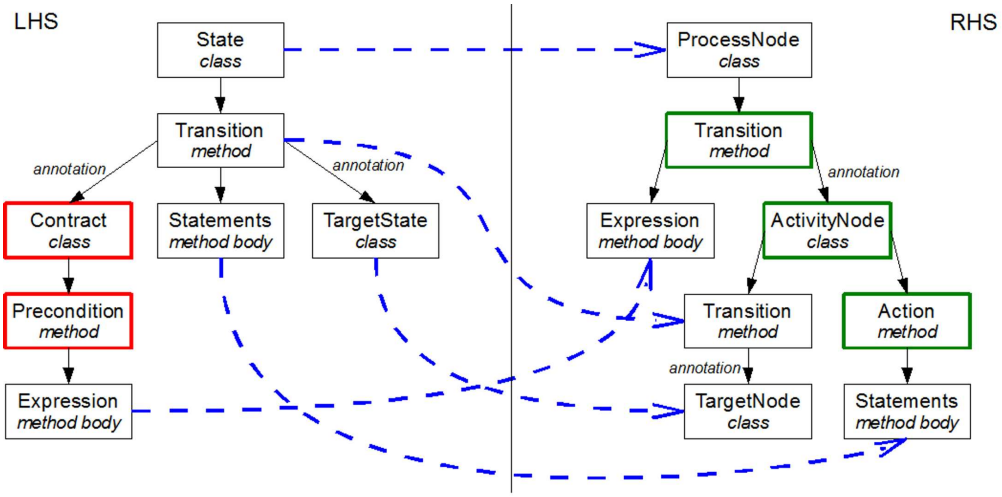


Figure 4: Simplified graph transformation rule for transforming states into process nodes. Nodes deleted from the syntax graph are marked in red while newly created nodes are marked in green. Some of the preserved nodes are renamed during transformation. Note how contents from the original state node are moved to a newly created activity node, while contents from the original extra contract node are moved into the existing process node.

allowed in embedded process models. Another set of seven rules is finally concerned with some adjustments to the code.

For a sample instance of a state machine containing 5 state classes and 5 contract classes, processing the 10 files with JAVA2GGX resulted in a graph with 331 nodes and 694 arcs. Transformation was finished after a total amount of 131 rule applications. Most applications where used for housekeeping on the graph (e.g. 36 for correcting "block" arcs, 32 for copying import statements and 14 for removing unused imports). On an Intel Core2 Duo CPU at 3.17 GHz and with 4 GB RAM the transformation took about 10 seconds.

4.3 Transformation for State Machine Verification

The transformations presented in the previous subsection can be called "internal" transformations since they consider only program code and its abstract syntax tree. However, we can of course also think of "external" transformations, where the abstract syntax tree of the program is part of a larger context. This is obviously the case if source code is derived from models by a sequence of transformation or generation steps. However, this larger context may also be a triple graph grammar, where the syntax graph is one element of the triple. The second element of the triple can be the representation of state machines as used by a verification tool (e.g. UPPAAL [LPY97]) as we have already shown [Str08]. Insufficient graph manipulating capabilities of UPPAAL do not allow to realize synchronous manipulations of models in UPPAAL and embedded models in source code, but transformation rules allow to propagate changes trough the triple graph structure in both directions.

5 Related Work

The principles of graph transformations for program manipulations have already been discussed some years ago in the context of refactoring [EJ04]. Tools like SPOON [PNP06] realize this based on graph traversals without providing the graph explicitly. More recent tools focus on meta modeling and integration into the Eclipse Modeling Framework (EMF) – e.g. JAMOPP [HJSW09] or the MODISCO project [MoD] – and thus come close to the high-level transformations described above in a more abstract way. For UML diagrams, such transformations have also been realized for AGG [FM07].

With JGRALAB it is possible to transform java source code into so-called TGraphs which can be used for declarative graph queries with a query language named GReQL [JGr]. While this allows for detailed program analysis based on graphs, it does not provide means for transformations of program code. The same applies to tools for static code analysis (like PMD as already mentioned above). They use syntax graphs or trees and allow to specify patterns to be searched, but do not make the graphs generally accessible for transformations.

6 Conclusion

In this contribution we introduced JAVA2GGX, a tool for generating explicit graph representations out of abstract syntax graphs of Java programs and vice versa. Using these graphs it is possible to apply static program analysis and low-level transformations for refactoring using graph transformation rules. The benefits are that any means known from graph transformations (e.g. use of additional node types as markers) can be applied to these problems this way. In addition, high-level transformations are also possible if appropriate code structures (e.g. embedded models) are used. By this means another benefit for software evolution is achieved, which is a tighter integration of model transformation and code transformation.

Future work is necessary with respect to content as well as to implementation. At the time of writing, JAVA2GGX can be used as a standalone tool and as a plugin for the development environment Eclipse. This is sufficient for internal transformations, but not for external ones. With respect to content, many other transformations can be developed than the ones shown in this contribution, for different purposes like analysis, verification, and evolution of software.

Bibliography

[AGG] AGG website. http://tfs.cs.tu-berlin.de/agg/.

[BEK+06] E. Biermann, K. Ehrig, C. Köhler, G. Kuhns, G. Taentzer, E. Weiss. EMF Model Refactoring based on Graph Transformation Concepts. In *Proceedings of Third International Workshop on Software Evolution through Transformations (SETra'06)*. Volume 3. Natal, Brazil, sept 2006. Electronic Communications of the EASST.

[BG09] M. Balz, M. Goedicke. Embedding Process Models in Object-Oriented Program Code. In *Proceedings of the First Workshop on Behavioural Modelling in Model-Driven Architecture (BM-MDA)*. 2009.

[BSG10] M. Balz, M. Striewe, M. Goedicke. Continuous Maintenance of Multiple Abstraction Levels in Program Code. In *Proceedings of the 2nd International Workshop on Future Trends of Model-Driven Development - FTMDD 2010, Funchal, Portugal*. 2010.

[Che] CheckStyle Project. http://checkstyle.sourceforge.net.

[EJ04] N. van Eetvelde, D. Jannsens. Extending graph rewriting for refactoring. In *Proceedings of International Conference of Graph Transformation (ICGT) 2004*. Springer, 9 2004.

[FM07] A. Folli, T. Mens. Refactoring of UML models using AGG. *ECEASST* 8, 2007.

[GSB09] M. Goedicke, M. Striewe, M. Balz. Support for Evolution of Software Systems using Embedded Models. In *Design for Future – Langlebige Softwaresysteme*. 2009.

[HJSW09] F. Heidenreich, J. Johannes, M. Seifert, C. Wende. JaMoPP: The Java Model Parser and Printer. Technical report, Technische Universität Dresden, 2009.

[HT06] B. Hailpern, P. Tarr. Model-driven development: The good, the bad, and the ugly. *IBM Systems Journal* 45(3):451–461, 2006.

[JGr] JGraLab. http://userpages.uni-koblenz.de/~ist/JGraLab.

[JKS06] J. Jakob, A. Königs, A. Schürr. Non-materialized Model View Specification with Triple Graph Grammars. In Corradini et al. (eds.), *Proceedings of the 3rd International Conference on Graph Transformation (ICGT) 2006, Natal*. Lecture Notes in Computer Science 4178, pp. 321–335. Springer, 2006.

[KG06] C. Köllmann, M. Goedicke. Automation of Java Code Analysis for Programming Exercises. In *Proceedings of the Third International Workshop on Graph Based Tools*. Electronic Communications of the EASST 1. 2006.

[KG08] C. Köllmann, M. Goedicke. A Specification Language for Static Analysis of Student Exercises. In *Proceedings of the International Conference on Automated Software Engineering*. 2008.

[LPY97] K. G. Larsen, P. Pettersson, W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer* 1(1–2):134–152, Oct 1997.

[MoD] MoDisco Project. http://www.eclipse.org/MoDisco/.

[PMD] PMD Project. http://pmd.sourceforge.net/.

[PNP06] R. Pawlak, C. Noguera, N. Petitprez. Spoon: Program Analysis and Transformation in Java. Technical report 5901, INRIA, 2006.

[Str08] M. Striewe. Using a Triple Graph Grammar for State Machine Implementations. In Ehrig et al. (eds.), *Proceedings of the 4th International Conference on Graph Transformations (ICGT) 2008, Leicester*. LNCS 5214, pp. 514–516. 2008.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Reachability Analysis on Timed Graph Transformation Systems

Christian Heinzemann, Julian Suck, Tobias Eckardt

12 pages

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

Reachability Analysis on Timed Graph Transformation Systems

Christian Heinzemann, Julian Suck, Tobias Eckardt

Software Engineering Group
Heinz Nixdorf Institute
University of Paderborn
Warburger Strasse 100
D-33098 Paderborn, Germany
chris227@upb.de|jsuck@mail.uni-paderborn.de|tobie@upb.de

Abstract: In recent years, software increasingly exhibits self-* properties like self-optimization or self-healing. Such properties require reconfiguration at runtime in order to react to changing environments or detected defects. A reconfiguration might add or delete components as well as it might change the communication topology of the system. Considering communication protocols between an arbitrary number of components, reconfiguration and state-based protocol behavior are no longer independent from each other and need to be verified based on a common formalism. Additionally, such protocols often contain timing constraints to model real-time properties which are of integral importance for the safety of the modeled system and thus need to be considered during the verification of the protocol. In current approaches either reconfigurations or timing constraints are not considered. Existing approaches for the verification of timed graph transformation systems lack important constructs needed for the verification of state-based real-time protocol behaviors. As a first step towards a solution to this problem, we introduced Timed Story Driven Modeling ([HHZ09]) as a common formalism integrating state-based real-time protocol behaviors and system reconfigurations based on graph transformations.

In this paper, we introduce a framework allowing to perform reachability analysis based on Timed Story Driven Modeling. The framework allows to compute the reachable timed graph transition system based on an initial graph and a set of timed transformation and invariant rules.

Keywords: Verification, Real-time Systems, Graph Transformation Systems, Reachability Analysis

1 Introduction

In recent years, software increasingly exhibits self-* properties like self-optimization or self-healing. Such properties require reconfiguration at runtime in order to react to changing environments or detected defects. This causes a significant increase in the complexity of the software as also the reconfiguration process has to be controlled by the software. As software often operates in safety critical environments, it has to meet highest quality standards. Formal verification of safety and liveness constraints as well as verification of joint structural and behavioral constraints ([KG07]) addresses these requirements.

For embedded or real-time systems timing constraints for the software have to be taken into account during verification. Model Checkers like Uppaal[1] address these issues as they allow to check timed temporal logic formulas based on timed automata ([Alu99]). Standard model checkers for real-time systems, however, are not able to consider changing system topologies resulting from system reconfigurations. Graph based model checkers like Groove ([Ren08]) support dynamic topologies, but are not capable of verifying timing constraints. Unfolding the state-space described by the graph transformation rules and using model checkers to verify the result does not work for constraints referencing both, structural and behavioral parts of the system. Existing approaches combining graph transformations and real-time constructs (cf. Section 6) come with restrictions that do not allow to model timed behavior to the extent that is needed for our systems. As a solution, we combine state-based real-time verification as it is realized in Uppaal with dynamic reconfiguration specified by graph transformation rules.

In this paper, we introduce a framework for timed reachability analysis based on Timed Story Driven Modeling ([HHZ09]). In [HHPS10], reconfiguration and state-based behavior are analyzed independently. There, only pairs of automata were checked and an induction over the reconfigurations of a regular architecture was used to show that no forbidden communication structures can arise. This ensures, that only such communication pairs can arise, that have been verified before. In this paper, we use Timed Story Charts ([HHZ09, HHH10]) as an explicit common formalism for the verification. This enables us to specify and verify arbitrary constraints that affect both, state-based protocol behavior and structural system state at the same time, e.g. that a reconfiguration may only take place if certain communication protocol is in a specific state.

Example Scenario Our example scenario stems from the RailCab[2] project. The RailCab system consists of autonomous RailCabs that are fully controlled by software. RailCabs can form contactless convoys to reduce energy consumption. A convoy of RailCabs always needs one coordinating RailCab in order to prevent oscillation of the shuttle distances. A component instance model of a convoy with three RailCabs is shown in Figure 1 where RailCab rc1 coordinates the convoy.

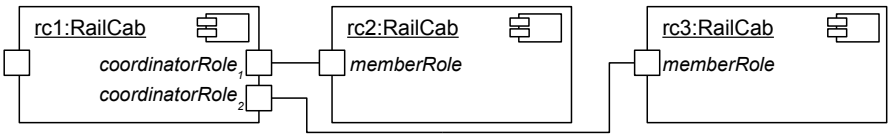


Figure 1: Component instances for the RailCab example.

Outline The remainder of this paper is structured as follows. First, we give a brief overview of Mechatronic UML. In Section 3, we explain how a reachability analysis is performed using Timed Story Driven Modeling while Section 4 provides a description of the framework. We present the results of our evaluation in Section 5 and related work in Section 6. Section 7 concludes the paper.

[1] http://www.uppaal.com/
[2] http://www.railcab.de

2 Mechatronic UML

In this section, we briefly introduce Mechatronic UML, an adaptation of the UML. It provides extensions for modeling and verifying real-time systems and hybrid systems integrating continuous control components. Section 2.1 describes the system architecture of the system and Section 2.2 gives a short overview of Timed Story Driven Modeling with respect to Mechatronic UML.

2.1 System architecture

We use a component-based system architecture based on Mechatronic UML components. The communication between components is modeled with parameterized coordination patterns as introduced in [HHPS10]. Parameterized coordination patterns are used to specify 1 to n communication protocols between communication partners, called *roles* for cardinality 1 or *multi roles* for cardinality *n*, respectively. Roles are instantiated at the ports of a component in order to provide the corresponding protocol.

The behavior of roles is specified by Real-Time Statecharts ([GB03]). In case of a multi role, a parameterized Real-Time Statechart ([HHPS10]) is used to define the behavior of all sub-roles. Figure 2 shows an example of a parameterized pattern including the Real-Time Statecharts defining the role behaviors.

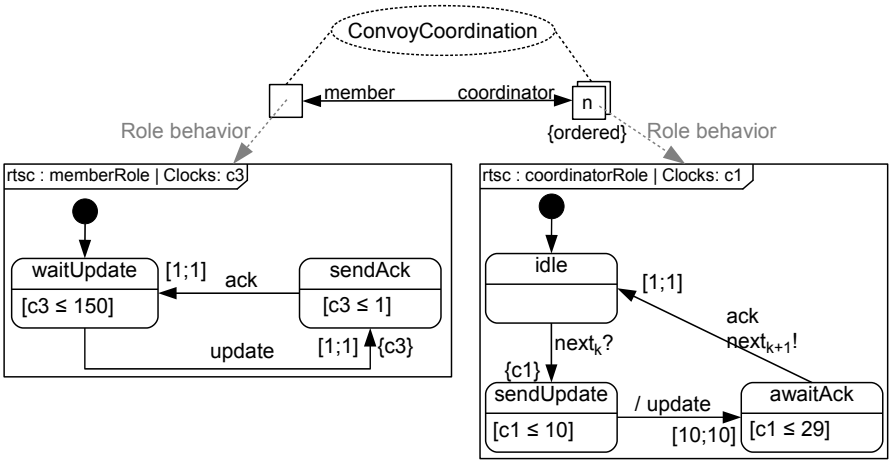


Figure 2: Definition of a Parameterized Coordination Pattern

The *coordinator* role sends an update event to the *member* role which answers with an acknowledgement. The internal synchronization channel *next* is used to synchronize the different sub-roles of the *coordinatorRole* as they are not independent in this scenario. The channel is parameterized with a parameter k referencing to the k^{th} instance of the statechart. Thus, one sub-role triggers the next one in the example.

Following [HHPS10], we use an additional adaptation statechart to synchronize the roles and to manage creation and deletion of roles. An excerpt of an adaptation statechart for the example is shown in Figure 3 on the left.

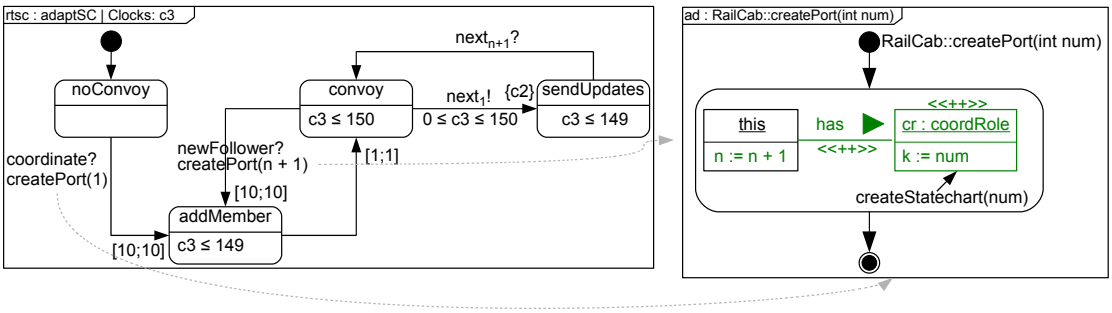


Figure 3: Adaptation Statechart for a Multi Port

The adaptation statechart is initially in state *noConvoy*. The decision which RailCab coordinates the convoy is made in another statechart which is omitted here. If the RailCab is chosen to coordinate, it changes its state to *addMember*, thereby performing the side effect *createPort(1)* of the transition. The side effect is a method of the component being specified by the story diagram ([Zün01]) on the right in Figure 3. The side effect in this example simply creates a new port. The deadline $[10;10]$ of the transition denotes that this reconfiguration takes at least 10 time units and at most 10 time units. After reaching the state *convoy*, the invariant forces the statechart to switch to state *sendUpdates* every 150 time units, thereby triggering the first *coordinatorRole* to send the update.

2.2 Timed Story Driven Modeling

The Timed Story Driven Modeling ([HHZ09]) approach is based on Timed Story Patterns and Timed Story Diagrams. Timed Story Patterns use the syntax of Story Pattern ([Zün01]), which are a short-hand notation for graph transformations. In Story Patterns, the left hand side and the right hand side are depicted in one diagram. Elements being deleted by the transformation are labeled with <<−−>>, elements being created with <<++>>. Additionally, Timed Story Patterns use the same time extensions introduced for timed graph transformation systems in [HHPS10, HHH10], i.e., they operate on timed graphs which contain clocks like timed automata ([Alu99]), each being associated with a subgraph of the graph. The same clock can occur multiple times, once for each occurrence of the subgraph. Therefore, we use the term *clock instance* to denote the instances of a clock. The representation of clock values is realized using clock zones ([Alu99, BY03]), as in timed automata.

In Timed Story Pattern, three kinds of rules are used: transformation rules, invariant rules, and clock instance rules. A clock instance rule adds clock instances to the graph which are used by the transformation rules and the invariant rules to specify time constraints. Due to space limitations, we only show how these rules are implemented in our framework in Section 4.2.

Timed Story Diagrams differ from normal story diagrams only by the fact that they use Timed Story Patterns instead of normal Story Patterns. Based on Timed Story Diagrams, we defined Timed Story Charts [HHH10] that allow us to execute Real-Time Statecharts using Timed Story Diagrams. Timed Story Charts preserve the semantics of Real-Time Statecharts while Real-Time Statecharts can be mapped to Uppaal timed automata ([BGHS04]). Thus, Timed Story Charts

have a proper semantics defined over timed automata. The core idea is to represent the statecharts and their states as objects and to provide graph transformation rules specifying the state changes of the transitions. The currently active state of the statechart is represented by an ActiveState-object. The transformation of Real-Time Statecharts to Timed Story Charts has been partially automated ([HSJZ10]).

3 Reachability Analysis

The reachability analysis will compute the Timed Graph Transition Systems (TTS) based on the given transformation rules and invariants. It represents the complete reachable behavior. In general, the TTS may be infinite. Thus, it cannot be guaranteed that the algorithm will eventually terminate for a given set of rules. This is a general problem when computing Graph Transition Systems. The TTS can be defined as follows, however.

Definition 1 (Timed Graph Transition System (TTS)) Let $\mathscr{G}$ be the set of all possible timed graphs, $\mathscr{R}$ a set of transformation rules, $\mathscr{I}$ a set of invariant rules. The Timed Graph Transition System (TTS) is a triple (S, s_0, T) where S represents the states of the TTS, $s_0 \in S$ is the initial state and T represents the transitions. A state $s \in S$ is a tuple $s = (g, z)$ with $g \in \mathscr{G}$ and z a non-empty clock zone over the clock instances contained in g. In s_0, all clock instances are 0.

There exists a transition t from s_1 to s_2, $s_1 \xrightarrow{t} s_2$, iff there exists a transformation rule $r \in \mathscr{R}$ such that s_2 is a successor state of s_1.

The states are tuples consisting of a timed graph and the current clock interpretations represented by a clock zone ([Alu99]). The clock zone contains intervals for all clock instances representing the possible values as well as the differences between those values. The definition of the TTS is analogous to the definition of zone graphs ([Alu99, BY03]), the only difference is that the states contain a timed graph instead of an automaton location.

The computation starts with the initial graph and all clocks being 0. Then, possible successors are computed according to Definition 2. The TTS contains transitions from one state to all its successors.

Definition 2 (Successor State) Let $s_1 = (g_1, z_1), s_2 = (g_2, z_2)$ states of a TTS. s_2 is a successor state of s_1 iff

- there exists a transformation rule $r \in \mathscr{R}$ such that r transforms g_1 into a graph isomorphic to g_2 and
- $z_2 = (((z_1 \wedge I(s_1)) \Uparrow) \wedge I(s_1) \wedge guard(r))[reset(r)]$ and z_2 non-empty.

The definition of a successor state is analogous to the one of timed automata ([Alu99]). The only difference is that the change in the TTS results from the application of a transformation rule instead of an automaton transition. The computation of the successor clock zone remains the same. First, the clock zone is intersected against all constraints of invariant rules applicable to g_1 denoted by $I(g_1)$. Then, time passes ($\Uparrow$), which is implemented by removing the upper bounds of all clocks (cf. [BY03]). Afterwards, the intersection against the invariants is repeated. Then, the resulting clock zone is intersected with the time guards of the applied transformation rule and

the rules' clock resets are executed. We currently restrict ourselves to guards of the form $c \sim n$ where c is a clock instance, $\sim \in \{<, \leq, =, \geq, >\}$, and $n \in \mathbb{N}$, i.e., comparing the value of a clock instance with an integer. Invariants are further restricted to comparisons $<$ and $\leq$. Please note that there can exist more than one possible successor state for the same transformation rule as multiple matchings can be found.

In order to obtain a finite TTS for reactive systems running in a loop, isomorphic states of the TTS are merged into one state. Two states are isomorphic, iff their graphs are isomorphic to each other and their clock zones are identical.

4 Verification Framework

We have implemented the reachability analysis introduced in Section 3 into our framework. In the following subsections, we introduce the general architecture of our framework at first. Second, we give an introduction how rules can be modeled, third, we explain how the timing computations are performed, and finally, we give a brief idea of constraints that can be checked using the framework. Part of the framework, not including the timing capabilities, has been shown in [HSJZ10].

4.1 Architecture

An implementation of a reachability analysis as specified in Section 3 requires additional rules which compute the TTS. Specifying concrete rules for the TTS generation for each example by hand is a tedious, error-prone task. Therefore, we provide a framework which requires the user to specify an initial graph and a set of rules, only. The remaining tasks, e.g. the application of rules, are integrated within the framework. Figure 4 shows the class diagram of the framework.

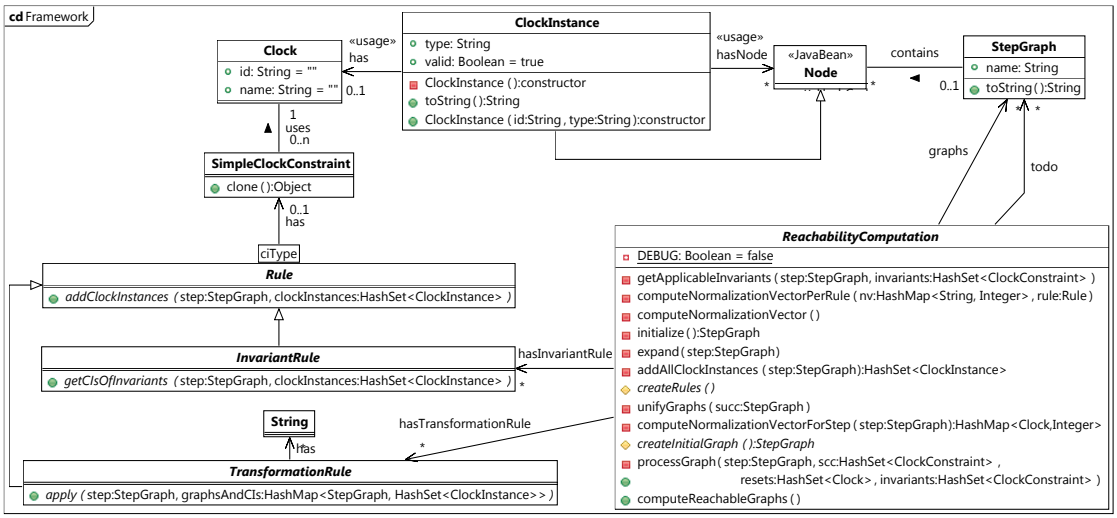


Figure 4: Class Diagram of the Verification Framework

The abstract class *ReachabilityComputation* encapsulates the whole functionality for comput-

ing the timed graph transition system. It contains two abstract methods *createInitialGraph()* and *createRules()*. Both have to be implemented by the user, whereas the former defines the initial graph and the latter specifies which graph transformation rules are to be used for the reachability computation and which time constraints these rules have.

Graphs are represented by objects of the class *StepGraph*. They contain objects of the class *Node*. To represent different types of nodes in a graph, subclasses of *Node* can be created. Edges between nodes are represented by associations. The *createInitialGraph()*-method mentioned above uses the specified nodes and edges to create the initial graph.

As already mentioned in Section 2.2, there exist three different kinds of rules in a timed graph transformation system, namely *timed graph transformation rules*, *invariant rules* and *clock instance rules*. The first two are represented by the classes *TransformationRule* and *InvariantRule*, respectively, whereas the third

is represented by the method *addClockInstances()*. The next section contains more detailed information on how to implement transformation and invariant rules.

4.2 Modeling Rules

Timed graph transformation rules are represented by the abstract class *TransformationRule*. It contains an abstract method *apply()*, which has to be implemented by subclasses to specify a concrete timed graph transformation rule. As parameters, this method receives a graph on which it is to be applied and a reference to a mapping. After the termination of the method, the mapping contains all reached successor graphs together with their respective clock instances used for the application of the rule.

Figure 5 shows an implementation of the method *apply()* as a Story Diagram modeling the transition from the state *sendAck* to the state *waitUpdate* of the statechart of the role *member* (cf. Figure 2). The Story Diagram is subdivided into three Story Activities. The first activity checks whether the rule is applicable, which is the case if the *sendAck*-state is the active state in the given statechart. If the rule is applicable, the second activity sets the active state from *sendAck* to *waitUpdate*. Finally, the third activity enqueues the *ack()*-Event into the event queue of the statechart of the coordinator port.

Invariant rules are specified by subclasses of *InvariantRule*. Subclasses implement the method *getCIsOfInvariant()*, which receives a graph on which the rule is to be applied and a set as parameters. After the termination of the method, the set contains all clock instances of the graph for which the time invariant specified by the rule is applicable.

Figure 6 shows a concrete example of a Story Diagram implementing the method *getCIsOfInvariant()*. The rule represents the invariant $c3 \leq 150$ of the state *convoy* of the adaptation statechart of the coordinator role (cf. Figure 2). The invariant has to hold whenever the adaptation statechart's active state is the *convoy*-state. The activity matches to all structures which model exactly this situation. Whenever a matching structure is found, the corresponding clock instance is inserted into the set.

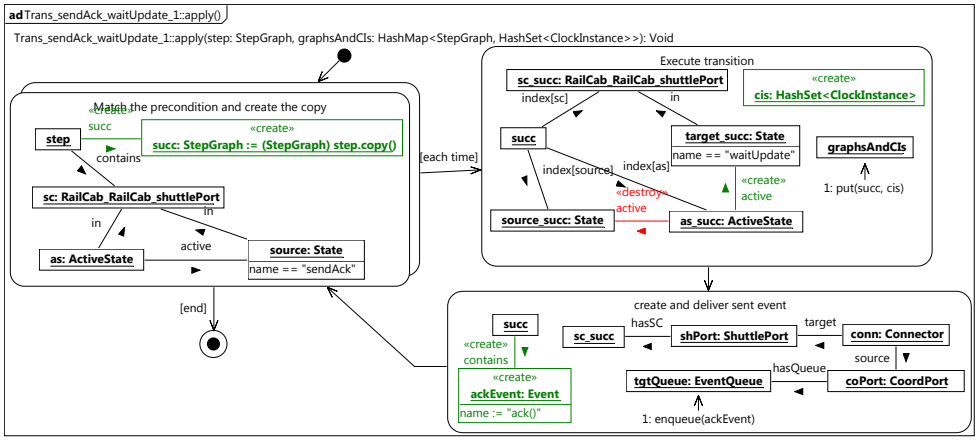


Figure 5: Story Diagram of the Transformation Rule, Modeling the Transition from *sendAck* to *waitUpdate*

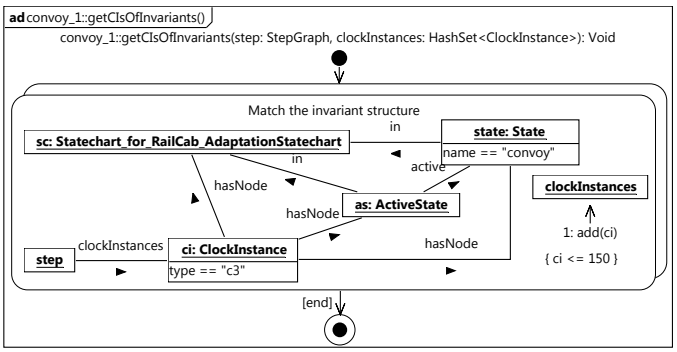


Figure 6: Story Diagram of the Invariant Rule, Modeling the Invariant of State *convoy*

4.3 Performing time computations

All time computations, i.e. operations on clock zones, are performed using the Uppaal DBM (UDBM) library[3]. The C/C++-library, as originally implemented for the Model Checker Uppaal, efficiently implements all necessary operations on clock zones (up or delay ($\Uparrow$), intersection ($\wedge$), clock resets ($[reset(r)]$)) by using the *Difference Bound Matrix (DBM)* data structure as defined by Dill ([Dil90]) and, in addition to that, by applying very efficient memory management.

Technically, we access the UDBM library using the provided Ruby binding in connection with a (local) client/server-communication between our Java implementation and the Ruby implementation. In Java, clocks, clock zones and clock constraints are represented as classes providing the corresponding operations on those instances as methods. This makes the binding to the UDBM library completely transparent for the developer and allows insertion and removal of clock instances which is not directly supported by the UDBM. In Java, clock instances can be created like normal objects. During runtime, a given clock zone is then transformed into ruby code as

[3] http://www.cs.aau.dk/~adavid/UDBM/

well as the desired operation is transformed. This ruby code is sent to the ruby server, which executes it and sends back the resulting clock zone as an encoded string. This string is finally transformed back into a clock zone object representing the result of the operation.

4.4 Verification of constraints

Currently, our framework only supports the verification of CTL (Computation Tree Logic) formulas having the form $EF\varphi$ or $\neg AG\varphi$ where φ is a graph invariant. Thus, it is possible to check whether a specific subgraph eventually occurs in the graph or to check whether a subgraph can never occurs in the graph. Such constraints are modeled as rules which print out a message in case they are fulfilled or not fulfilled, respectively. A formula $EF\varphi$ is fulfilled if the left hand side of the corresponding rule can be matched to the graph. A formula $\neg AG\varphi$ is not satisfied if the left hand side cannot be matched. Both is possible using Story Diagrams. Using the TTS, it is also possible to identify deadlocks, as these are simply reachable states with no outgoing transitions.

5 Evaluation

We implemented the convoy coordination example shown in Figures 2 and 3 using the pattern and the provided statecharts. Then, we generated partial Timed Story Charts and added the missing parts, e.g. the statechart synchronization channels and the recipients of sent messages. This resulted in 15 transformation rules and 13 invariant rules. In this case, we only needed three clock instance rules, one for each statechart, as we could create a statechart instance as a whole, along with all its clocks.

The number of RailCabs in a convoy was restricted to a maximum number in order to obtain a TTS for different maximum convoy sizes. Then, we used our framework described in Section 4 to compute the TTS. The results are shown in Table 1.

max convoy size	# graphs in TTS	runtime (s)	runtime (s) optimized	max. graph size
2	17	2	1	37
3	52	21	2	52
4	112	125	6	67
5	203	545	16	82
6	329	1856	39	97

Table 1: Evaluation results for different convoy sizes

A convoy size of 2 corresponds to one leading RailCab and one convoy member RailCab, i.e., the leading RailCab has one *coordinatorRole* statechart instance. For each additional RailCab in the convoy, an additional instance is added. We computed the TTS for our example with a maximum convoy size of 6 RailCabs because the timing constraints in our example do not support larger convoys. We recently found a major performance bug in our implementation that yielded a significant improvement of our runtime as shown in Table 1. The results indicate that the runtime grows exponentially in the number of reached graphs while the maximum graph size

grows constantly as expected. The growth in runtime results from the high number of clock instances and the expensive timing computations which consume about 66% of the runtime. Additionally, our isomorphism check on graphs turned out to be inefficient ([HSJZ10]). The timing computations along with the definition of isomorphic states cause a certain blowup in the number of reached states, as isomorphic graphs had to be expanded more than once because of differences in the clock zones. Finally, isomorphic states were reached on all paths and the computation terminated.

6 Related Work

In the field of timed graph transformation systems, there exist several other approaches. The MOMENT2 framework ([BÖ10]) provides graph transformations based on MOF meta models. The approach supports one unresetable clock per object, timers, that trigger actions, and timed values which can be increased or decreased at a certain, fixed rate. The real-time graph rewrite model checker Real-Time Maude ([ÖM07]) provides object oriented graph transformations in a textual syntax, but it does not support invariant rules requiring subgraph changes. The approach by Rivera et al. ([RDV09]) provides only one global clock and durations for the execution of rules, but no verification or guarding of rules by time constraints. De Lara et al.([LV10]) map their graph transformation rules to timed Petri nets. Time is not actually part of the model, but annotated as an interval in which the transformation can be executed after a match has been found. In [THRB10], stochastic graph transformations are introduced. The simulation of these transformations incorporates a scheduling that is based on continuous time and executes a rule at a randomly generated point in time. A drawback of all approaches is their lack of support for flexible clock creation with resets and the specification of time guards at the same time. This, however, is needed for the system models we employ.

There exist some approaches for checking graph transformations without the possibility to consider timing constraints. The Groove project supports reachability analysis on labeled graphs, as well as checking graph based LTL formulas on the graph transition system ([Ren08]). The approach by König et. al. [KK08] uses an approximation technique that maps a possibly infinite graph transition system to finite Petri graphs and verifies the specified formula on this Petri graph structure. The inductive invariants introduced in [GS04] support infinite state spaces, as well as they only require a static analysis on the set of rules showing that a forbidden graph cannot be produced. It is not possible, however, to verify properties that cannot be depicted as a graph, like deadlock freedom, for example.

Bauer et. al. provide a verification approach for dynamic communication protocols ([BSTW06]) using over- and underapproximation of the system in order to verify LTL (Linear-time Temporal Logic) formulas with first-order quantification on objects. The approach supports infinite numbers of communicating objects and finite message queues.

The timed model checker Uppaal provides the ability to check timed systems, but not the evolution of the system in terms of adding new statechart behaviors at runtime. The BIP framework ([BS10]) also provides real-time components and connectors with extensive analysis approaches, but does not support reconfigurations, either.

7 Conclusions

In this paper, we have shown a technique to perform a reachability analysis on Timed Story Diagrams, a dialect of graph transformation systems extended by the notion of time. We use Timed Story Charts as a common formalism to perform a reachability analysis for dynamic real-time communication protocols whose structural evolution is specified by graph transformations.

As future work, we plan to extend our framework by the possibility to verify more complex timing constraints, as introduced in [KG07]. We will also investigate the possibility of applying existing abstraction and approximation techniques to the timed graph transformation systems in order to be able to handle larger state spaces and to obtain a more efficient verification procedure. Finally, we will try to fully automate the Timed Story Chart generation by adding generation of synchronizations and message recipients.

Bibliography

[Alu99] R. Alur. Timed Automata. In Halbwachs and Peled (eds.), *Proc. of the 11th International Conference on Computer Aided Verification (CAV '99), July 6-10, 1999, Trento, Italy*. Lecture Notes in Computer Science 1633, pp. 8–22. Springer, 1999.

