Using Graph Transformation for Practical Model Driven Software Engineering

Lars Grunske[1], Leif Geiger[2], Albert Zündorf[2], Niels Van Eetvelde[3], Pieter Van Gorp[3], and Dániel Varró[4]

[1] School of Information Technology and Electrical Engineering, University of Queensland, Brisbane, QLD 4072, Australia
grunske@itee.uq.edu.au
[2] Department of Computer Science and Electrical Engineering, University of Kassel, Wilhelmshöher Allee 73, 34121 Kassel, Germany
leif.geiger|albert.zuendorf@uni-kassel.de
[3] University of Antwerp, Middelheimlaan 1, 2020 Antwerpen, Belgium
niels.vaneetvelde|pieter.vangorp@ua.ac.be
[4] Department of Measurement and Information Systems, Budapest University of Technology and Economics, H-1117, Budapest, Magyar tudósok krt. 2.
varro@mit.bme.hu

Summary. Model transformations are one of the core technologies needed to apply OMG's model driven engineering concept for the construction of real world systems. Several formalisms are currently proposed for the specification of these model transformations. A suitable formalism is based on graph transformation systems and graph transformation rules. The chapter provides an overview about the needed concepts to apply graph transformations in the context of model driven engineering and we show the technical feasibility based on several tools and applications.

Key words: Model Driven Engineering, Model Transformation, Graph Transformation

1 Introduction

Model Driven Engineering (MDE) is a software engineering approach that promotes the usage of models and transformations as primary artifacts. The Object Management Group (OMG)[1] proposed the Model Driven Architecture (MDA) as a set of standards for integrating MDE tools. These standards focus on the usage of Platform Independent Models (PIM's), which help to develop software on a higher level of abstraction by hiding platform specific details. Thus they solve some of the problems that are caused by the ever-increasing complexity of software systems. For further reduction of complexity PIM's can be modelled with several viewpoints (e.g. structure models,

behaviour models, quality assurance models, test cases), in order to focus on particular concerns within the system separately.

For the construction and evolution of these viewpoint models, it is necessary to ensure consistency between the different models. To enable this consistency, model transformations are used to update all other viewpoint models in case of one model has been changed. Furthermore, model transformations can help to construct a new viewpoint model based on the existing models. All these model transformations are PIM-to-PIM transformations or model-to-model transformations, which can be also called horizontal transformations, because they are used to transform models on the same level of abstraction. To execute the PIM on the target platform, a platform specific model (PSM) must be generated. This generation also needs appropriate model transformations that enrich the PIM with platform specific details. These model transformations are vertical transformations, respectively called as PIM-to-PSM transformations.

To conclude this, model transformation is the heart and soul of model driven software engineering [2]. To standardize these model transformations the OMG recently announced a request for proposals (RFP MOF 2.0 Query / Views / Transformation) [3], which includes requirements for the transformational language. This transformation language and the underlying formalism should provide the following characteristics:

1. The formalism should support the specification of horizontal and vertical model transformations
2. The formalism should enable the automatic application of the model transformation rules
3. The transformation rules should be easy to understand
4. The transformation rules should be adaptable and reusable

Based on these requirements, we propose to use graph transformations to specify and apply model transformations in model driven engineering. The reasons for this are: (a) graphs are a natural representation for models, since most modeling languages are formalized by a visual abstract syntax definition, (b) graph transformations provide a formal theory and some established formalisms for the automatic application, (c) we believe that graph transformation rules can be easily and intuitively specified (unfortunately there are currently no empirical studies to prove this) and (d) the complexity of the graph transformation rules and the application formalisms can be hidden for the end user.

This chapter is structured as follows. Section 2 summarizes the theoretical background of graphs and graph transformations. Furthermore, alternative approaches are discussed. In Section 3 and 4 the state-of-the-art graph transformation tools are described to illustrate their applicability for horizontal and vertical model transformation. In Section 5 it is presented how the correctness of the applied graph transformations can be verified. Finally, conclusions are drawn and directions for future work are discussed.

2 A Basic Introduction to the Graph Transformation Concepts

In this section, we introduce the basics of graph-based structures. Furthermore, we describe the fundamental graph transformation theory and give an overview of useful graph transformation variants that can be used to specify model transformations.

2.1 Directed Typed Graphs and Graph Morphisms

We choose directed typed graphs as the basic structure for graph-based model transformations, because they are well suited for specifying different types of models [4]. These graphs contain nodes and edges, which are instances of node and edge types. The instance relation between the nodes and edges and their types is similar to the relation between objects and classes in object-oriented software engineering. Due to this, a node or edge type can contain a set of application specific attributes and operations. To model the graph-based structure each edge is associated to a source and a target node. Formally, a typed graph can be defined as follows:

Definition 1 (Directed Typed Graphs). Let L_V be a set of node types and L_E be a set of edge types, then a directed typed graph G from the possible set of graphs $\mathcal{G}$ over L_V and L_E is characterized by the tuple $\langle V, E, source, target, type \rangle$, with two finite sets V and E of nodes (or vertices) and edges, a function $type : V \rightarrow L_V \cup E \rightarrow L_E$ which assigns a type to each edge and node and two functions $source : E \rightarrow V$ and $target : E \rightarrow V$ that assign to each edge a source and a target node.

Another preliminary for the definition of graph transformation systems are graph morphisms. These graph morphisms are structure and type-preserving mappings between two graphs, which can be defined as follows:

Definition 2 (Graph Morphism). Let $G = \langle V, E, source, target, type \rangle$ and $G' = \langle V', E', source', target', type' \rangle$ be two graphs, then a graph morphism $m : G \rightarrow G'$ consists of a pair of mappings $\langle m_V, m_E \rangle$, with $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ which satisfy the following conditions:

- $\forall e \in E : type'(m_E(e)) = type(e)$
- $\forall v \in V : type'(m_V(v)) = type(v)$

If both mappings $m_V : V \rightarrow V'$ and $m_E : E \rightarrow E'$ are injective (surjective, bijective) then the mapping $m : G \rightarrow G'$ is injective (surjective, bijective).

2.2 Graph Variants

In addition to the introduced directed typed graphs, several other variants have been proposed in the graph transformation community. One basic variant

uses undirected edges. These undirected edges can be modelled in a directed graph with two contrary edges for each undirected edge. Another variant are hypergraphs [5], where each edge is associated to a sequence of source and target node. Due to this, these edges are also called hyperedges. For the construction of hierarchical models, hierarchical graphs are important. These hierarchical graphs model the hierarchical structure ether by (hyper)edge [6] or node [7] refinement.

2.3 Graph Transformation and Graph Transformation Systems

Basic Principles

Graph transformation systems make use of graph rewriting techniques to manipulate graphs. A graph transformation system is defined with a set of graph production rules, where a production rule consists of a left-hand side (LHS) graph and a right-hand side (RHS) graph. Such rules are the graph equivalent of term rewriting rules, i.e., intuitively, if the LHS graph is matched in the source graph, it is replaced by the RHS graph. Formally, a graph transformation rule can be defined as follows:

Definition 3 (Graph Transformation Rule). A graph transformation rule $p = \langle G_{LHS}, G_{RHS} \rangle$ consist of two directed typed graphs G_{LHS} und G_{RHS} which are called the left-hand side and right-hand side of p. Furthermore the graph G_I is an interface graph, satisfying $G_I \subseteq G_{LHS}$ and $G_I \subseteq G_{RHS}$

For the application of a graph transformation rule to an application graph G_{APP} the following simplified algorithm can be used, which contains the following steps:

1. Identify the left-hand side G_{LHS} within the application graph G_{APP}. For this, it is necessary to find a total graph morphism $m : G_{LHS} \rightarrow G_{APP}$ that matches the left-hand side G_{LHS} in the application graph G_{APP}.
2. Delete all corresponding graph elements, w.r.t m, in the application graph G_{APP} that are part of the left-hand side G_{LHS} and are not part of the interface graph G_I.
3. Create a graph element in the application graph G_{APP} for each graph element that is part of the right-hand side G_{RHS} and is not part of the interface graph G_I. Connect or glue these added graph elements with the rest of the application graph G_{APP}.