[BGHS04] S. Burmester, H. Giese, M. Hirsch, D. Schilling. Incremental Design and Formal Verification with UML/RT in the FUJABA Real-Time Tool Suite. In *Proc. of the International Workshop on Specification and Validation of UML Models for Real Time and Embedded Systems, SVERTS2004*. Pp. 1–20. October 2004.

[BÖ10] A. Boronat, P. C. Ölveczky. Formal Real-Time Model Transformations in MOMENT2. In *Proc. of the 13th International Conference on Fundamental Approaches to Software Engineering, FASE 2010*. Pp. 29–43. 2010.

[BS10] S. Bliudze, J. Sifakis. Causal semantics for the algebra of connectors. In *Formal Methods in System Design*. Volume 36(2), pp. 167–194. Springer, 2010.

[BSTW06] J. Bauer, I. Schaefer, T. Toben, B. Westphal. Specification and Verification of Dynamic Communication Systems. In *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*. IEEE Computer Society Press, 2006.

[BY03] J. Bengtsson, W. Yi. Timed Automata: Semantics, Algorithms and Tools. In Desel et al. (eds.), *Lectures on Concurrency and Petri Nets*. Lecture Notes in Computer Science 3098, pp. 87–124. Springer, 2003.

[Dil90] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*. Lecture Notes in Computer Science 407, pp. 197–212. Springer-Verlag, London, UK, 1990.

[GB03] H. Giese, S. Burmester. Real-Time Statechart Semantics. Technical report tr-ri-03-239, Software Engineering Group, University of Paderborn, Germany, June 2003.

[GS04] H. Giese, D. Schilling. Towards the Automatic Verification of Inductive Invariants for Infinite State UML Models. Technical report tr-ri-04-252, Software Engineering Group, University of Paderborn, Germany, December 2004.

[HHH10] C. Heinzemann, S. Henkler, M. Hirsch. Refinement Checking of Self-Adaptive Embedded Component Architectures. Technical report tr-ri-10-313, Software Engineering Group, University of Paderborn, Mar. 2010.

[HHPS10] S. Henkler, M. Hirsch, C. Priesterjahn, W. Schäfer. Modeling and Verifying Dynamic Communication Structures based on Graph Transformations. In *Proc. of the Software Engineering 2010 Conference, Paderborn, Germany*. 2010.

[HHZ09] C. Heinzemann, S. Henkler, A. Zündorf. Specification and Refinement Checking of Dynamic Systems. In Gorp (ed.), *Proceedings of the 7th International Fujaba Days*. Pp. 6–10. Eindhoven University of Technology, The Netherlands, November 2009.

[HSJZ10] C. Heinzemann, J. Suck, R. Jubeh, A. Zündorf. Topology Analysis of Car Platoons Merge with FujabaRT & TimedStoryCharts - a Case Study. In Gorp et al. (eds.), *Transformation Tool Contest*. Malaga, 2010.

[KG07] F. Klein, H. Giese. Joint Structural and Temporal Property Specification Using Timed Story Scenario Diagrams. In *Formal Approaches to Software Engineering*. Lecture Notes in Computer Science 4422, pp. 185–199. Springer, 2007.

[KK08] B. König, V. Kozioura. Towards the Verification of Attributed Graph Transformation Systems. In *ICGT '08: Proc. of the 4th international conference on Graph Transformations*. Pp. 305–320. Springer-Verlag, Berlin, Heidelberg, 2008.

[LV10] J. de Lara, H. Vangheluwe. Automating the transformation-based analysis of visual languages. In *Formal Aspects of Computing*. Volume 22(3), pp. 297–326. Springer, 2010.

[ÖM07] P. C. Ölveczky, J. Meseguer. Semantics and Pragmatics of Real-Time Maude. *Higher-Order and Symbolic Computation* 20(1-2):161–196, 2007.

[RDV09] J. E. Rivera, F. Duran, A. Vallecillo. A graphical approach for modeling time-dependent behavior of DSLs. *Visual Languages - Human Centric Computing* 0:51–55, 2009.

[Ren08] A. Rensink. Explicit State Model Checking for Graph Grammars. In *Concurrency, Graphs and Models*. Lecture Notes in Computer Science 5065, pp. 114–132. Springer, 2008.

[THRB10] P. Torrini, R. Heckel, I. Ráth, G. Bergmann. Stochastic Graph Transformation with Regions. In *GM-VMT'10*. Electronic Communications of the EASST 29. 2010.

[Zün01] A. Zündorf. *Rigorous Object Oriented Software Development*. University of Paderborn, 2001.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Neighbourhood Abstraction in GROOVE — Tool Paper

Arend Rensink and Eduardo Zambon

6 pages

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

Neighbourhood Abstraction in GROOVE — Tool Paper

Arend Rensink and Eduardo Zambon*

rensink@cs.utwente.nl, zambon@cs.utwente.nl
Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: In this paper we discuss the implementation of neighbourhood graph abstraction in the GROOVE tool set. Important classes of graph grammars may have unbounded state spaces and therefore cannot be verified with traditional model checking techniques. One way to address this problem is to perform graph abstraction, which allows us to generate a *finite* abstract state space that over-approximates the original one. In previous work we presented the theory of *neighbourhood abstraction*. In this paper, we present the implementation of this theory in GROOVE and illustrate its applicability with a case study that models a single-linked list.

Keywords: Graph Abstraction, Graph Transformation, Model Checking, GROOVE

1 Introduction

Many verification methods rely on the exploration of the state space of systems. However, even for small systems the state space size tends to blow up exponentially. Moreover, one would like to be able to analyse systems independently of their instantiated size. An approach that can in principle solve both these problems is *state abstraction*. The idea behind this is that "similar" states are actually grouped together, and such a group of similar states is modelled in such a way the distinction between them is no longer visible. The behaviour of the abstract state is the collection of possible behaviours of the original states.

This principle has been long known and studied, e.g., in abstract interpretation [CC77] and shape analysis [SRW98, SRW02]. In the context of graph transformation we have seen several theoretical studies on suitable abstractions [Ren04, RD06, BBKR08, RN08, BKK03, KK06]. However, to the best of our knowledge only the last of these is backed up by an implementation, namely AUGUR2 [KK08].

In this paper we report an extension of GROOVE that implements the neighbourhood abstraction principle of [BBKR08], showing its feasibility at least on a small example. This gives us a basis for experimenting with different, more expressive notions of abstraction.

2 Neighbourhood Abstraction

Our notion of abstraction is based on neighbourhood similarity: two nodes are considered indistinguishable if they have the same incoming and outgoing edges, and the opposite ends of

* The work reported herein is being carried out as part of the GRAIL project, funded by NWO (Grant 612.000.632).

those edges are also comparable (in a parameterisable sense). Graphs are abstracted by folding all indistinguishable nodes into one, while keeping count of their original number up to some bound of precision. The incident edges are also combined. Counting up to some bound is done using *multiplicities*. We use $M_k = \{0, \ldots, k, \omega\}$ with $k \in \mathbb{N}$ consisting of exact numbers up to k (which is typically a low value such as 1 or 2) and the value ω standing for "many".

Formally, a graph is a tuple $G = \langle V, E \rangle$ of nodes and edges, where (in our case) the edges are triples (v, a, w) of source node, label, and target node. The abstractions are called *shapes*: they are 5-tuples $S = \langle G, \sim, \mathsf{mult}_{\mathsf{node}}, \mathsf{mult}_{\mathsf{in}}, \mathsf{mult}_{\mathsf{out}} \rangle$ in which

- G is the underlying graph structure of the shape;
- $\sim \subseteq V \times V$ is a *neighbourhood similarity* relation;
- $\mathsf{mult}_{\mathsf{node}} \colon V \to M_\nu$ is a *node multiplicity* function, which records how many concrete nodes were folded into a given abstract node, up to bound ν;
- $\mathsf{mult}_{\mathsf{in}}, \mathsf{mult}_{\mathsf{out}} \colon (V \times L \times V/\!\sim) \to M_\mu$ are incoming and outgoing *edge multiplicity* functions, which record how many edges with a certain label the concrete nodes had that were folded into an abstract node, up to a bound μ and a group of $\sim$-similar opposite nodes.

The following is pseudo-code for generating the abstract state space. Q is the set of all shapes and F the set of fresh, yet to be explored shapes; P is the set of rules and G the start graph.

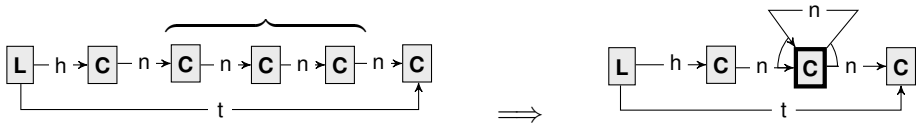
```
1  let S := abstract_i(G), Q := ∅, F := {S}
2  while F ≠ ∅
3  do choose S ∈ F        (which S is selected depends on the exploration strategy)
4     let F := F \ {S}
5     for p ∈ P, m ∈ prematch(p,S), S' ∈ materialise(m,S)
6     do let R := normalise(apply(p,m,S'))
7        if R ∉ Q
8        then let Q := Q ∪ {R}, F := F ∪ {R}
9        fi
10    od
11 od
```

The important phases in this algorithm are:

- abstract computes the shape of a graph. This is controlled by a parameter i expressing the *radius* of the neighbourhood to be considered in the neighbourhood similarity relation $\sim$, which is built iteratively. Level 0 considers only node labels, and additional levels look at outgoing and incoming edges, bounded by i;
- prematch computes morphisms of a rule p into a shape S. Such a morphism is not yet a match, because the images of p's left hand side may be nodes with multiplicity > 1; in that case they have to be materialised.
- materialise creates concrete nodes and edges for the image of p in S. This is a non-deterministic step, as there may be choices involved in choosing multiplicities for the instantiated nodes and edges.
- apply is rule application, which can be carried out as usual because the rule now acts upon a concrete subgraph of S'.

- normalise merges the transformed graph back into the rest of the shape; it is thus similar to abstract except that it acts upon a (partially materialised) shape rather than a graph.

For instance, $\mathsf{abstract}_i$ converts the following graph into a shape, using $i = 1$ and $\nu = \mu = 1$. The indicated **C**-nodes in the graph are folded because they are indistinguishable by their incoming and outgoing edges.



In the right hand side figure, all **C**-labelled nodes are $\sim$-similar, and all edge multiplicities are 1. The "joined" incoming and outgoing edges indicate that the multiplicities apply to a bundle of edges, rather than a single edge. The fat **C**-node has multiplicity ω.

The main challenge of implementing the pseudo-code given above lies in the complexity of the algorithms for each five phases, in particular of materialise, which requires the enumeration of all concrete nodes and edges configurations for the pre-match of a rule. An implementation has to properly address these efficiency issues in order to be usable in practice.

Due to space limitations we refrain from giving additional information on the theory of neighbourhood abstraction. For the details, we refer the interested reader to [BBKR08].

3 Case study

As a test case for our abstraction implementation we use a graph transformation system that models a single-linked list. The list is formed by *cells*, representing the elements in the list, which are connected by a *next* pointer. Additionally, a list has a root object that indicates the first and last elements of the list, by way of pointers called *head* and *tail*, respectively. The modelling of such structure as a graph in GROOVE is trivial, as shown in the previous section. The root of the list is represented by a **L**-node and the cells by **C**-nodes. Pointers *head*, *tail*, and *next* are modelled by edges labelled h, t, and n, respectively.

We consider two list operations: one that *puts* a new element to the tail of the list, and another that *gets* the head element from the list. These operations are modelled in our graph transformation system by two rules, shown below. GROOVE combines the left and right hand side of a rule in a single graph, and colours and shapes are used to distinguish different elements. Blue (dashed thin) elements are deleted by the rule application; green (continuous fat) nodes and edges are created; and black (continuous thin) elements are preserved. For simplicity, we assume that our lists always have at least one element[1].

put rule get rule

[1] otherwise, two more rules are necessary to insert an element to an empty list and to remove the last element.

It is clear that the concrete state space of this example is unbounded: the put rule is always enabled, and successive applications of this rule keep producing longer and longer lists. However, the abstract state space produced by our abstraction mechanism is finite. For an abstraction radius of one, the state space has 10 states and 21 transitions, as shown in Fig. 1. Each dashed box represents a state, with its numbering on the lower left corner, and the corresponding shape drawn with solid lines. The transitions between states are shown by dashed arrows, labelled with the rule applied.

There are many interesting points to note in the state space of Fig. 1. First, as long as node and edge multiplicities stay within their bounds, the abstract graph transformation corresponds to the concrete one. This is seen on states s0, s1, and s2, where the shapes are concrete.

Second, an abstract state may represent an unbounded number of concrete ones. State s3, for example, is an abstract representative for lists with four or more elements. This is illustrated by the put transition from s3 to itself.

Third, the non-determinism of the materialisation algorithm can be seen on the two get transitions from state s3. Although there is only one pre-matching of the rule, when materialising this pre-match two distinct shapes are produced.

Fourth, we can see that the abstract state space has spurious configurations. For example, states s6 to s9 represent lists with unconnected elements, which do not occurr in the concrete state space. This spurious shapes arise from the fact that the neighbourhood abstraction mechanism does not keep information regarding connectivity.

4 Conclusion

The results reported above are the very first steps toward the capability for GROOVE to incorporate abstraction. We look upon this as a key factor in the eventual success of the tool. Though currently we have merely implemented the theory described in [BBKR08], we know from experience that having the ability to actually experiment with smaller and larger cases provides a lot of additional motivation and can be a source of new ideas and developments.

For instance, only a working implementation makes it possible to obtain figures about actual abstract state space sizes, which is one of the most important factors in the feasibility of any abstraction-based methods. Some very first figures about the effect of increasing node multiplicity bounds v and radii i for our linked list example are collected in the following table. Clearly, the radius has a much greater effect on the state space size than the node multiplicity. All results took less than 60 seconds to be generated.

	$v=1$		$v=2$		$v=3$	
	states	trans.	states	trans.	states	trans.
$i=1$	10	21	14	29	18	37
$i=2$	389	1060	613	1486	969	2318

As noted in [BBKR08], neighbourhood abstraction is adequate for cases where updates are determined locally and reachability is not important. Since one of our goals is to use graph abstraction for the verification of software with dynamically allocated data structures, we are working on improving the current theory and implementation in order to trim down the spurious

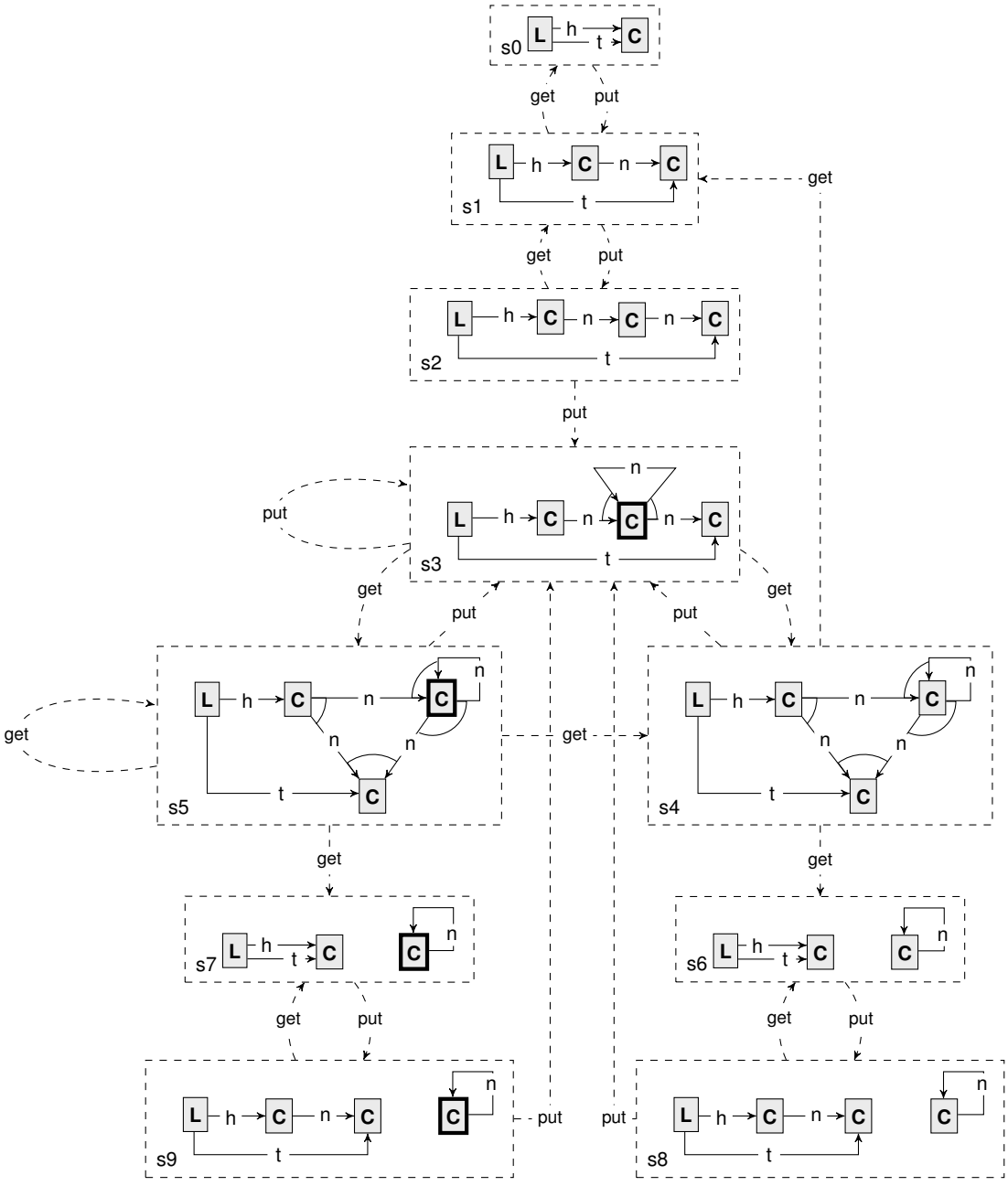


Figure 1: The abstract state space for the parameters $i = 1$ and $\nu = \mu = 1$

configurations in the state space. We expect this to reduce the size of the state spaces considerably, which will allow GROOVE to be used on more complex and realistic examples.

Bibliography

[BBKR08] J. Bauer, I. B. Boneva, M. E. Kurban, A. Rensink. A Modal-Logic Based Graph Abstraction. Pp. 321–335 in [EHRT08].

[BKK03] P. Baldan, B. König, B. König. A Logic for Analyzing Abstractions of Graph Transformation Systems. In Cousot (ed.), *Static Analysis Symposium (SAS)*. LNCS 2694, pp. 255–272. Springer, 2003.

[CC77] P. Cousot, R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *POPL*. Pp. 238–252. 1977.

[EHRT08] H. Ehrig, R. Heckel, G. Rozenberg, G. Taentzer (eds.). *International Conference on Graph Transformations (ICGT)*. LNCS 5214. Springer, 2008.

[KK06] B. König, V. Kozioura. Counterexample-Guided Abstraction Refinement for the Analysis of Graph Transformation Systems. In Hermanns and Palsberg (eds.), *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. LNCS 3920, pp. 197–211. Springer, 2006.

[KK08] B. König, V. Kozioura. AUGUR2— A New Version of a Tool for the Analysis of Graph Transformation Systems. *ENTCS* 211:201–210, 2008.

[RD06] A. Rensink, D. S. Distefano. Abstract Graph Transformation. In Mukhopadhyay et al. (eds.), *Software Verification and Validation*. ENTCS 157, pp. 39–59. May 2006.

[Ren04] A. Rensink. Canonical Graph Shapes. In Schmidt (ed.), *Programming Languages and Systems (ESOP)*. LNCS 2986, pp. 401–415. Springer, 2004.

[RN08] S. Rieger, T. Noll. Abstracting Complex Data Structures by Hyperedge Replacement. Pp. 69–83 in [EHRT08].

[SRW98] S. Sagiv, T. W. Reps, R. Wilhelm. Solving Shape-Analysis Problems in Languages with Destructive Updating. *ACM ToPLaS* 20(1):1–50, 1998.

[SRW02] S. Sagiv, T. W. Reps, R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM ToPLaS* 24(3):217–298, 2002.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing Techniques

Steffen Mazanek, Christian Rutetzki, and Mark Minas

13 pages

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

Sketch-based Diagram Editors with User Assistance based on Graph Transformation and Graph Drawing Techniques

Steffen Mazanek, Christian Rutetzki, and Mark Minas

(Steffen.Mazanek, Christian.Rutetzki, Mark.Minas)@unibw.de
Universität der Bundeswehr München, Germany

Abstract: In the last years, tools have emerged that recognize *sketched diagrams* of a particular visual language. That way, the user can draw diagrams with a pen in a natural way and still has available most processing capabilities. But also in the domain of conventional diagram editors, considerable improvements have been achieved. Among other features, powerful *user assistance* like auto-completion has been developed, which guides the user in the construction of correct diagrams. The combination of these two developments, sketching and guidance, is the main contribution of this paper. It not only shows feasibility and usefulness of the integration of user assistance into sketching editors, but also that novel user strategies for identifying and dealing with recognition errors are made possible that way. The proposed approach heavily exploits graph transformation and drawing techniques. It was integrated into a meta-tool, which has been used to generate an editor for business process models that comprises the features described in this paper.

Keywords: sketching, meta-tools, user assistance, graph transformation, graph drawing, process models

1 Introduction

An important benefit of sketch-based diagram editors is that diagrams can be drawn with maximal freedom in a very natural way. With the appearance of powerful and permissive approaches to their subsequent recognition – among others [13, 7, 8] – many advantages of traditional WIMP interfaces (Window, Icon, Menu, Pointer) can be carried over. Most importantly, diagrams, once recognized, can be further processed. However, one feature of state-of-the-art conventional diagram editors, namely user assistance, has not yet been integrated into sketch tools. The user assistance we aim at guides the user in the construction of correct diagrams. Indeed, the only existing attempt in this direction we are aware of is the work on *symbol completion* by Costagliola et al. [9]. This approach helps the user in completing individual symbols (lexical level), but the overall diagram structure (syntactical level) is not at all considered – not even to mention language semantics or pragmatics. Moreover, this approach has not been integrated into a visual environment yet. In this paper we fill this gap by integrating a user assistance component into a sketching meta-tool, i.e., a framework for generating sketch editors from a language specification. We report on the challenges that had to be addressed and how graph transformation and graph drawing techniques have been used for solving them.

Fig. 1 shows a bird's eye view of the proposed approach, i.e., the overall architecture of sketching editors with assistance. The *user*, who is represented by the stickman in the mid-

dle, draws *strokes*, which are the basic input of most sketch recognition tools. The *recognizer* transforms these strokes either on-line or on user's request into a *set of diagram components*. Next, a language-specific *analysis* of the diagram is performed, e.g., a syntax check. For the diagram given in Fig. 1, it might be checked that there are no arrows without proper source and target components or that processing components (rectangles) are connected to data structures (ellipses) only. The result of this analysis step is passed back to the user as visual *feedback*.

The novel aspects of this work are surrounded by a dashed line. Concretely, we propose that the analysis step might also result in a *set of suggestions* for the user, e.g., on how the diagram can be completed. The user can choose among these suggestions, e.g., by using a preview of the corresponding diagram changes. The selected suggestion then has to be integrated into the sketch. Therefore, a *translator* component generates a set of new strokes and adds them to the user's sketch. That way, the next analysis cycle will directly consider the selected suggestion.

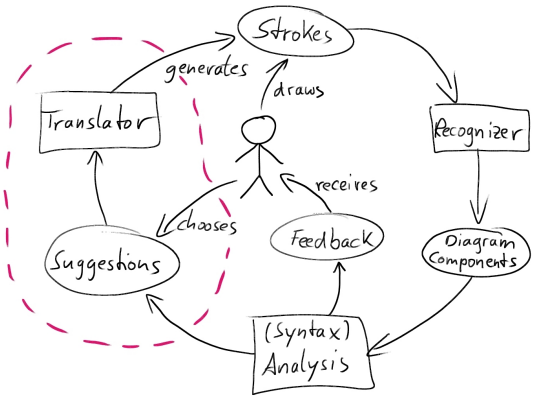


Figure 1: Proposed editor architecture

This paper covers the following assistance features, which are all based on *syntax* [16]:

- *auto-completion*: the computation of missing diagram components that transform the incomplete diagram into a proper member of the underlying visual language,
- *auto-link*: the derivation of missing edges in graph-like languages according to node arrangement and other kinds of editing accelerators,
- *example generation*: the generation of correct example diagrams that can be explored by the user for the sake of language learning.

Suggestions that remove parts of a sketch are not considered.

Sketch tools with powerful recognizers [13, 7, 8] as well as tools for the computation or specification of suggestions [1, 15, 19] already exist. Therefore, this paper focuses on the following three aspects:

- *User interaction*: how can the user invoke and control assistance,
- *Stroke generation*: how and where should the translator generate strokes from the suggestion (this actually is a graph drawing problem),
- *User feedback*: indeed, syntactical assistance not only provides clues for syntactical problems, but also simplifies the identification of recognition errors.

This paper is structured as follows: Sect. 2 introduces the running example language, namely business process models (BPMs). Our implementation relies on existing frameworks for sketch recognition and user assistance; Sect. 3 recapitulates their concepts. How these two approaches actually have been combined and integrated is described in Sect. 4. A discussion is provided in Sect. 5. Finally, related work is reviewed and the paper is concluded (Sect. 6 and 7).

2 Business Process Models

BPMs are used to represent the workflows within an enterprise and, thus, are a highly relevant diagrammatic language today. In recent years a standardized visual notation, the Business Process Modeling Notation BPMN [18], has been developed. Since BPMs are frequently developed in creative team meetings, this language ideally should be supported by sketch editors. Fig. 2 shows a small sales process, which has been drawn and recognized by the sketching editor described in this paper. The magnified (assistance) toolbar will be described later.

BPMs basically are graphs, where the connecting arrows represent sequence flow. The example process starts with the receipt of an order, which is expressed by a start event (circle). Thereafter, the sequence flow is split by an exclusive gateway (diamond shape). If the ordered product is available, it is prepared and shipped, which is expressed by activities (rectangles). Otherwise, a notification is sent to the customer. Thereafter, the sequence flow is joined again by another gateway, and the process terminates as indicated by the end event (circle).

In the following, only well-structured BPMs are treated, i.e., we require splits and joins to be properly nested. This restriction improves the understandability of process models in the same way as structured programming improves the understandability of program code. For well-structured BPMs, moreover, powerful syntactical user assistance is available [16].

3 The Frameworks *PerSUADE* and *DSketch*

The general approach proposed in this paper (Fig. 1) is generic as it is not restricted to a particular visual language. Hence, it requires frameworks for sketch recognition as well as syntax analysis and user assistance that are generic as well, i.e., they must be adaptable to different visual languages. Concretely, we have chosen the *DiaGen* approach [17] as a base, where hypergraphs are used as a model for diagrams and hypergraph grammars as a means for syntax definition. Accordingly, this formalism is introduced at first. Thereafter, the *PerSUADE* approach, an extension of *DiaGen* by syntax-based user assistance, is introduced. Finally, the sketching approach *DSketch*, which is also based on *DiaGen*, is recapitulated.

3.1 Hypergraphs and Hypergraph Grammars

Hypergraphs are generalized graphs whose edges can connect an arbitrary number of nodes. This notion of graphs allows a uniform representation of all kinds of diagrams. The key idea is that diagram components are represented by hyperedges and their attachment areas by nodes. Fig. 2 also shows the chosen hypergraph representation of the example BPM. The hyperedges are drawn as rectangular boxes and the nodes as black dots. If a hyperedge and a node are incident, they are connected by a line called tentacle. Activities and events have two attachment areas, i.e., incident nodes: one for incoming and one for outgoing sequence flow. Gateways have four attachment areas (namely their corners). Note that sequence arrows do not explicitly occur in this hypergraph representation. They are rather represented implicitly by the fact that connected components visit the same node: the source component via its outgoing tentacle, the target component via its incoming tentacle, respectively. The hypergraph shown in Fig. 2 actually is the result of a lexical analysis step, which performs such simplifications.

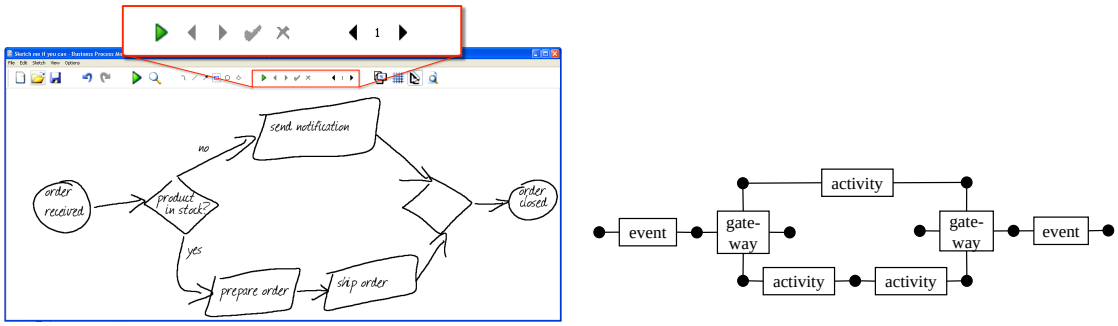


Figure 2: A sketched BPM fragment and its hypergraph representation

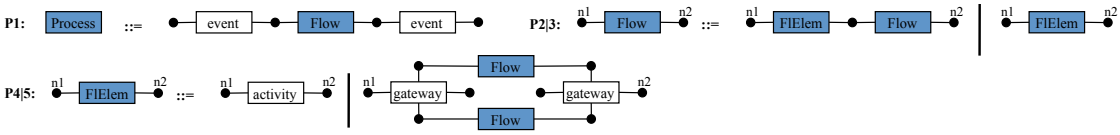


Figure 3: Hypergraph grammar for BPMs

In *DiaGen*, hypergraph grammars are used for language definition. For this paper, only context-free ones are considered [11]. Such hypergraph grammars consist of two finite sets of terminal and nonterminal hyperedge labels and a starting hypergraph that contains only a single nonterminal hyperedge. Syntax is described by a set of productions. The hypergraph language generated by a grammar is defined by the set of terminally labeled hypergraphs that can be derived from the starting hypergraph. Fig. 3 shows the productions of a hypergraph grammar G_{BPM} for very simple process models. A more comprehensive version that includes pools (process containers), different kinds of intermediate events, and embedded messages has been shown in [16]. The types *event*, *activity*, and *gateway* are the terminal hyperedge labels. The set of nonterminal labels consists of *Process*, *Flow*, and *FlElem*. The starting hypergraph consists of just a single *Process* edge. The application of a context-free production removes an occurrence e of the hyperedge on the left-hand side of the production from the host graph and replaces it by the hypergraph H_r on the right-hand side. Matching node labels of both sides of a production determine how H_r has to fit in after removing e.

3.2 User Assistance with *PerSUADE*

For visual languages defined by hypergraph grammars, *hypergraph patches* have been proposed as a means for the realization of Syntax-based User Assistance in Diagram Editors (*PerSUADE*) [15]. A patch basically describes a modification of a given hypergraph H that transforms H into a valid member of the language defined by a given grammar G. Two different kinds of atomic modifications are considered: merging nodes and adding edges. The application of a patch for a hypergraph H then corresponds to the construction of a so-called quotient hypergraph $H/\sim$ whose nodes are equivalence classes of the original nodes of H. Correcting patches indeed can be computed while parsing hypergraphs [15]. Consider the hypergraph H given in Fig. 4 as an

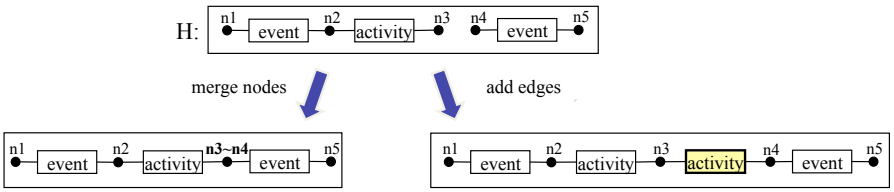


Figure 4: Hypergraph patches by example

example. Hypergraph H does not belong to the language of G_{BPM}, but it can be corrected by merging the nodes n3 and n4. It can also be corrected by inserting an *activity* hyperedge at the proper position. Note that there usually is an infinite number of correcting patches. Actually, according to G_{BPM}, an arbitrary number of activities could be inserted between the *activity* and the *event* hyperedge at the right. So, the size of desired patches, i.e., the number of additional hyperedges, must be restricted by the user. A special case of patches is the empty input hypergraph. Its patches can be used for exhaustive example generation.

Assistance based on hypergraph patches has been integrated into *DiaGen* editors as follows: The editor automatically maintains the hypergraph representation of the diagram. On user's request, the patch-computing parser is applied to this hypergraph representation with the desired size of patches as a parameter. It computes all possible correcting hypergraph patches of this size. From those, the user has to choose via a preview functionality. The selected patch is translated to diagram modifications by a language-specific *update translator*. Finally, the diagram is beautified by a *layout* component. That way, powerful syntax-based user assistance for BPMs has been realized already [16] – however, only in the context of a conventional WIMP editor. A screencast is available at www.unibw.de/inf2/DiaGen/assistance/bpm.

For the computation of patches, *PerSUADE* can only consider the context-free part of a hypergraph [16]. This limitation naturally also applies to the sketch editors with guidance to be discussed in Section 4.

3.3 Diagram Recognition à la *DSketch*

DSketch is an extension of *DiaGen* that complements the conventional WIMP-based GUI of diagram editors by a drawing canvas, which readily accepts all kinds of user strokes freely entered with a stylus. The integrated recognizer allows diagrams to be analyzed and further processed [6]. The main characteristics of this approach are: (i) Little restrictions to drawing components, e.g., a rectangle can be drawn clockwise, counterclockwise, or even interleaved with other components. (ii) Syntactic and semantic information is used to resolve ambiguities that occur in the recognition process. For instance, if a sloppily drawn BPM component could be both an activity or an event, the actual decision is postponed to the analysis stage where the interpretation of the respective strokes might get clear from the context. And finally (iii), the approach is generic, i.e., editors for a wide range of languages can be specified.

Fig. 5 shows the overall architecture of this sketching approach. The first processing step is the *recognizer*, which analyzes the sketch's strokes and creates a corresponding set of diagram components. Actually, several primitive recognizers (called transformers in [5]) for lines,

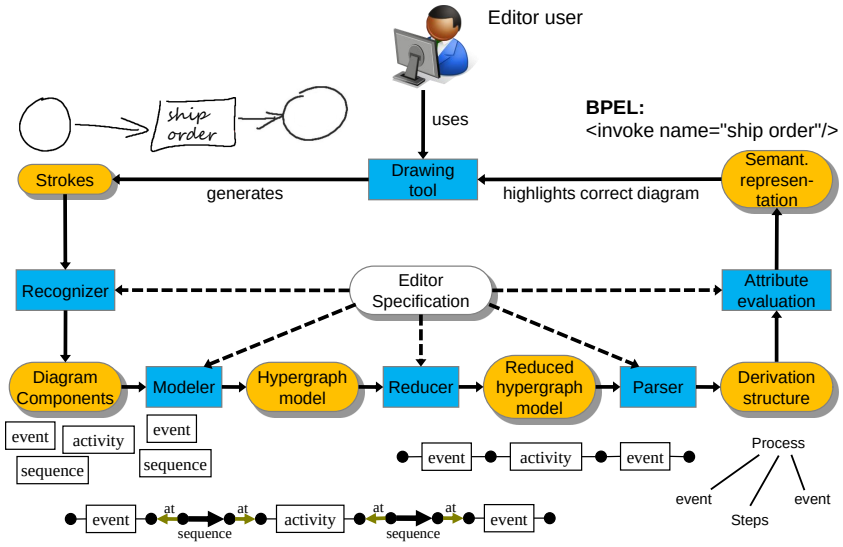


Figure 5: Architecture of *DSketch* [6]

arcs, circles, etc. search for corresponding primitives in the sketch. The main recognizer queries these primitive recognizers and – directed by the language specification – assembles the diagram components from those primitives. Generally, the recognition is very tolerant to avoid false negatives. The inevitably resulting false positives are resolved not until parsing. The actual analysis of the diagram now works in several steps similar to the analysis in conventional *DiaGen* editors: First, a hypergraph model is created from all components. Then the reducer is applied (lexical analysis) and yields the reduced hypergraph model as shown in Fig. 2. The parser syntactically analyzes this hypergraph and builds up a derivation structure that is similar to a derivation tree, but that also reflects non-context-free aspects of the diagram. The parser ensures that no two possible interpretations of the same stroke are integrated into the same derivation structure. That way, ambiguities are effectively resolved. Each derivation structure then represents a correct diagram and is rated according to its quality. Finally, a semantic representation of the best-rated derivation is computed via attribute evaluation. If this is not possible, the next best derivation is tried and so on. Details about this process can be found in [4].