For a formal description of the rule application formalisms, we refer to [8, 9], where the formal foundations of the single pushout (SPO) and double pushout (DPO) approach are reviewed. Currently these approaches have the most impact in the graph transformation community.

Application Conditions

In graph transformation systems with a large number of graph transformation rules it is often necessary to restrict the application of single rules. Therefore, in [10] the concept of positive and negative application conditions (PACs and NACs) are introduced. These application conditions are formally graphs that define a required context (i.e., the presence of some nodes or edges) or a forbidden context (i.e., the absence of some nodes or edges). The fulfillment of these application conditions must be checked before the rule is applied. Therefore, the presented algorithm must be extended with an additional step between the first and the second step, that checks the application conditions. With the introduction of application conditions graph transformation rules become conditional productions, which enhances the expressiveness of the graph transformation system (especially if NACs are used)[10].

Specification of a Graph Transformation Rule

In traditional approaches for specification of graph transformation rules, the right-hand side and the left-hand side of a rule are drawn separately. Throughout this chapter, the notation of story diagrams [11] is used, that combines both sides. Story diagrams use UML collaboration diagrams to model graph transformation, cf. [12, 13]. Consequently, nodes become objects and edges become links between objects. Objects and links in such collaboration diagrams marked with the ≪*destroy*≫ stereotype appear only on the left-hand side of the corresponding graph transformation, i.e. they are deleted. The stereotype ≪*create*≫ marks elements only used on the right-hand side, i.e. such elements are created. Objects and links that have no stereotypes appear on both sides of the graph transformation rule (cf. Fig. 1).

Story diagrams use programmed graph transformation rules. Due to this, a control structure can be specified that manages the order of the execution of transformation rules. Such control structure is modeled using UML activity diagrams. The transformation rules are then embedded into the activities.

Story diagrams use typed graphs. Graph schemata are modeled using UML classdiagrams (cf. Fig. 2). In graph transformations, the type is specified after the object name separated by a colon. By omitting the type, bound objects are marked. Bound objects are objects that are already known to the system either from previous matchings or because they are passed as parameters to the transformation rule (as done for the object *afterElem* in Fig. 1. Thus, a bound object does not compute a new match but it reuses its old match.

More elaborated elements of graph transformations, like negative application conditions, multi objects, non injective matching are also supported by the story diagram language. Some of these features will be discussed in chapter 3.

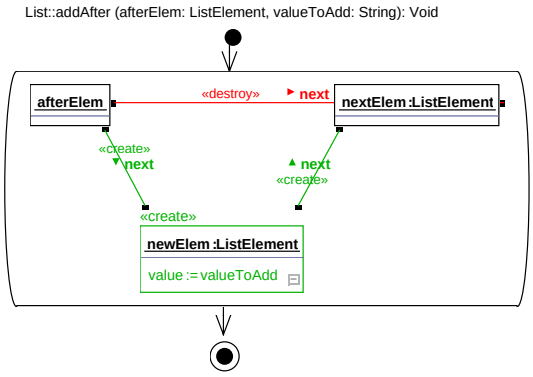


Fig. 1. Graph transformation "Fujaba-style"

The rule shown in Fig. 1 models the behavior of the *addAfter* method of a class emphList. This method simply adds a new String value (passed as parameter *valueToAdd*) into a list after a given element (passed as *afterElem*). This method consists only of one activity, that means only one transformation rule. The pattern matching starts with the bound *afterElem* node. From this node, an edged labeled *next* to a node of type *ListElement* is searched. If such an edge is found, the targeting node is called *nextElem*. Having found such a node, the pattern matching is completed and the changes can be executed. First the *next* edge between *afterElem* and *nextElem* is deleted. After that a new node *newElem* is created and its *value* attribute is set to the passed *valueToAdd* parameter. Then two new *next* edges connecting this node with the other two nodes are created. After that, the rule is completed and the method is left.

2.4 Graph Transformation Variants

In this section we briefly introduce two alternative approaches that are suitable to specify model transformations as presented in the following sections. These approaches are pair grammars [7] and triple graph grammars [14].

Pair Graph Grammars

Pair grammars and pair grammar rules were introduced by Pratt[7] in the early seventies to specify graph-to-string translations. A pair grammar rule rewrites two models: a source graph and a target string. Thus it contains a pair of production rules (a graph and a string production rule), which modify simultaneously the two participating models. Based on this, pair grammars are well suited to specify transformations between graphs and strings. If the string production rule is substituted by a graph production rule, pair grammars can be used for graph-to-graph translations.

Triple Graph Grammars

Triple graph grammars, as introduced in the early nineties [14], are an extension of pair graph grammars [7] to specify graph-to-graph translations and evolution. Each triple graph grammar rule contains three graph productions, one operates on a source graph, one on the target graph and one on a correspondence graph. This correspondence graph describes a graph-to-graph mapping, which relates elements of the source graph to elements of the target graph. Based on this mapping an incremental change propagation is possible, which updates the target graph if an element in the source graph is changed.

2.5 Alternatives for a graph transformation approach

Gerber et al. provide a good overview of mainstream transformation approaches [15]:

- One alternative for graph transformation, developed by the OMG, would be the Common Warehouse Metamodel (CWM) Specification [16], which provides concepts for *black-box* and *white-box* transformation specifications. Still, fine grained graph transformations can complement these CWM concepts. CWM white-box specifications leave the actual production of target model elements from source model elements unspecified in a string of source code that can be implemented in any programming language. CWM black-blox transformations are even more abstract.
- XSLT [17] has become a popular alternative for describing model transformations. Peltier et al. [18] propose to consider metamodels as models: they use MOF [19] as the metamodel of such models and use XMI [20] to serialize their instances. An XSLT specification is also an XML document, describing a transformation using both declarative and imperative constructs. Gerber et al. found that the verbosity of the XML syntax leaded to specifications that were difficult to read and to maintain.
- Text based tools like perl and awk are also useful but only for simple transformations, because they cannot cope with the abstract syntax of models.
- Their own tool, GenGen, describes transformation rules as instances of a CWM - inspired metamodel. The rules are implemented by transforming them to java code which can then manipulate models in MOF repositories. It is a rather procedural transformation language, and therefore, although powerful, lacks the capabilities like unification from declarative languages.
- The Mercury programming language [21], in contrary, is a purely declarative, strongly typed logic language and offers these pattern matching features. This leads to rules that are more compact and easier to understand. However, the main problems experienced with this approach is that it is necessary, and hard to capture the semantics of the source and target models using the language type system.

- Finally, F-Logic [22] is a complete formal model for deductive object oriented languages. It offers a flexible and compact syntax for defining rules that can be interpreted at both the model and instance levels, and does not suffer from the restrictions of the Mercury language.

The authors describe their attempt to map an EDOC Business Process [23] model to the Breeze Workflow [24] model, using both the declarative and procedural approaches. From their experiment the authors derive a set of both functional and usability requirements, needed to model mapping rules. They conclude that a declarative approach is preferred, due to the simpler semantic model required to understand the transformation rules, but acknowledge that for certain types of transformations a procedural specification may remain necessary.

Küster et al. performed an initial comparison between a graph transformation approach and a relational approach to model transformation [25]. Using the translation of statecharts to CSP as a benchmark for comparing the Consistency Workbench [26] to the QVT-Merge proposal [27], the authors concluded that both approaches were similar regarding the matching of patterns in a host model. Therefore, the new relational approach can build upon the insights of efficient model matching as discussed by Dorr [28], Vizhanyo et al. [29] and Varró et. al. in [30].