The *DSketch* approach is efficient and fully functional, but it cannot recognize dashed lines nor distinguish different line widths. BPM messages, which are usually drawn as dashed lines, and BPM end events, which are drawn as fat circles, thus, must be represented with another notation. Moreover, text recognition is not integrated into *DSketch*. Textual labels, hence, must be entered via keyboard or an extra text recognizer.

4 Integration of User Assistance into *DSketch*

In this section we describe how the assistance provided by *PerSUADE* has been integrated into *DSketch*. The overall architecture of the editors generated by the realized system is shown in Fig. 6, which basically refines Fig. 1. The right-hand side of Fig. 6 comprises the analysis steps

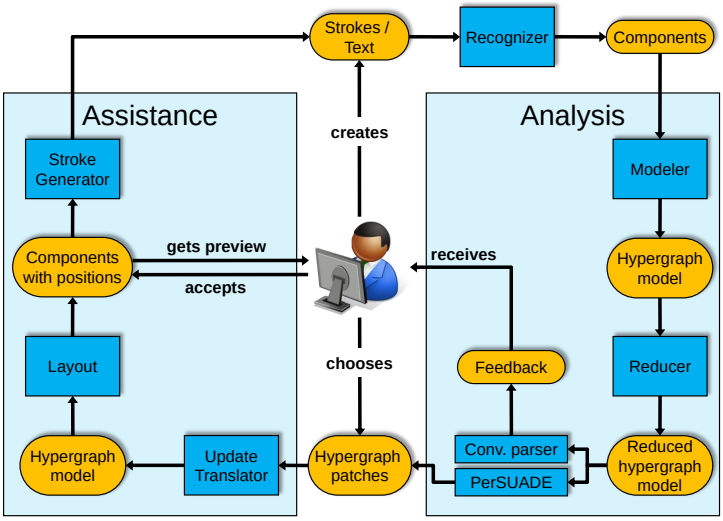


Figure 6: Novel architecture of sketch editors

of *DSketch*. The analysis performed by the *PerSUADE* parser belongs to this side, too. The left-hand side comprises the novel part of the system where the results of *PerSUADE* are further processed for the sake of assistance.

The processing steps up to the parser remain almost unchanged. Just the recognizer needed to be slightly adapted. Recall that in *DSketch* the recognizer is very error-tolerant. So, often the same stroke is accepted by several different primitive recognizers. This results in double findings that are resolved in *DSketch* during syntax analysis. The *PerSUADE* framework cannot deal with such ambiguities yet. Therefore, we have enforced the recognizer to make the decision if one of the assistance functions is invoked. Basically, the recognizer now selects the interpretation with the highest rating from the double findings. This rating depends on how precisely a primitive is drawn, how close the connections at the junctions are, and how particular constraints are met.

The next adapted component is the parser, i.e., the process of syntactically analyzing the diagram. Actually, the *DSketch* parser has remained unchanged, but an additional parser component from the *PerSUADE* framework now complements it. All kinds of assistance are supported by this parser instead of the normal *DSketch* parser. On user's request, this parser computes hypergraph patches for (the recognized parts of) the diagram's reduced hypergraph model. The user can explore these patches and choose one of them using a preview functionality.

Let us assume that the user has selected one of the patches. Consider the example traced in Appendix A. There, the smallest existing patch just merges the outgoing node of the activity and the incoming node of the right-most event. The update translator translates this patch into changes of the hypergraph model. For our example, an arrow needs to be introduced that is attached to the activity and the event (indicated by the spatial relationship edges "at"). Thereafter, it is up to the layouter to find an appropriate position for the newly introduced components. For the example of Appendix A, this is an easy task because the source and target components of the sequence arrow already exist. The more complex completion examples given in Fig. 7 and the generated example diagrams given in Fig. 8, however, show that this step is not always that

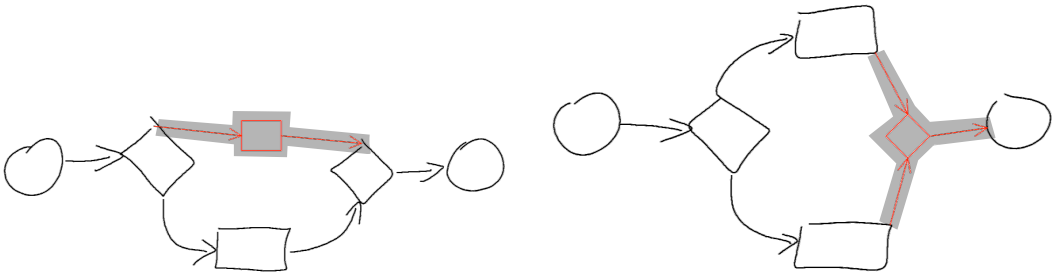


Figure 7: Auto-completion examples. Suggested completions are drawn in red.

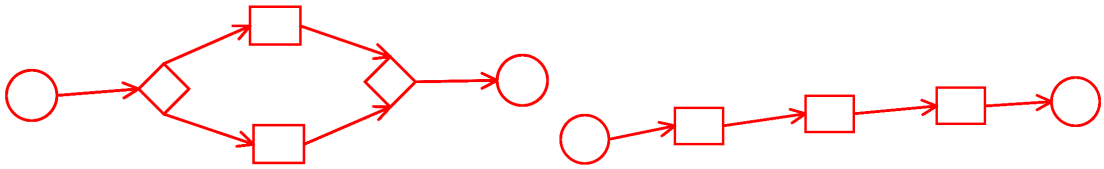


Figure 8: Example generation

simple. The used layout approach and the actual user interface are discussed in the following subsections. The last processing step, i.e., the stroke generator, is rather simple. It just draws perfect components with the optimal sample rate, thus maximizing the recognition rate.

4.1 Placement of New Components by the Use of Graph Drawing Techniques

A basic assumption of our implementation is that user strokes remain unchanged (in contrast to conventional *PerSUADE* editors, where the existing components can be adapted during assistance). That way, surprises are prevented and the special flavor of sketching is preserved. So, we need a flexible layout engine for graph-like languages (other languages would require some adaptations) that only integrates the new components and leaves the remaining diagram unchanged. These requirements can be satisfied by layout algorithms based on physical analogies [3]. Concretely, we have adapted a spring embedder, which interprets edges as springs with their particular attraction forces. Furthermore, special repulsive forces take effect between the node components. During layout, the node components move in increments according to the respective sum of forces until an equilibrium state has been reached. However, in our context not all nodes can be moved around freely, but only the new ones introduced by the update translator. The existing nodes, in contrast, are locked into their positions. An important benefit of spring embedders besides their simplicity is that they can also be used for static layout in a straightforward way. Static layout is required here, e.g., for example generation (cf. Fig. 8).

There are two problems with spring embedders in our context: The top diagram of Fig. 7 would look much better if the new activity were positioned further to the top. However, spring forces pull the new activity to the presented position, i.e., springs prevent bent arcs that way. This behavior can only be avoided by introducing invisible components in a context-sensitive way.Another problem is that new components, if positioned randomly at the beginning, cannot

"pass" existing components due to the repulsive forces. This may result in strange layouts. We have prevented this problem by introducing an additional processing step that guesses more appropriate initial coordinates for new node components to be refined afterwards.

Other layout algorithms might yield more common looking process models; actually, most professional business modeling tools apply Sugiyama-style layout algorithms. However, such algorithms are less suited in the context of this paper where the (usually user-chosen) position of existing nodes must be preserved when new nodes or edges have been introduced.

4.2 User Interface

The actual user interface is quite simple and easy to use — the complete editor window is shown in Fig. 2. There is a button for starting the computation of patches (see the magnified part of Fig. 2). After pressing this button, the first solution is shown immediately. Arrow buttons can be used for browsing through other solutions. In particular the generation of examples usually results in a large number of solutions. A check button has to be pressed in order to accept the currently previewed solution. Strokes are then generated from the previewed diagram components. The resulting diagram looks like the preview, but the new components are not highlighted anymore, but drawn as normal, although perfect, strokes. Note that the user does not need to accept the previewed solution, but can use it as a kind of draft and draw the same components with his own strokes. That way, he will get a diagram that looks more homogeneous than the one with the generated perfect strokes. To continue with the UI, the preview also can be canceled, of course. Finally, the patch size can be set via plus and minus buttons. This parameter basically indicates how many new diagram components (or more precisely, terminal hyperedges) are to be introduced. In the figure this parameter is set to its default value 1.

5 Discussion

Although an elaborate user study still remains to be done, the results so far are promising. As before, the user can freely sketch diagrams. He is not restricted in any way in the creative process of sketching. This actually is the reason why we have not realized a more pervasive assistance, e.g., on-line after every single stroke. In many conventional sketch editors, the user is in trouble if his sketch cannot be recognized. With the developed editor, he can ask for syntactical guidance instead. But this is not the only help he can get as we describe next.

5.1 Location of Recognition Errors

An important benefit of our approach is that it helps identifying *recognition errors* (besides syntax errors). Consider Fig. 9 as an example. A human can easily see that the sketched diagram is a structured business process. Still the diagram is not correctly recognized by *DSketch*. Normally, the user would have no clue what is wrong here. Has the start event mistakenly been recognized as an activity? Or has the end event been drawn too sloppily? Invoking assistance yields the answer. The red arrow between the activity and the end event clearly points out the problem: either the existing arrow has not been correctly recognized, or the gap between its head and the end event is too large. In either case, the user now can correct this problem without the need

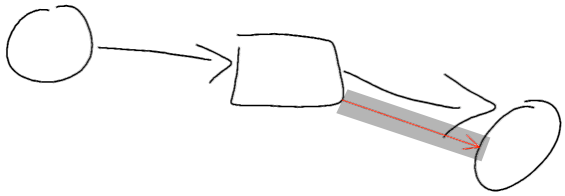


Figure 9: Identification of recognition errors

to redraw the whole diagram. It would even be possible to automatically mask or remove those strokes that do not contribute to the solution.

This problem, however, mainly arises for quite restricted visual languages where either the whole diagram is correct or nothing. Indeed, a single misrecognized arrow affects the correctness of the whole BPM. It simply is not well-structured anymore as we have required by the grammar. With a more relaxed syntax definition at least sub-diagrams would be recognized correctly so that the visual feedback given by the *DiaGen* parser might indicate what is wrong. Actually, languages, where either the whole diagram or nothing is recognized as correct, have been very critical for sketching systems so far, because the recognition rate exponentially drops down with the size of the diagram. This problem is solved with our approach (although it would be even better to re-feed the analysis result in the recognizer, so that it can try harder at the weak points).

5.2 Limitations

A problem of integrating *PerSUADE* into a sketching system is that it may happen that a sketch is not recognized as correct after a suggested patch has been accepted by the user. When using *PerSUADE* in conventional WIMP editors, this cannot happen: diagrams resulting from the application of assistance are always correct. In the context of sketching, newly generated strokes may interfere with existing strokes that, e.g., had been ignored by stroke recognition before.

6 Related Work

Of course, there are also other sketch editors for BPMs such as [14]. Moreover, due to the practical relevance of this language, various kinds of guidance have been developed for conventional WIMP-based BPM environments (an example is [2]). However, to our best knowledge this guidance has not been integrated into sketch editors yet.

As already noted in the introduction, the most closely related work is [9] by Costagliola et al. Here, an LR parser as known from textual languages is used for syntax analysis with respect to a so-called sketch grammar. Thereby, syntactic information is exploited to resolve ambiguities similar to the *DSketch* approach [10]. The symbol table of the parser then can be exploited to realize symbol completion in sketch editors (and so-called symbol prompting in conventional diagram editors). The strong points of this approach are that it is generic, that direct feedback is provided (the approach is actually incremental), that the user's own drawing style is used for completion (a stroke repository is filled by the different symbol recognizers to this end), and that the recognition of complex symbols can generally be improved that way. But, like with

our approach, explicit user interaction is still required. In contrast to [9], our approach does not stop at the lexical level, but also considers the overall diagram structure. Even other kinds of assistance not necessarily based on syntax could be integrated.

Another meta-tool where it should be possible to combine assistance with sketching is the Marama toolkit. For Marama, both a critic authoring tool [1] for the specification of user feedback and a sketching framework [12] are available. Here, however, critics would have to be specified manually whereas we gain the feedback automatically from the parser. The strong points of [12] are that only very little extra specification effort is needed for complementing a normal diagram editor with a sketching editor and that the user can easily overrule the recognizer when it makes a mistake.

7 Conclusion

In this paper we have shown that user assistance functionality can be successfully added to sketch editors. The presented approach allows to generate sketching editors with user assistance from a language specification based on the existing sketching editor generator *DSketch* and the user assistance library *PerSUADE*. As a representative example, we have created a sketch editor for business process models with assistance features such as auto-completion or example generation.

But we have noticed yet another benefit of this approach besides helping the user with the language. The very same assistance features actually can be put to a good use in locating recognition errors. Those often directly result in syntax errors, whose fixes then point the user precisely to the recognition error. If a new component is suggested as a correction where already a component exists, the user can conclude that the existing component had been drawn too sloppily and needs to be redrawn.

The developed sketch editor for business process models is demonstrated in several screencasts and can be downloaded from www.unibw.de/inf2/DiaGen/assistance/sketching.

Future Work

In the future we want to experiment with relaxations of the assumption that the existing user strokes must not be changed. It is certainly imaginable that sketched components are moved around or even resized similar to the assistance in conventional *DiaGen* editors [16]. In this context it should also be possible to integrate existing component fragments into the newly introduced components in order to reuse as many strokes of the user as possible.

It would be also important to integrate the suggestions into the diagram closely following the user's drawing style. Perfect components mixed with sloppily drawn components make the diagram look inhomogeneous. Costagliola et al. have proposed a stroke repository to this end, which is used already for their symbol completion [9]. Alternatively, the user strokes could be beautified to close this gap.

Finally, *DSketch* and *PerSUADE* need to be more deeply intertwined. While *DSketch* originally postpones final decision of stroke recognition until syntax analysis in order to improve the recognition rate and to make ambiguity resolution possible, we had to enforce early recognition decisions in order to integrate *PerSUADE* into *DSketch*.

Bibliography

[1] N. M. Ali, J. Hosking, J. Huh, and J. Grundy. Critic authoring templates for specifying domain-specific visual language tool critics. In *Proc. 2009 Australian Software Engineering Conference*, pp. 81–90. IEEE, 2009.

[2] M. Born, C. Brelage, I. Markovic, D. Pfeiffer, and I. Weber. Auto-completion for executable business process models. In *Business Process Management Workshops*, *LNBIP* 17, pp. 510–515. 2009.

[3] U. Brandes. Drawing on physical analogies. In *Drawing Graphs: Methods and Models*, *LNCS* 2025, pp. 71–86. 2001.

[4] F. Brieler and M. Minas. Ambiguity resolution for sketched diagrams by syntax analysis based on graph grammars. In *Proc. GT-VMT'08*, *ECEASST*, vol. 10. 2008.

[5] F. Brieler and M. Minas. A model-based recognition engine for sketched diagrams. In *Proc. VL/HCC Workshop on Sketch Tools for Diagramming*, pp. 19–28, 2008.

[6] F. Brieler and M. Minas. Recognition and processing of hand-drawn diagrams using syntactic and semantic analysis. In *Proc. AVI'08*, pp. 181–188. ACM, 2008.

[7] R. Chung, P. Mirica, and B. Plimmer. InkKit: A generic design tool for the tablet PC. In *Proc. 6th ACM SIGCHI NZ chapter's Int. Conf. on CHI*, pp. 29–30. ACM, 2005.

[8] G. Costagliola, V. Deufemia, and M. Risi. Sketch grammars: A formalism for describing and recognizing diagrammatic sketch languages. In *Proc. ICDAR'05*, pp. 1226–1231. IEEE, 2005.

[9] G. Costagliola, V. Deufemia, and M. Risi. Using grammar-based recognizers for symbol completion in diagrammatic sketches. In *Proc. ICDAR'07*, pp. 1078–1082. IEEE, 2007.

[10] G. Costagliola, V. Deufemia, and M. Risi. Using error recovery techniques to improve sketch recognition accuracy. In *Proc. 7th Int. Workshop on Graphics Recognition*, *LNCS* 5046, pp. 157–168. 2008.

[11] F. Drewes, A. Habel, and H.-J. Kreowski. Hyperedge replacement graph grammars. In *Handbook of Graph Grammars and Computing by Graph Transformation. Vol. I: Foundations*, pp. 95–162. World Scientific, 1997.

[12] J. Grundy and J. Hosking. Supporting generic sketching-based input of diagrams in a domain-specific visual language meta-tool. In *Proc. 29th ICSE*, pp. 282–291. IEEE, 2007.

[13] T. Hammond and R. Davis. LADDER, a sketching language for user interface developers. *Computers&Graphics*, 29(4):518–532, 2005.

[14] N. Mangano and N. Sukaviriya. Liberating expression: A freehand approach to business process modeling. In *Proc. 12th IFIP TC 13 Int. Conf. on HCI*, *LNCS* 5727, pp. 834–835. 2009.

[15] S. Mazanek, S. Maier, and M. Minas. Auto-completion for diagram editors based on graph grammars. In *Proc. VL/HCC'08*, pp. 242–245. IEEE, 2008.

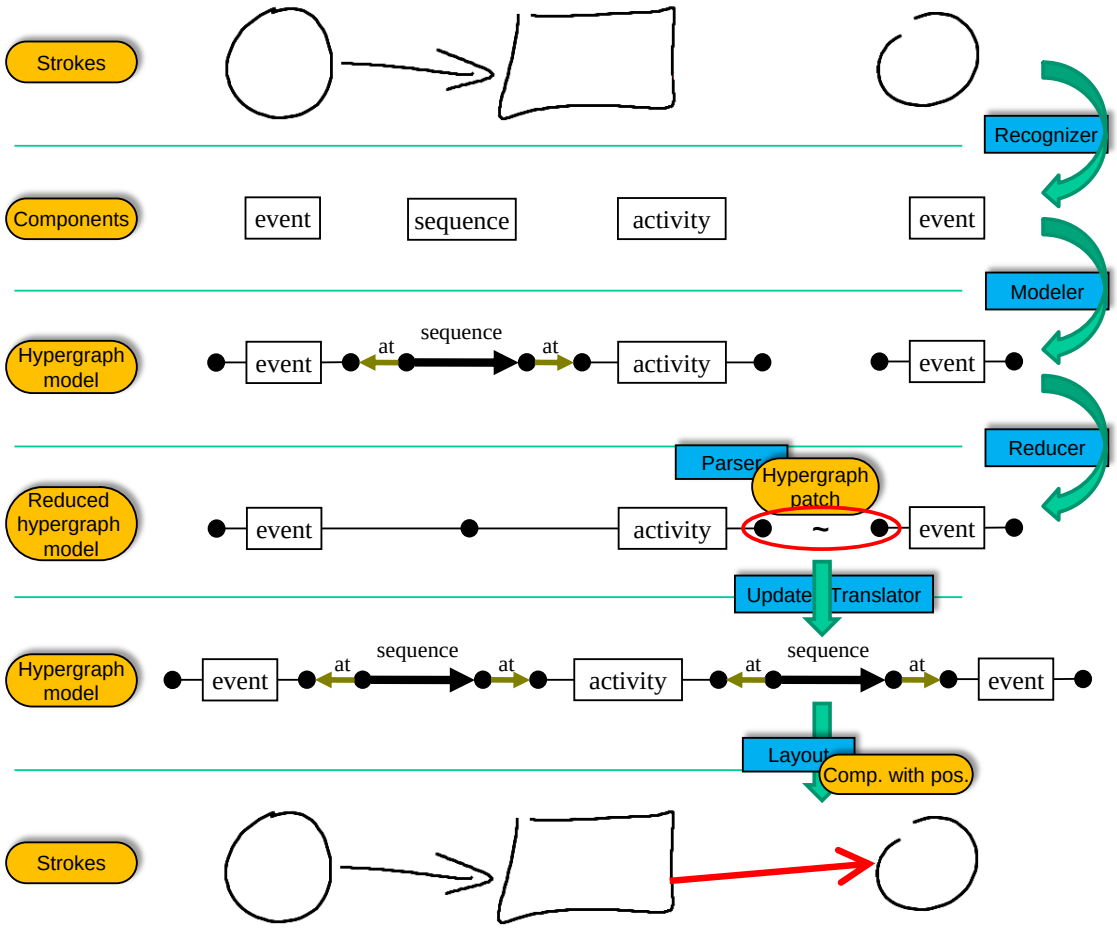
[16] S. Mazanek and M. Minas. Business process models as a showcase for syntax-based assistance in diagram editors. In *Proc. MoDELS'09*, *LNCS* 5795, pp. 322–336. 2009.

[17] M. Minas. Concepts and realization of a diagram editor generator based on hypergraph transformation. *Science of Computer Programming*, 44(2):157–180, 2002.

[18] Object Management Group. Business Process Model and Notation (BPMN). http://www.omg.org/docs/formal/09-01-03.pdf. 2009.

[19] S. Sen, B. Baudry, and H. Vangheluwe. Domain-specific model editors with model completion. In *Models in SE*, *LNCS* 5002, pp. 259–270. 2008.

A Processing steps by example



Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation

Torsten Strobl, Mark Minas, Andreas Pleuß, Arnd Vitzthum

13 pages

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

From the Behavior Model of an Animated Visual Language to its Editing Environment Based on Graph Transformation

Torsten Strobl[1], Mark Minas[1], Andreas Pleuß[2], Arnd Vitzthum[3]

[1] [Torsten.Strobl,Mark.Minas]@unibw.de, Univ. der Bundeswehr München, Germany
[3] Andreas.Pleuss@lero.ie, Lero, Univ. of Limerick, Ireland
[4] Vitzthum@informatik.tu-freiberg.de, Technische Univ. Bergakademie Freiberg, Germany

Abstract: Animated visual models are a reasonable means for illustrating system behavior. However, implementing animated visual languages and their editing environments is difficult, so guidelines, specification methods, and tool support are necessary. A flexible approach for specifying model states and system behavior is to use graphs and graph transformations. Thereby, a static graph does not necessarily represent a static view on the system; instead, it can be visualized with animations. Graph transformations are triggered over time in order to control the system behavior. This means starting, modifying, and stopping animations as well as changing the system structure, e.g., by adding or removing elements to or from system components, respectively. These concepts had already been added to *DiaMeta*, a framework for generating editing environments, but they provide only low-level support for specifying and implementing animated visual languages; specifying complex dynamic systems was still a challenging task. This paper proposes the modeling language *AML*, which allows for modeling behavior and animations on a higher level of abstraction by decomposing a dynamic system into its basic constituents. Models of this language are then translated into a low-level specification based on graph transformations. The approach is demonstrated using a traffic simulation.

Keywords: animated visual language, behavior modeling

1 Introduction

Visual modeling languages (VLs) are widespread in engineering and computer science. Several frameworks and tools have been realized that make implementing VLs, i.e., providing tool support for such models, easier. *DiaGen/DiaMeta* [6], GenGED [1] or AToM^3 [5] are only a few of them. The majority of VLs are static; each model is a static diagram. However, there are also dynamic VLs with animated diagrams, e.g., *Pictorial Janus* [3] or *ToonTalk* [8] and many further examples in simulation, biology, or chemistry. However, there is still a lack of tool support, so VLs usually have to be implemented manually. Recently, a new approach for specifying interactive dynamic VLs based on graph transformation (GT) has been proposed and realized within the *DiaMeta* tool [12]. Previous approaches based on GT (e.g., [1]) use graphs for representing static model states whereas the effects of GTs can be animated. Graphs in the new approach [12], in contrast, do not necessarily represent the static aspect of a model, but rather its dynamic aspects. GTs, when triggered at specific points of time, modify such graphs and implement the dynamic behavior of the system. As a consequence, GTs can start, change, or stop

animations, for example. GTs can easily describe interactions within the model or between user and model, too. This approach allows for specifying dynamic VLs including rather complicated animations and interactions, e.g., several concurrent animations may simultaneously take place in a model. However, GTs without further abstraction mechanisms do not provide sufficient support for the easy specification of complex animations. This paper addresses this problem and proposes the Animation Modeling Language (*AML*), which is based on previous work with the Multimedia Modeling Language (MML) [10] and the Scene Structure and Integration Modeling Language (SSIML) [14]. Its purpose is the modeling of behavior and animations on a higher level of abstraction. It allows to decompose a dynamic system into its basic constituents and to describe behavior by hierarchical automata. *AML* models can then be refined and transformed into a specification for *DiaMeta* following the approach presented in [12].

The rest of the paper is structured as follows: The next section introduces a traffic simulation as the running example. Section 3 then briefly outlines the specification approach for animated interactive VLs presented in [12]. *AML* and the translation of *AML* models into a specification for *DiaMeta* based on GT are described in Sections 4 and 5. Section 6 reports on related work, and Section 7 concludes the paper.

2 Running Example: *Traffic*

Traffic simulations can be considered as complex dynamic systems. The modeling of the behavior for each traffic participant in such simulations is non-trivial because participants have to take care of many different situations. For this running example, simplified but still complex aspects of a traffic simulation have been chosen and realized in a system called *Traffic*. The road network in this system contains the following components: Roads, Intersections and EntryExit points. Thereby, a Road always connects two of the other components. It is not necessarily straight, i.e., it can also have turns based on cubic curves. An Intersection always has a predefined dimension and one connection point for each cardinal direction. Finally, there are EntryExit points which are special elements where cars can enter or exit the simulation.

Each Intersection has one 4-state TrafficLight (states Go, Caution, Stop, and Ready) for each direction. The duration of states Go and Stop is configurable by a property interval available for each Intersection, and as a special feature for interactivity, the user is also allowed to click on an Intersection in order to trigger an immediate switch. Each EntryExit "produces" cars randomly. A parameter randomNext specifies the maximal amount of time between two cars. Cars that arrive at an EntryExit are "consumed" by the EntryExit and disappear.

Cars can accelerate and brake, but they have a fixed maximum speed. Cars must stop at a traffic light if it is in states Stop or Caution as long as there is enough time for stopping. At a predefined distance in front of each Intersection, Cars also indicate the aim to turn left or right with the corresponding turn signals, and their straight motion without operating the signals. This decision is also randomized for each Intersection. Cars have to obey apparent rules before and while turning, e.g., left-turning drivers do not have priority when oncoming traffic blocks the road. In addition, drivers also have to watch cars in front of them and must start braking if the car in front is stopping, and if the safety distance is violated otherwise. If the front car is starting again, a minor delay time for restarting the car behind is applied in order to imitate real world

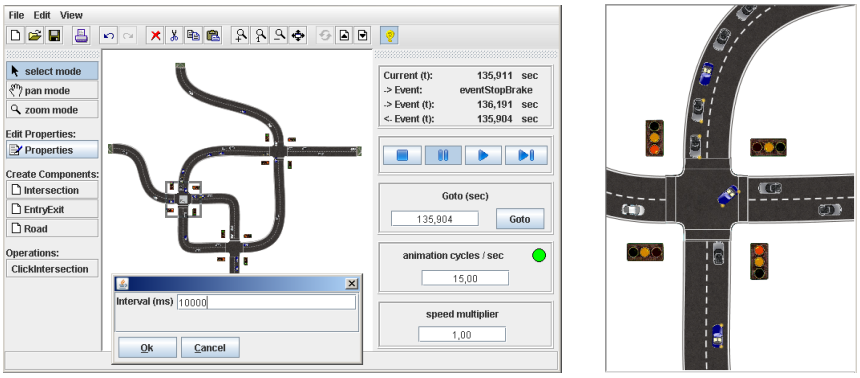


Figure 1: Screenshots: (a) editor during animated simulation, (b) zoomed

flow of traffic. Finally, Cars also have to watch traffic jams, i.e., they must stop in front of an Intersection if the destination street is jammed.

Fig. 1 shows two screenshots of a *Traffic* editor whose code has been generated based on a *DiaMeta* specification. An animated example can also be found online[1]. While (a) shows the whole editor which was used to build the shown road network, (b) is a zoomed illustration during animation runtime. Although the specification allows for the generation of a *Traffic* editor, the following mainly focuses on dynamic aspects of *Traffic* shown in this editor.

3 Animation by Graph Transformation with *DiaMeta*

The GT-based approach for specifying animated interactive VLs described in [12] uses hypergraphs for internally representing animated visual models. Each model component (e.g., a Car, a Road, or an Intersection in the *Traffic* example) is represented by a *component hyperedge* that visits the nodes representing the component's attachment points. These hyperedges carry attributes representing properties of their component, e.g., its position. Model hypergraphs also contain *relation edges* (binary hyperedges), that stand for relationships between components, and further *link hyperedges* which have multiple purposes as shown later. The visual model is just a view of the hypergraph and depends on the hypergraph's attributes, but also on the continuously proceeding time. This means that a hypergraph without changing its structure or its attributes may still represent a visual model that is currently being animated, i.e., the hypergraph represents both the current structure of a visual model and its current animation state. Changes of the structure and animation state are the result of events, which may be triggered externally, e.g., by the user, or internally, e.g., when a running car must start breaking because it is approaching an intersection with a red light. With regard to such events, our approach is closely related to *DEVS* (cf. [12]).

External as well as internal events are GT programs consisting of GT rules together with a control program that modify the hypergraph of the animated visual model together with its attributes. However, GTs representing external events are handled differently from GTs rep-

[1] http://www.unibw.de/inf2/DiaGen/animated

resenting internal events: A GT that represents an external event is performed immediately as soon as the external event is triggered, e.g., when the user pushes a button. An internal event, in contrast, occurs after a certain amount of time depending on the current structure of the visual model and its animation state. Therefore, the specification of an internal event consists of a GT and a *time calculation rule*. This special rule is used for determining the point of time when such an internal event occurs: Whenever the hypergraph of a visual model is changed due to an event, the runtime system has to check which internal events may happen next. This is done by examining the GTs of all internal events and checking whether they are enabled. However, the enabled transformations are not yet actually performed. Instead, the time calculation rules are used to compute the points of time when the events will occur. The GT of the earliest internal event is actually performed at the computed point of time if no external event has been triggered meanwhile, changing the model's hypergraph and possibly removing other scheduled internal event. This procedure of computing the next internal event is repeated after each modification of the model's hypergraph.

This specification approach has been realized within the *DiaMeta* tool and has been used for several animated VLs as described in [12]. However, this specification approach can hardly be applied to more complex animated VLs without analyzing required structures and events first. The large amount of required events (resp. GT programs), which appear confusing, is a inhibition threshold for realizing the VL specification. The following section introduces the more abstract modeling language *AML*, which addresses this problem.

4 Animation Modeling Language

Because of the complexity of *Traffic*, we propose to use an appropriate modeling language before specifying an animated VL and its behavior for *DiaMeta*. This section introduces the Animations Specification Language *AML*, which is based on previous work with two other modeling languages MML [10] and SSIML [14]. The long-term goal of *AML* is to define general concepts for modeling interactive animations which can be applied in other modeling languages, e.g. for multimedia or augmented reality. In the context of this paper, *AML* is used for a high-level specification of animated VLs from which the GT-based implementation can be derived. A simplified metamodel of *AML* is shown in Fig. 2 which presents *AML* as extension of UML elements.

The main structural elements in *AML* are visual elements. We adopt the concept of *media components* [10] to model them. A media component encapsulates some media content together with basic functionality, e.g., methods that render or play the media content, such as audio, video, 3D graphics. However, we will focus on 2D graphics in the following. Each media component, such as 2D graphics, provides some standard properties and operations depending on its media type, e.g. size and position for graphics. In addition, custom properties, operations, and associations can be specified in the same way as for UML components. For instance, Fig. 3 shows the graphics component for Intersection and its custom property interval, but without custom operations. Standard properties and operations need not to be specified explicitly in the model.

A media component usually consists of several parts, e.g., a video is composed of multiple images or a graphic consists of different shapes. These inner parts can be important for the application's behavior: For instance, complex animations often consist of multiple parts, such

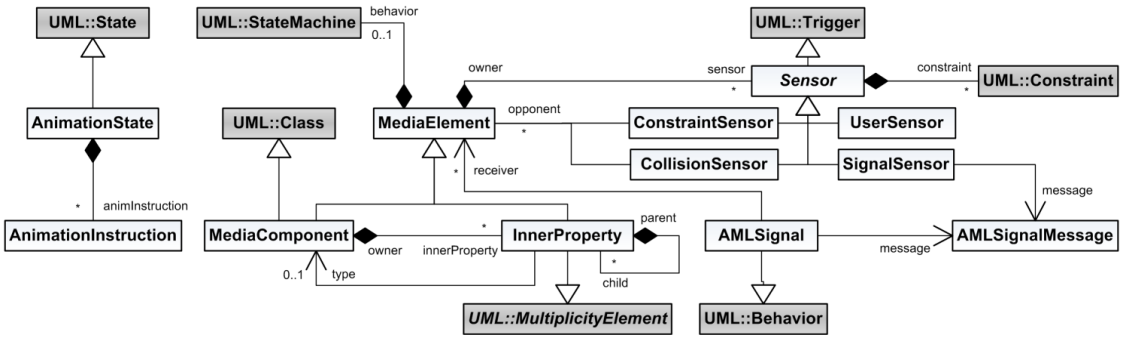


Figure 2: *AML* metamodel (simplified)

as a moving car whose turn signals should blink when it drives while turning. This requires that the graphics of the car's turn signals are not part of the car's graphic itself, but attached graphics which can be accessed at runtime. Moreover, distinction between different parts is also important for event handling: For instance, the user might trigger different functionality when clicking on the intersection itself or one of its traffic lights. Therefore, the inner structure of media components needs to be defined in *AML*.

A media component's inner structure is defined in terms of *inner properties* which are organized in a hierarchical manner. Manipulations of parent properties also affect their children; e.g., if the car turns, the attached turn signals must turn as well. An inner property has a name, an optional type, and an optional multiplicity. A type needs only to be specified to indicate that this inner part is an instance of another media component. If no type is specified, the media part is just an instance of an anonymous media component. A multiplicity can be specified to indicate multiple instances. For instance, the Intersection graphics in Fig. 3 consists of a roadsCrossing which contains four trafficLights. The latter ones are instances of another media component TrafficLight while for roadsCrossing no further type is specified.

Dynamic behavior of animations is modeled using specific kinds of events. In conventional interactive applications, events are mainly triggered by user actions, such as pressing a button. However, applications with complex animations require additional kinds of events resulting from the dynamic behavior of media components. For instance, behavior should be triggered if a moving object reaches a specific position or touches another object on the screen (e.g., a car reaches a traffic light) or if a certain point of time is reached.

In *AML*, this is modeled by different kinds of *sensors*. Four kinds are presented in this paper. Fig. 3 shows several sensors which are denoted similar to an accept event action in UML. Fundamentally, a sensor is owned by a media component or by a media component's inner property. A *user sensor* listens for user events, such as clicks on a specific component. For instance, ClickIntersection listens for a user click on roadsCrossing which is a part of an Intersection. A *signal sensor* such as NextLightSignal is similar, but is used for passing events internally, so media components can pass messages to each other. A *collision sensor* has a relationship to one or more other graphic components (called *opponents*) and triggers an event when its owner collides with its opponent(s), i.e., when they overlap on the screen. The collision sensor DecideDirection owned

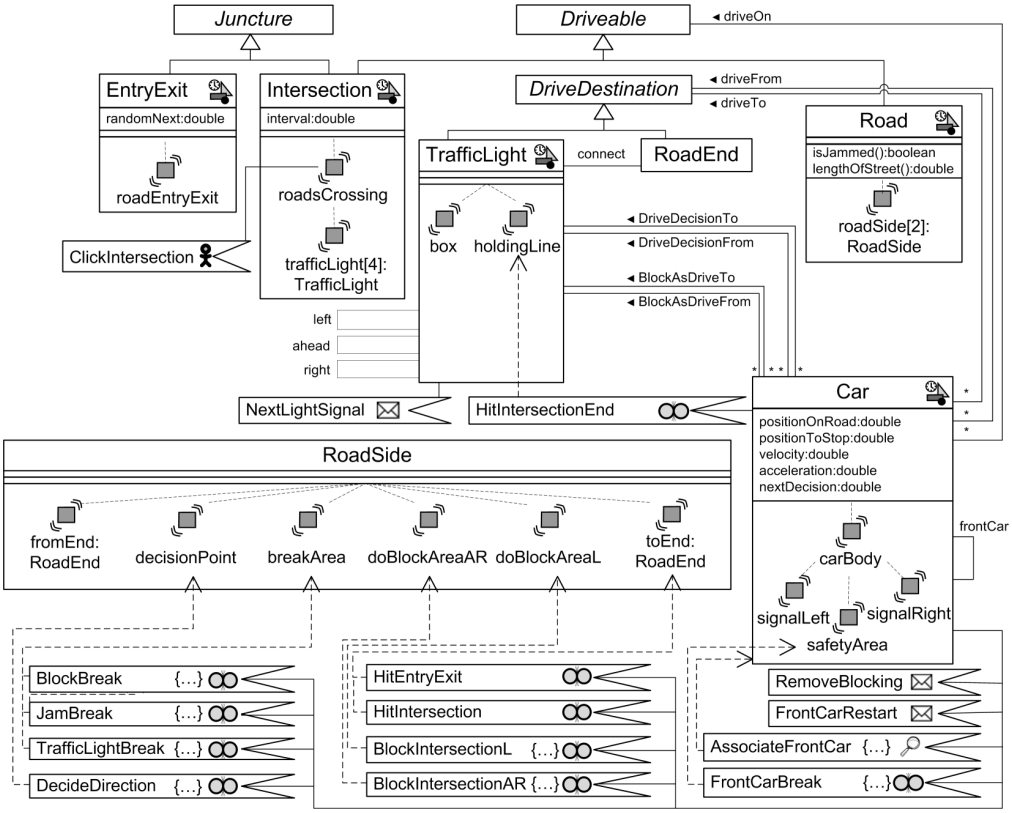


Figure 3: *AML* model: media components, associations, inner structures and sensors

by the graphic component Car triggers an event as soon as the car reaches a specific area on its street. As a consequence, the driver decides upon a direction and starts indicating, if necessary.

A sensor can also be associated with OCL constraints to specify that the sensor is active only under certain conditions. Most sensors in Fig. 3 contain constraints, but details are not illustrated there. Several additional keywords such as owner, opponent (sensor) or parent (inner property) are available in constraint expressions allowing to refer to involved components or to navigate through their structures. A special kind of sensors, the *constraint sensor*, is always modeled with OCL expression. The purpose of this sensor type is to observe connected components for satisfying the OCL constraint. An example is sensor AssociateFrontCar which checks whether a car is driving behind another one on the same RoadSide, but they have not been associated using frontCar yet. The sensor can fire an event which associats both cars then.

The behavior and animations of an media component or an inner property is modeled by a special kind of state machine (see Fig. 4). As different aspects of one component can have their own behavior, multiple state machines resp. regions can be executed in parallel. The state machines basically support the same concepts as state machines in UML, i.e., states, pseudo states, parallel states, state transitions, guards, and activities associated with states or transitions. Expressions in the state machines can refer to all properties and operations of its owner.

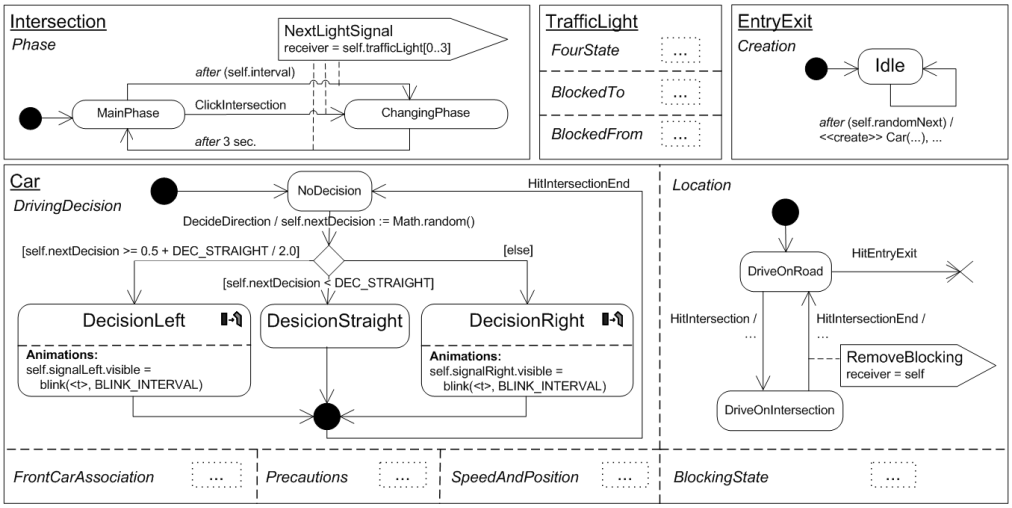


Figure 4: *AML* model: state machines and animation behavior (excerpt)

The most important triggers for transitions are the sensors (see above). The sensor's name can be denoted at the transition which means that the transition is performed when the sensor triggers an event. Next to events effected by sensors, it is also possible to use *elapsed time events* or *change events* as trigger for transitions. The former is indicated by keyword after and occurs automatically after a period of time in the state, the latter is indicated by when and occurs as soon as annotated constraints are satisfied.

Finally, transitions can also send signals to other components. In *AML*, this is denoted explicitly by a special kind of send signal action where one or more receivers of the signal can be specified explicitly (using the keyword receiver). Such signals are designed for corresponding signal sensors of the receiver. After receiving a signal, these sensors trigger other events, if possible. An example in Fig. 4 is NextLightSignal which is sent by an Intersection to each of its TrafficLights at the same time one of the attached state transitions is performed.

The most important *AML*-specific concept is a special kind of state: *animation states*. They define the change of properties over time while the owner of the state machine is in this state (i.e., the animation of the graphic component or a part of it). Animation states have a small symbol in the top right corner and *animation instructions* at the bottom. Within these instructions, properties of media components, e.g., position or angle, can be bound to expressions. Fig. 4 shows two examples of animation states. For instance, state DecisionLeft describes the blinking of the car's left turn signal by switching the visibility attribute of graphic component SignalLeft on and off depending on the elapsed time (denoted by $<t>$) and a constant for the interval.

5 Translating *AML* Models to *DiaMeta* Specifications

AML models describe all dynamic aspects of an animated VL. This section describes how a *DiaMeta* specification (called "specification" in the following) can be derived from an *AML* model

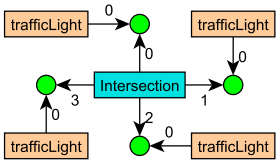


Figure 5: Intersection and TrafficLight

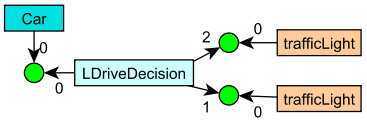


Figure 6: Car associated with TrafficLights

(called "model"). So far, there is no automatic translation process from model to specification. Instead, a model is manually translated into a specification using five steps (see Fig. 8):

(1) Static components and dynamic components of the VL represented by *AML* media components must be specified. (2) The resulting specification must be extended by constructs allowing for the representation of all states within the *AML* model. (3) GT rules must be derived from the model in order to map each state transition. (4) Resulting GT rules must be specified as *DiaMeta* events. (5) Animations during animation states must be considered.

Each of the following subsections describes one step for *Traffic*. Please note, that only a rough picture of these steps can be presented due to the space limitations of this paper. The translation starts with the specification of static VL components (1.1) and dynamic VL components (1.2).

(1.1) As a first step, the *AML* media component model (see Fig. 3) must be investigated for static language elements. While independent media components are declared as regular VL components, media elements that are used as inner properties must be specified as *sub-components*, i.e., they are represented by so-called *sub-component (hyper-)edges* being connected to the component hyperedge of its parent component through appropriate nodes. Fig. 5 shows an Intersection as a component hyperedge and its trafficLights as sub-component edges.

(1.2) The *AML* model usually contains further media components and associations that are used during animation only (e.g., Car); such elements, hence, are maintained by state transitions of the behavioral model. For example, *AML* associations belonging to dynamic aspects require the specification of so-called *animation (hyper-)edges* which can link components in order to represent associations. Fig. 6 shows animation edge LDriveDecision which corresponds to both *AML* associations DriveDecisionFrom and DriveDecisionTo. The shown graph represents a car which has decided to drive from the first traffic light attached to LDriveDecision to the second one.

(2) The next step is to translate the state machine states (see Fig. 4). Thereby, the state of the dynamic system must be expressible by the graph representing the model (cf. Sec. 3), i.e., specifications which allow for the characterization of each component's state are required. In some cases, no additional specification is necessary because a state can already be determined implicitly because of animation edges. The state of a Car's state machine DrivingDecision, for instance, is already characterized by edge *LDriveDecision* (cp. Fig. 6). Depending on the linkage to trafficLight edges, it is determined if the driver wants to drive straight, left, right, or has not decided yet (no such edge). Another specification option for embedding state information is to create an explicit state attribute for a (sub-)component edge storing the currently active state.

(3) Next, state transitions modeled in *AML* are analyzed in order to derive GT rules that, when executed, realize the behavior of the *AML* state machines. As described above, states are encoded

in the hypergraph of an animated model. A GT rule simulating a transition from state A to state B, hence, must be enabled if the state machine is in state A and also the guard conditions of the transition are met, which includes guards of triggers. As a result, the rule has to modify the hypergraph model such that it contains the encoding of state B. In the simplest case, switching from state A to B means changing the state attribute of a component. In addition, the established GT rule resp. program also has to perform actions described by the *AML* state transition.

(4) Each state transition shown in the last section is triggered by events. As explained in Sec. 3, two kinds of events can be distinguished and specified for *DiaMeta*, so the following subsections address the specification of external events (4.1) and internal events (4.2).

(4.1) In *AML*, events are observed by sensors, i.e. user sensors such as ClickIntersection in case of external events. In terms of *DiaMeta*, the specification of an external event means the specification of a GT program which is executed directly if the user selects a component and pushes a defined key or button. This event-related GT program, and the following is also true for internal events, must choose the applicable state transition depending on the recipient's current state and execute the according GT rule specified in the previous section. In addition, OCL constraints of sensors must be included in the GT program.

In a sense, but not correct in terms of definition, receiving signals (e.g. NextLightSignal) can also be considered as external event for the recipient. Therefore, signal sensors can be realized by a regular GT program which can be executed directly.

(4.2) The *DiaMeta* specification of internal event specifications follows the guidelines described in the previous step, except that internal events are not published for external systems resp. the user. However, the time calculation rule for internal events must still be specified. In order to derive meaningful additional constraints and time calculation rules, different *AML* elements (cf. Sec. 4) indicating internal events are described in the rest of this step:

The simplest internal events found in the *AML* models are *elapsed time events* indicated by keyword *after*. In this case, the GT rule is already complete, and the required time calculation rule can be deduced directly by adding the argument of the elapsed time event (e.g., "3 sec") to the time the state has been entered, which implies that this *state entry time* must be tracked at corresponding state transitions (e.g., by maintaining an appropriate component attribute). Another kind are *change events* indicated by keyword *when*. They can also result in a simple time calculation rule, i.e., the rule that always returns the time when the internal event is calculated. This means that the state transition is performed immediately if the graph pattern and additional conditions match. This implies that arguments of the change event must become part of the GT rule's graph pattern and conditions. However, if a condition depends on a value which changes during animation, the time calculation rule, which has to calculate the first point of time the condition holds true, becomes more complex.

Finally, constraint and collision sensors must trigger internal events. Both are similar to change events (see above). A usual difference is that constraint sensors observe the relation of multiple components instead of values of a single component. Therefore, the graph pattern is extended by the hyperedges of each (sub-)component which is observed by the sensor. Collision sensors are similar to constraint sensors, but they imply additional constraints that indicate colliding components. This constraint is often related to movement animations of components. Because animations can be visualized between state changes, the time calculation rule has to supply the point of time when the first collision happens. An exemplary collision sensor is Traf-

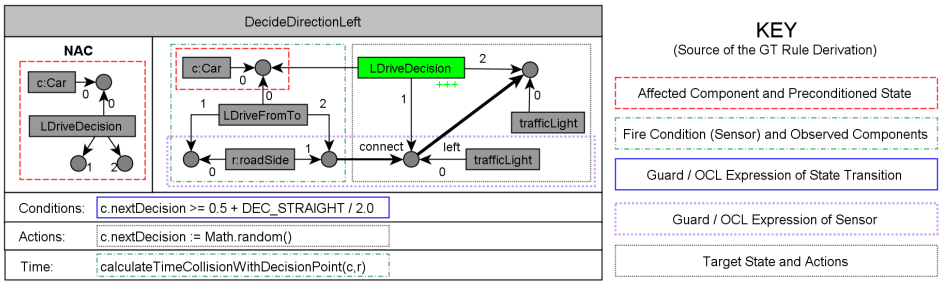


Figure 7: Event specification for DecideDirectionLeft and according state transition

ficLightBreak. The required internal event specification can be found in Fig. 7. The figure[2] also illustrates from which *AML* elements specific parts of the rule have been derived from.

(5) The remaining part is the interpretation of the *animation states* within the *AML* model, e.g., a Car which is moving along a Road or a Car with an activated turn signal signalLeft (see Fig. 4). The animation instructions stated in animation states indicate which and how the attributes of the component must be changed while an animation state is active (cp. Sec. 4). These instructions must be transfered into the *DiaMeta* specification. In terms of *AML*, property changes are performed while in a state without any transition necessary. For *DiaMeta* this means that graphical primitives are not drawn using static attributes (such as the position), but using dynamic values derived from such attributes and modified by referring to the current animation time and to the associated animation instruction. Finally, when leaving or entering an animation state, it is often necessary to update static attributes which have been the basis for animation visualization before. Within the *DiaMeta* specification this must be done by the GT rule which realizes the corresponding state transition. For instance, a Car is moved during an animation state called Drive. The car's position (static attributes) must be updated when leaving this state. Otherwise, the system would have no information about the position of the Car after the movement animation.

6 Related Work

AML integrates concepts from two existing modeling languages: the *Scene Structure and Integration Modeling Language* (SSIML) for 3D development and the *Multimedia Modeling Language* (MML) for interactive multimedia applications. SSIML/Behaviour [14] is a modeling language based on UML2 state machines which also introduced animation states for modeling animation details. Although SSIML/Behaviour focuses on 3D applications. MML [10] is a platform-independent language for model-driven development of multimedia applications. It provides concepts such as complex media components with inner structure, user interface elements, and sensors, which have been reused here. However, it does not support modeling animations yet and will be extended with the concepts presented in this paper in the future.

[2] It shows the GT rule within one segment: parts which are added by the rule are drawn in green with "+++". Attribute conditions and modifications are illustrated by expressions within the two extra boxes *Conditions* and *Actions*. The time calculation rule is shown in box *Time*.

In the area of interactive multimedia, there are only few other modeling approaches so far. OMMMA [11] allows modeling multimedia applications including media objects. Dynamic behavior is specified by statecharts while static animations are modeled by extended UML sequence diagrams. However, the models published so far are on a high level of abstraction and not intended for code generation. Kienzle et al. [4] presented a modeling approach for computer games illustrated by a tank game. They use statechart on multiple abstraction layers for different aspects which are specially suited for game design: sensors, memorizers, strategical deciders, tactical deciders, and actuators.

AML allows for separated behavior specifications of different components, which interact in a common environment. This possibility makes *AML* also attractive for the specification of domain-specific visual languages for agent-based modeling and simulation. In this field of research other tools are already available, e.g., in [9] a toolkit for specifying the behavior of agents and their visual appearance within a modeling environment is demonstrated.

There are also other concepts for the animation of VLs based on GT. An example is the approach described by Ermel [1] which basically allows for the specification of animations for discrete event simulations. However, animations are visualized for state transitions resp. GTs which restricts VLs especially in terms of interactivity and parallelism of independent animations. The resulting animations are self-running movies, and amalgamated GT rules are already required for the specification of less complex examples such as animated Petri nets.

In [7] a set of visual languages is introduced in order to describe the behavior of metamodeled languages. The concepts are also based on events and state machines, but most languages are less close to UML. In addition, the concepts include the modeling of user interfaces, whereas aspects of smooth graphical animations are not covered. Concerning the flexibility, the approach is less adequate for a language with many independently animated actors resp. agents such as in *Traffic*, because the behavior is represented by one state machine.

Another approach for the application of models in order to implement graph-based simulations is shown by Syriani [13]. He describes how DEVS models can be used as a semantic domain for programmed GTs which allows for simulation-based design. However, the concepts describe GTs consuming time and because of possible needs for parallel executions or interruptions, additional control structures are required.

7 Conclusions

In order to generate the animated *Traffic* editor, the dynamic aspects of the *DiaMeta* specification have been systematically derived from an *AML* model. Thereby, *AML* has been used for modeling the behavior of components, their interaction, and reasonable animation descriptions. Different levels of details can be chosen, so *AML* can illustrate rough ideas as a starting point for *DiaMeta* specifications, but also models which are very close to the resulting specifications. On the other side, fully featured *AML* diagrams even document *DiaMeta* specifications which are usually less explanatory. Using *AML* and *DiaMeta* is a promising approach for simplifying the specification effort for complex VLs.

We now aim at an automated transformation of customized *AML* diagrams into *DiaMeta* specifications, so further research objectives include platform-specific extensions of *AML* allowing

for the automated generation of graph-based VL specifications. In this context, a reasonable set of predefined animation instructions must also be found and offered by our framework in order to avoid self-programmed routines realizing animations or collision detection.

Bibliography

[1] C. Ermel. *Simulation and Animation of Visual Languages based on Typed Algebraic Graph Transformation*. PhD thesis, Tech. Univ. Berlin, Books on Demand, Norderstedt, 2006.

[2] S. Gyapay, R. Heckel, D. Varró. Graph Transformation with Time: Causality and Logical Clocks. In *Proc. ICGT '02*, LNCS 2505, Springer, 2002, pp. 120–134.

[3] K. Kahn, V. Saraswat. Complete Visualizations of Concurrent Programs and their Executions. In *Proc. 1990 IEEE Workshop on VLs*. 1990, pp. 7–15.

[4] J. Kienzle, A. Denault, H. Vangheluwe. Model-based Design of Computer-Controlled Game Character Behavior. In *Proc. Models 2007*, LNCS 4735, Springer, 2007, pp. 650–665.

[5] J. de Lara, H. Vangheluwe. AToM3: A Tool for Multi-formalism and Meta-modeling. In *Proc. FASE '02*, LNCS 2306, Springer, 2002, pp. 174–188.

[6] M. Minas. Generating Meta-Model-Based Freehand Editors. In *Proc. GraBaTs'06*. Electronic Communications of the EASST 1. 2006.

[7] T. Mészáros, G. Mezei, H. Charaf. Engineering the Dynamic Behavior of Metamodeled Languages. In *Simulation* 85(11):793–810, 2009.

[8] L. Morgado, K. Kahn. Towards a specification of the ToonTalk language. In *J. of Visual Languages and Computing* 19:574–597, 2008.

[9] M.J. North, E. Tatara, N.T. Collier, J. Ozik. Visual Agent-based Model Development with Repast Simphony. In *Proc. of the Agent 2007 Conf. on Complex Interaction and Social Emergence*. 2007.

[10] A. Pleuß. MML: A Language for Modeling Interactive Multimedia Applications. In *Proc. 7th IEEE Int. Symp. on Multimedia (ISM'05)*. IEEE, 2005, pp. 465–473.

[11] S. Sauer and G. Engels. UML-based Behavior Specification of Interactive Multimedia Applications. In *IEEE Symp. on Human-Centric Computing Languages and Environments (HCC 2001)*. IEEE, 2001, pp. 248–255.

[12] T. Strobl, M. Minas. Specifying and Generating Editing Environments for Interactive Animated Visual Models. In *Preproc. GT-VMT'10*, 2010. http://www.cs.le.ac.uk/events/gtvmt10/GT-VMT07PreProceedings.pdf

[13] E. Syriani, H. Vangheluwe. DEVS as a Semantic Domain for Programmed Graph Transformation. In *Discrete-Event Modeling and Simulation: Theory and Applications*. CRC Press, 2009.

[14] A. Vitzthum. SSIML/Behaviour: Designing Behaviour and Animation of Graphical Objects in Virtual Reality and Multimedia Applications. In *Proc. 7th IEEE Int. Symp. on Multimedia (ISM'05)*. IEEE, 2005, pp. 159–167.

A Appendix

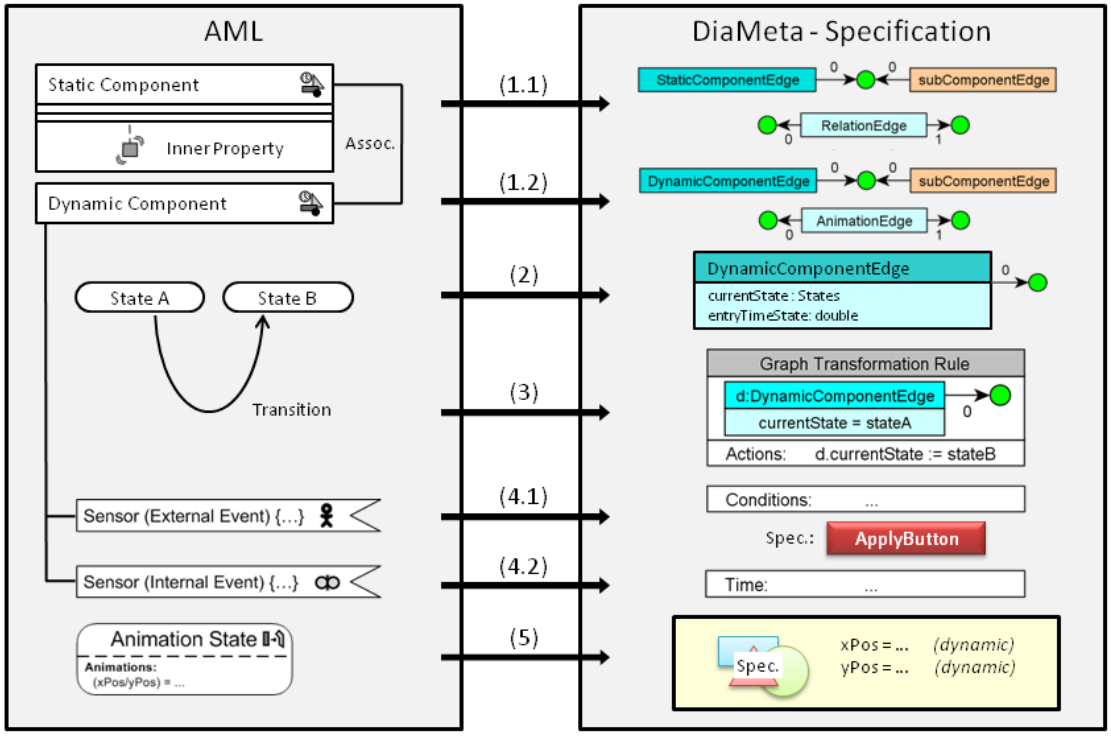


Figure 8: Overview of the translation steps

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Design of a SOM Business Process Modelling Tool based on the ADOxx meta-modelling Platform

Domenik Bork and Elmar J. Sinz

12 Pages

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

Design of a SOM Business Process Modelling Tool based on the ADOxx Meta-modelling Platform

Domenik Bork and Elmar J. Sinz

Department of Information Systems – Systems Engineering
University of Bamberg
Feldkirchenstraße 21
D-96045 Bamberg
{domenik.bork | elmar.sinz}@uni-bamberg.de

Abstract: The Semantic Object Model (SOM) is a comprehensive methodology for business systems modelling. An emphasis of SOM is on modelling of business processes. According to SOM, a business process model specifies the task layer of a business system from an inside perspective. SOM business process modelling is grounded in systems theory and organisational theory. A SOM business process model is perceived as a distributed system, consisting of business objects which are coordinated in business transactions. Both business objects and business transactions can be refined recursively. SOM business process models are specified using a graph-based multi-view approach which comprises a structural view, a behavioural view as well as views on the decomposition of business transactions and business objects.
This article reports on the design of a tool which facilitates multi-view modelling of SOM business processes. The tool is based on the ADOxx[1] meta-modelling platform. The focus of the article is on the design of the multi-view approach and corresponding tool functions on the basis of the ADOxx platform.

Keywords: Semantic Object Model (SOM), ADOxx, graph-based multi-view modelling, tool design

1 Introduction

The Semantic Object Model (SOM) is a comprehensive methodology for business systems modelling (Ferstl and Sinz 1995, Ferstl and Sinz 1996, Ferstl and Sinz 2008). The conceptual framework of the SOM methodology is an enterprise architecture which comprises the layers of *enterprise plan*, *business process model* and *specification of resources* (Figure 1). The enterprise plan constitutes an outside perspective on an enterprise. It focuses on the global enterprise task and the resources to fulfil this task. The business process model constitutes an inside perspective on an enterprise, specifying the tasks and task relations collectively carrying out the global enterprise task. Thus, a business process model can be considered as a procedure for executing the enterprise task. Finally, the third layer specifies the resources needed to fulfil the business processes, particularly personnel for the execution of non-automated tasks and business application systems for the execution of automated tasks.

[1] ADOxx is a registered trademark of BOC AG

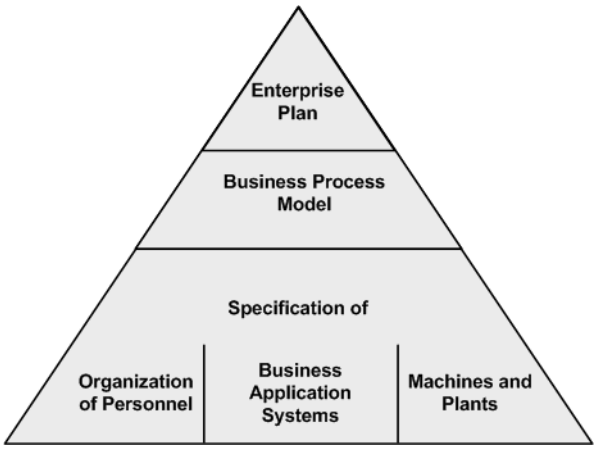


Figure 1: Enterprise Architecture of the SOM methodology (Ferstl and Sinz 2006)

This paper concentrates on the middle layer of the SOM enterprise architecture, the business process model. A SOM business process model is specified according to a graph-based multi-view approach. Based on an integrated internal representation, two different diagrams, an *interaction schema (IAS)* and a *task-event schema (TES)*, representing a structural view and a behavioural view on the business process model, are defined.

In contrast to other modelling approaches which focus on the drawing of a specific diagram at a particular time, the SOM methodology utilises an integrated kind of modelling. At any time, the different views are derived from an integrated model. Moreover, the decomposition of the different artefacts of a business process model is an integral part of the model, too. This characteristic allows navigation through the model by zooming in and zooming out the areas considered at a particular time.

Over the last two decades several software tools supporting the SOM methodology have been developed (e.g. Ferstl et al. 1994). These tools helped to utilise the SOM methodology and enabled its application even to industry-sized projects. However, based on native software platforms like C++, these tools were not suitable to provide enduring availability. To overcome this shortness, it was decided to use a specialised software platform for tool development, allowing high software productivity, easy adaption and extension of the methodology, and integration into a tool family sharing the same platform as well as bridging different methodologies.

This article reports on the design of a new SOM tool which is aimed to meet the requirements mentioned above. The tool is based on the ADOxx[2] meta-modelling platform. After a short introduction to SOM business process modelling and the ADOxx platform, the design of the software tool is outlined by means of mapping the SOM meta-model to the ADOxx meta-meta-model, graph-based visualisation of the multiple views on a business process model as well as tool functions and modelling transactions.

[2] ADOxx is a registered trademark of BOC AG

The paper is organised as follows: Chapter 2 gives a brief introduction to basic concepts of the SOM methodology, especially to SOM business process modelling. Chapter 3 shortly introduces the meta-modelling platform ADOxx. The core of the paper is chapter 4, outlining the design of the SOM business process modelling tool on the ADOxx platform. The paper ends with some conclusions and an outlook to future work.

2 SOM Business Process Modelling

The meta-model for SOM business process modelling is shown in figure 2. According to this meta-model, a SOM business process model consists of a set of business objects, each belonging either to the considered business system (symbol: rectangle) or to its environment (symbol: oval). A business object encapsulates one to many tasks sharing common states and pursuing joint goals. A task drives one to many transactions (symbol: arrow), each of them driven by exactly two tasks belonging to different business objects. Each transaction either transmits goods or services from one business object to another or it participates in coordinating business objects or other transactions. Related tasks within a business object are coupled by internal events (symbol: circle). External events model occurrences in the environment of a business system (e.g. “the first day of a month”).

The SOM methodology for business process modelling utilises two different coordination principles (Ferstl and Sinz 2006).

- According to the *negotiation principle* a transaction is decomposed into a sequence of three transactions: (1) An initiating transaction T_i, where the objects learn to know each other and exchange information on deliverable goods or services, (2) a contracting transaction T_c, where both objects agree to a contract on the delivery of goods or services, and (3) an enforcing transaction T_e where the objects transfer the goods or services. The negotiation principle can be formally specified as
  $T(O,O') ::= [[T_i(O,O') \text{ seq }] T_c(O',O) \text{ seq }] T_e(O,O')$.

- According to the *feedback control principle* a business object O is decomposed into two sub-objects and two transactions: A management object O’, an operational object O’’ as well as a control transaction T_r from O’ to O’’ and a feedback transaction T_f in the opposite direction. These components establish a feedback control loop, specified by a set of components:
  $O ::= \{ O', O'', T_r(O',O'') [, T_f(O'',O')] \}$.

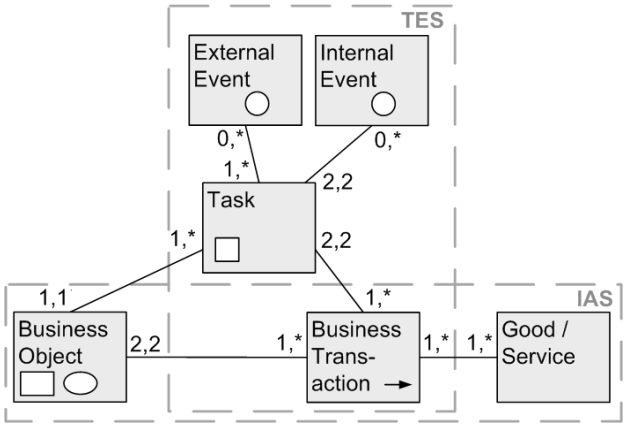


Figure 2: Business Process Meta-model of the SOM methodology (Ferstl and Sinz 2008)

The structural view (IAS) and the behavioural view (TES) are defined by projections onto the meta-model (figure 2). Except for the notion of business transaction the views are disjoint. Business transactions appear in both views representing the structural aspect (communication channel) and the behavioural aspect (event) respectively. Beyond that, decomposition of business objects and transactions establish two further views on a business process model.

To give an example, figures 3 and 4 demonstrate the IAS and TES of the simple business process model of a trading company. An initial transaction *distribution of goods* from *trading company* to *customer* has been decomposed according to the negotiation principle into an initiating transaction *information*, a contracting transaction *order* and an enforcing transaction *delivery*. The trading company itself has been decomposed applying the feedback control principle into a management object *sales*, an operational object *storage*, a control transaction *delivery order* and a feedback transaction *delivery report*. These two decompositions result in the IAS depicted in figure 3. Figure 4 shows the corresponding process flow by means of a TES. The TES specifies the sequence of the tasks obtained from the decompositions named above. The *information* transaction is performed by the tasks *I>* ("send information") of the business object *sales* and *>I* ("receive information") of the business object *customer*. After customer has received necessary information, he/she is able to send an *order* (task *C>*) to *sales* and so on.

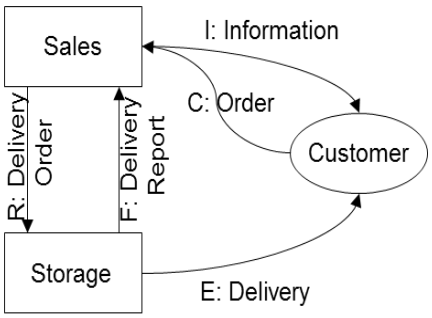


Figure 3: SOM Interaction Schema (IAS)

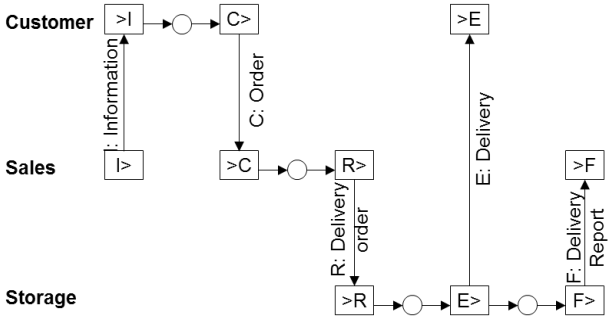


Figure 4: SOM Task-Event Schema (TES)

3 The ADOxx meta-modelling Platform

ADOxx is a meta-modelling platform which is aimed to facilitate design and implementation of modelling tools for domain specific languages (DSL). The platform has been developed by the BOC-Group[3], a spin-off of the University of Vienna. Over the last decades, ADOxx has been used to implement modelling tools for a wide area of domains like e-learning, knowledge management, strategic management and many others more (Schwab et al. 2010, Fill 2005, Lichka et al. 2002, Karagiannis and Bajnai, Karagiannis and Telesko 2000, Junginger et al. 2000).

ADOxx provides the designing engineer of a modelling tool with basic functions for representation and editing of diagrams, persistent storage of models, simulation and evaluation of models as well as import/export of models. To utilise these functions, a tool designer has to map the meta-model of the DSL onto the meta-meta-model of ADOxx. In other words, the meta-model of the DSL has to be specified using the concepts provided by the ADOxx meta-meta-model. In this way the ADOxx platform allows efficient design and implementation of both powerful and flexible DSL modelling tools.

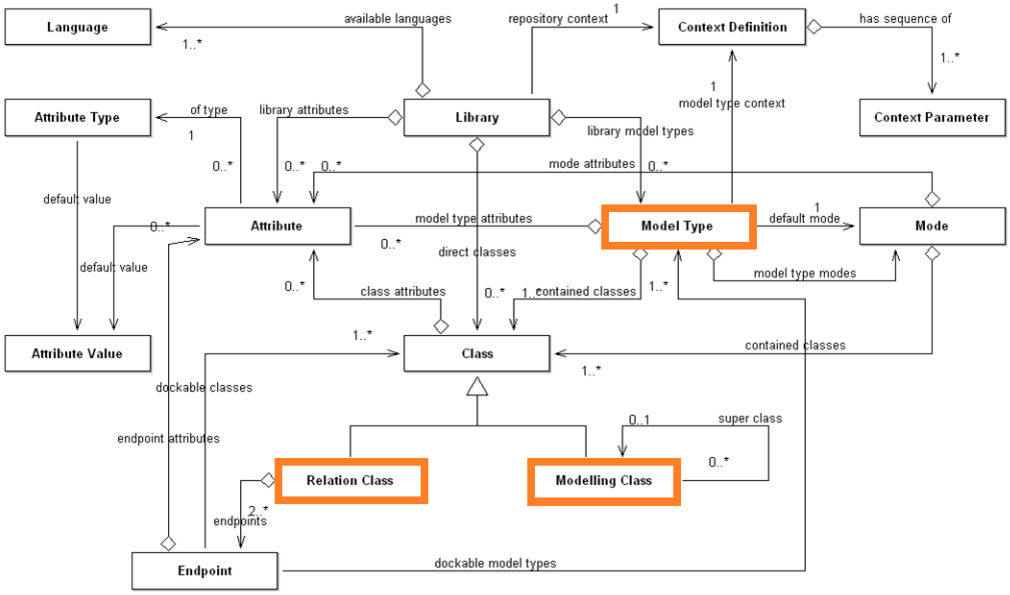


Figure 5: Extract of the ADONIS meta-meta-model (Junginger et al. 2000)

The ADOxx platform fosters the concept of meta-modelling which is widely used in the field of modelling (see e.g. OMG Meta Object Facility Object Management Group 2006). Within a hierarchy of meta levels a model (schema) at level 1 represents an instance of a corresponding meta-model at level 2 (Ferstl and Sinz 2008). The meta-model in turn is an instance of a meta-meta-model at level 3. Conversely, several meta-models which comply with a given meta-meta-model usually can be specified and many schemata may meet a given meta-model.

[3] http://www.boc-group.com/de/, last visit: 18.05.2010

On level 3 the ADOxx platform provides a meta-meta-model (often also called meta²model) defining some generic modelling classes and relations as well as corresponding attributes and constraints. The meta-meta-model is implemented in C++ and cannot be modified by a tool developer. The core of the ADOxx meta-meta-model is shown in figure Figure 5. The most important concepts, *Modelling Class, Relation Class and Model Type* are highlighted.

To implement a specific DSL on the ADOxx platform, the meta-model of the DSL has to be specified as an instance of the meta-meta-model. Thereby the concepts of the meta-model have to be mapped to those of the meta-meta-model (e.g. an activity of the BPMN language is mapped onto the concept modelling class). Afterwards, the meta-model is described in the ADONIS Library Language (ALL). Finishing this step, the meta-model of a methodology is defined within ADOxx. The tool is now ready to create simple diagrams, meaning that objects can be placed and connected by arcs. For methodologies like SOM, the more important and challenging step is to implement the tool functions and modelling transactions (e.g. decomposition rules, model visualisation, model consistency, zooming etc.). For this purpose the scripting language AdoScript is provided by the platform.