From a consistency maintenance viewpoint, the relational approach is currently more generic than graph transformation since the former provides a framework to automatically keep the related source and target models synchronized. Graph transformation theory needs to be extended with the notions of consistency contracts and traceability links to support such incremental updates. Consistency contracts can be described declaratively in OCL while the graph transformation approach described in Section 3.2 can be used establish and maintain the contracts [31]. Initial ideas on graph transformation with incremental updates are discussed by Varró et. al. in [32].

On the other hand, from a model analysis viewpoint graph transformation has clear advantage. In such cases, the transformation should describe either a complex computation (e.g. flattening hierarchical statecharts concurrent regions into flat statecharts) or powerful abstractions need to be performed to avoid state space explosion (thus the descriptive power of source and target languages are very different). Declarative (relational) approaches have not yet proved their practical feasibility in these areas.

3 Graph Transformations for Vertical Model Transformation

Vertical Transformations are transformations towards platform specific models and towards a specific model implementation. Generally, vertical transformations may use the same techniques as horizontal transformations just for a

different purpose. Only some aspects e.g., code generation require additional concepts.

3.1 The Fujaba approach

Our approach uses the Fujaba project and tool set developed at the University of Paderborn [11]. Fujaba is a graph based tool which uses the Unified Modeling Language UML for design and realization of software projects. Fujaba uses UML class diagrams for the specification of graph schemata. As mentioned before, it uses a combination of activity diagrams and collaboration diagrams, so-called story diagrams for the specification of operational behavior. The semantics of story diagrams are based on programmed graph rewriting rules [13]. The story diagrams offer many powerful constructs of graph transformation like multiobjects, non-injective matching, NACs etc. to create a powerful language which is usable for modeling even complex problems in an elegant way. The operational behavior modeled with such story diagrams can then be tested using the graph based object browser DOBS (Dynamic Object Browsing System, cf. Fig. 3) which is part of the Fujaba Tool Suite, cf. [33].

In contrast to other graph based tools (cf. [34, 35]), Fujaba does not rely on proprietary runtime environments. Instead, Fujaba generates standard Java source code that is easily integrated with other Java program parts and that runs in a common Java runtime environment. The Fujaba code generation for graph rewrite rules uses a sophisticated query optimizer that translates the left-hand side of a rule into nested search loops, cf. [36, 13]. In general this results in efficient rule execution. Altogether, this enables the use of graph based concepts in all kinds of Java applications. Support for other target languages is planned, too.

Meta Model

To illustrate our approach, a small case study on a statechart environment is used. This case study was first introduced in [37]. The Fujaba approach is used to describe how model transformation to PSM and operational semantics of the PSM can be done.

The statechart case study is used to show Fujabas abilities to model a visual language. Since the Fujaba approach uses typed graphs, one needs a graph schema to model graph transformations. Such graph schemata are modeled using UML class diagrams in Fujaba. Note, that the usage of MOF would be applicable here, too. Fig. 2 shows such a graph schema / metamodel / class diagram for the statechart environment.

From such a class diagram, Fujaba generates Java classes for the different kinds of objects, their attributes and their relationships. From the developers point of view, Fujaba's implementation of relationships turns Java object

structures into graphs with bi-directional edges. Provided with a class diagram, our dynamic object browser DOBS may already be used as a simple editor for models / object diagrams / graphs, cf. Fig. 3. DOBS shows the abstract syntax of our model.

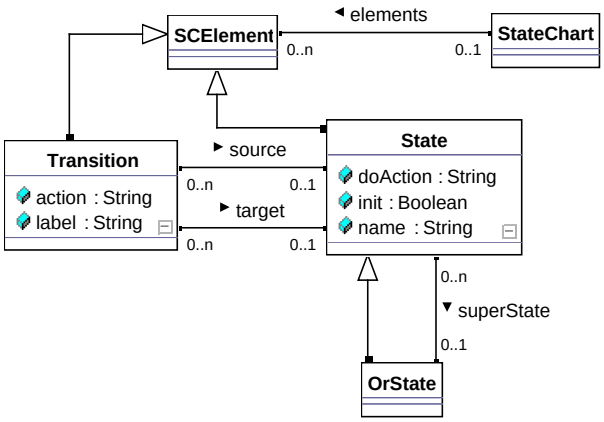


Fig. 2. Class diagram for statechart meta model

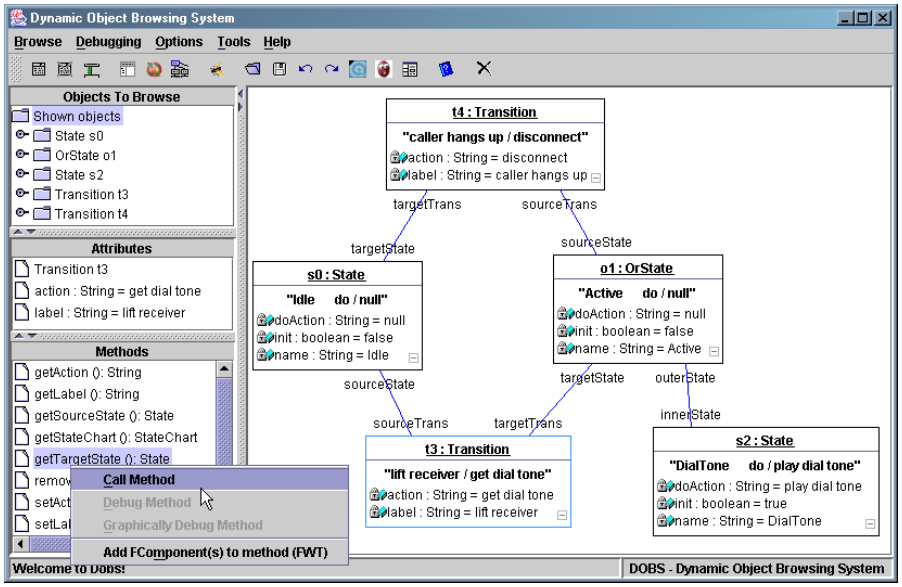


Fig. 3. Abstract syntax in DOBS

Model transformations to PSM

Based on the meta model, model transformations are discussed in the sense of model driven architecture(MDA). As an example for a simple model transformation the flattening of complex statecharts to plain state machines is specified. This flattening is discussed first, since this allows to simplify the specification of operation semantics and of consistency checks later on.

Flattening of statecharts with or-states deals with the replacement of transitions targeting or-states, and with the replacement of transitions leaving or-states and with the removal of or-states that have no more transition attached.

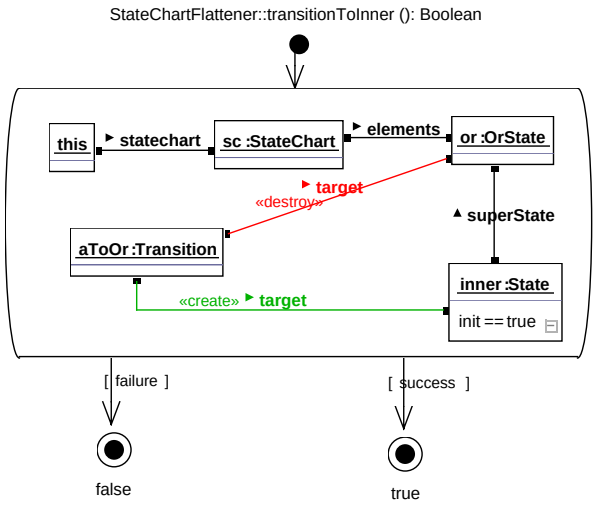


Fig. 4. Replacing transitions targeting or-states

Fig. 4 specifies the replacement of transitions targeting or-states. Such transitions are simply re-targeted to the initial state of the statechart embedded within the or-state. Note, Fig. 4 employs a new (functional) class *StateChartFlattener* that has a *statechart* reference to *StateChart* objects. The graph transformation in Fig. 4 matches a statechart object *sc* containing an or-state *or* that is targeted by a transition *aToOr*. In addition, the graph transformation identifies a sub-state object *inner*, where the *init* attribute has value *true*, i.e. the initial sub-state. As indicated by the ≪*destroy*≫ and ≪*create*≫ markers, the graph transformation of Fig. 4 removes the *target* link connecting transition *aToOR* and or-state *or* and adds a new target link leading to sub-state *inner*. If this rule is applied as often as possible, all transitions leading to or-states are redirected to the corresponding initial states.

In Fig. 2 class OrState inherits from class State. This means, any time we need a node of type State, a node of type OrState does the job as well

(substitutability). For our graph rewrite rule this means node inner may either match a plain state or an or-state. Thus, our graph transformation works for nested or-states as well.

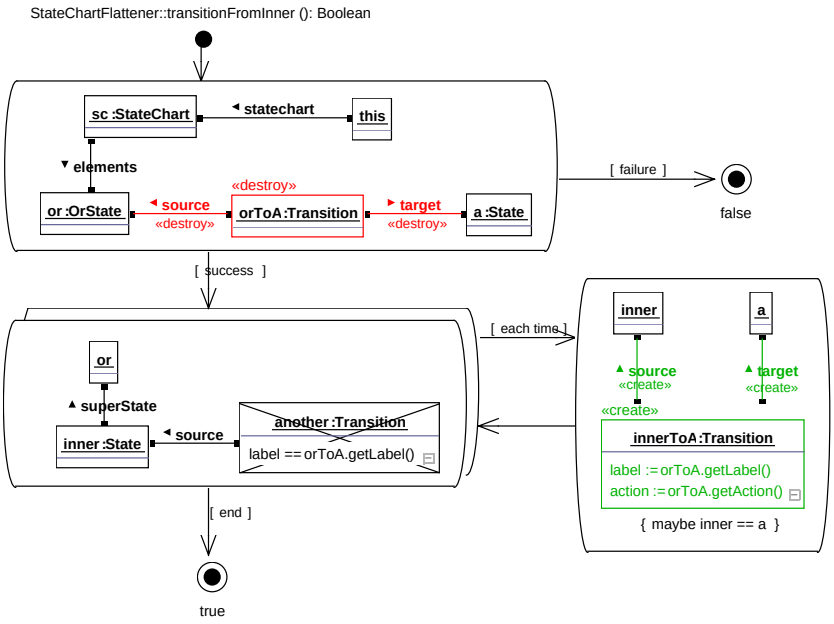


Fig. 5. Replacing transitions leaving or-states

The graph transformation of Fig. 5 replaces transitions leaving or-states. This is done in three steps. The first graph rewrite rule identifies a transition *orToA* with a source link to an or-state *or* and destroys it. If this rule has been applied, successfully, the second graph rewrite rule identifies inner states of *or* that do not already have a leaving transition with the same label. Story diagrams use crossed out elements to specify negative application conditions. The second graph rewrite rule has two stacked shapes. Such a rule is called a *for-each activity*. For-each activities are iteratively applied as long as new matches are found. Due to the *each time* transition in Fig. 5, each time when the second graph rewrite rule identifies an inner state without an appropriate leaving transition, the third graph rewrite rule is executed. The negative node *another* prevents the creation of a new transition if the *inner* state has already such a transition. This implements the priority rules of UML statecharts. The third graph rewrite rule creates a new transition leaving the corresponding *inner* state, targeting the same state *a* as the old transition. In addition, the transition label and the transition action are transferred.

In general story diagrams employ isomorphic rule matching, only. However, the *maybe inner==a* clause of the third graph rewrite rule allows nodes *inner* and *a* to be matched on the same host graph object. This handles self transitions.

The graph rewrite rule of Fig. 6 employs two negative nodes ensuring that the considered or-state has no out-going and no incoming transition. For simplicity, a third negative application condition ensures that the considered or-state is not embedded in another or-state. This means, we handle nested or-states outside in. If all conditions hold, the or-state is destroyed and all its sub-states are added to the statechart *sc*. In addition, the init flag of the or-state is transferred to its initial sub-state. Thus, if the or-state was a usual state, its initial sub-state becomes a usual state, too. If the or-state was the initial state of the whole statechart, its initial sub-state becomes the new initial state of the statechart.

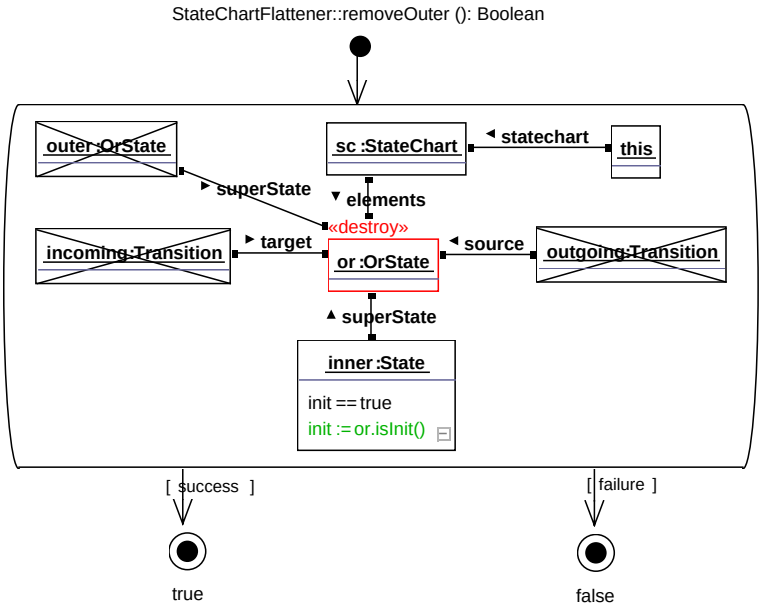


Fig. 6. Removing obsolete or-states

In story diagrams, the graph grammar like application of a set of rules as long as possible needs to be programmed, explicitly. This may be done as shown in the (pseudo) graph transformation of Fig. 7, which employs a boolean constraint calling our three model transformations. If one of the above transformation is applied (and returns true), we follow the *success* transition, and the boolean constraint is evaluated, again. If no transformation succeeds, the

transformation terminates. Thus, the application of transformation *flattenStateChart* removes all (even nested) or-states and results in a simple state machine.

Note, the boolean *or* operators connecting our three basic model transformations use left precedence and short circuit evaluation. This means, transitionFromInner has higher priority than transitionToInner which again has higher priority than removeOuter. Thus, the proposed way of applying a set of graph transformation implies precedences on the transformation rules. Relying on these precedences, the negative nodes of 6 could have been omitted..

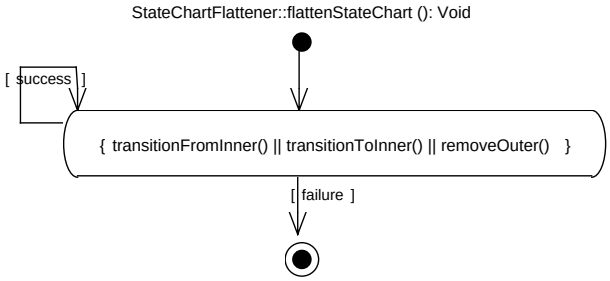


Fig. 7. Employing the transformation rules as long as possible

Operational semantics of PSM

This chapter provides the operational semantics for our statecharts. Of course we could interpret statecharts with (nested) or-states directly. However, this would need some more complicated rules. Thus, to facilitate the example this chapter assumes that the state chart is first flattened and all or-states are properly replaced. Then, the statechart may be executed using the graph transformation of Fig. 8.