4 Design of a SOM Business Process Tool on ADOxx

The design of a modelling tool for SOM business process models within the ADOxx meta-modelling platform consists of three major steps. First, the meta-model for SOM business process models is mapped to concepts provided by the meta-meta-model of ADOxx (section 4.1). In the second step the visualisation of the models is conceived using a graph-based multi-view approach (section 4.2). The third step comprises the design of the tool functionality including the concept of modelling transactions (section 4.3). A modelling transaction is perceived as a sequence of editing operations which transform a consistent state of the model into a new state, which again is consistent according to syntax and semantics.

4.1. Meta-model Mapping

As mentioned above, the first step in creating a modelling tool with ADOxx is to define a mapping between the domain-specific meta-model and the meta-meta-model provided by the platform (Figure 5). In case of the SOM business process meta-model the mapping is basically defined as follows:

SOM Meta-model	ADOxx Meta-meta-model
Business Object	Modelling Class
Task	
Business Transaction	Relation Class
External/Internal Event	
Interaction Schema	Model Type
Task-Event Schema	
Object Decomposition	
Transaction Decomposition	

Table 1: Meta-model mapping

Furthermore, the mapping requires a specification of the syntax of the modelling language, particularly the feasible connections between classes. Finally the attributes (*AttrRep*) and a graphical representation (*GraphRep*) of the classes have to be defined.

As mentioned before, a SOM business process model consists of an internal representation and several corresponding views. The views are diagrams, representing an interaction schema, a task-event schema as well as the decomposition of business objects and business transactions. Both the internal representation of a model and the views require the definition of each a model type in ADOxx. A model type specifies a set of classes and relations between classes.

Having finalised these specifications, a modeller is able to create diagrams by dropping model elements on the drawing board and linking them by arcs. It is worth mentioning, that at this point the implementation of a tool for a common modelling language on ADOxx is done. However, SOM business process modelling is a sophisticated methodology which requires some more effort for tool implementation.

4.2. Graph-based multi-view visualisation

A SOM business process model is represented by several complementary views derived from one integrated internal model. The model-view-controller (MVC) paradigm (Reenskaug 1979) has been approved to be a suitable design pattern for such a requirement. For the design of the SOM modelling tool the MVC paradigm is utilised to propagate any action the modeller performs on a particular view to the internal model representation and from there to all other views as far as these views are concerned by these actions. For example, changing the name of a business object in one view causes changing the name in all other views showing the considered business object.

For the visualisation of SOM business process models three different types of diagrams, each corresponding to an ADOxx model type, have been designed and implemented on the platform:

1. *Decomposition diagram*
   Decomposition of both business objects and business transactions is represented using a tree-based graph. This style of visualisation helps the modeller to quickly recognise the recursive decomposition of objects and transactions. The decomposition of objects is displayed on the upper right side, the decomposition of transactions is shown on the upper left side of the tool window. Within a decomposition diagram, no information about the connections of business objects and business transactions is displayed. This information is subject of the following two diagrams.

2. *Structure diagram*
   This type of diagram is used to represent the structural view on a SOM business process model, the interaction schema (IAS). An IAS consists of business objects which are connected by business transactions. The IAS is displayed on the lower left side of the tool window.

3. *Behaviour diagram*
   Complementary to the structure diagram, this type of diagram shows the behavioural view on a SOM business process model by a task-event schema (TES). A TES consists of tasks, each belonging to a business object. Tasks are connected by internal events (if the tasks belong to the same business object) or business transactions (if the tasks belong to different business objects). The TES is displayed on the lower right side of the tool window.

Figure 6 illustrates the graph-based multi-view visualisation of SOM business process models on the ADOxx platform.

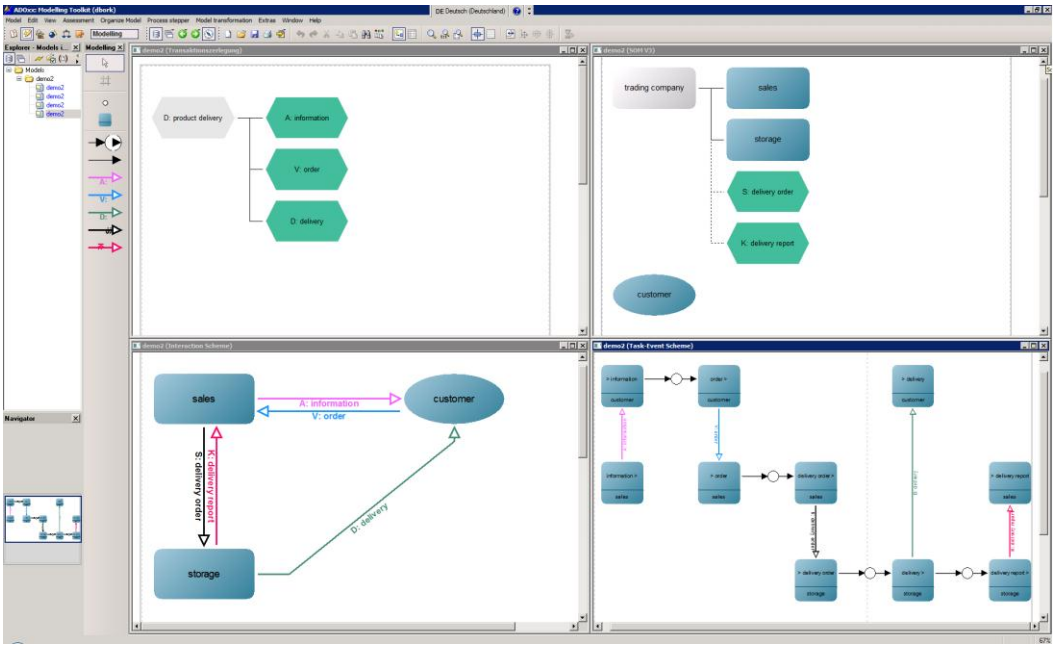


Figure 6: Graph-based multi-view visualisation in ADOxx

With increasing model size, comprehension of a SOM business process model, presented by four different views, can easily overstrain the modeller. Therefore, the visualisation of the model has been improved effectively using different techniques.

First, any object or transaction that is currently not visible in the IAS and TES will be grey-shaded in the object and transaction decomposition window. This retains the modeller from getting lost when handling large models. Furthermore, a red border is drawn around any object that is selectable in the process of reconfiguring transactions to new objects which are resulting from the decomposition of a given object (section 4.3 for a detailed description of the tools' functions).

Additionally, two layout algorithms have been implemented in order to optimize the positioning of business objects and the routing of the business transactions in the interaction schema:

- *Auto-Layout*
  The auto-layout algorithm draws every business transaction either directly, if the connected business objects are placed on the same vertical or horizontal position or with a right angle otherwise. Any additional ingoing or outgoing transaction is shifted a bit to the left or right alternatingly in order to prevent overlapping transactions.

- *Smooth Edges*
  Calling this algorithm ensures, that any business transaction is drawn on the most direct way between the connected business objects. Again, multiple ingoing or outgoing transactions are shifted a bit to the left or right.

Besides the discussed algorithms, the modeller is able to optimize the positioning of modelling elements and relations using the functionality provided by the platform (e.g. adding an edge to a relation or moving of elements with the associated relations on the modelling area using the mouse or keyboard).

4.3. Tool Functions and Modelling Transactions

As any modelling tool, the SOM tool provides a set of functions via its user interface. In many cases, a modelling step requires the execution of a sequence of functions. Such a sequence, transforming a consistent state of the model into a new state, which again is consistent with respect to syntax and semantics, is perceived as a modelling transaction. In the following, tool functions as well as modelling transactions are illustrated by means of use cases. Each use case refers to a typical scenario a modeller carries out when using the tool.

- *Decomposition of business objects and business transactions*
  The SOM methodology comprises a set of rules for the decomposition of business objects and business transactions (chapter 2 for selected rules). The rules can be applied recursively in order to refine a business process model thereby revealing its coordination.

- *Reconfiguration of relationships within a business process model*
  After decomposing a business object or a business transaction, the modeller must reconfigure the new objects or transactions in order to adjust the IAS or TES to the more detailed level of the business process model.

- *Navigation within a business process model*
  Within the hierarchy of business objects and business transactions, the modeller is able to navigate. The navigation is supported by a zoom operator, meaning to perform no changes on the integral model. Navigation enables browsing through a large business process model in order to retain overview.

A typical modelling transaction consists of the decomposition of some business object or business transaction followed by a reconfiguration step to adjust the IAS and TES to the new refinement level. The SOM tool supports modelling transactions by providing guidance for the modeller through the steps necessary to transform a consistent state of the model to a new one.

Generally, usage of the tool is dialog-driven and guided by the context menu of an object or a transaction. In contrast to other modelling tools, drag & drop technique is utilised scarcely. The only case in which the modeller draws a relation between two modelling elements is when adding internal events to a task-event schema in order to determine the behaviour of the business process. Figure 7 illustrates the dialog-driven procedure of reconfiguring the relations on a more detailed level of the business process model. As reconfiguration has no effect on the object decomposition view as well as the transaction decomposition view, only IAS and TES are displayed.

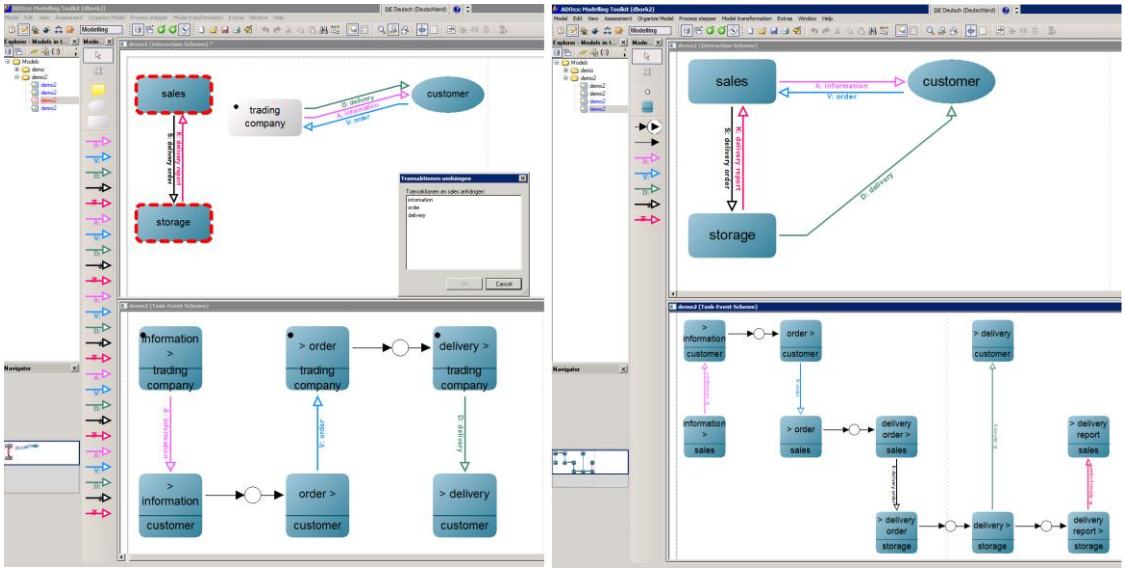


Figure 7: Reconfiguring the relations within a business process model

On the left side the modeller has decomposed the business object *trading company* into the objects *sales* and *storage* according to the feedback control principle (chapter 2). The decomposed trading company object is still displayed on the left side, but grey-shaded to give the modeller a hint about which objects are going to be removed and which transactions he has to reconfigure using the new objects. The dialog box in the middle of the left window displays the transactions currently not considered and asks the modeller alternatingly which transactions he wants to reconfigure to the currently regarded object. This process is done until all transactions are related to either *sales* or *storage*. The objects selectable while connecting the transactions are highlighted using a dashed edge around them.
On the right side then, all transactions are reconfigured and the procedure ends with an update of the IAS and TES diagrams using the implemented graph algorithms. Finally, the dashed edges around the objects sales and storage are removed.

5 Conclusion and future work

This article outlines the design of a new software tool for SOM business process modelling based on the meta-modelling platform ADOxx. The focus is on how the ADOxx platform can be used to establish a graph-based multi-view visualisation of comprehensive business process models based on an integrated meta-model. The paper is research-in-progress. A first tool prototype is available and going to be used in modelling projects.

Next steps will concentrate on the third layer of the SOM enterprise architecture, namely the functional specification of business application systems. In the SOM methodology, application systems are specified from a functional viewpoint by two complementary schemata: a s*chema of task classes (TAS)* which refers to the workflow of an application system and a s*chema of conceptual classes (COS)* which provides corresponding business functions. Both TAS and COS can be derived initially from a SOM business process model via a model-driven approach. Because of its wide acceptance, BPMN (Allweyer 2008, White and Miers 2008) is a candidate language for workflow schemata. Pütz and Sinz report on model-driven derivation of BPMN workflow schemata from SOM business process models (Pütz and Sinz 2010).

Literatur

Allweyer T (2008) BPMN - Business Process Modeling Notation. Einführung in den Standard für die Geschäftsprozessmodellierung. Books on Demand, Norderstedt

Ferstl OK, Sinz EJ (2006) Modelling of business systems using SOM. In: Bernus Peter, Mertins Kai, Schmidt Günter (Hrsg) Handbook on Architectures of Information Systems, second edition. Springer, Berlin, S. 347–367

Ferstl OK, Sinz EJ (2008) Grundlagen der Wirtschaftsinformatik, 6th, revised and extended edition. Oldenbourg, München

Ferstl OK, Sinz EJ (1995) Der Ansatz des Semantischen Objektmodells (SOM) zur Modellierung von Geschäftsprozessen. Wirtschaftsinformatik 37(3):209-220

Ferstl OK, Sinz EJ (1996) Multi-Layered Development of Business Process Models and Distributed Business Application Systems. An Object-Oriented Approach. Distributed Information Systems in Business. Springer, Berlin:159-179

Ferstl OK, Sinz EJ, Amberg M, Hagemann U, Malischewski C (1994) Tool-Based Business Process Modeling Using the SOM Approach. In: Wolfinger B. (Hrsg) 24. GI Jahrestagung 1994. Innovationen bei Rechen- und Kommunikationssystemen, S. 430–436

Fill H (2005) UML Statechart Diagrams on the ADONIS Metamodeling Platform. Electronic Notes in theoretical Computer Science (ENTCS)(127):27–36

Junginger S, Kühn H, Strobl R, Karagiannis Dimitris (2000) Ein Geschäftsprozessmanagement-Werkzeug der nächsten Generation - ADONIS: Konzeption und Anwendungen. Wirtschaftsinformatik 42(5):392–401

Karagiannis D, Bajnai J ADVISOR® - An Educational Management Tool. submitted to the Symposium Towards the New Education Society, Zvolen, Slovakia

Karagiannis D, Telesko R (2000) The EU-Project PROMOTE: a process-oriented approach for knowledge management. In: Proc. of the Third Int. Conf. of Practical Aspects of Knowledge Management.

Lichka C, Kühn H, Karagiannis D (2002) ADOscore® - IT gestützte Balanced Scorecard. wisu - das wirtschaftsstudium(7):915–918

Object Management Group (2006) OMG Meta Object Facility (MOF) Version 2.0. http://www.omg.org/spec/MOF/2.0/, last visit: 2010-05-18

Pütz C, Sinz EJ (2010) Modellgetriebene Ableitung von BPMN-Workflowschemata aus SOM Geschäftsprozessmodellen. In: Gregor Engels, Dimitris Karagiannis and Heinrich C. Mayr (Hrsg) Modellierung 2010. GI, Klagenfurt, Austria, S. 253–268

Reenskaug T (1979) Thing-Model-View-Editor, an example from a planningsystem. In: technical note, Xerox Parc

Schwab M, Karagiannis D, Bergmayr A (2010) i* on ADOxx®: A Case Study. In: Proceedings of the 4th International i* Workshop

White SA, Miers D (2008) BPMN modeling and reference guide. Understanding and using BPMN ; develop rigorous yet understandable graphical representations of business processes. Future Strategies Inc., Lighthouse Point, Fla.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Visualization of Traceability Models with Domain-specific Layouting

Ábel Hegedüs, Zoltán Ujhelyi, István Ráth and Ákos Horváth

12 pages

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

Visualization of Traceability Models with Domain-specific Layouting

Ábel Hegedüs, Zoltán Ujhelyi, István Ráth and Ákos Horváth *

(hegedusa,ujhelyiz,rath,ahorvath)@mit.bme.hu, http://www.inf.mit.bme.hu/FTSRG/
Department of Measurement and Information Systems (MIT)
Budapest University of Technology and Economics (BME), Budapest, Hungary

Abstract: Traceability models are often used to describe the correspondence between source and target models of model transformations. Although the visual representation of such models are important for transformation development and application, mostly ad-hoc solutions are present in industrial environments. In this paper we present a user interface component for visualizing traceability models inside transformation frameworks. As generic graph visualization methods fail to emphasize the underlying logical structure of our model, we used domain-specific layouts by customizing generic graph layout algorithms with data from the metamodels used during the transformation. This approach was evaluated, among others, with the traceability models generated by a BPEL verification transformation, which serves as our running example.

Keywords: traceability, graph visualization, domain-specific layout algorithms

1 Introduction

Model transformations are applied increasingly in various fields of software engineering, from business process modeling to formal verification to code generation. The models acting as source and target for the transformations often represent different domains, thus the identification of correspondence between them is non-trivial. Although in the field of critical systems and services the precise recording of *traceability information* is a strict requirement, in many industrial applications only ad-hoc solutions are used for handling this information.

Throughout the lifecycle of a system or product, traceability information is generated and used for various tasks. The correspondence information are most often created at the time when the target model is produced using the source model during the execution of the transformation. This information can be later used for validation, verification, change management, maintenance or back-annotation. Traceability information itself can be accessed with model transformations thus *model-based traceability solutions* are advantageous [DKPF09].

A development environment that supports both the generation and visualization of traceability information can enhance both *automatic verification* and *human reviews during certification*. We argue that instead of viewing this information textually or as structured data, it should be stored as *traceability models and visualized graphically*. After examining the most common *generic*

* This work was partially supported by the SecureChange (ICT-FET-231101) and the CERTIMOT (ERC_HU_09) projects.

graph visualization methods with traceability models, we concluded that they fail to emphasize the underlying logical structure and mental map, thus a different approach is required.

In this paper we present a *domain-specific visualization* method for traceability models by customizing generic layout algorithms. Our approach results in a comprehensive visualization better suited for model transformation debugging purposes than existing approaches. We also outline the general techniques used for constructing domain-specific layouts for traceability models, and discuss various usage scenarios of the visualization during the transformation development process.

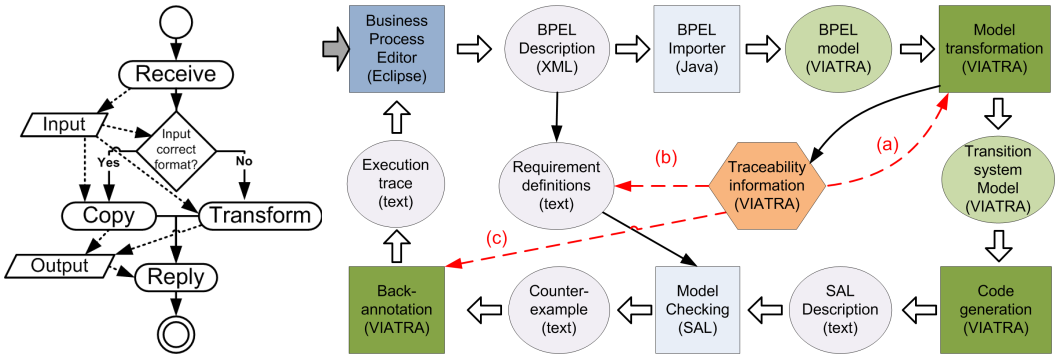
Our concepts are presented on a complex running example (implemented during the SENSORIA European project [SEN05]) that aims at providing automated support for the verification of processes defined in the Business Process Execution Language (BPEL) [OAS07]. The example includes a complex model transformation, which generates a formal transition system description from the selected BPEL process together with a traceability model, which stores the correspondence information between the source and target models. Apart from transformation development, the traceability model is used for aiding the verification and for supporting back-annotation by projecting the verification results from the transition system level to the business process level.

The rest of the paper is structured as follows. Section 2 summarizes our running example of the model checking of BPEL business process using the Symbolic Analysis Laboratory (SAL) [Sha00] model checking framework. Section 3 presents the traceability aspects of the example and the traceability model, including its generation and use. Then, Section 4 presents how generic and domain-specific layout algorithms can be used to visualize traceability models, and demonstrates them using an implementation in the VIATRA2 framework [V2]. Section 5 assesses the related work and finally, Section 6 concludes our paper by evaluating the presented method and suggesting possible future research directions.

2 Case study: Formal Verification of BPEL Processes

Business processes implemented in BPEL are often used to create business-to-business collaborations and complex web services. Their quality is critical to the organization and any malfunction may have a significant negative impact on financial aspects. To minimize the possibility of failures, designers and analysts need powerful tools to guarantee the correctness of business workflows. As the running example of our paper we use such a tool (BPEL2SAL) implemented based on the method presented in [GHV10].

In order to verify BPEL processes, first the input process description is *transformed* into a formal model (i.e. state transition systems, see Figure 1b), thus defining precise formal semantics for BPEL. In the second stage, this transition system is projected into the language of SAL by *code generation* executed using the VIATRA2 framework. The actual verification is then carried out by *model checking techniques* using the SAL framework. Requirements against the business process are captured as Linear Tempolar Logic (LTL) expressions (defined automatically for generic and manually for process-specific requirements), while model checking results in a counter-example (sequence of transitions) if the requirement is violated by the model. Finally, the *back-annotation* of the verification results is provided by another transformation using the traceability information generated during the first transformation [HRV10].



(a) Example BPEL process (b) BPEL Verification approach steps

Figure 1: Running Example

Example business process The input of the tool is a BPEL process therefore, for illustration purposes, we will use the process shown in Figure 1a. The process represents a simple web service responsible for checking the data format of the input and transforming it if required. The `Receive` activity stores the incoming message from the requester in the `input` variable, then the format check decides whether the data is well-formed. If it is the `Copy` activity copies the data into the `output` variable, otherwise the `Transform` activity executes some manipulation before writing the result into the `output` variable. Finally the `Reply` activity sends the data from the `output` variable to the requester and the process finishes.

Traceability aspects In the BPEL2SAL tool all steps including *model generation, model checking and back-annotation* require adequate traceability information concerning the relation between the source (BPEL) and target (SAL) models (see the dashed arrows in Figure 1b: (a) to *simplify the model transformation* by providing pointers to target model elements in different phases of the structural transformation (b) in order to *capture the requirements* properly in LTL expressions where SAL variables have to be used instead of BPEL elements, (c) in back-annotation to *derive the BPEL process execution* from the SAL counter-example. Furthermore, during the development of the transformations it was used for debugging purposes.

Throughout the paper, we will use this traceability information as a domain-specific model for which a graph layout is defined using the presented algorithm. Although presented in the context of the BPEL2SAL tool, these aspects can be identified in many other transformation problems as general concerns [GLMD09]. The traceability information is (i) reused in the transformation, (ii) examined for the verification or model-checking and (iii) used to drive back-annotation.

3 Static Traceability Models

The collected traceability information, which stores the correspondence between the structure of models is called a *static traceability model*. Such models are used for various purposes, some of which are presented through our running example.

Traceability Metamodel In order to store traceability information a third model called the *traceability model* is created during transformation execution. The BPEL, SAL and traceability

models are stored as separate models. The traceability metamodel (illustrated on Figure 2) used in the BPEL2SAL approach has similarities to those presented e.g. in [RÖV09, DKPF09].

The core of the traceability metamodel is the *traceability record* (TR), which represents a correspondence relation between source and target model elements. The record stores relations, which either point to source model elements (`ref_source` relation) or target model elements (`ref_target` relation).

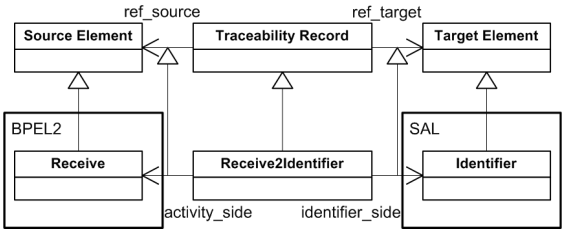


Figure 2: Traceability metamodel

Although the TR could be used directly for creating traceability model instances the definition of *subtypes* (e.g. `Receive2Identifier`) provides a better solution for using the traceability model (e.g. by simplifying pattern matching).

Traceability Use Cases: Reuse, Identification and Back-annotation The importance of traceability can be illustrated on the running example (see Figure 1b) by pointing out three scenarios how the traceability model supports the execution of the different steps of the method: (a) *reuse*, (b) *identification* and (c) *back-annotation*. These steps are found in many transformation use cases, while visualization is highly convenient for these scenarios as an assistance tool for transformation development and verification.

(a) Reuse The SAL transition system has separate parts, which are generated at different phases of the transformation. Thus the traceability model is used repeatedly to find the corresponding variables to the relevant BPEL elements (e.g. find the SAL element corresponding to the BPEL variable `input`, which is written during the execution of the `Receive` activity).

(b) Identification The requirements, which are *validated against the business process*, can be best described using the source model (i.e. the BPEL process itself [XLW08]). However they have to be *specified as an LTL formula* using the formalism of the target model (in this case, the SAL transition system variables). The traceability model can be used to identify corresponding SAL and BPEL elements (e.g. SAL variable to describe that `Receive` always finishes).

(c) Back-annotation The result of the model checking is a counter-example if the requirement is violated by a *sequence of transitions* from the initial state leading to a violating state. However the interpretation (or back-annotation) in the original BPEL process is far from trivial [HRV10]. The traceability model is used to *find corresponding source elements* for the variables in order to derive the BPEL execution from the steps of the counter-example (e.g. the assignment stating that the SAL variable corresponding to the `Receive` activity changed its value after a given transition execution). Ideally, back-annotation itself may be automated, the visualization of traceability models is still advantageous during its development.

4 Visualization of Traceability Models

Displaying generated traceability models gives an overview of the status of model transformation, as ideally for every created target model element there should be one or more element in the source model referenced by traceability records. However, not all the source model elements have traceability records, while the visualization displays the nodes with records, as the subset with

traceability records is usually the significant part.

As the connections are their central element, traceability models can be meaningfully visualized as a graph with the model elements as nodes and the traceability relations as arcs between them. In this section our main contribution is to describe a domain-specific graph layout algorithm created from generic and parameterizable algorithms to visualize such traceability models.

4.1 Visualization of Static Traceability Models

We examined various generic layout algorithms [KW01] to visualize static traceability models. These algorithms evaluate aesthetic conditions to determine the layout of the graph, without any specific information about the logical structure of the underlying model.

The simple *grid layout* displays nodes in rows and columns, but without additional information the node positions had no connection with the structure, and the arcs were crossing both nodes and each other, similar to layout in Figure 4a.

The *radial layout*, developed for the drawing of trees by positioning the nodes in concentric circles, displayed the traceability model in two concentric circles: in the middle the traceability records were displayed, while the outer circle contained both the source and target model elements, mixed together. Although the traceability records are clearly identifiable, the created graph does not reflect the structure of the model very well.

Spring layout, which defines the layout as minimal energy-state of a similar spring-system, displayed the graph as isolated tuples (in most cases triplets) of related model elements. Thus the corresponding elements are placed close to each other in an easily understandable way, but random positioning of isolated tuples makes it hard to find typical problems.

Visualization Requirements The empirical evaluation of the generic graph layout algorithms lead us to identifying the following requirements for a graph layout algorithm to visualize static traceability models:

R1. The displayed nodes should not overlap, as it makes identification of nodes difficult.

R2. The crossings of the displayed arcs should be minimized, as this simple aesthetic criteria is shown to affect human understanding greatly [Pur97].

R3. The corresponding source, target and traceability model elements should be placed close to each other for emphasizing the relations between the model elements. By putting the related elements close, requirement R2 becomes easier to fulfill, as it reduces the arc lengths.

R4. The visualization should clearly separate the source, target and traceability models. As the distinction of the three models forms the basis of the underlying structure, we consider this requirement critical for an understandable visualization.

R5. In order to provide a meaningful visualization during the transformation execution, it is important to handle changes of the input models. The layout should change fast enough, and should keep unchanged parts similar (as in maintaining the mental map [ELMS91]).

All listed generic layouts clearly violate the R4 requirement. In addition to that the grid and spring layouts violate the R2 crossing requirement, the radial layout the R3 closeness requirement (as traceability records get far from the connected source and target elements).

	Grid	Radial	Spring
R1	Yes	Partial	Partial
R2	No	Yes	No
R3	No	Partial	Yes
R4	No	No	No
R5	No	Yes	Partial

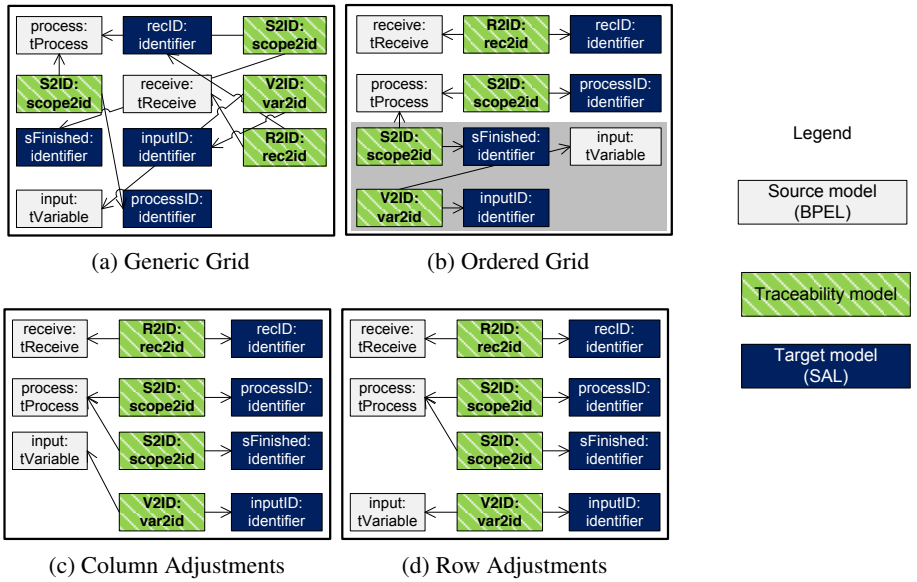
Figure 3: Layouts and Requirements

Requirements R1, R2 and R3 are simple aesthetic properties of the created graph visualization, so they can be fulfilled using generic graph layout algorithms. On the other hand, as the source and target models are symmetrically connected to the traceability record (as seen in the traceability metamodel in Figure 2), the R4 separation requirement can only be fulfilled by a model-dependent differentiating between the model elements.

For this reason we propose a domain-specific layout algorithm, that is created by adding model-dependent customizations to existing generic layout algorithms.

4.2 Domain-specific Layout Algorithm for Static Traceability Models

In this paper we propose the use of domain-specific layouting by customizing generic and parameterized algorithms: we try to give extra information based on the source, traceability and target metamodels. The customization takes the following steps: (1) filtering the model, (2) creating a custom ordering of the model elements and (3) adjusting the layout algorithm to satisfy further visual constraints.



(a) Generic Grid (b) Ordered Grid

(c) Column Adjustments (d) Row Adjustments

Figure 4: Visualizing the Traceability Model

The different problems we addressed with the use of domain-specific layout algorithms for traceability visualization are illustrated in Figure 4. In each figure we use a small subset of the the traceability model: a `receive`, a `process` and an `input` node from the BPEL model, their corresponding identifiers from the SAL model (SAL variables) and the traceability records in-between. The different parts of the graph are colored differently, and the arc captions are removed to maintain readability of the graphs.

Algorithm selection The selected layout algorithm depends on the properties of the visualized model - this idea is hard to generalize. We have chosen a three-column grid layout algorithm, as columns provide a clean separation of the different parts (thus fulfilling requirement R1). Although the generic grid layout gives the worst visualization results without further information, it can be extended to incorporate domain-specific data in a simpler way.

The generic grid layout does not distinguish between different parts of the model, instead it places the nodes in an unpredictable way similar to the layout in Figure 4a. It is important to note, that the generic layout algorithm does not utilize domain-specific filtering, however for the sake of readability we omitted irrelevant elements from the figures, like the internal structure of the source or target models.

Filtering The use of (meta)model-dependent filters helps to provide more relevant domain-specific layout algorithm: by removing unnecessary entities or relations the resulting visualization becomes more focused.

When displaying the traceability model, the intra-model relations of source or target models are often unimportant. By filtering out these relations the resulting visualization is more relevant. Removing irrelevant nodes or relations also reduces the number of crossing arcs (requirement R2).

The removed intra-model relations can either be evaluated using the existing model space editors or a less strict filter should be applied. When applying such a filter, in order to keep the visualization readable, the source, target and traceability models should be filtered similarly. The development of such filters are planned in the future.

Filters can also be defined in the user interface: the user can decide which elements are relevant, and others can be filtered out. This can be used as a kind of search functionality inside the traceability model: it is possible to list only a type from the source or target model, and display its corresponding nodes.

The filtering can happen on both the model and metamodel level: in the first case it is possible to filter out some model elements (typically initiated by the user on the user interface), or entire types (typically built-in filters, provided by the framework).

Ordering Custom ordering can be used to force an algorithm to place the nodes in a predefined order. It is important to note, that some layout algorithms, such as the grid layout are dependent on the ordering of nodes, but others, typically force-based algorithms are ordering-insensitive.

The grid layout implementation used for traceability visualization places the nodes in the order it receives them. This means by creating an ordering that puts the corresponding nodes next to each other in source-traceability-target order, the layout will place them next to each other. Secondary ordering can be used to order the tuples by their relations.

Our solution ordered the items by the name of the traceability records (source and target models are ordered by the names of their corresponding traceability nodes). The ordering ensures that corresponding elements are placed close to each other, and can often be connected without arc crossings (requirement R3 and R2 respectively).

This simple ordering works very well if there is one-to-one correspondence between the different source, traceability and target model elements, otherwise elements get misplaced between columns. As both `S2ID` elements in the grayed area of Figure 4b are connected to the same `process` node, they should be placed into the second column, but the ordering misplaced them.

Further Visualization Constraints Domain-specific knowledge (e.g. the type of the nodes) makes the layout algorithm capable of more efficient visualization. Filtering and ordering cannot utilize all this information, so slight alterations of the layout algorithm might be needed.

For our traceability visualization the grid layout has been altered in two ways: (1) the algorithm decides which model the model elements belong to (simple categorization based on the metamodel), and places it into its corresponding column (requirement R4), and (2) grid cells are left empty to align the corresponding model elements together (requirement R3). The second adjustment is needed, as the first one only ensures that every element appears in the intended column, but an element can get into a wrong row (as the `input` element in Figure 4c). This issue is addressed by ensuring that for every source, traceability and target tuple a new row is started in the layout (see Figure 4d).

Change handling Changes in the underlying model could be handled by simply reapplying the layout algorithm, because the grid layout is simple to calculate (if the nodes are ordered correctly, the execution time is linear to the number of nodes), and results in a layout similar to the original.

If new elements are added, typically a new row has to be appended to the visualized graph. The previously discussed ordering might put this new row into the middle of the layout, thus shifting the rows under the insertion point, but this can be avoided by altering the ordering method to always add new nodes to the end of the list. It is important, that only entire new rows should be handled this way, otherwise unwanted arc crossing could be introduced.

The deletion of elements often results in the deletion of a row, also causing row shifting. If instead of deleting the elements they would be marked dirty (i.e. a signal representing the deletion), the layout could be altered to leave their places empty.

We consider that even the basic version fulfills the change handling requirement R5, as typically complete rows are shifted, so locally the mental map is maintained. By applying the mentioned optimizations the changes can be emphasized.

Evaluation The visualization fulfills all requirements (R1–R5), and emphasizes the logical structure of the transformation: the three models are clearly separated in columns, while the connected elements are grouped in rows. A big drawback of our layout algorithm is the large space consumption, especially vertically. This means, vertical scrolling is almost always needed, but that is easy to understand, and does not affect the usability of our solution. It is also possible to enhance the usability of the solution by more advanced filtering options (by increasing readability), or some kind of searching option (to make the model navigable).

The created visualization can handle large models, we used the industrial size Finance case study [AD07] for evaluation, whose BPEL implementation contains more than hundred elements. Together with traceability and SAL elements, it can be visualized in less than a second.