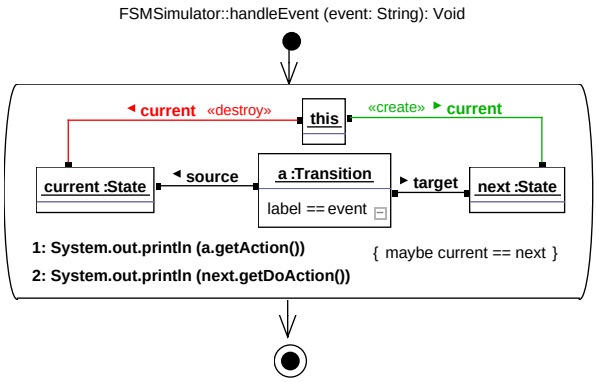


Fig. 8. Firing transitions

For handling events, we employ an object of type *FSMSimulator*. This simulator object has a *current* edge marking the currently active state. If method *handleEvent* is called, it tries to identify an outgoing transition *a* with the label provided in parameter *event*. The *maybe current==next* clause allows to handle self transitions). If such a transition exists, the *current* edge is redirected to the target state of the transition. In addition, the transition action and the do-action of the target state are executed. For simplicity reasons, this is simulated using *System.out.println*. Alternatively, the actions might employ Java syntax and a Java interpreter like the bean shell [38] could have been used to actually execute the actions.

3.2 Comparison with other related approaches

Since vertical transformations and horizontal transformations are just transformations for different purposes, the comparison given in the following chapter holds here, too. However, there are some specific techniques in current CASE tools related to vertical transformations.

Usually, platform specific information is added to a platform independent model with the help of stereotypes. For example, a platform independent server class might be stereotyped to use CORBA as its underlying communication mechanism. Such a stereotype parameterizes the code generator of the underlying CASE tool to generate platform specific stub classes and communication means. Similarly, a stereotype might specify different priority options within a statechart. Again, this parameterizes the code generator of the underlying CASE tool. In some CASE tools, stereotypes may be exploited in template based code generators. This means the user may provide a meaning for new stereotypes by editing specific code generation templates. Altogether, we do not consider such stereotype mechanisms as actual model transformations. For example they seem not appropriate for state chart flattening or other more complex transformation tasks.

The only notable mechanism for model transformation provided by CASE tools so far is the design pattern expansion mechanism provided e.g. by Rational ROSE XDE [39] and Artisan. This is a kind of macro mechanism allowing to instantiate and adapt multiple design elements with a single pattern expansion command. In addition, methods generated by such a pattern expansion may already have an implementation that is adapted to the specific pattern occurrence during pattern expansion. Actually, this is a helpful mechanism for vertical model transformations. However, in graph grammar terms this is a specific node replacement system. Compared to a full graph rewriting system, such node replacement systems provide very limited modeling means. While both kinds of graph grammars in principle generate the same classes of graphs, our experiences have shown, that comfortable modeling of complex transformations require complex object patterns in the left-hand side, various kinds of application conditions and control structures for the combination of simple rules to complex transformation algorithms.

While the design pattern expansion mechanism provided by some CASE tools is not rich enough from the point of view of graph grammars, the design pattern expansion mechanism in these tools has the advantage, that the rules are written in an extended UML notation, i.e. at the level of concrete syntax. Story diagrams are a general modeling language. Therefore, story diagram rules work on the meta model level, i.e. at the level of abstract syntax. For end users, dealing with the internal meta model of some tool is a major obstacle. To facilitate the writing of story diagram rules for such end users we plan to allow concrete syntax in story diagram rules, too.

The example transformations, shown in this chapter, utilize many of the more sophisticated features of story diagrams e.g. programmed graph rewriting, method invocations, for-each activities, multi objects, maybe clauses, negative application conditions, path expressions, etc. Similar language elements are provided by Progres graph transformations [35], only. Due to our experiences, such sophisticated modeling constructs are mandatory for the specification of complex functionality as required for CASE tools.

The MoTMoT approach [40, 41] also uses story diagrams to specify model transformations. But unlike Fujaba, MoTMoT does not offer an editor to create story diagrams, but provides a UML 1.4 profile which uses annotated UML class diagrams and annotated UML activity diagrams to model story diagrams. This way, story diagrams can be drawn with every UML 1.4 compliant editor, like Together, MagicDraw or Poseidon.

The MOLA language [42] developed at the University of Latvia uses very similar concepts as used in story diagrams. MOLA also uses typed graphs and programmed graph transformations. But until now, MOLA remained only a language specification and an editor. Interpreter or compiler support is still missing.

The GReAT tool [43] uses similar transformation rules. To structure more rules, GReAT makes use of data flow diagrams rather that control flow diagrams used in Fujaba.

The VIATRA tool [44] uses a combination of graph transformation rules (to describe structural modifications) and abstract state machines [45] (used as control structures to arrange elementary rules into complex transformations) and it is integrated into the Eclipse environment.

ATOM3 [46] is a multi-paradigm visual modeling framework also using graph transformation for defining semantics of individual modeling languages and transformations. In addition to discrete modeling languages (like Petri nets, statecharts, etc.) the tool also aims at integrating domains of continuous models (as widely used control theory).

4 Graph Transformations for Horizontal Model Transformation

The vertical model transformations from the previous section are used to implement a refinement (or its inverse abstraction) relationship. The input and output models conform to metamodels that represent system properties at different levels of abstraction. In contrast, horizontal model transformations are used to implement mappings between models at the same level of abstraction.

In some cases, the output model is an in-place updated version of the input model. Using refactorings as an example, it will be illustrated that one implements such transformations by rephrasing a system in the same language: a transformation is defined on only one metamodel that serves as input and output.

In other cases, the output models are constructed from the input models by translating their information to other modeling languages. In this section, it will be shown that such horizontal translations are defined on a different metamodel for input and output. The generative programming community also recognizes the distinction between horizontal and vertical transformations that are defined either within the same metamodel or across different metamodels [47]. For a brief classification, we refer to the introduction of Visser's survey of rewriting strategies in program transformation systems [48].

4.1 The Fujaba Approach

Refactoring as an example of horizontal rephrasing

Refactorings are transformations that change the structure of a program while maintaining its external behavior [49]. Other examples of horizontal rephrasings are normalization and optimizations. In a model driven engineering context one implements such program transformations following the same approach: the program is parsed to a model which is then transformed horizontally while preserving certain properties.

When implementing refactorings, one first has to decide what properties characterize external program behavior. From this, one has to agree upon

a metamodel on which the transformations and properties can be formalized without navigating through irrelevant details. In order to express the transformations, one finally needs a language that assists reasoning about the correctness of a refactoring implementation, again while hiding unimportant details. In [50], it is shown that graph rewriting is a promising formalism for this.

A typical example of a graph rewriting rule implementing a refactoring is given in Fig. 9. It represents the Pull Up Method refactoring, expressed as a Fujaba SDM specification. The metamodel used to express the refactoring, is Fujaba's internal metamodel. The diagram shows how *method*, which is implemented in *container*, is moved to *superclass*. By expressing the refactoring

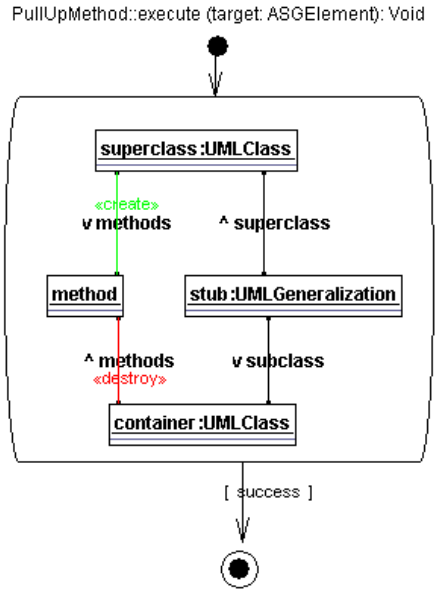


Fig. 9. Pull Up Method refactoring expressed using Fujaba's Story Driven Modeling (SDM) language.

on Fujaba's internal metamodel, the code generated from this graph rewriting can be called from a Fujaba plugin for refactoring UML class diagrams [51].

Refactorings are not the only applications of horizontal rephrasing with the target of improving an existing infrastructure. Similar approaches exist for the restructuring of software architectures: for example, Fahmy and Holt [52] use graphs to represent software architectures and conditional graph rewriting rules to improve quality attributes of a software architecture.

Horizontal Translation

To be able to focus on different aspects of a program, software engineers usually employ different views on the software. A typical example, which was introduced in [14], considers syntax trees and flow diagrams. Both diagrams contain the same information, but allow one to extract a certain type of information more easily. For example, a syntax tree representation of a method is excellent for specifying low-level code transformations, while flow diagrams help the programmer to get a better understanding of parts of the code, and the calculation of certain properties like cyclomatic complexity. Since the same program is represented in different modeling languages, horizontal translations are required for maintaining the consistency between the models. Pair grammars and Triple graph grammars offer a declarative solution to implement this.

For the example, a triple graph grammar rule is given in Fig. 10. It relates the addition of an assignment construction to an abstract syntrax tree to that of a control flow diagram. This rule is triggered whenever the syntax tree is adapted and updates the control flow graph. Apart from this forward rule, two other rules are part of the triple rule. The backward rule specifies how the addition of the assignment in the flow graph must update the syntax graph. The last rule analyses and updates the correspondence graph when both structures evolve. Another example relates a program structure graph to an architectural description of a program [53]. Different graph rewriting environments like PROGRES [54] and Fujaba support the specification of triple graph grammars.

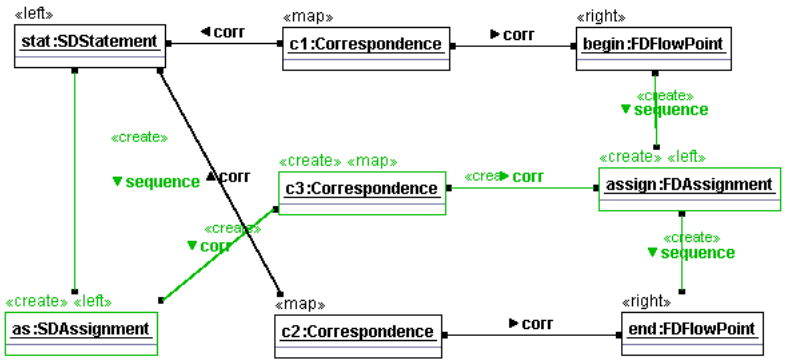


Fig. 10. Forward triple graph rule

4.2 Comparison with other related approaches

Hypergraph transformation approaches

Next to using simple, typed graphs as a formalism for specifying horizontal transformations, other approaches can use more advanced structures like hypergraphs [5] and hypergraph grammars. A typical use of these structures is described in [55], where a network is represented as a hypergraph. In such a network, the basic entities, like servers and clients are modeled as hyperedges, while the nodes represent communication between these entities. The reconfiguration of such a network architecture is then described as a set of hyperedge replacement rules. The main advantage of using hypergraphs are the improved flexibility in modeling, and the possibility of creating hierarchical graphs. Hierarchical graph transformation [6] can then be used to specify high-level translations or rephrasings as described in [56].

Metamodel design

Fujaba's metamodel is similar to the UML metamodel. In [57], it is shown which key shortcomings to the UML metamodel need to be resolved for expressing source-consistent refactorings. By extending the UML 1.4 metamodel with constructs for modeling method bodies in a language neutral way, one can reuse (parts of) refactoring implementations (and related bad code smell specifications) for different OO languages.

Based on a list of criteria for UML refactoring (like simplicity, backward compatibility with previous UML versions, the ability to integrate with code smell detectors and maintain the consistency with the source code), the action semantics package of UML 1.5 was found inadequate as a basis for expressing UML refactorings. Therefore, the authors proposed a refactoring oriented set of extensions to the UML 1.4 metamodel. Research on the FAMIX metamodel [58] indicated that the notion of access-, call- and update-behavior had to be augmented with the notions of type-casting and nested scopes containing local variables.

In 2001, researchers from the reengineering community [59] agreed on the Dagstuhl Middle Metamodel [60] for representing software on a medium level of abstraction and on Datrix [61] for a low level representation. It appears to be unfeasible to design a metamodel that satisfies the needs of both low-level refactoring tools and high-level visualization tools. Therefore, it may be more promising to investigate how the consistency between models in different modeling languages can be maintained. Source code refactorings could be implemented on full-fledged abstract syntax graphs while visualization tools could be implemented using minimalistic metamodels for class diagrams, statecharts, sequence diagrams, etc. Changes resulting from refactoring applications on the concrete models could be propagated to the visualization models by triggering model translations.

Consistency through Distributed Graph Transformation

Bottoni et al. use distributed graph rewriting to decompose a refactoring on a redundant metamodel into several smaller units [62]. This technique is useful in a UML context where structural program information is redundantly stored in both the core package as the class diagram package of the metamodel. By coordinating the distributed rules, one can maintain the consistency between all diagrams.

5 Graph Grammars for Model Analysis and Verification

Although graph grammars provide a visual yet formal language for capturing a wide range of model transformations with automated code generation and tool support, automation and formality does not alone guarantee correctness as the specifications of model transformations can also be erroneous. This problem can be severe since if some bugs are detected during model driven development in a PSM or in the target program, the designers have to be ascertained that the automated model transformations do not introduce new flaws into the system model. As a consequence, one has to prove precisely (and automatically, if possible) that a model transformation is correct [63].

5.1 Correctness criteria for transformations

The most elementary requirements of a model transformation are syntactic. The minimal requirement is to assure *syntactic correctness*, i.e., to guarantee that the generated model is a syntactically well–formed instance of the target language. An additional requirement (called *syntactic completeness*) is to completely cover the source language by transformation rules, i.e., to prove that there exists a corresponding element in the target model for each construct in the source language (and we did not forget about any situations when specifying the transformation rules) .

However, in order to assure a higher quality of model transformations, at least the following *semantic requirements* should be verified for a model transformation. First one must guarantee that a transformation is *terminating* and *confluent* (thus it yields a unique result). As model transformations may also define a projection from the source language to the target language, semantic equivalence between models cannot always be proved. Instead we aim to prove the *property preservation* of a transformation for certain (transformation specific) correctness properties. For instance, in a statechart-to-code transformation, one may prescribe the natural criterion for correctness that each state configuration that is reachable from the initial configuration in a statechart model should be reachable in the target code as well.

5.2 CheckVML: A tool for model checking graph grammars

The main idea of the CheckVML approach [64, 65, 66, 67] is to exploit off-the-shelf model checker tools like SPIN [68] for the verification of graph grammars. More specifically, it translates a graph transformation system parameterized with a type graph and an initial graph (via an abstract transition system representation) into its Promela equivalent to carry out the formal analysis in SPIN. Furthermore, graph properties that capture the requirements for the system visually are also translated into equivalent temporal logic formulae.

Graph and rule model.

CheckVML uses directed, typed and attributed graphs as model representation. Inheritance between node types is also supported.

Concerning the rule application strategy, it prescribes that a matching in the host graph should be an injective occurrence of the LHS (and NAC) graphs. Furthermore, all dangling edges are implicitly removed when deleting a node. Arbitrary creation and deletion of edges are allowed while there is an *a priori* upper bound for the number of nodes (of a certain type) potentially created during a verification run, which is passed as a parameter to the translator.

The model checking problem.