4.3 Implementation and Usage Scenarios

VIATRA2 model space visualization The VIATRA2 framework organizes its models using a model space [VP03] that allows a hierarchical modeling framework similar to the one provided by the Eclipse Modeling Framework or ontologies.

The model elements can be either entities (graph nodes) or relations (graph edges). Entities represent the basic concepts of the modeling domain, while relations represent a general relationship

between model elements.

The VIATRA2 framework uses a containment-based editor to display and edit model spaces in the user interface (similar to EMF tree editors), but in many cases this is not the most suitable display for the user of the framework. For this reason, we created a model space visualization component for VIATRA2, based on the Zest [Bul08] visualization framework. Zest is built up on general purpose graph visualization techniques, which can be parameterized.

The visualization component is tightly integrated into the transformation development environment: it reacts to changes of the model space (occurring either during transformation execution or model editing), and links back to the containment hierarchy-based editor. This integrated approach helps in using both the containment-based editor and the graph layout together.

The implemented visualization component is depicted in Figure 5: next to the containment-based editor the graph viewer shows the visualized traceability model.

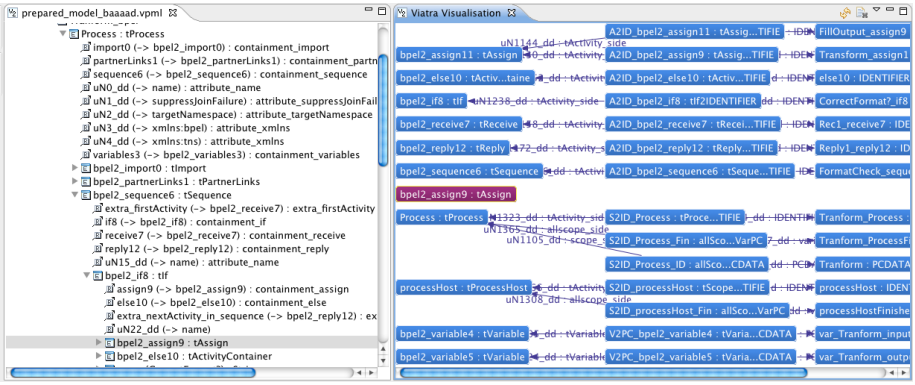


Figure 5: The Visualization of an Erroneous Traceability Record

Usage Scenarios Our traceability visualization helps the manual uses of traceability, such as the ones defined in Section 3, as the traceability links are displayed explicitly. The fact that the visualization is integrated into the transformation IDE helps debugging transformation programs in two ways: the detection of erroneous traceability links is easier.

It is possible to detect missing (or maybe misplaced) traceability records by looking for model elements without connections in the visualization: as the visualization of Figure 5 shows, the selected BPEL model element is not connected to a traceability record. This shows that no traceability record has been created for the source element and suggests an error in the transformation program (alternatively, a target element with no corresponding source model part suggests incorrect traceability handling as well).

The dynamic update mechanism also gives an overview of the transformation status: before the transformation is executed, only the source model is displayed, then as the transformation is executed, the trace records and the relation targets are displayed as they are created.

Our visualization layout (and component) can be used to visualize similar traceability models, which may use different metamodels. A more detailed evaluation is available on our website[1].

[1] http://home.mit.bme.hu/~ujhelyiz/pub/traceabilityvisualization.html

	Visualization	R1	R2	R3	R4	R5	Pros/Cons
Eclipse AM3	Model editors placed next to each other	Yes	N/A	No	Yes	Yes	+ Easy editing
ModeLink							- Implicit traceability links (no arcs)
TraceViz	Display of structured traceability links between selected source and target model elements	Yes	N/A	N/A	Yes	Part	+ Easy directed searches
							- Traceability visualization is restricted to selected elements
Transformation Chain Visualization with GEF3D	Graphical model editors displayed and connected within a 3D space	Yes	Part	Part	Yes	Yes	+ Reuse of existing graphical editors
							+ Visible traceability links
							- No automatic 3D layouting
Domain-specific Graph Layout (this work)	Displaying the traceability model with a modified grid layout algorithm	Yes	Yes	Yes	Yes	Yes	+ Visible traceability links
							+ Transformation IDE integration
							- Space consumption

Figure 6: Overview of the Traceability Visualization Solutions

5 Evaluation of Related Work

In this section a brief overview of various initiatives in the graph visualization field is provided with special focus on model-specific visualization techniques and their use on traceability models.

Visualization Graph drawing is a critical area of information visualization. The graph drawing community researched layout algorithms [KW01], such as different tree layouts or algorithms based on physical analogies such as springs or energy levels. All these algorithms can be used as the base of domain-specific layouts.

The publication information visualizer tool SHriMPBib [All03] uses domain-specific layout algorithms by customizing the Jambalaya ontology visualization tool. Its method is similar to our approach, as filters were defined together with the settings of visual and layout parameters, but the actual design decisions were not published.

The DiaMeta editor generator framework allows specifying metamodel-dependent layout algorithms [MM08]. The visualization is a constraint-based enhancement process: in every step it tries to enhance the existing layout, then checks whether some metamodel-based layout constraints hold. The approach is general and flexible, especially in terms of mental map preservation after modifications, although the evaluation of the constraints could be resource-intensive.

Traceability Model Visualization We compare several traceability model visualization approaches in Figure 6. Both the Eclipse AM3 [JVB+10] and the ModeLink [MLi] projects ease the manual editing of trace models by putting two or three EMF editors side by side, and making it easy to add links between the editors. This approach works well for manual editing, but it is harder to get an overview of the connection between the source and target models, as the connections are not displayed graphically but by the setting of attributes. As no arcs are used, requirement R2 is not applicable, while the corresponding nodes are not grouped together (requirement R3).

The TraceViz [MXP05] tool follows another approach: it displays a list of source and target model elements, and displays the traceability links between the selected elements in a large central area. This interface allows efficient, user-directed search, but is harder to visualize changes in this way (as the changed elements might be hidden - partial support for requirement R5). As the connections are not displayed, requirement R2 and R3 cannot be applied.

The transformation chain visualization of [PVSB08] also includes a three dimensional visualization of traceability links. The main idea of the approach is to put the existing model editors in a three dimensional space, and connect them with traceability links, on the other hand although the layouting options of the existing editors are reused, the traceability links are simply connect the corresponding nodes. This way requirement R2 and R3 are only partially solved, as they depend on the positioning of the editors and the camera position.

Triple Graph Grammars (TGG) are introduced in [Sch95], which also defines a basic layout that separates the three models but lacks further placement guidelines. TGGs are also used for defining transformations in [GLMD09] include traceability information inherently, while the VizMODLE tool supports the visualization of correspondence structures. However these do not include specific layouting for traceability models, therefore these approaches are not evaluated against the requirements.

6 Conclusion and Future Work

In case of complex model transformations (e.g. for automatic model analysis) debugging and back-annotation of the transformation necessitates the visualization of traceability connections between the source and target models in an intuitive, easy to understand way. Unfortunately generic purpose graph layout algorithms frequently fail to properly display the underlying logical structure of traceability models. To solve this problem we proposed a semi-automatic technique with domain-specific layouting by customizing generic and parameterized layout algorithms, and introduced our techniques on a complex case study.

The use of domain-specific layout algorithms seems a promising direction for visualizing traceability models, although further research on automating possibilities is required. In the future we plan to investigate how to identify the core structure of the source and target models, and use this information to make the visualization more intuitive.

Bibliography

[AD07] M. Alessandrini, D. Dost. SENSORIA Deliverable D8.3.a: Finance Case Study: Requirements modelling and analysis of selected scenarios. Technical report, S&N AG, August 2007.

[All03] M. M. Allen. Empirical Evaluation of a Visualization Tool for Knowledge Engineering. Master's thesis, University of Victoria, 2003.

[Bul08] I. Bull. *Model Driven Visualization: Towards A Model Driven Engineering Approach For Information Visualization*. Ph.D. thesis, University of Victoria, BC, Canada, 2008.

[DKPF09] N. Drivalos, D. Kolovos, R. Paige, K. Fernandes. Engineering a DSL for Software Traceability. In *Software Language Engineering*. Pp. 151–167. Springer Berlin / Heidelberg, 2009.

[ELMS91] P. Eades, W. Lai, K. Misue, K. Sugiyama. Preserving the mental map of a diagram. In *Proceedings of COMPUGRAPHICS*. Volume 91, p. 2433. 1991.

[GHV10] L. Gönczy, Á. Hegedüs, D. Varró. Methodologies for Model-Driven Development and Deployment: an Overview. In Wirsing (ed.), *Rigorous Software Engineering for Service-Oriented Systems: Results of the* SENSORIA *project on Software Engineering for Service-Oriented Computing*. Springer-Verlag, 2010. To appear.

[GLMD09] E. Guerra, J. de Lara, A. Malizia, P. Díaz. Supporting user-oriented analysis for multi-view domain-specific visual languages. *Information & Software Technology* 51(4):769–784, 2009.

[HRV10] Á. Hegedüs, I. Ráth, D. Varró. Back-annotation of Simulation Traces with Change-Driven Model Transformations. In *Proceedings of the Eigth International Conference on Software Engineering and Formal Methods*. 2010. To appear.

[JVB+10] F. Jouault, B. Vanhooff, H. Bruneliere, G. Doux, Y. Berbers, J. Bezivin. Inter-DSL coordination support by combining megamodeling and model weaving. In *Proceedings of the 2010 ACM Symposium on Applied Computing*. Pp. 2011–2018. ACM, Sierre, Switzerland, 2010.

[KW01] M. Kaufmann, D. Wagner. *Drawing Graphs*. Lecture Notes in Computer Science Volume 2025/2001. Springer Berlin / Heidelberg, 2001.

[MLi] The ModeLink Project. http://www.eclipse.org/gmt/epsilon/doc/modelink/.

[MM08] S. Maier, M. Minas. A Generic Layout Algorithm for Meta-model Based Editors. In *Applications of Graph Transformations with Industrial Relevance*. Volume Volume 5088/2008, pp. 66–81. Springer Berlin / Heidelberg, Oct. 2008.

[MXP05] A. Marcus, X. Xie, D. Poshyvanyk. When and how to visualize traceability links? In *Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. Pp. 56–61. ACM, Long Beach, California, 2005.

[OAS07] OASIS. Web Services Business Process Execution Language Version 2.0 (OASIS Standard). 2007. "http://docs.oasis-open.org/wsbpel/2.0/wsbpel-v2.0.html".

[Pur97] H. Purchase. Which aesthetic has the greatest effect on human understanding? In *Graph Drawing*. Lecture Notes in Computer Science, pp. 248–261. Springer Berlin / Heidelberg, 1997.

[PVSB08] J. von Pilgrim, B. Vanhooff, I. Schulz-Gerlach, Y. Berbers. Constructing and Visualizing Transformation Chains. In *Model Driven Architecture Foundations and Applications*. Pp. 17–32. 2008.

[RÖV09] I. Ráth, A. Ökrös, D. Varró. Synchronization of Abstract and Concrete Syntax in Domain-specific Modeling Languages. *Journal of Software and Systems Modeling*, 2009.

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

[SEN05] SENSORIA (Software Engineering in Service-Oriented Overlay Computers) EU FP6 Project. 2005. http://sensoria-ist.eu.

[Sha00] N. Shankar. Symbolic Analysis of Transition Systems. In Gurevich et al. (eds.), *ASM 2000*. LNCS 1912, pp. 287–302. Springer-Verlag, Monte Verità, Switzerland, 2000.

[V2] VIATRA2 Model Transformation Framework. http://www.eclipse.org/gmt/VIATRA2/.

[VP03] D. Varró, A. Pataricza. VPM: A visual, precise and multilevel metamodeling framework for describing mathematical domains and UML. *Journal of Software and Systems Modeling* 2(3):187–210, October 2003.

[XLW08] K. Xu, Y. Liu, C. Wu. BPSL Modeler – Visual Notation Language for Intuitive Business Property Reasoning. *Electron. Notes Theor. Comput. Sci.* 211, 2008.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Methods and Tools for the Verification of Finite-State and Infinite-State Graph Transformation Systems

Barbara König

1 pages

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

Methods and Tools for the Verification of Finite-State and Infinite-State Graph Transformation Systems

Barbara König

Abteilung für Informatik und Angewandte Kognitionswissenschaft,
Universität Duisburg-Essen, Germany

Abstract: In this talk we will review several tools and methods for the verification of graph transformation systems. Especially we distinguish between techniques for verifying finite-state and infinite-state graph transformation systems. Then, in the second part of the talk, we will focus on partial-order methods, which can be used in both scenarios. We will specifically describe the tool AUGUR, which uses approximated unfoldings in order to over-approximate graph transformation systems with Petri nets.

Keywords: graph transformation systems, verification tools, partial-order methods

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Distributed Graph-Based State Space Generation

Stefan Blom and Gijs Kant and Arend Rensink

12 pages

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

Distributed Graph-Based State Space Generation

Stefan Blom and Gijs Kant and Arend Rensink*

s.c.c.blom@utwente.nl kant@cs.utwente.nl rensink@cs.utwente.nl
Formal Methods and Tools Group
Department of Computer Science
University of Twente, The Netherlands

Abstract: LTSMIN provides a framework in which state space generation can be distributed easily over many cores on a single compute node, as well as over multiple compute nodes. The tool works on the basis of a vector representation of the states; the individual cores are assigned the task of computing all successors of states that are sent to them. In this paper we show how this framework can be applied in the case where states are essentially graphs interpreted up to isomorphism, such as the ones we have been studying for GROOVE. This involves developing a suitable vector representation for a canonical form of those graphs. The canonical forms are computed using a third tool called BLISS. We combined the three tools to form a system for distributed state space generation based on graph grammars.

We show that the time performance of the resulting system scales well (i.e., close to linear) with the number of cores. We also report surprising statistics on the memory consumption, which imply that the vector representation used to store graphs in LTSMIN is more compact than the representation used in GROOVE.

Keywords: Graph Transformation, Symmetry Reduction, State Space Generation, Distributed Computing, GROOVE, LTSMIN

1 Introduction

For the last two years, the development in modern computer processors has been to put more cores on a single processor, rather than to speed up individual cores. To benefit from this development, it is therefore important to find ways to utilise the power of parallel processing. So far, there is no general way to achieve this for arbitrary applications.

In the context of graph transformation, this topic has been investigated by Bergmann et al. in [BRV09] for the tool VIATRA2. The core functionality of VIATRA2 is to compute a sequence of transformations, controlled by a predefined set of rules, as fast as possible. The paper proposes parallellisation of the matching algorithm.

In this paper, we address parallellisation of GROOVE [Ren04], which differs from other graph transformation tools in that it aims at *complete state space exploration* for a given set of rules, rather than computing a single sequence — where a state equates to a graph. One of the most important aspects of GROOVE, furthermore, is that states are compared *modulo isomorphism*; that is, two graphs are considered to represent the same state if they are isomorphic. Though checking

* any projects?

graph isomorphism is thought to be non-polynomial, the resulting reduction in state space size can more than make up for the cost of isomorphism checking; see, e.g., [CPR08].

At the core of our solution lies LTSMIN [BPW10], an existing framework specifically designed to enable distributed state space exploration with support for multiple specification languages. To use LTSMIN, an application has to:

1. Provide a serialisation of states in the form of fixed-length *state vectors*. State vectors are minimised to so-called *index vectors* (see [BLPW08]), which can be efficiently stored and transmitted.
2. Be able to generate all successors of a given source state, where both the source state and the successor states are communicated in the form of such a state vector.

LTSMIN will then run parallel copies of this application on every available core; the copies communicate using message passing, so that this works equally well with parallel and distributed cores. This method of parallellisation is particularly promising for GROOVE because the time-intensive step of isomorphism checking is done concurrently for many states.

In the case of GROOVE, Step 2 is present by default, but Step 1 is challenging. It is not enough to "flatten" graphs to a vector representation of some kind: in order to reduce the state space up to graph isomorphism, we have to make sure that the representative vector is the same for isomorphic graphs. For this purpose, we can make use of an existing tool called BLISS [JK07], which computes *canonical graphs* based on the principles developed in [McK81]. The LTSMIN vector representing a graph is thus the "flattening" of its canonical form.

We have experimented with this combination of LTSMIN, GROOVE and BLISS. In this paper we report two results:

- For larger cases, the time performance of the parallellised system scales well (though not linearly) with the number of processors. On a single core the setup is a good deal less efficient than GROOVE, but a system with eight or more cores easily outperforms the stand-alone version of GROOVE.
- Given a good vectorisation of the canonical form, the memory performance of the combination of LTSMIN, GROOVE and BLISS is also a good deal better than that of the stand-alone version. This is surprising given the fact that, in contrast to GROOVE, the data structures that LTSMIN uses in its tree compression and central state store are not at all optimised towards the storage of graphs. The gain is large enough to make us consider moving to the compressed vector representation even in the stand-alone, sequential version.

We introduce GROOVE in Section 2 and the relevant features of the LTSMIN framework in Section 3, especially the canonical graph vector representation. In Section 4 we report and analyze the outcome of the experiments. Section 5 draws conclusions and discusses future work.

2 Graph-based state space generation

Graph transformation is a declarative formalism, based on a set of *rules* that are applied to *graphs*. In the context of this paper, graphs are edge-labelled, with labels drawn from a global set Lab;

moreover, nodes are drawn either from a set of node identities Node, or from the set of primitive data values $\mathsf{Val} = \mathsf{Bool} \cup \mathsf{Int} \cup \mathsf{Real} \cup \mathsf{String}$.

Definition 1 (graph, isomorphism) A graph G is a tuple $\langle V, E \rangle$ where $V \subseteq \mathsf{Node}$ is a finite set of nodes and $E \subseteq V \times \mathsf{Lab} \times (V \cup \mathsf{Val})$ is a finite set of edges. We use $src(e)$, $tgt(e)$ and $lab(e)$ to denoted the source, target and label of an edge e. The set of all graphs is denoted Graph. Graphs G, H are *isomorphic*, denoted $G \cong H$, if there exists a bijection $f : V_G \to V_H$ such that $(f(v), a, \bar{f}(w)) \in E_H$ if and only if $(v, a, w) \in E_G$, where $\bar{f} = f \cup id_{\mathsf{Val}}$. We sometimes write $f(G) = H$.

There is no need to precisely define rules; we merely formalise their actions upon graphs. A rule is an object r that can be applied to a *host graph* G if there exists a so-called *match* m for r in G (not formalised here, either). The rule and the match together determine a transformation of G, formally expressed by a derivation relation $G \xrightarrow{r,m} H$, where H is called the *target graph*. This derivation relation is well-defined and deterministic modulo isomorphism:

- $G \xrightarrow{r,m} H$ and $G' \cong G$ implies $G' \xrightarrow{r,m} H'$ for some $H' \cong H$.
- $G \xrightarrow{r,m} H_1$ and $G \xrightarrow{r,m} H_2$ implies $H_1 \cong H_2$.

Using these concepts we define the graph transition system generated by a set of rules.

Definition 2 (graph transition system) The graph transition system (GTS) for a set of rules R and a start graph S is given by $\langle Q, \to, S \rangle$, where $\to$ is the derivation relation restricted to Q, and Q is the smallest set of graphs such that (i) $S \in Q$, and (ii) $H \in Q$ for all $G \in Q$, $r \in R$ and $G \xrightarrow{r,m} H$.

The GTS is a labelled transition system as used in many verification methods, in particular model checking [BK09]. Unfortunately, the GTS can easily be infinite, and even when finite can grow extremely large even for small start graphs — a phenomenon called *state space explosion*. One way to combat state space explosion is through *symmetry reduction* (see, e.g., [CJEF96]). In the case of graphs, symmetries show up as isomorphisms; the state space can be reduced by collapsing all isomorphic states, or in other words, taking the quotient of the GTS under $\cong$. The following algorithm generates this quotient $\langle Q, T, S \rangle$ (where T is the set of transitions).

```
1  let Q := {S},  T := ∅,  F := {S}        (F is the collection of fresh states)
2  while F ≠ ∅
3  do choose G ∈ F        (which G is chosen depends on the structure of F)
4     let F := F \ {G}
5     for G -r,m-> H
6     do if ∃H' ∈ Q : H' ≅ H
7        then let H := H'
8        else let Q := Q ∪ {H},  F := F ∪ {H}
9        endif
10       let T := T ∪ {(G, r, m, H)}
11    endfor
12 endwhile
```

The crux is in Line 6, which tests for *membership up to isomorphism*: given a graph H and a set of graphs Q, find $H' \in Q$ such that $H' \cong H$. Testing $H' \cong H$ for given graphs H, H' is believed to be non-polynomial in $|H|$ (see [Wei02]), and clearly membership up to isomorphism generalises the pairwise test. However, we have shown in [Ren07, CPR08] that the gain by symmetry reduction can be huge, and hence can be worthwhile despite its complexity. We now discuss two ways to implement membership modulo isomorphism.

Graph certificates. The current implementation of GROOVE, as reported in [Ren07], uses *certificates* to obtain a data structure for Q allowing a membership-up-to-isomorphism test that performs well in many practical cases.

A *node certifier* is a function $nc\colon \mathsf{Graph} \to \mathsf{Node} \rightharpoonup \mathsf{Nat}$, which for every graph G results in a function $nc_G\colon V_G \to \mathsf{Nat}$ with the property that $nc_G = nc_H \circ f$ for all isomorphisms f from G to H. A *graph certifier* is a function $gc\colon \mathsf{Graph} \to \mathsf{Nat}$ such that $G \cong H$ implies $gc(G) = gc(H)$. An easy example of a node certifier is to count the number of incident edges ($nc_G\colon v \mapsto |\{e \in E_G \mid src(e) = v \vee tgt(e) = v\}|$ for all $v \in V_G$). Every node certifier nc gives rise to a graph certifier $gc\colon G \mapsto \sum_{v \in V_G} nc(v)$.

GROOVE currently implements Q as a map $\mathsf{Nat} \to \mathbf{2}^{\mathsf{Graph}}$ such that $n \mapsto \{G \in Q \mid n = gc(G)\}$. Finding $H' \in Q$ such that $H' \cong H$ comes down to searching $Q(gc(H))$, which for a good graph certifier gc is almost always either empty or a singleton set. Moreover, pairwise testing $H' \cong H$ for the $H' \in Q(gc(H))$ is made easier by using a node certifier.

Canonical forms. One can take the idea of graph certifiers one step further by also requiring that $gc(G) = gc(H)$ implies $G \cong H$. This is the idea behind the concept of *canonical forms*.

A *graph canoniser* is a function $can\colon \mathsf{Graph} \to \mathsf{Node} \rightharpoonup \mathsf{Node}$, which for every graph G results in an injective function $can_G\colon V_G \to \mathsf{Node}$ such that $G \cong H$ if and only if $can_G(G) = can_H(H)$. (Note that, in combination with a hash function $hash\colon \mathsf{Node} \to \mathsf{Nat}$, this gives rise to a node certifier $nc = hash \circ can$.) Q can then be implemented as a set of canonical form graphs. Obviously, computing canonical forms is as complex as testing for isomorphism; nevertheless, in practice the complexity often turns out to be bearable. In particular, the algorithm developed by McKay [McK81] as implemented in the tools NAUTY [McK09] and BLISS [JK07] does well in practice.

We have used BLISS in our experimentation in the distributed setting. There is a discrepancy in that BLISS uses node-labelled rather than edge-labelled graphs; however, our graphs can be converted to BLISS graphs without loss of information, though with a slight blowup due to the need to encode edge labels in some way. Another noteworthy property is that the canonical forms produced by BLISS always map to an initial fragment of Nat; that is, $can_G(V_G) = \{0, \ldots, |V_G| - 1\}$ for all graphs G. BLISS reorders the nodes such that for isomorphic graphs G and H for all $v \in G$ the same number is assigned to v and $f(v) \in H$ for some isomorphism f (with $H = f(G)$).

3 The LTSMIN framework

LTSMIN is meant to be used as a module in a tool chain that enables state space generation on parallel or distributed systems, consisting of many independent cores. The modular design ensures that the framework can be used for a variety of formalisms. The communication between LTSMIN and the application that uses it, hereafter called the *user module*, is through an interface

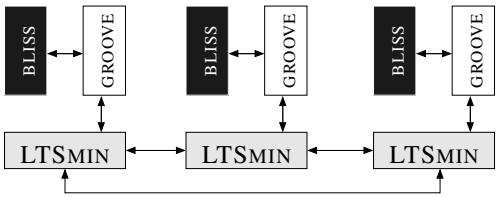


Figure 1: 3-core configuration of LTSMIN with GROOVE +BLISS as user module.

called PINS, for *Partitioned Next-State function*. We will briefly explain the underlying concepts.

To run an application on top of the LTSMIN framework, an LTSMIN client as well as a copy of the user module is started up in parallel on every core. These copies communicate by message passing, so that it does not matter (from the protocol view) whether cores are on a single machine or distributed over different machines. State space exploration then proceeds as follows:

- LTSMIN defines a function that associates a fixed core with each state, on the basis of the state's vector representation. When a state is generated, it is sent to the associated core for further processing. The exploration is kicked off by sending the initial state to the appropriate core.
- Each core keeps a store of all states sent to it so far, remembering also whether the states are closed (i.e., already fully explored) or fresh.
- Upon reception of a state, a core adds it to its state store, marking it as fresh if it was not already in the store.
- Each core computes the successor of each fresh state, and sends the successors to their associated cores. This computation is done by the user module.

An example configuration with GROOVE and BLISS is depicted schematically in Figure 1.

3.1 State vectors and tree compression

The central concept enabling the modularity of LTSMIN is the *state vector*. Every state has to be presented as a vector $\langle p_1, \ldots, p_n \rangle$ for fixed n. The nature of the elements p_i actually does not matter, as these are immediately mapped to table indices for each position. That is, for $i = 1, \ldots, n$ LTSMIN builds up an injective mapping $t_i \colon P_i \to \mathsf{Nat}$, where P_i is the set of all values encountered so far at position i and Nat is a finite fragment of natural numbers; e.g., that fragment which can be represented in 32 bits. Every state vector $\langle p_1, \ldots, p_n \rangle$ is then converted to an *index vector* $\langle t_1(p_1), \ldots, t_n(p_n) \rangle \in \mathsf{Nat}^n$. The mappings t_i are generated on the fly: once a value is encountered for the first time (on position i) it is added to t_i; from then on the same value on that position will always be mapped to the same index. The function associating a core with each state is computed as a hash on the index vector, modulo the number of available cores.

The tables $(t_i)_{1 \le i \le n}$, together called the *leaf database*, are duplicated in the system. It is essential that all workers use the same tables, but they are impossible to build beforehand, as it is unknown which values will be encountered at each position. For this reason, the LTSMIN framework also has the task of distributing the tables over the workers, which in turn means that all the t_i are replicated over all LTSMIN clients. On the other hand, the tables also need to be

known on the side of the user module, since this is where the coding of state vectors to and from index vectors actually takes place. Thus, in a system with c cores, all t_i are replicated $2c$ times.[1]

Index vectors are further compressed using so-called *tree compression* (see [BLPW08]): without going into details, this comes down to repeatedly grouping neighbouring positions of the index vector and building a new table of all combinations of values at those positions that are found during exploration. All these tables together form what is called the *tree database*.

The success of the method crucially depends on finding a state vector representation that has as few values at each vector position as possible; i.e., each of the P_i should be small. This does not contradict a huge overall state space size: for $\max_i |P_i| = m$, the number of states that can potentially be represented is m^n. In the worst case, for one or more i $|P_i|$ approaches the total state space size, and hence so does the size of the tree database; the advantage of this compression method is then completely lost.

3.2 Serialising canonical form graphs

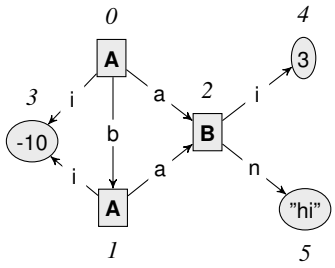
We will now describe the steps necessary to use GROOVE as a user module in the LTSMIN framework. The main difficulty is to find a suitable state vector representation. This is entirely up to the user module: LTSMIN gets to see the state vectors only after they have been produced, and treats the values in the P_i as completely unstructured.

It is absolutely necessary that the state vectors uniquely represent states. This means that, if we want to benefit from symmetry reduction, we have to put graphs into canonical form *before* communicating them to LTSMIN. Moreover, as explained above, the vector representation should ideally have only few possible values at each slot.

In Section 2 we have explained that the canonical form computed by BLISS essentially assigns a sequence number from 0 to $|V|-1$ to each node of a graph G. This imposes a total ordering $\leq$ on V; we will use $\vec{v} = v_0 \cdots v_k$ to denote the ordered sequence of nodes in V. Furthermore, we use the natural total orders on the primitive values Int, String, Bool and Real and we assume a total order on Lab (for instance, the alphabetical ordering). This also gives rise to a lexicographical ordering on edges. In the sequel, $ord(X)$ for a set X with an implicit order will denote the ordered vector of X-elements, and $\vec{x}\restriction_I$ for a sequence $x \in X^*$ and an index set $I \subseteq \{0,\ldots,|\vec{x}|-1\}$ will denote the sequence of elements at positions I.

The vector $\vec{p}_G$ representing G will consist of n slots, of which the first contains a sequence of *node colours* (i.e., the primitive value in the case of value nodes or the set of self-edges for the other nodes; this is also used in the conversion to coloured graphs, needed for the use of BLISS), one for each node, in the order imposed by the canonical form; the second to fifth contain the sets of primitive values from Val used as target nodes, seperated per primitive type; and the remaining slots contain outgoing edges for the individual nodes. If $k > n-5$ (where $k = |V_G|$ and $n = |\vec{p}|$) then nodes are "wrapped around", e.g. for $n = 12$ slot p_5 would be used for $v_0, v_7, v_{14}, \ldots$. This way graphs can be encoded into a fixed size vector, even if the size of the graphs is not fixed.

[1] This description is actually still slightly simplified with respect to the implementation: there the encoding of the *outgoing* states may be different from that of the *incoming* ones; the former is then local to each core, and the LTSMIN clients translate the local to the global encoding.



$p_0 = \{\mathbf{A}\}\ \{\mathbf{A}\}\ \{\mathbf{B}\}\ (\mathsf{Int},1)\ (\mathsf{Int},0)\ (\mathsf{String},0)$ list of nodes
$p_1 = -10\ 3$ list of Ints
$p_2 =$ "hi" list of Strings
$p_3 = \varepsilon$ (the empty sequence) list of Bools
$p_4 = \varepsilon$ list of Reals
$p_5 = \{(\mathsf{a},2),(\mathsf{b},1),(\mathsf{i},3)\}\ \emptyset$ edges of *0* and *4*
$p_6 = \{(\mathsf{a},2),(\mathsf{i},3)\}\ \emptyset$ edges of *1* and *5*
$p_7 = \{(\mathsf{i},4),(\mathsf{n},5)\}$ edges of *2*
$p_8 = \emptyset$ edges of *3*

Figure 2: An example graph with $|V| = 6$, represented by a state vector with $|\vec{p}| = 9$. The canonical node numbers are in italic. Node labels **A**, **B** are self-edges; oval nodes are data values.

Formally this is defined by

$$p_i = \begin{cases} color_G(v_0)\cdots color_G(v_k) & \text{if } i = 0 \\ \mathsf{X}_G & \text{if } i = 1+j \text{ and } \mathsf{X} = (\mathsf{Int}\ \mathsf{String}\ \mathsf{Bool}\ \mathsf{Real})\restriction_j \\ out_G(w_0)\cdots out_G(w_m) & \text{if } i = 5+j \text{ and } \vec{w} = \vec{v}\restriction\{l \mid l = j \bmod (n-5)\} \end{cases}$$

where $color_G(v)$ denotes the colour and $out_G(v)$ the outgoing edges of v, defined as follows:

$$color_G(v) = \begin{cases} self_G(v) & \text{if } v \in \mathsf{Node} \\ (\mathsf{X},i) & \text{if } v = \mathsf{X}_G\restriction_i, \end{cases}$$
$$self_G(v) = \{a \mid (v,a,v) \in E_G\},$$
$$\mathsf{X}_G = ord(\mathsf{X} \cap tgt(E_G)) \qquad \text{for } \mathsf{X} = \mathsf{Int},\mathsf{String},\mathsf{Bool},\mathsf{Real},$$
$$out_G(v) = \{(a, can_G(w)) \mid (v,a,w) \in E_G, v \neq w\}.$$

An example state vector is shown in Figure 2. As related above, this is translated to an index vector together with a set of tables $t_0,\ldots,t_n$, so that a value at position i which recurs in another state vector at the same position is encoded by the same index. For instance, if the graph in Figure 2 is modified by $i := 3$ in node 0, only slot 5 of the state and index vectors would change (namely to $\{(\mathsf{a},2),(\mathsf{b},1),(\mathsf{i},4)\}\ \emptyset$) and only t_5 might have to be updated with this new value.

4 The experiments

We have carried out experiments based on three rule systems with varying characteristics.

le A leader election protocol. In this case there is a fixed number of nodes representing network nodes and a varying number of nodes representing messages. The number of nodes is an upper limit on the number of messages. This case study has been used for the GraBaTs 2009 tool contest (see http://is.tm.tue.nl/staff/pvgorp/events/grabats2009).

unflagged-platoon A protocol for forming car platoons. In this model there is always a fixed number of nodes. The behaviour shows extensive symmetries; reduction modulo isomorphism shrinks the state space by many orders of magnitude. This case study has been used for the Transformation Tool Contest 2010 (see http://planet-research20.org/ttc2010).

Table 1: Results for the largest start graphs where both GROOVE (sequential) and LTSMIN (1, 8 and 64 cores) were able to generate the state-space. The memory usage shown is the average per core. The last column shows the number of elements in the global leaf database for LTSMIN.

Grammar/ Start State	States/ Transitions	Tool	Cores	Time (s)	Speedup	Mem (MB)	Leaf db
le	3.724.544	GROOVE		5.128		2.751	
start-7p	16.956.727	LTSMIN	1	–	–	–	
			8	2.005	2,6	52	
			64	307	16,7	52	1.819
unflagged-platoon	1.580.449	GROOVE		1.016		1.259	
start-10	10.200.436	LTSMIN	1	6.621	0,2	120	
			8	889	1,1	54	
			64	156	6,5	54	8.534
append	261.460	GROOVE		202		372	
append-4-list-8	969.977	LTSMIN	1	3.285	0,1	99	
			8	352	0,6	76	
			64	92	2,2	70	69.147

append A model of list appenders that concurrently add a value to the same list. In this case the number of nodes grows in each step. The maximal number of nodes equals the number of appenders plus 1 times the number of elements in the list. There is hardly any nontrivial isomorphism in the transition system.

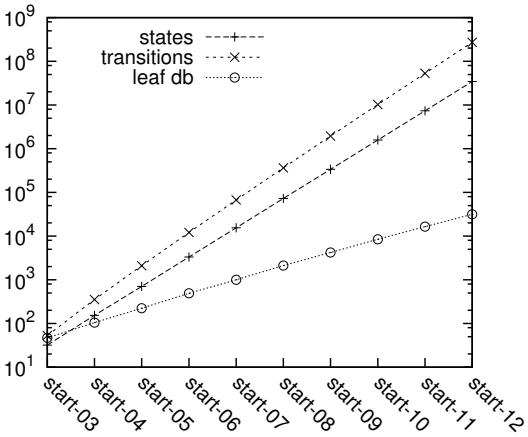
The experiments have been performed on a cluster consisting of 8 compute nodes with 4 dual Intel E5520 CPUs each and 24GB RAM, for a total of 8 cores per compute node and 64 cores in total. GROOVE 4.0.1 has been used with a Sun Java 1.6.0 64-bit VM with a maximum of 2GB of memory for each core. For computing canonical forms we used BLISS 0.50. We used LTSMIN 1.5, with an added dataflow module to facilitate the communication between LTSMIN and GROOVE. For all experiments, the combined system was given a time limit of 4 hours. The state vector size for the first two cases was chosen such that $n \geq k+5$, hence no slot needs to encode the edges of more than one node; however, this is not the case for the third case.

We have compared the performance of the distributed setting with the default, sequential implementation of GROOVE (without computation of canonical forms), running on the same machine but with a memory upper bound of 20GB. For the leader election with start state `start-7p` a different machine with 60GB of memory has been used, because with 20GB of memory the result could not be calculated.

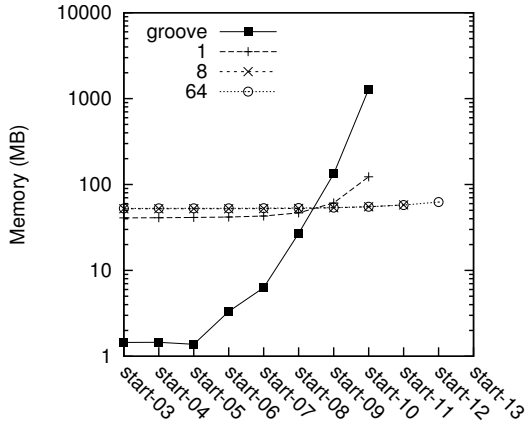
Where there are no values in the table or figures of this section, either the time limit of 4 hours was exceeded or there was not enough memory.