The *model checking problem* is to automatically decide whether a certain correctness property holds in a given system by systematically traversing all enabled transitions in all states (thus all possible execution paths) of the system. The correctness properties are frequently formalized as LTL formulae.

Traditional model checkers are based on so-called *(state-) transition systems (TS)*, where the structure of a state consists of a subset of a finite universe of propositions. This determines the storage structures used (usually Binary Decision Diagrams or a variant thereof), the logic used to express properties (propositional logic extended with temporal operators, usually LTL or CTL) and the model checking algorithms (automata-based or tableau-based).

From graph grammars to transition systems: an overview.

In graph grammars, a state is constituted by a graph, while a transition means the application of a rule for a certain matching of the LHS in such a graph. Traversing all enabled transitions then means applying all rules on all possible matchings. During this process, it is important to realize whether a certain state has been investigated before; therefore the model checker has to store all the graphs that it has encountered.

Since graph transformation is a meta-level specification paradigm (i.e. it defines how each instance of a type graph should behave) while the transition system formalism of Promela is a model-level specification language (i.e.

a Promela model describes how a specific model should behave), the main challenge in this approach is *rule instantiation*, i.e. to generate one Promela transition for all the potential application of a graph transformation rule *in a preprocessing phase* at compile time.

State variables and initialization

A model element (object, link or attribute) is considered to be dynamic if there is at least one rule that potentially modifies (creates, destroys, updates) the element. The encoding of dynamic model elements into state variables is driven by the metamodel (type graph). A one-dimensional boolean state variable array (a unary relation symbol) is defined for each dynamic node type; and a two-dimensional boolean state variable array (a binary relation symbol) for each edge type.

In traditional model checker tools, the dimension of each array and all the enumeration types must be restricted to be finite at compile-time. For the corresponding graph grammar, this restriction implies that there exists an *a priori* upper bound for the number of nodes in the model for each node type given explicitly by the user of CheckVML. In this respect, we suppose that when a new node is to be created it is only activated from the bounded "pool" of currently passive objects (deletion means passivation, naturally), and the same applies to the interpretation of links.

Note that the restriction for the existence of a priori upper bounds is a direct consequence of using SPIN, which prescribes that the domains of all state variables have to be a priori finite. Fortunately, one can insert special assert statements in the Promela code which checks if these limits were exceeded during verification, and then the user can increase the corresponding upper bound before the next compilation and verification run. Alternatively, one can switch to another model checker (like dSPIN [69]) that supports object creation without restrictions, which needs further investigations.

In general terms, the initial configuration of the application model is projected into the initial state of the TS. In this respect, exactly those locations of state variable arrays evaluate to true in the initial state for which the related model elements are existent in the initial configuration.

Translation of rules.

Potential applications of the graph transformation rules that specify the dynamic behavior of the style are encoded into transitions (guarded commands) of the corresponding TS.

Since the encoding only introduces state variables for dynamic model elements, we also have to eliminate conditions that refer to the static parts of the model. For that reason, the generation process of transitions is driven by a graph pattern matching engine, which collects all the matching instances of the static parts of the preconditions of a rule. If the guard of a certain guarded

command can never be satisfied due to the failure of pattern matching in the static structure then this transition is not generated at all in the target TS.

Although this compile-time preprocessing can be time-consuming, since all the potential matches of a rule have to be encountered, we only have to traverse a relatively small part of the state space for this step as graph transformation rules define local modifications to the system state thus it is typically negligible when compared with the time required for traversing the entire state space during model checking.

Properties to be verified.

The requirements against the system under design are most typically captured by *safety* and *reachability* properties. A safety property defines a desired property that should always hold on every execution path or (equivalently) an undesired situation which should never hold on any execution paths. A reachability property describes, on the contrary, a desired situation which should be reached along at least one execution path. From a verification point of view, safety and reachability properties are dual: the refutation of a safety property is a counter-example which satisfies the reachability property obtained as the negation of the safety property. On the other hand, if a safety property holds (or a reachability property is refuted) the model checker has to traverse the entire state space.

A safety or reachability property can be interpreted as a special graph pattern (called *property graph* in the sequel) which immediately terminates the verification process if it is matched successfully. It is shown in [70] that the properties expressible in this way are equivalent to the $\exists\neg\exists$ fragment of ($\forall$-free) first order logic with binary predicates.

The CheckVML tool: Pros and Contras.

The potential benefits of the CheckVML tool are the following:

1. The tool considers typed and attributed graphs which fits well to the metamodeling philosophy of UML and other modeling languages.
2. The size of the state vector depends only on the dynamic model elements (i.e., elements that can be altered by at least one graph transformation rule) while immutable static parts of a model are not stored in the state vector. This is a typical case for data-flow like systems (e.g., dataflow networks, Petri nets, etc).
3. CheckVML can be easily adapted to various back-end model checker tools (like SAL [71], Murϕ [72]) due to the usage of XML input and output formats.

The essential disadvantage of the approach is that dynamic model elements (that are not restricted by static constraints) easily blow up both the verification model and state space, moreover, symmetries in graphs can be handled only for limited cases.

Verification experiments

Concerning the practical usefulness of the approach, two questions might immediately arise: (i) what is the size of the models that can be verified, and (ii) how does one obtain meaningful initial models.

To answer the first question, a detailed experimentation have been carried out in [67] where two main roads of model checking graph transformations (namely, Groove [73] vs. CheckVML) were compared on various case studies having essentially different characteristics concerning the dynamic and symmetric nature of the problem.

For instance, SPIN (CheckVML) managed to verify the graph transformation version of dining philosophers problem with 10 philosophers — which is a relatively good problem size concerning (i) the use of explicit state model checkers and (ii) the automatic generation of the Promela code without further manual optimizations. On the other hand, verification failed much sooner in case of inherently dynamic and symmetric problems (like the mutual exclusion example of [74]).

These experiments also revealed that (i) the space consumption of these tools were comparable, (ii) SPIN (CheckVML) had a clear advantage concerning the time required for verification of problems of the same size, but (iii) Groove were able to handle problems with a larger dimension due to its sophisticated algorithms for graph isomorphism checks.

Now, to answer the second question based upon these experiments, it seems unrealistic that correctness properties could be verified on a large initial graph (of a complex system model). But the practical use of model checkers is most frequently to automatically find conceptual flaws in the specification (i.e. refutation instead of verification), for which one can use much smaller initial models. Furthermore, the execution traces retrieved during the verification of reachability properties can immediately be used as (automatically derived) *test cases* for a more complex system. The exploitation of these issues is a direction of future research.

5.3 Reachability analysis of flattened statecharts

Based upon the operational semantics rule of statecharts (see Figure 8), a reachability analysis is carried out to decide whether all states of a flattened statechart are reachable or not. For this purpose, we first translate a sample flattened statechart model (consisting of states `s1`, `s2` and `s3` and transitions `t1: s1->s2`, `t2: s2->s3` and `t3: s1->s3`) and the rule into a corresponding TS.