Global results. Table 1 shows some global results for the three cases, using the largest start graphs for which the sequential setting could compute the entire state space. We can observe that with 8 cores, the distributed setting starts to outperform the sequential, and also that the speedup

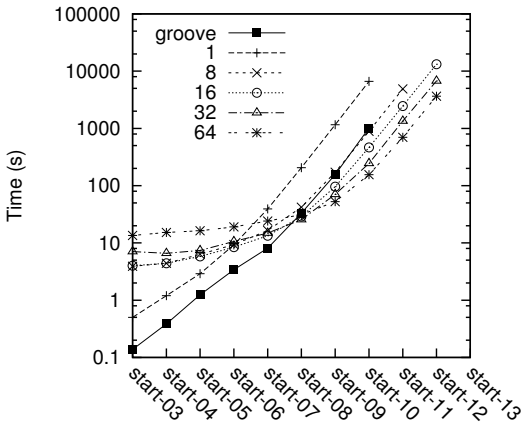
increase from 1 to 8 cores and from 8 to 64 cores is sizeable, though below the optimal value of 8. Furthermore, the leaf database of the append rule system grows much larger than for the others, despite the fact that the state space is much smaller. This is a consequence of the fact that the graph size outgrows the vector size for this case.



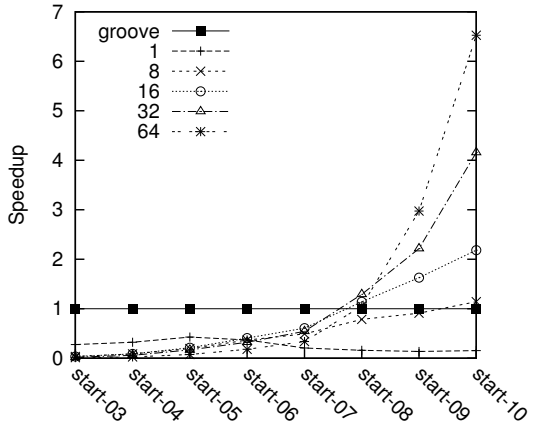
(a) Number of states, transitions and leaf values.



(b) Memory usage for GROOVE and LTSMIN (per core).



(c) Execution time.

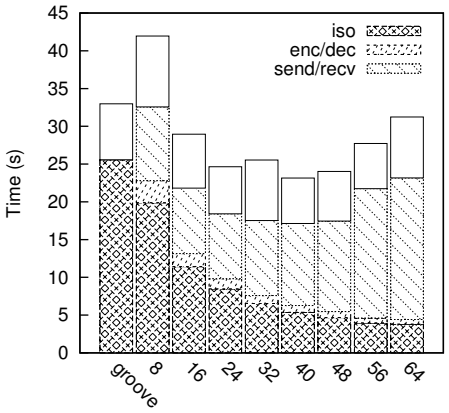


(d) Speedup compared to GROOVE.

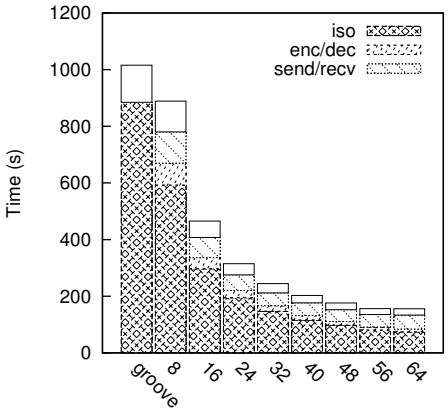
Figure 3: Figures for the car platooning case for different start states.

Time and memory distributions. For the car platooning case, more detailed results are shown in Figures 3 and 4. First of all, Figure 2a shows that even though the size of the problem grows exponentially, the number of elements of the leaf value database of LTSMIN does not. This is also reflected by the per-core memory usage of LTSMIN (Figure 2b), which seems hardly to grow, in contrast to the more than exponentially growing memory usage of GROOVE.

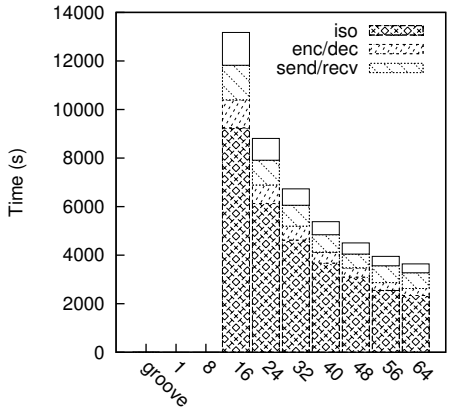
Figures 2c and 2d show the execution time respectively the speedup of LTSMIN with different numbers of cores compared to GROOVE. The execution time of LTSMIN with one core is much worse than GROOVE, but the speedup is growing fast. For the start states with 11 and 12 cars, GROOVE cannot generate the state-space within 4 hours, but LTSMIN with 8 respectively 16 or more cores can.



(a) Execution times for `start-08.`*



(b) Execution times for `start-10.`*



(c) Execution times for `start-12.`

Legend

iso	Computing canonical forms, including the conversion of GROOVE to BLISS and back
enc/dec	Encoding and decoding of GROOVE graphs into state vectors and back
send/recv	Waiting for the next assignment from LTSMIN
rest	Matching in GROOVE

* LTSMIN with one core is left out, because it is more then five times slower than the fastest after that.

Figure 4: Decomposed execution times for the car platooning case for different numbers of cores.

Execution time decomposition. Figure 4 shows how the execution time is built up. For smaller start states, the communication between cores (labelled "send/recv" in the figure) is a major factor in the computation time for LTSMIN, but for larger start states most of the time is spent on computing canonical forms (isomorphism reduction). As the number of cores grows, however, the communication again starts to play a larger relative rule — which is to be expected since this is the only task that is *not* parallellised; indeed, the communication overhead grows more than linearly with the number of cores.

Analysis. For lack of space we cannot include all results in this paper, but the trends for the leader election and append rule systems are very similar to the ones reported above for car platooning. Based on these results, we come to the following observations:

- The chosen serialisation of graphs works well in the reported cases. The total number of values stored, the so called *leaf database*, is orders of magnitude smaller than the total number of states. Indeed, the number of states grows exponentially with the problem size for all three modelled systems, but the number of leaf nodes grows less than exponentially.

Although the canonical form calculation can renumber nodes in an unpredictable manner, which in the worst case could blow up the number of leaf values, apparently the different states are really a combinatorial result of the different parts of the vector. This is especially true for the leader election and car platooning cases, where the number of nodes is a priori bounded and the vector size can be chosen to accomodate this; in the append case, where the state vector representation has to reuse slots for multiple nodes, the results are less spectacular, though still quite good.

- The memory performance of the distributed LTSMIN solution is better than that of the sequential GROOVE system. This is a direct consequence of the success of the serialisation, but it deserves a separate mention. GROOVE uses dedicated data structures, which store only the difference (delta) between successive graphs; nevertheless, the very general tree compression algorithm of LTSMIN turns out to beat this hands down. This came as a big surprise to us, and is reason to reconsider the data structures of GROOVE.

- The time performance of the distributed LTSMIN solution scales well with the number of cores, especially for larger start graphs. The performance of a single core is quite bad compared to GROOVE, taking in the order of 8-10 times as much time, but the distributed system with 8 or more cores is faster. For the largest cases that GROOVE still can compute, we get speedups up to 16 (for 64 cores); moreover, the LTSMIN solution continues to scale well for larger start graphs, which GROOVE on its own cannot cope with at all any more.

- The canonical form computation in the LTSMIN-based system lasts as much as 5 times longer than isomorphism checking in stand-alone GROOVE. As the certificate-based solution of GROOVE uses the same underlying technique as BLISS' canonical form computation (namely, repeated partition refinement), there is no obvious reason for this performance penalty; we hypothesize that it is a consequence of the required encoding of edge-labelled GROOVE graphs as node-labelled BLISS graphs, which increases the graph size. It therefore seems interesting to reimplement the BLISS algorithm for edge-labelled graphs. Given the fact that isomorphism checking is a major fraction of the total time, we expect that this may further improve the distributed performance.

5 Conclusion

We showed a successful way of parallellising graph-based state space generation, using a combination of three tools: GROOVE, BLISS and LTSMIN. A nontrivial step is the encoding of arbitrary graphs into fixed-sized state vectors. We concluded that the resulting system scales well with the number of cores, and has a surprisingly good memory performance — so good, in fact, that it might be worth replacing the current GROOVE data structures. We also observed that a further performance gain can probably be made by reimplementing the functionality of BLISS in order to take advantage of the structure of edge-labelled graphs.

An interesting question raised in the course of this work is whether isomorphism checking is a good idea at all. Omitting the canonical graph computation would ensure that rules have only local effect on the state vector, giving rise to nontrivial (in)dependencies between transitions.

This in turn would allow more of the functionality of LTSMIN to be used, namely the symbolic storage of states. Though there are examples where symmetry reduction has a huge payoff, the same is true, to an even larger degree, for symbolic representations. This is a subject for future investigation.

Bibliography

[BK09] C. Baier, J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2009.

[BLPW08] S. C. C. Blom, B. Lisser, J. C. van de Pol, M. Weber. A Database Approach to Distributed State Space Generation. In Cerná and Haverkort (eds.), *Parallel and Distributed Methods in verifiCation (PDMC)*. Electr. Notes Theor. Comput. Sci. 198, pp. 17–32. Elsevier, 2008.

[BPW10] S. C. C. Blom, J. C. van de Pol, M. Weber. LTSMIN: Distributed and Symbolic Reachability. In *Computer-Aided Verification (CAV)*. LNCS 6174. Springer, 2010. See http://fmt.cs.utwente.nl/tools/ltsmin/.

[BRV09] G. Bergmann, I. Ráth, D. Varró. Parallelization of Graph Transformation Based on Incremental Pattern Matching. In Boronat and Heckel (eds.), *Graph Transformation and Visual Modeling Techniques (GT-VMT)*. Electr. Comm. of the EASST 18. 2009.

[CJEF96] E. M. Clarke, S. Jha, R. Enders, T. Filkorn. Exploiting Symmetry in Temporal Logic Model Checking. *Formal Methods in System Design* 9(1/2):77–104, 1996.

[CPR08] P. Crouzen, J. C. van de Pol, A. Rensink. Applying Formal Methods to Gossiping Networks with mCRL and Groove. *ACM SIGMETRICS Performance Evaluation Review* 36(3):7–16, December 2008.

[JK07] T. Junttila, P. Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *9th Workshop on Algorithm Engineering and Experiments*. Pp. 135–149. SIAM, 2007. See http://www.tcs.hut.fi/Software/bliss/.

[McK81] B. D. McKay. Practical graph isomorphism. *Congressus Numerantium* 30:45–87, 1981.

[McK09] B. D. McKay. NAUTY User's Guide (Version 2.4). Nov. 2009. See http://cs.anu.edu.au/~bdm/nauty/nug.pdf.

[Ren04] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*. LNCS 3062, pp. 479–485. Springer Verlag, 2004.

[Ren07] A. Rensink. Isomorphism Checking in GROOVE. In Zündorf and Varró (eds.), *Graph-Based Tools (GraBaTs)*. Electr. Comm. of the EASST 1. September 2007.

[Wei02] E. W. Weisstein. Isomorphic Graphs. From MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/IsomorphicGraphs.html, 2002.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Applying Offline Verification of Model Transformations to Mobile Social Networks

Márk Asztalos, Péter Ekler, László Lengyel, Tihamér Levendovszky

12 pages

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

Applying Offline Verification of Model Transformations to Mobile Social Networks

Márk Asztalos, Péter Ekler, László Lengyel, Tihamér Levendovszky

Budapest University of Technology and Economics
Department of Automation and Applied Informatics
{asztalos, ekler.peter, lengyel, tihamer}@aut.bme.hu

Abstract: Offline verification of model processing programs has become a fundamental issue in model-based software development. Offline means that only the definitions of the program and the languages of the models to be transformed are analyzed, therefore, the results are independent of the concrete input models and will hold for every possible model. Obviously, the offline analysis is very complex, but it must be performed only once. There are offline analysis results that analyze the formal background or present experimental results, but we believe that formal, automated model transformation verification is still in an initial phase and needs to be improved by providing more general solutions. In previous work, we have provided fundamental formal and algorithmic background of an offline, (semi-)automated verification approach. This work concludes these components and put them together to introduce the implementation of a verification system fully integrated into a modeling and model transformation framework. We believe that the strong points of our approach is its usability, its implementation in an existing tool, and its extendibility, which are demonstrated on a case study in the application domain of mobile centric social networks. Our results show that the verification of graph rewriting-based model transformations can be largely automated.

Keywords: model transformation, automated verification, offline analysis

1 Introduction

In model-based software engineering, developers use programs to process models in a repeatable and automated way. With the increasing need of reliable systems, the verification of such model processing programs has become a fundamental issue. Verification means determining the correctness of the program in the sense that the output satisfies certain functional and non-functional conditions.

Graph rewriting-based model transformation is a frequently used model processing technique, which is well suited to describe several model processing scenarios. We analyze graph rewriting-based model transformations that are based on the formal background graph transformation systems as defined in [EEPT06]. A graph transformation system is defined by a set of rewriting rules (productions), the applications of rules are the elementary operations on graphs. In our terminology, a *model transformation* is the

definition of a model processing program specified by a a set of rewriting rules (based on the double-pushout approach [EEPT06]) and an additional control flow graph that explicitly defines the execution order of the rules. In this control flow, branches and loops may be defined.

In this work, we concentrate on the automated, formal, *offline* verification of *model transformations*. A verification technique is called *offline* if only the definition of the transformation and the specification of the languages that describe the models to be transformed are used during the analysis process. The results of the offline analysis are general in the sense that they are independent from the concrete input models. Moreover, the analysis needs to be performed only once. The disadvantage of the offline approach is the increased complexity of the analysis.

Many published examples can be found for the verification of individual model transformations, however, these methods usually lack generalization possibilities, since the analysis is performed manually or the methods can be applied only to a certain transformation class or to the analysis of only a certain type of property. Therefore, there is an increasing need for automated verification methods and tools. There are few results [Pen09, Ore08, Sch09] related to offline analysis. These analyze the formal background or present experimental results, but we believe that the area of formal, automated model transformation verification is still in an initial phase and needs to be improved. For a more detailed discussion on related work, see Section 5.

In this paper, we present the concept of an automated verification framework implemented in Visual Modeling and Transformation System (VMTS) [Vis]. We have defined a first-order logic-based language (Model Condition Description Language [ALL09b], detailed in Section 3.1), which is capable of expressing most of the properties that are important to be verified. Our system is capable of analyzing model transformations implemented in VMTS and examine the properties described by our language. The result of the analysis can be a proof, a refutation, or that the property cannot be decided.

In previous work [AML10] [ALL09b] [ALL09a] [ALL10], we have described individual components of our verification approach. In this paper, we introduce each of these components shortly to set the grounds for our discussion, then we present the architecture of our implemented verification system. We demonstrate the operation of our system on a real-world case study of refactoring social network models.

The structure of this paper is organized as follows. Section 2 presents the application domain of mobile-centric social networks with its metamodel and introduces a refactoring model transformation. In Section 3, components of our approach are detailed. Section 4 outlines our implemented framework and the verification of the refactoring model transformation. Section 5 contains discussion on related work and Section 6 concludes this paper.

2 Case Study

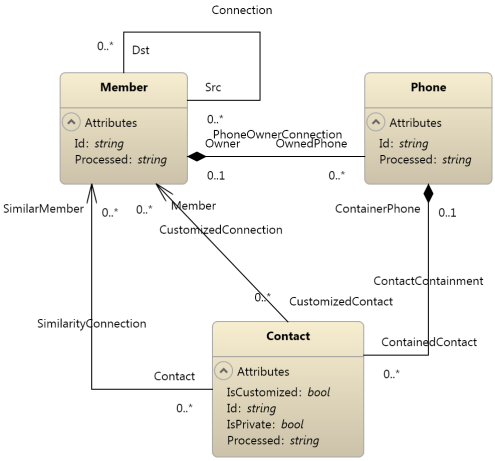
Support of mobile devices is generally marginal in most social networks, however, the phone book in our mobile phone is a small part of a social network because every contact has some kind of relationship to us. Given an implementation that allows us to upload as well as download our contacts to and from the social networking application, we can

completely keep our contacts synchronized. In this paper, we refer to this solution as a *phone book-centric social network*.

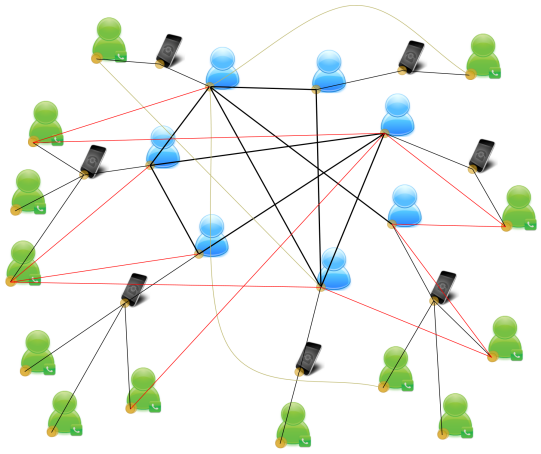
Phonebookmark [EIA09, EL10] is a phone book-centric social network implementation by Nokia Siemens Networks. We took part in the implementation and before the public introduction it was available for a group of general users from April to December of 2008. It had 420 registered members with more than 72000 private contacts. In the following, we present the model-based representation of *Phonebookmark* networks in VMTS. We also show a model transformation used for the refactoring of *Phonebookmark* models. In this paper, we demonstrate the operation of our verification system in the application domain of phone book-centric social networks.

2.1 Metamodeling Central Social Network

Visual Modeling and Transformation System (VMTS) [Vis] is a metamodeling and model transformation system, we can create models not only for predefined modeling languages, but we can define new modeling languages as well. In VMTS, a domain-specific modeling environment has been created for the *Phonebookmark* social network, the metamodel is presented in Figure 1a.



(a) *Phonebookmark* Metamodel



(b) Sample *Phonebookmark* Instance Model

Figure 1: *Phonebookmark* Application Domain in VMTS

A *member* is a user of the social network, a *phone* is a mobile device of a member that can contain phone book entries, a *contact* corresponds to a phone book entry of a phone. Relations between the entities have been defined as follows: each member can own several phones (*PhoneOwnerConnection*), each phone can contain several contacts (*ContactContainment*). A contact can be connected to a member with a *CustomizedConnection* or a *SimilarityConnection* edge. A *ContactContainment*, or shortly customization edge, means that the current entry corresponds to the member of the social network. Whenever the owner member of the entry connects to the social network, the data can be synchronized. A *SimilarityConnection*, or shortly similarity edge, between a member and a contact, denotes that a special *similarity detecting algorithm* has found similarities

between their data so the user has to decide whether to accept this relation and convert it into a customization edge or reject it. For this purpose, *ApprovalState* attribute has been defined for similarity edges, whose value can be *approved*, *rejected*, or, the default value, *ignored*, which means that the user has not decided yet.

In VMTS, the domain-specific environment includes the metamodel and a concrete syntax extension for the instance models. A sample instance model is presented in Figure 1b. The entities can be easily differentiated by their icons. Similarity edges are denoted by red, customization edges by goldenrod colors.

2.2 Similarity Handling Transformation

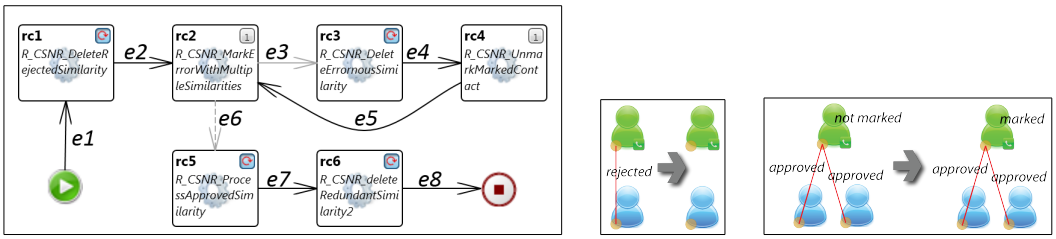
In VMTS, the graph rewriting-based transformations are defined with the use of two modeling languages: the Visual Control Flow Language (VCFL) and the Visual Transformation Definition Language (VTDL) [AAL+09]. The activity diagram-like VCFL models controls the execution order of the rewriting rules, while the rewriting rules are described with VTDL models. These VTDL models define the (left hand side, LHS) pattern to be found and the replacement (right hand side, RHS) pattern of a rule.

Phonebookmark provides a semi-automatic similarity detecting and resolving mechanism, which detects similarities between phone book contacts and the members of the network. Similarity means that the algorithm suggest to the user that the contact and the member represent the same person. In this case, a similarity edge is created between the contact and the appropriate member.

Our model transformation (similarity handling transformation) program processes a *Phonebookmark* model. The user starts it manually after finishing the classification of the similarity edges, where classification means setting the value of the *ApprovalState* attribute. The transformation processes the classified similarity edges as follows: approved edges are converted into customization edges and rejected edges are removed. Our algorithm is defined in VMTS, and C# code is generated from the definition, since the refactoring needs to be executed on the mobile devices.

Before presenting the model transformation, we need to introduce the fundamental concepts of our approach related to the transformation definitions. (i) Firstly, we assume that formally, each model transformation processes one model, which is modified during the execution. It means that the output of an execution is the modified input model. However, it is not a restriction on the generality of the model transformation, since assuming that we have multiple input and multiple output models, we can always compose their union and treat them as a single model. (ii) Secondly, we need to explain the concept of the control flow graph of the transformations. Start node and end nodes are used to mark the starting point and possible end points of the transformations. We call an application of a rule successful if a match has been found, otherwise unsuccessful, this property can be used in the control flow to define branches. To each flow edge the value *success*, *failure*, or *dontcare* is assigned. Success means that the flow edge is followed if the application of the source rule was successful. Failure means that the flow edge is followed if the application of the source step was unsuccessful. *Dontcare* means that the edge is followed in both cases.

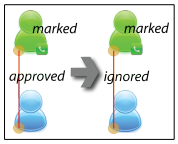
The control flow graph of the transformation, as implemented in VMTS, is presented in Figure 2a. The dashed, gray control flow edges are followed, if the application of the source rules was unsuccessful, which happens when no matches of the left hand side can be found. The solid, gray edges are followed if the application of the previous rule was successful, while solid black edges are followed always. Rules with a circle in the top right bottom are executed exhaustively, which means that the rule is applied repeatedly, until it cannot be applied any more. For a more detailed specification of our model transformation language, see [AM09]. Figure 2 also contains the definition of the rules of the transformation. Here, we use a concrete syntax-based formal representation of each rule with specifying the left hand side (LHS) and right hand side (RHS) of each of them. Recall that the application of rewriting rules is based on the double pushout approach.



(a) Transformation Control Flow

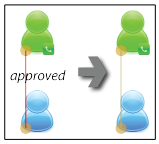
(b) Rule *rc1*

(c) Rule *rc2*

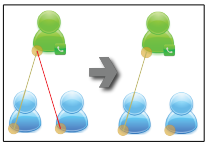


(d) Rule *rc3*

(e) Rule *rc4*



(f) Rule *rc5*



(g) Rule *rc6*

Figure 2: Similarity Refactoring Transformation

Informally, the implemented transformation works as follows: (i) $rc1$ removes all rejected similarity edges. (ii) If there is a contact that has two approved similarity edges, $rc2$ marks the contact. Marking means setting an attribute of the entity. (iii) $rc3$ changes the approval state of an approved similarity edge of a marked node to ignored. This rule is reached only when $rc2$ has been applied successfully. (iv) $rc4$ removes the mark from a marked node. (v) $rc5$ replaces all approved similarity edges with customization edges. This rule is reached only when $rc2$ cannot be applied. (vi) $rc6$ removes all similarity edges which comes from a member that already has a customization edge.

This model transformation will be used to demonstrate the verification system presented in the following section. In the following, two edges are called parallel if they have the same source node. The requirements of this transformation are as follows: (i) Inconsistent states, when the model has parallel approved edges, should be identified. In this case, all parallel approved edges should be modified to be ignored. (ii) All rejected similarity edges should be deleted. (iii) In a consistent state, all approved similarity edge should be transformed to a customization edge. (iv) Whenever a customization edge is present, all similarity edges that are parallel with it should be deleted.

3 Components of a verification framework

In this section, we summarize the fundamental components of our verification approach that have been presented in [AML10, ALL09b, ALL09a, ALL10].

We start with the concept of patterns of models, which are used in the formalisms presented later in this section. Informally, a pattern P defines a submodel, more precisely, the elements of the submodel (i.e. nodes and edges). A pattern does not specify the values of the attributes, but may contain attribute constraints that states something about the attributes. For example, the LHS of rule *rc1* defines a model pattern, which consists of two nodes and an edge between them. One of the nodes is of type *Contact*, the other is of type *Member*, and the edge is a *SimilarityConnection*. Moreover, the pattern specifies the *ApprovalState* property of the edge has the value *rejected*. In this paper, we will use the concrete syntax of the application domain to define model patterns shortly as we did in the presentation of the rewriting rules.

3.1 Model Condition Description Language

The first component of our verification approach is the Model Condition Description Language (MCDL) that is a formal language for writing first-order logic-based expressions (formulae). These logical expressions can describe several properties of models. In the following, MCDL is presented informally, based on our previous work [ALL09b, ALL10].

With the simplest expression of MCDL, we can define that a match of a certain pattern must exist in the model: $\exists\, P$ is satisfied by a model M if an isomorphic occurrence of P can be found in M such that all attribute constraints in P are satisfied. We can combine expressions of MCDL with the standard logical operators $\neg$, $\wedge$, and $\vee$. In this paper, formulae of MCDL are denoted by Greek letters.

For example, the condition $c = \exists$  $\wedge \nexists$  indicates that there exists a member in the model, but there are no contacts ($\nexists$ is the abbreviated form of $\neg\exists$). Given a model M, M satisfies c if and only if there is at least one member element in the model, but there are no contact elements. We can give attribute constraints in the patterns, e.g. in the left hand side of rule $rc1$, we specified that the similarity edge must be rejected, i.e. its *ApprovalState* attribute must have the value *rejected*.

MCDL has already been extended to make it possible to specify more complex conditions with patterns, e.g. that for each member element of a model, at least one phone has to be connected, but their complete presentation would exceed the limits of this paper. In our case study, we will only use the first type of expressions during the verification. VMTS provides a user interface to define formulae of MCDL.

3.2 Model Condition Inference Logic

Given two MCDL expressions ϕ_1, ϕ_2, we may want to prove or refute the logical implication $\phi_1 \Rightarrow \phi_2$. Model Condition Inference Logic (MCIL) is an inference logic defined over expressions of MCDL. An implementation exists in VMTS, and deduction rules for the calculus have been proposed in [ALL10]. The following illustrative examples show two possible inferences: $\exists$  $\Rightarrow \exists$  , $\nexists$  $\Rightarrow \nexists$  .

Given two arbitrary expressions ϕ_1 and ϕ_2, MCIL analyzes the logical implication $\phi_1 \Rightarrow$

ϕ_2. The result of the analysis can the proof described by the steps of the logical reasoning, or a refutation that is the proof of $\phi_1 \Rightarrow \neg\phi_2$. However, if not enough information is provided neither the proof, nor the refutation can be provided. In this case the result of the analysis is *unknown*. The most important property of MCIL is that it is designed in a way that the reasoning algorithm terminates even when the satisfiability of a property is not decidable, therefore, it can be applied in real-world analysis tools as well.

MCIL is used during the verification of model transformations. For example, assume that we can prove that ϕ_1 is true for each output model of the transformation, and let ϕ_2 be a verifiable property. In this case, if the implication $\phi_1 \Rightarrow \phi_2$ can be proved, the property is verified. If the implication can be refuted, it is proved that no output models will satisfy ϕ_2.

3.3 Propagation of Formulae

Propagation of an MCDL formula ϕ_{in} through a rewriting rule r means the derivation of formula ϕ_{out} by the definition of the rule, such that if ϕ_{in} is true for an arbitrary model M and M is modified by rule r, then ϕ_{out} is the formula that is satisfied by the modified model. For example, let ϕ_1 and ϕ_2 be as follows:

$\varphi_1 =$ 

, and $\varphi_2 =$ 

By the definition of $rc5$ (Figure 2f), it can be proved that, if given a model M that satisfies φ_1 and M is modified by executing rule $rc5$, the resulting model will also satisfy φ_1 (since $rc5$ does not create rejected similarities). Moreover, we can infer that the modified model must satisfy φ_2 independently of the fact that M satisfied φ_2 before the application of the rule (because $rc5$ is applied exhaustively).

We have already proposed formal propagation methods for rewriting rules in [ALL10, ALL09a]. We have also given algorithms for the propagation of formulae of MCDL through complex control flows of model transformations, which is based on the propagation of formulae through individual rules and the traversing of the control flow graphs. In this case, given the initial conditions that must be satisfied by any possible input models of the transformations, the propagated formula is the condition that is satisfied by any possible output model of the whole transformation. Moreover, the discovery of formulae on the edges of the control flow graph is also the goal of the algorithms. This means that the formulae that must be true for the model under transformation are computed in different locations of the control flow graph, these locations are the control flow edges. Therefore, after the discovery algorithm for each control flow edge e, an MCDL formula ϕ_e will be assigned. Given the formulae $\phi_1^{final}, \phi_2^{final}, \ldots \phi_j^{final}$ on all incoming edges of all end nodes, we can say that the formula $\phi^{final} = \phi_1^{final} \vee \ldots \phi_j^{final}$ will be satisfied by all possible transformed model of the transformation.

What are the benefits of the discovery algorithm outlined above? (i) As mentioned in Section 3.2, given a verifiable condition ϕ^{ver} specified as an MCDL formula, the verification is the proof of the expression $\phi^{final} \Rightarrow \phi^{ver}$. (ii) Formulae are discovered on all edges of the control flow graph, this can help to localize problematic points of the

transformations during the testing phase. We will show examples for the propagation in Section 4.1, where the transformation handling the similarity is verified.

4 Automated Verification Framework

The verification framework for model transformations has been implemented in VMTS. The main elements of the verification process and the components of the framework are presented in Figure 3. Rounded rectangles are artifacts that are created by the developers or by the verification framework automatically, while not rounded rectangles are the components of the verification system implemented in VMTS.

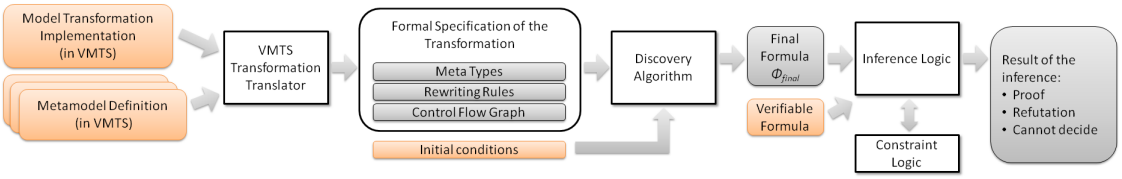


Figure 3: Components of the Verification Framework

The phases of a model-based development (i.e. the implementation of transformations and the verification process), and the roles of the components are as follows:

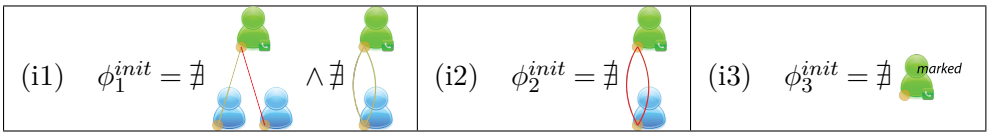
1. Domain experts define the metamodel of the modeled application domains.
2. Model transformation developers implement a model transformation.
3. From these artifacts, *VMTS Transformation Translator* automatically generates the formal specification of the model transformation. This specification is a formal, declarative description, which makes the further automated analysis of the control flow and rules possible.
4. The *Discovery Algorithm* traverses the specification of the transformation, propagates the initial conditions, discovers the formulae on all edges, and generates the final formula (ϕ_{final}).
5. MCIL is used to prove or refute the implication $\phi_{final} \Rightarrow \phi_{ver}$, where ϕ_{ver} is the verifiable formula provided by the developer or the tester of the transformation. The *Inference Logic* component of VMTS is the implementation of the MCIL. The result of the reasoning can be the proof, or the refutation of the implication, or that the algorithm cannot decide.

4.1 Verification of Model Transformation Handling Similarity

In this section, we present the verification of the similarity handling transformation and demonstrate the operation of the discovery algorithm. Primarily, we provide initial conditions. Informally, we assume that each input model satisfies the following conditions: (i1) there cannot exist parallel customization edges, (i2) parallel similarity edges cannot have the same target, and (i3) initially all contacts is *not marked* (a contact can be *marked* or *not marked*, which is expressed by the attribute *Processed*). The initial conditions are formally specified in Table 1, let $\phi^{init} = \phi_1^{init} \wedge \phi_2^{init} \wedge \phi_3^{init}$.

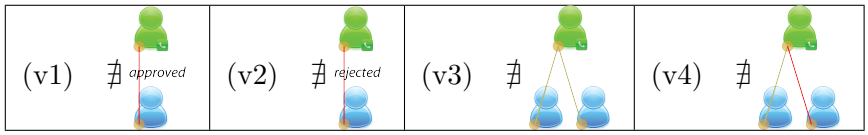
For the verification, we need to define the verifiable properties, which are as follows (the properties are formalized in Table 2): (v1) *After the application of the transformation,*

Table 1: Initial Conditions



no approved similarity edge should be present in the model. Each approved edge should be transformed to a customization edge, or should be deleted if there are more than one approved similarity edge from the same contact. (v2) *After the application of the transformation, no rejected similarity edge should be present in the model.* All rejected similarity edges should be deleted. (v3) *After the application of the transformation, it is forbidden that a contact has a similarity and a customization edge at a time.* In this case, the similarity edge should have been deleted. (v4) *After the application of the transformation, it is forbidden that a contact has two customization edge at a time, provided that before that transformation started this pattern was also forbidden.* This would result an inconsistent state.

Table 2: Verifiable Properties



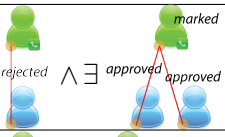
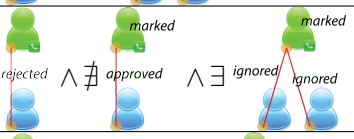
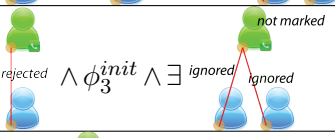
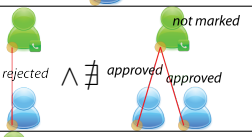
Given the initial conditions, the discovery algorithm traverses the control flow of the transformation and derives the formulae, which means that for each flow edge a formula is assigned. Table 3 shows the discovered expressions. After the initial conditions are provided and the discovery algorithm is executed, the final formula can be derived, which is $\phi^{final} = \phi_{e8}$ in our case. MCIL can derive all four verifiable properties from the final formula, therefore, all properties can be verified. VMTS also provides the derivation steps of the inference, but its presentation would exceed the limits of this paper.

5 Related Work

In this section, we discuss work related to our verification method. The offline analysis of model transformations have been performed in several cases [Var02, BH07, BBG+06], but the presented approaches can usually be applied to only certain (class of) transformations, or only for certain (type of) properties, hence, usually cannot be generalized. [ABK07] presents an approach similar to ours: UML metamodels along with embedded well-formedness rules (typically OCL constraints) can be translated to the formalism Alloy. Then the Alloy Analyzer can conduct fully automated analysis of the transformation. The difference between our approach and the one presented in the paper is that the Alloy Analyzer uses a simulation that generates a random instance model of the input metamodel, then analyzes the behavior of the transformation by transforming this instance model.

Formal methods that can be composed in a complex framework for the analysis of

Table 3: Results of the Discovery Algorithm

Edge	Discovered Formula
$e1$	$\phi_{e1} = \phi^{init}$
$e2$	$\phi_{e2} = \phi^{init} \wedge \nexists$ 
$e3$	$\phi_{e3} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists$ 
$e4$	$\phi_{e4} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists$ 
$e5$	$\phi_{e5} = \phi_1^{init} \wedge \phi_2^{init} \wedge \nexists$ 
$e6$	$\phi_{e6} = \phi^{init} \wedge \nexists$ 
$e7$	$\phi_{e7} = \phi_{e6} \wedge \nexists$ 
$e8$	$\phi_{e8} = \phi_{e7} \wedge \nexists$ 

certain properties of model transformations have been introduced. [Pen09] presents a formalism that is similar to our concept of conditions on models. Nested conditions that are based on traditional application conditions of graph rewriting systems for high-level structures are formalized. Additionally, a sound and complete satisfiability algorithm for graph conditions is investigated and a fragment of conditions is identified, for which the algorithm decides. However, this solution does not take attribute constraints into account. [Ore08] also introduces formalization for attributed graph constraints. The new notion of attributed constraint combines a (standard) graph constraint with a formula describing a condition on the attributes of the graphs involved in the constraint. Moreover, [Ore08] also presents inference rules for the classes of constraints considered, showing their soundness and completeness. In [Sch09], the authors introduce a formalism to describe a model transformation in a declarative way, hereby, verification of soundness conditions can be performed using an interactive theorem prover. The propagation of conditions through individual rewriting rules is investigated in [Roz97] and [EEPT06],

where (negative) application conditions on graph productions and their propagation are analyzed. The work does not take attribute constraints into account either.

6 Conclusions

In this paper, we have outlined an offline, formal, automated framework for the verification of graph rewriting-based model transformations. We have presented how the components of the framework work together in an implementation of our verification methods in a modeling tool, VMTS. We have demonstrated the usability of our methods on a case study of the verification of refactoring mobile-centric social network models.

We believe that the strong point of our approach is its formal background that was developed with considering the possible implementation kept in mind. Moreover, our approach is not only theoretical: it is completely implemented in a real-world modeling and model transformation framework VMTS.

In future work, we would like to complete the formalism behind each of presented components of our solution, and present more complex case studies

Acknowledgements: This paper was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences.

This work is connected to the scientific program of the "Development of quality-oriented and harmonized R+D+I strategy and functional model at BME" project. This project is supported by the New Hungary Development Plan (Project ID: TÁMOP-4.2.1/B-09/1/KMR-2010-0002).

Bibliography

[AAL+09] L. Angyal, M. Asztalos, L. Lengyel, T. Levendovszky, I. Madari, G. Mezei, T. Mészáros, L. Siroki, T. Vajk. Towards a Fast, Efficient and Customizable Domain-Specific Modeling Framework. In *Software Engineering*. 2009. Innsbruck, Austria.

[ABK07] K. Anastasakis, B. Bordbar, J. M. Küster. Analysis of Model Transformations via Alloy. In *MoDeVVA'07*. Pp. 47–56. October 2007.

[ALL09a] M. Asztalos, L. Lengyel, T. Levendovszky. Toward Automated Verification of Model Transformations: A Case Study of Analysis of Refactoring Business Process Models. In *MPM*. Denver, Colorado (USA), October 2009.

[ALL09b] M. Asztalos, L. Lengyel, T. Levendovszky. A formalism for describing modeling transformations for verification. In *MoDeVVa '09*. Pp. 1–10. ACM, New York, NY, USA, 2009.

[ALL10] M. Asztalos, L. Lengyel, T. Levendovszky. Towards Automated, Formal Verification of Model Transformations. In *ICST*. Paris, France, April 2010.

[AM09] M. Asztalos, I. Madari. VTL: an improved model transformation language. In István Vajk (ed.), *AACS*. Pp. 185–195. BME Kiadó, Budapest, Hungary, June 2009.

[AML10] M. Asztalos, I. Madari, L. Lengyel. Towards Formal Analysis of Multi-paradigm Model Transformations. *SIMULATION* 86(7), 2010.

[BBG+06] B. Becker, D. Beyer, H. Giese, F. Klein, D. Schilling. Symbolic invariant verification for systems with dynamic structural adaptation. In *ICSE*. Pp. 72–81. ACM, New York, NY, USA, 2006.

[BH07] D. Bisztray, R. Heckel. Rule-Level Verification of Business Process Transformations using CSP. *ECEASST* 6, 2007.

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

[EIA09] P. Ekler, Z. Iványi, K. Aczel. Similarity Management in Phonebook-centric Social Networks. In *ICIW*. Venice, Italy, 2009.

[EL10] P. Ekler, T. Lukovszki. Experiences with phonebook-centric social networks. In *CCNC*. Las Vegas, USA, 2010.

[Ore08] F. Orejas. Attributed Graph Constraints. In *ICGT*. Pp. 274–288. Springer-Verlag, Berlin, Heidelberg, 2008.

[Pen09] K.-H. Pennemann. *Development of Correct Graph Transformation Systems*. PhD thesis, Department of Computing Science, University of Oldenburg, 2009.

[Roz97] G. Rozenberg. Handbook on Graph Grammars and Computing by Graph Transformation, Foundations, Vol.1. *World Scientific,*, 1997.

[Sch09] B. Schätz. Formalization and Rule-Based Transformation of EMF Ecore-Based Models. *SLE, Toulouse, France, September 29-30, 2008. Revised Selected Papers*, pp. 227–244, 2009.

[Var02] D. Varró. Towards Formal Verification of Model Transformations. In *PhD Student Workshop of FMOODS, Enschede, Hollandia*. 2002.

[Vis] Visual Modeling and Transformation System (VMTS) website. http://vmts.aut.bme.hu/.

Electronic Communications of the EASST
Volume X (2010)

Proceedings of the Fourth International Workshop on Graph-Based Tools (GraBaTs 2010)

Incremental Pattern Matching in Graph-Based State Space Exploration

Amir Hossein Ghamarian, Arash Jalali, Arend Rensink

12 pages

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

Incremental Pattern Matching in Graph-Based State Space Exploration

Amir Hossein Ghamarian[1], **Arash Jalali**[2], **Arend Rensink**[3]

[1] ghamarian@cs.utwente.nl
[3] rensink@cs.utwente.nl
Department of Computer Science,
University of Twente, The Netherlands

[2] arash@netstairs.com
NetStairs.com, Inc.

Abstract: Graph pattern matching is among the most costly operations in any graph transformation system. Incremental pattern matching aims at reducing this cost by incrementally updating, as opposed to totally recalculating, the possible matches of rules in the graph grammar at each step of the transformation. In this paper an implementation of one such algorithm is discussed with respect to the GROOVE toolset, with a special emphasis put on state space exploration. Specifically, we shall discuss exploration strategies that could better harness the positive aspects of incremental pattern matching in order to gain better performance.

Keywords: Graph Transformation, Incremental Pattern Matching, State-space Exploration

1 Introduction

Graph transformation (GT) applications range from model transformation [GBG+06, FUJ06, VB07] to software verification [KK08, Ren04b]. Irrespective of its applications, GT consists mainly of the operations of finding the images of the rules in the host graph, i.e., matching, and transforming the host graph according to the rules. One of the major problems of GT in general is the complexity of the matching operation. Different algorithms have been proposed in the literature as optimized matching algorithms [HVV07, GBG+06, BÖR+08, BGT91]. These algorithms are mainly based on one of the two approaches of *search plan* [GBG+06, HVV07] or *incremental matching* [BÖR+08, BGT91].

In the search plan approach, a match for a rule is found based on a plan, i.e., a set of primitive matching operations, custom-made for each rule using a heuristic algorithm. Once the host graph on which the GT rules are to be applied, is modified by a rule application, the plans must be employed anew to once again find the matches of all the rules in the newly updated host graph.

Incremental matching, on the other hand, relies on a special data structure based on all the rules of the GT system, which is capable of maintaining information about partial and complete matches of all the rules within the host graph. Any changes made to the host graph are also applied to this data structure, and the information about partial and total matches is incrementally

updated where needed. In this way, all the matches of all rules are always readily available, with the extra cost of having to keep the network up to date.

Prior work in the GT literature suggests that incremental matching generally outperforms the search plan approach [BGT91]. In particular, a very efficient and intuitive algorithm for incremental pattern matching is based on the idea of RETE networks [For82]. However, incremental matching has only been used in tools that focus on a model transformation [BÖR+08]. GROOVE [Ren04b], on the other hand, is a general purpose graph transformation tool with the main distinguishing capability of generating the entire (finite) state space of a graph transformation system. The current implementation uses search plans in its matching engine. In this paper, we investigate the use of a RETE-based algorithm in GROOVE, with the hypothesis that, because we actually apply all rule matches at every state, it should be possible to gain performance at least as much as for the VIATRA case reported in [BÖR+08]. In addition to extending RETE to support some special features of GROOVE rules, in particular the quantified rules described in [Ren04a, RK09], the main contributions of this paper are as follows:

- It introduces a state space exploration strategy that makes efficient use of RETE as the matching algorithm.
- It reports experiments showing that the resulting RETE implementation outperforms the previous search plan-based approach.

The next section explains the basic functionality of GROOVE. Section 2 specifies the RETE approaches together with our adaptation. Section 3 describes state space exploration and proposes an efficient strategy suitable for RETE. Section 4 shows the experimental results. Finally, Section 5 discusses directions for future work.

1.1 Introduction to GROOVE

In this section we briefly provide an overview of the GROOVE tool.

Graphs and rules.
Graphs in GROOVE consist of nodes and labelled edges. An edge is a binary arrow between two nodes, or from a node to itself. Node labels can either be node types or flags, which are a special kind of loop edges. Graphs are transformed by applying rules. A rule consists of: A) A pattern that must be present in the host graph in order for the rule to be applicable; B) Subpatterns that must be absent in the host graph in order for the rule to be applicable; C) Elements (nodes and edges) to be deleted from the graph; D) Elements (nodes and edges) to be added to the graph; E) Pairs of nodes that are to be merged. All these elements are combined into a single graph; colours and shapes are used to distinguish them. Figure 1 shows a small example.

The overall effect of the rule is to search for **A**- and **C**-nodes connected by a child-edge but without a parent-edge to a **P**-node, and to modify this by removing the child-edge and adding a parent-edge to a fresh **P**-node. For instance, the rule can be applied to the graph on the left hand side of Figure 2 in two ways, one of which results in the graph on the right hand side. (The other application removes the other child-edge.)

Quantification.
One of the special features of GROOVE is the support of universal quantification in rules (see

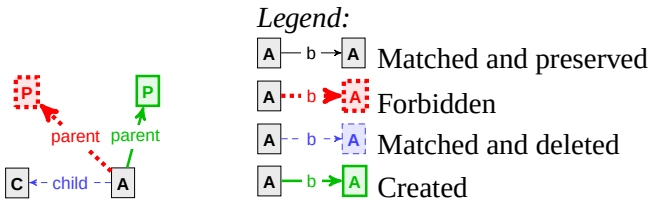
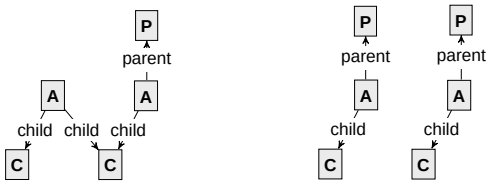


Figure 1: Example GROOVE rule and legend



(a) *Source graph* (b) *Target graph*

Figure 2: Example application of the rule in Figure 1

[RK09]). A universally quantified (sub)rule is one that will be applied to *all* subgraphs that satisfy the relevant application conditions, rather than just a single one as in the standard case. Such a rule can itself be much more concise, and also result in a smaller state space, than the equivalent set of rules that would ordinarily be needed. In fact, quantification can be *nested* in the sense that universally quantified rules can contain existential subrules, and vice versa. Among other things, this makes it possible to formulate powerful application conditions (see [Ren04a]).

State space exploration.
The core functionality of GROOVE is to recursively apply all rules from a predefined set (the graph transformation system) to a given start graph, and to all graphs generated by such applications. This results in a *state space* consisting of the generated graphs, which is a rich source of information for further analysis.

In fact, GROOVE offers a choice of the exploration strategy to be used: *depth-first full exploration*, which also allows on-the-fly LTL model checking; *breadth-first full exploration*, which enables finding shortest paths to certain graphs; and *linear*, *random linear*, and *conditional* exploration which allow simulation without covering all states, for instance if the state space is too large. (In other words, for the latter strategies, GROOVE behaves like other GT tools.)

2 RETE basics

The RETE algorithm was first proposed by Forgy [For82] as an efficient means of pattern matching in production rule systems, in which the system is expected to apply those rules whose left-hand side (LHS) has a matching pattern in a given knowledge-base or state. This original algorithm was meant to be used in text-based expert systems; Bunke et al. generalized and adapted the idea behind the RETE algorithm to graph grammars [BGT91].

The basic idea behind RETE is that the pattern represented by the LHS of a rule can be gradually broken down into smaller sub-patterns all the way down to the basic elements of a graph

pattern, i.e. nodes and edges. By applying the same process to all the rules in a graph grammar, it is possible to construct a network of patterns, starting from simple nodes and edges, to more complex combinations that ultimately lead to the full pattern of the LHS of the rules. This network, called the RETE network, is itself a directed acyclic graph.

2.1 Classic RETE

Figure 3 shows a simple graph grammar with two rules and its corresponding RETE network. To avoid confusion, a node in the RETE network is usually referred to as an *n-node*. As originally defined in [BGT91], a RETE network consists of the following types of n-node:

Root.
This is the only node with no incoming edges. This node is in charge of receiving and passing nodes and edges down the network during runtime, i.e. when matching is being performed. The root is always succeded by edge-checker and node-checker nodes.

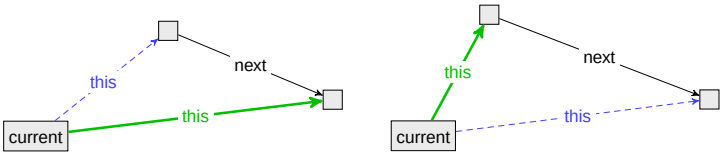
Edge-checkers.
Edge-checkers pass down the network those edges in the host graph that have a specific label. Some edge-checkers only accept loop edges while some accept any edge. In Figure 3c there are three edge-checkers under the root, one of which only accepts loop edges with the label 'current'.

Node-checkers.
Node-checkers appear only immediately after the root, but their use in GROOVE is much more limited as nodes have no labels of their own in GROOVE. A node-checker can therefore be present in a RETE network only when the LHS of a rule consists of one or more isolated nodes.

Subgraph-checkers.
They combine the matches they receive from the upper n-nodes, also known as *antecedents*, and combine them into bigger matches if they overlap on certain nodes. These are indicated in Figure 3c by node equality relations at the bottom of each subgraph-checker. Subgraph-checkers always have two antecedents, commonly referred to as *left* and *right* antecedents. Matches received from these antecedents are stored in two memories called the left and right memories.

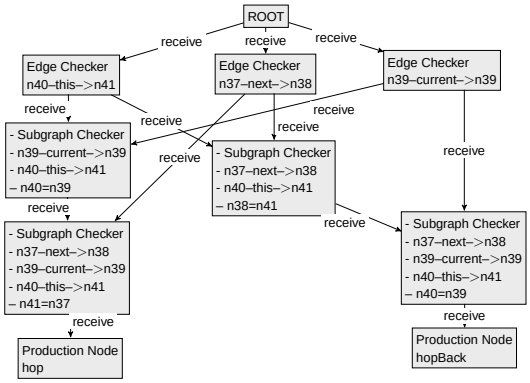
Production nodes.
These nodes represent the LHS of a rule. If a match reaches a production node, it would mean that it is a valid match for the LHS of that rule. The set of all possible matches of a production node in the host graph is called its *conflict set*.

The classic RETE algorithm for graphs consists of two phases: the *static construction* and the *dynamic* phase. During the static construction phase, the RETE network is built by going through each rule of the grammar starting with the very basic elements in its LHS, i.e. nodes and edges. Initially the network consists only of the root. When processing the first rule in the grammar, a separate edge-checker is created and added under the root for each edge in the LHS. The algorithm will then try to combine those edge-checkers into subgraph-checkers. Each subgraph-checker has a set of node equality relations, which signify the way the left and right antecedents of the subgraph-checker are connected to each other. This n-node merging process continues until a single subgraph-checker corresponding to the LHS of a rule is built. At this point a production node is added under the subgraph checker. For the subsequent rules in the



(a) Grammar rule 'hop'

(b) Grammar rule 'hopBack'



(c) Rete Network for 'hop' and 'hopBack'

Figure 3: Two rules and their associated RETE network

grammar, the algorithm will try as far as possible to reuse the existing n-nodes. For details of the static RETE network construction algorithm please refer to [BGT91].

During the dynamic phase, sometimes simply referred to as *runtime*, the RETE algorithm will make use of the network to collectively find the conflict sets of all the rules in the grammar. At the very beginning of the dynamic phase, the RETE network is initialized by feeding all the edges and nodes of the host graph into the root node. The root node will pass down what it receives to its node- and edge-checker successors, and they in turn pass down legitimate matches to their subgraph-checker successors. The subgraph-checkers try to combine the partial matches they receive from their left and right antecedents if possible, and will pass the combined matches down to their own successors. This process continues until all the edges and nodes of the host graph are propagated through the RETE network. The matches that end up in the memories of production nodes will constitute the conflict set of the corresponding rule. The contents of all the memories of n-nodes collectively define the *state* of the RETE network.

The RETE state should be updated after a rule is applied to reflect the changes made to the host graph by that rule. This is where the incremental nature of RETE can be seen, as the network is updated by propagating only those elements that have been deleted or added by the rule's RHS.

2.2 Extensions added to RETE in GROOVE

RETE has already been implemented in other GT tools such as VIATRA [BÖR+08]. Specific features in each tool call for changes to be made to the classic RETE algorithm to make pattern matching for rules that use them possible. Among such features in GROOVE are NACs, quantifiers, and support for rules with disconnected LHSs. In this section, we shall briefly cover these

enhancements.

Quantifiers and nested conditions.
Universal and existential quantifiers in GROOVE have been implemented based on the idea of nested transformation rules [RK09]. Essentially in this scheme, a graph predicate is represented by a multi-level nested rule, where each subrule or subcondition is allowed to find its own matches as if it were an autonomous rule. Through a process called *rule amalgamation*, the matches of the lower level conditions are collected based on the semantics of the quantifier at each level to form the overall set of matches for a grammar rule.

The RETE algorithm has been implemented in GROOVE to accommodate this nested predicate approach. In other words, the RETE network has been extended in GROOVE to support the idea of a subcondition rather than a quantifier, leaving the complexities of amalgamation out of the RETE network data structure. To accomplish this, we have added a new type of n-node to the RETE network called a *condition-checker*. A condition-checker is in many ways like a production node, in fact production nodes are implemented in GROOVE as a specialized condition-checker. During the static construction phase, the construction algorithm not only iterates through the grammar's rules but also through the sub-rules of any complex rule that has one or more levels of quantification in its LHS. During runtime, the RETE network can therefore be queried for the conflict set of any (sub)condition at any level. This approach not only helps keep the complexity of quantifiers and their semantics out of the basic pattern matching algorithm, but it also allows us to exploit any possible overlap of patterns among subconditions, as no discrimination is made between subgraph-checkers based on the level of subcondition they are associated with.

Negative application conditions.
In GROOVE, negative application conditions (NACs) are implemented as subconditions. In other words a rule with both positive and negative parts is represented as an upper level positive rule together with a sub-rule that consists of all the negative nodes and edges with a *root map* that specifies the connection points between the negative and the positive subgraphs.

Following this structure, the RETE implementation in GROOVE will also treat the NAC subconditions as any other subcondition represented by a condition-checker. The condition-checker associated with a NAC along with its antecedent n-nodes in the RETE network do their job as if they are looking for positive matches. Once a new match appears in the conflict set of the condition-checker associated with the NAC, that match is sent to the parent positive condition as an *inhibitor* match, i.e., a match that can potentially inhibit the positive matches of the parent positive condition. The positive condition-checker keeps the record of its inhibitor matches as well as the positive matches inhibited by them, and when it is queried for its conflict set, it will only return those positive matches that are uninhibited.

Again, this implementation allows the RETE network to maintain its original simplicity and to provide the possibility of reuse of subgraph-checkers among all condition-checkers. The implementation of NACs in VIATRA [BÖR+08] is effectively like our implementation, except that the inhibition map in our condition-checkers are explicitly implemented as a special kind of subgraph-checker successor to a positive and a negative subgraph-checker in the RETE network.

Rules with disconnected LHSs.
The original RETE algorithm as outlined in [BGT91] assumes that each rule's LHS is a connected

graph. There, it is suggested that in order to support disconnected left hand sides, one can assume special *dummy edges* that bridge between the disjoint components of the LHS.

In GROOVE a slight modification has been made to the algorithm as well as to the structure of the RETE network so that any production node (or more generally a condition-checker) can have not one but several subgraph-checkers as antecedents, where each subgraph-checker corresponds to one of the disjoint components of the LHS. In turn, such a condition-checker with several antecedents does not directly put matches received from its antecedents into its conflict set, but are rather collected as *partial matches* that can be combined, on demand, with the partial matches from other antecedents to form the whole matches for the condition. Unlike the dummy edge approach, this scheme remains faithful to the general philosophy of reusing of subgraph-checkers.

Domino removal.
For performance reasons, we have implemented the removal updates to RETE like a domino, i.e. matches are linked together and once a deleted edge match reaches the first subgraph-checker, GROOVE will follow the dependency links between smaller and larger matches and drops them from the memories they reside in rather than going through the process of overlap-checking and combining them in each subgraph-checker.

3 State space exploration

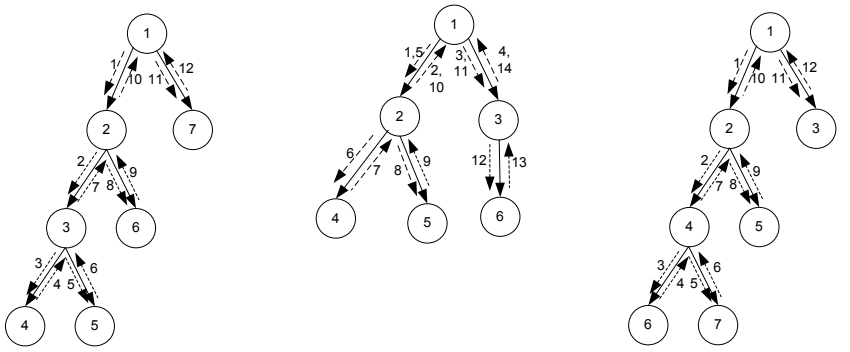
Having explained the static and dynamic part of the RETE network, in this section, we first briefly show how RETE is used in the context of other existing GT tools in linear strategies. Subsequently, we focus on how RETE can be used to explore the state space exhaustively.

3.1 Linear exploration and random linear exploration

In most GT tools, especially those with a model transformation focus (see [FUJ06, VB07]), it is only needed to support a linear path through the state space. In other words, at each state, only one match out of all the matches of all rules is picked and applied to generate the next new state.

In RETE, the state of the network is updated based on the changes made to the host graph as a result of the application of a rule. This process repeats at each newly generated state. There are different methods for choosing a match at each state. In linear strategy, the selected match can be the very first match that is found, or rules can have priorities and the matches of the rules with higher priorities are chosen first. If this match is selected randomly, the strategy is called random linear. The overall scheme of this strategy is given in Algorithm *(Random) LinearStrategy*.

Algorithm *(Random) LinearStrategy*(G, H)
Input: A Graph Grammar G
Input: Input graph H
1. rete ←buildReteNetwork(G)
2. rete.initialize(H)
3. **while** desirable
4. **do** (rule, match) ←rete.selectAMatch()
5. (∗ match corresponds to the rule ∗)
6. δ ←applyMatch(rule, match, H)
7. Rete.update(δ)



(a) An example of DFS strategy using RETE.

(b) An example of BFS strategy using RETE.

(c) An example of a customized RETE DFS.

Figure 4: Different state space exploration strategies using RETE

3.2 Exhaustive state space exploration

There are different strategies for state space generation. GROOVE supports the main two algorithms, i.e., Depth-First Search (DFS) and Breadth-First Search (BFS). As explained in the linear strategy, the RETE network must be initialized in the beginning of the strategy, and as changes occur to the host graph, the RETE state also needs to be updated accordingly. The number of RETE updates as well as the size of updates (number of elements added and removed from the graph) can substantially affect the performance of the exploration strategy. Figure 4a and Figure 4b show two sample state spaces traversed using the conventional DFS and BFS strategies respectively. They also show the order in which the RETE state is updated, if these strategies are used together with RETE as the matching algorithm.

Circles represent the states and the solid arrows between them represent transitions (rule applications) between two states. The dashed arrows signify updates performed on the RETE state. The number on each state is the order in which it is generated. The numbers next to each dashed line show the order in which the RETE state is updated, which could be more than once.

In Figure 4a the exploration starts from state 1, then in the DFS scheme the first match is selected and applied, consequently the successor state is discovered. The RETE state is updated according to the changes resulting from the rule application in state 1, which leads to state 2 (dashed arrow 1). The same procedure then repeats untill it reaches state 4, where there are no more successors and the exploration strategy needs to backtrack. The backtrack requires the matches of state 3 to be found again. In other words, the RETE update 4 is the reverse of update 3. This time however, the second match must be chosen from the RETE conflict sets, and this process continues until all states are fully explored. It is important to note that in every backtrack, a new (next) match ought to be selected, which implies that a fixed order among the matches needs to be maintained within RETE; a requirement that can be very expensive. It is also important to note that during each backtrack, the RETE network will inevitably find *all* the possible matches at that state, including those that have already investigated.

Figure 4b follows the same principle but in a BFS fashion. As can be seen, there are many duplicate RETE updates, as a state needs to be visited several times.

In order to avoid duplicate updates, as well as to avoid having to keep the conflict sets of the

rules sorted in a fixed order, we propose another strategy which is more suitable for a matching algorithm like RETE, in which all matches are readily available at each state. Figure 4c shows this customized strategy, which we call *rete-dfs*. In this approach, once a state is reached, *all* the successor states are generated and stored in a stack. The stack a global data-structure that contains the generated but not yet fully-explored states. The next state to be explored is thus the state at top. This will cause the rest of the exploration strategy to continue in a DFS fashion.

The rete-dfs algorithm is explained in detail in Algorithm *reteDfsStrategy*. The RETE network is built based on the grammar G, and then gets initialized with the host graph H. The host graph is added to the stack, and as long as there are states in the stack, the *next* procedure is called.

Algorithm *reteDfsStrategy*(G, H)
Input: A Graph Grammar G
Input: Input graph H
1. rete ←buildReteNetwork(G)
2. rete.initialize(H)
3. stack.push(H)
4. **while** stack is not empty
5. **do** *next(G, rete)*

Algorithm *next*(G,*rete*)
Input: A Graph Grammar G
Input: Updated RETE state *rete*
1. atState ←stack.top()
2. matchSet ←reteState.getMatchSet()
3. **for** every (rule, match) in matchSet
4. **do** newState ←applyMatch(rule, match, atState)
5. stack.push(newState)
6. *updateAtState(atState, rete)*

In procedure *next* all the matches of the current state are queried from the updated RETE state. Each rule match is applied according to its corresponding rule and the newly generated states are added to the stack. Then *updateAtState* is invoked in order to update the RETE state.

Algorithm *updateAtState*(*atState*,*rete*)
Input: The currrently explored state *atState*
Input: Updated RETE state *rete*
1. $\delta \leftarrow 0$
2. **if** atState $\neq$ stack.top()
3. **then** atState ←stack.top()
4. **else** (∗ no new state is put on stack in the last call to *next* ∗)
5. **repeat**
6. $\delta \leftarrow \delta +$ atState.getFromParentDelta() (∗ getFromParentDelta returns the changes when going from the parent of atState to atState ∗)
7. triedState ←atState
8. stack.pop()
9. atState ←stack.top()
10. **until** atState.parent() = triedState.parent() **or** atState = null
11. **if** atState $\neq$ null
12. **then** $\delta \leftarrow \delta^{-1} +$ atState.getFromParentDelta()
13. rete.update(δ)

If in procedure *next* any new state was added to the stack (atState is not equal to the top of stack), that would mean that the exploration is going forward. In that case, *updateAtState* just applies the changes made by the last rule application (δ) to that state and the RETE state is updated according to δ. Each state saved on the stack maintains the changes occurred when moving from its parent to itself; information which is obtained by calling *getFromParentDelta*.

However, if during the course of the procedure *next* nothing new was added to the stack, *updateAtState* would start backtracking by removing states from the stack. During backtrack,

Grammar	States	Trans.	32-bit 3GB			64-bit 48GB		
			Incr. [s]	Search Plan [s]		Incr. [s]	Search Plan [s]	
			rete-dfs	DFS	BFS	rete-dfs	DFS	BFS
append1	4046	11210	2.4	2.5	2.5	4.6	4.4	4.3
append2	261460	969977	253.0	792.0	831.0	199.0	190.0	184.0
carPlat1	4881	11280	0.6	0.5	0.6	1.2	1.0	1.3
carPlat2	110366	323405	110.0	109.2	111.6	9.9	8.5	8.7
carPlat3	2988061	10632648	1371.0	1539.0	-	203.0	202.6	205.1
crashCars1	131072	548864	21.6	26.7	31.7	18.9	21.5	27.4
crashCars2	278528	1236992	48.2	59.0	70.6	37.5	40.7	50.9
crashCars3	589824	2768896	343.0	367.5	446.7	74.5	86.1	105.4

Table 1: Exploration strategies' execution times

updateAtState accumulates in δ all the parent-to-child changes of the states taken from the stack.

The backtracking continues until *updateAtState* reaches a state where forward exploration is once again possible. This happens when the top of the stack is a sibling of (shares a parent with) the last state popped out of the stack. At this stage the backtracking phase is over and it is time to apply the accumulated changes stored in δ so far. However, since the changes stored in δ are from parent to child and we have been travelling in the opposite direction, δ needs to be inverted first (as indicated by δ^{-1} in line 12). In addition to the inverted δ, the update to the RETE state should also include one forward step from the shared parent to the sibling found at the stack top.

It is important to note that these δ's between the states are accumulated and the state of RETE gets updated only once. This approach decreases the amount of changes needed to be propagated through RETE. Maximum gain is achieved in cases when the changes made by successive rules cancel each other out, e.g. an edge is added by one transition and removed by another.

4 Experimental results and evaluation

We compared the rete-dfs exploration strategy, which uses RETE as an incremental matching algorithm, with DFS and BFS strategies, which use search plan for matching. We executed all of these strategies on three different grammars *car platooning*, *crashing cars* and *append* and each grammar on multiple start graphs (available through GROOVE repository [Ren04b]). *Car platooning* was taken from this year's GT tool contest, and the other two have been devised by the GROOVE team. These grammars are chosen as they not only make use of the special features of GROOVE grammars such as NACs, etc. but they also cover various levels of rule complexity with regards to the size of their LHS. The experiments were ran on a machine with two Quad core Xeon 3GHz processors. Two series of experiments were performed over 32-bit and 64-bit Java Virtual Machines with 3GB and 48GB of RAM respectively. The results are shown in Table 1.

The first column shows the name of the grammar along with a sequence number representing the start graphs used to run the exploration. The larger the sequence number, the larger the size of the corresponding start graph. The second and third columns are the numbers of states and transitions in the state space, respectively. The next six columns show the execution times of the three strategies in seconds, over the 32-bit and the 64-bit JVMs. A hyphen (-) in the execution time means that the experiment did not finish due to an out-of-memory exception. Please note

Grammar	User time			Real time		
	Incr. [s]	Search Plan [s]		Incr. [s]	Search Plan [s]	
	rete-dfs	DFS	BFS	rete-dfs	DFS	BFS
append2	231.4	240.0	233.2	199.0	190.0	184.0
carPlat3	350.6	467.5	513.3	203.0	202.6	205.1
crashCars3	105.9	134.4	158.1	74.5	76.1	105.4

Table 2: User time versus real time in 64-bit JVM

that the reported time is the whole execution time, and the matching time is a fraction of this time. To make the total time a more precise indication of the matching time, isomorphism checking has been turned off in these experiments.

Table 1 shows that in the 32-bit case, rete-dfs almost always outperforms the other two strategies, with substantial gains of up to 300%. Overall, it scales much better for larger cases.

On a 64-bit architecture, depending on the size of the host graph and the number of matches at each state, rete-dfs can perform a bit better or worse than the other two strategies. In general, there is no substantial performance difference between the three strategies.

The great disparity in the observed performance gain between the two architectures is primarily caused by the significant difference between the size of the available memory in each case, and the significant decline in the performance of DFS and BFS in the 32-bit case can be taken as an indication of their extravagance in memory allocation (and de-allocation), which will result in the production of a larger amount of garbage memory, thus calling for more frequent garbage collection cycles to be performed during their execution. It is worth mentioning that garbage collection in the 64-bit case is performed in parallel by 8 cores. This is evident from the measured user time (the time spent in all threads including the garbage collectors) as shown in Table 2 for the largest cases of all three grammars. A significant improvement in total computation time for rete-dfs over the other two strategies can be observed.

That RETE generates less garbage memory than search plans can be explained by this inherent property of RETE that matches unaffected by an update to RETE remain in the n-nodes' memories and are used in the subsequent states, whereas matches found by search plan at each state will be thrown away after each transition even if they are needed again at the next state.

5 Conclusion and future work

We proposed an adapted version of a RETE-based incremental pattern matching algorithm that could accommodate the special features and requirements of GROOVE. As our main contribution, we used this incremental pattern matching algorithm in exhaustive state space exploration. To harness the full potential of RETE we proposed the rete-dfs exploration strategy. The comparison of our proposed strategy with both DFS and BFS which use search plan showed that under common execution configurations rete-dfs visibly outperforms other existing strategies (especially with larger start graphs), and that only under very special circumstances with an unusual abundance of resources (eight 64-bit CPU cores and as much as 48GB of memory) can the existing strategies perform as well as rete-dfs.

In our agenda for future work, first and foremost is the adding of support for some of the more

advanced features in GROOVE, e.g. mergers, regular expressions and attributes. Furthermore, possible optimizations of the structure of RETE need to be more thoroughly investigated. Currently, the network is built in an ad-hoc way, leaving a lot of room for further optimization as the order in which the rules and their nodes and edges are processed during the static phase can affect the efficiency of the RETE network both in terms of speed and memory. Another clue for further optimization of the exploration strategy is that sometimes during backtracking, the size of update (δ) to be performed on the network can be very large; so much so that it might occasionally be less costly if the RETE states are stored in their entirety at all or some of the states.

Bibliography

[BGT91] H. Bunke, T. Glauser, T.-H. Tran. An Efficient Implementation of Graph Grammars Based on the RETE Matching Algorithm. In *Proceedings of the 4th International Workshop on Graph-Grammars and Their Application to Computer Science*. Pp. 174–189. Springer-Verlag, London, UK, 1991.

[BÖR+08] G. Bergmann, A. Ökrös, I. Ráth, D. Varró, G. Varró. Incremental pattern matching in the VIATRA model transformation system. In *GRaMoT '08: Proceedings of the third international workshop on Graph and model transformations*. Pp. 25–32. ACM, NY, USA, 2008.

[For82] C. Forgy. RETE, a fast algorithm for the many pattern / many object pattern match problem. *Artificial Intelligence* 19:17–37, 1982.

[FUJ06] The FUJABA Toolsuite. 2006. Homepage: http://www.fujaba.de.

[GBG+06] R. Geiß, G. V. Batz, D. Grund, S. Hack, A. Szalkowski. GRGEN: A Fast SPO-Based Graph Rewriting Tool. In Corradini et al. (eds.), *International Conference on Graph Transformations (ICGT)*. LNCS 4178, pp. 383–397. Springer, 2006.

[HVV07] Á. Horváth, G. Varró, D. Varró. Generic Search Plans for Matching Advanced Graph Patterns. *ECEASST* 6, 2007.

[KK08] B. König, V. Kozioura. Augur 2 — A New Version of a Tool for the Analysis of Graph Transformation Systems. In Bruni and Varró (eds.), *Graph Transformation and Visual Modeling Techniques (GT-VMT 2006)*. Electronic Notes in Theoretical Computer Science 211, pp. 201–210. 2008.

[Ren04a] A. Rensink. Representing First-Order Logic Using Graphs. In Ehrig et al. (eds.), *International Conference on Graph Transformations (ICGT)*. LNCS 3256, pp. 319–335. Springer Verlag, 2004.

[Ren04b] A. Rensink. The GROOVE Simulator: A Tool for State Space Generation. In Pfaltz et al. (eds.), *Applications of Graph Transformations with Industrial Relevance, (AGTIVE)*. LNCS 3062, pp. 479–485. Springer, 2004. See http://sourceforge.net/projects/groove.

[RK09] A. Rensink, J.-H. Kuperus. Repotting the geraniums: on nested graph transformation rules. In Boronat and Heckel (eds.), *Graph transformation and visual modelling techniques (GT-VMT)*. Electronic Communications of the EASST 18. EASST, 2009.

[VB07] D. Varró, A. Balogh. The model transformation language of the VIATRA2 framework. *Science of Computer Programming* 68(3):187–207, 2007.