Since there is a single dynamic element in the class diagram (namely, the `current` edge type), the TS encoding of our model contains only a single state variable array. All other static elements are eliminated by CheckVML in the preprocessing phase since they will never change during the verification (model execution).

```
/* upper bound for the domain of variables  */ #define MAXFSM 1
#define MAXSTATE 3 /* instances (individuals) */ #define a1 0
#define s1 0 #define s2 1 #define s3 2

/* state variable: one or two-dimensional arrays */ /* for dynamic
node, edges, and attributes */
 bool current[MAXFSM][MAXSTATE];

/* Initialization of state variables */ init /* All locations in
the state variable array are set properly */
 current[a1][s1] = true; current[a1][s2] = false; current[a1][s3] = false;
```

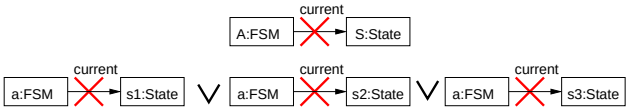
All potential matches of the graph transformation rule of the statechart semantics are collected also at compile-time and translated into the following guarded commands (which is, essentially a pair of guards and elementary update operations). Below we present the TS equivalent of one potential matching, namely, for transition `t1`.

```
/* transition: Boolean guard -> update1; update2; */ :: atomic
 fsm[a1] && current[a1][s1] && state[s1] && source[t1][s1] &&
 transition[t1] && target[t1][s2] && state[s2] ->
   current[a1][s1] = false; current[a1][s2] = true;
```

Static parts of the model and dead guarded commands with a guard always evaluated to false due to some static parts of the model are removed this time as well. As a result, we end up with the following very compact representation of our statechart model.

```
/*  non-deterministic selection between transitions; */ /*
executed as an atomic step; */ do :: atomic current[a1][s1] ->
   current[a1][s1] = false; current[a1][s2] = true;
:: atomic current[a1][s2] ->
   current[a1][s2] = false; current[a1][s3] = true;
:: atomic current[a1][s1] ->
   current[a1][s1] = false; current[a1][s3] = true;
od
```

The required reachability property stating that each state of the statechart should be reachable can be expressed by the graph pattern of Figure 11. The parametric property graph in the upper part (expressing that a state S never becomes current, which is an undesired situation) is first instantiated to enumerate all potential matchings in the instance graph. Then the corresponding LTL formula for the safety property can be easily derived, which simply collects all these matchings into a disjunctive formula with global (G) temporal quantifiers (which prescribes that the formula should hold for all states on all execution paths).




```
/* Definitions for each match of the property pattern */ #define
match1 (fsm[a] && state[s1] && NOT current[a][s1]) #define match2
(fsm[a] && state[s2] && NOT current[a][s2]) #define match3 (fsm[a]
&& state[s3] && NOT current[a][s3])

/* G (p): globally p, i.e. the property p holds */ /* for all
states on all execution paths        */
  G (match1) || G (match2) || G (match3)
```

Fig. 11. The reachability property of states

When running the model checker, a counter example for the property of Figure 11 shows an execution of the flattened statechart where it is proved that all states of the statechart become reachable.

5.4 Comparison with other related approaches

The theoretical basics of verifying graph transformation systems by model checking have been studied thoroughly by Heckel et. al. in [74, 75] (and subsequent papers). The authors propose that graphs can be interpreted as states and rule applications as transitions in a transition system. Which idea is adopted more or less in all existing model checking approaches of graph grammars.

The main current alternative for CheckVML is provided by the GROOVE framework [73], which uses the core concepts of graphs and graph transformations all the way through during model checking (instead of translating them into existing model checking tools). This means that states are explicitly represented and stored as graphs, and transitions as applications of graph transformation rules; moreover, properties to be checked should be specified in a graph-based logic, and graph-specific model checking algorithms should be applied. A more detailed comparison between GROOVE and CheckVML can be found in [67].

A theoretical framework by Baldan et. al. [76] aims at analyzing a special class of hypergraph rewriting systems by a static analysis technique based on foldings and unfoldings of a special class of Petri nets. This framework is able to verify infinite state systems by calculating a representative finite complete prefix. Unfortunately, no supporting tools have been reported in the literature.

Dotti et. al. uses object-based graph grammars [77] for modeling object-oriented systems and defines a translation into SPIN to carry out model checking. The main difference (in contrast to CheckVML) is that the authors allow

a restricted structure for graph transformation rules that is tailored to model message calls in object-oriented systems. Therefore, CheckVML is more general from a pure graph transformation perspective (i.e. any kind of rules are allowed). However, the framework of [77] relies on higher-level SPIN/Promela constructs (processes and channels), which might result better run-time performance.

6 Conclusions and Future Work

In this chapter we used graph transformation to specify and apply model transformations in the context of Model Driven Software Engineering. For this reason, we presented the theoretical background of graphs and graph transformations. Thereafter, we demonstrated the practical feasibility of graph transformations for horizontal (PIM-to-PIM), vertical (PIM-to-PSM) model transformations and the verification of these model transformations. For each domain typical case studies were introduced and the existing tool support was described. Based on these examples, we believe that graph rewriting systems are well suited in the context of model-transformations. Especially pair grammars and triple graph grammars can become the dominant formalism for automated PIM-to-PIM and PIM-to-PSM transformations.

Throughout the examples, we used the syntax of Fujaba's graph rewriting language which is very close to standard UML activity diagrams and object diagrams. In order to realize the MDA vision of platform independence, a recent investigation demonstrated how this syntax could be completely aligned with UML stereotypes and MOF [78]. The result of this work makes it possible to specify model transformations in one UML tool and deploy them on the repository of another UML tool.

To continue this movement towards standardization and cross-fertilization among formalisms and tools, one may investigate how OCL can be integrated with the standardized graph transformation language to support textual constraint specifications embedded in graph rewriting rules. This may result in a concrete proposal for supporting queries, views and transformations on models based on the synergies between graph transformation and UML/MOF.

References

1. OMG (The Object Management Group). MDA specifications, http:// www.omg.org/ mda/ specs.htm., 2002-2004.
2. Shane Sendall and Wojtek Kozaczynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20 (5):42–45, 2003.
3. OMG (The Object Management Group). OMG MOF 2.0 Query, Views, Transformations request for proposals (QVT RFP), http://www.omg.org/ techprocess/ meetings/ schedule/ MOF 2.0 Query View Transf.RFP.html, 2003.

4. Luciano Baresi and Reiko Heckel. Tutorial introduction to graph transformation: A software engineering perspective. In *International Conference on Graph Transformation, ICGT, LNCS*, volume 2505 of *Lecture Notes in Computer Science*, pages 402–439. Springer, 2002.
5. Annegret Habel. *Hyperedge replacement: grammars and languages*, volume 643 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1992.
6. Frank Drewes, Berthold Hoffmann, and Detlef Plump. Hierarchical graph transformation. *Journal of Computer and System Sciences*, 64(2):249–283, 2002.
7. Terrence W. Pratt. Pair grammars, graph languages and string-to-graph translations. *Journal of Computer and System Sciences*, 5:560–595, 1971.
8. Andrea Corradini, Ugo Montanari, Francesca Rossi, Hartmut Ehrig, Reiko Heckel, and Michael Löwe. Algebraic approaches to graph transformation I: Basic concepts and double pushout approach. In G. Rozenberg, editor, *Handbook of Graph Grammars and Computing by Graph transformation, Volume 1: Foundations*. World Scientific, 1997.
9. Hartmut Ehrig, Reiko Heckel, Martin Korff, Michael Löwe, Leila Ribeiro, Annika Wagner, and Andrea Corradini. Algebraic approaches to graph transformation II: Single pushout approach and comparison with double pushout approach. In G. Rozenberg, editor, *The Handbook of Graph Grammars and Computing by Graph Transformation, Volume 1: Foundations*. World Scientific, 1997.
10. Annegret Habel, Reiko Heckel, and Gabriele Taentzer. Graph grammars with negative application conditions. *Fundamenta Informaticae*, 26(3/4):287–313, 1996.
11. Albert Zündorf. The fujaba toolsuite. http://www.fujaba.de/, 1999.
12. Thorsten Fischer, Jörg Niere, Lars Torunski, and Albert Zündorf. Story Diagrams: A New Graph Rewrite Language Based on the Unified Modeling Language and Java. In *Proceedings of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT)*, volume 1764 of *LNCS*, pages 296–309. Springer Verlag, November 1998.
13. Albert Zündorf. Rigorous object oriented software development, habilitation thesis, 2001.
14. Andy Schürr. Specification of graph translators with triple graph grammars. In *Proceedings 20th Workshop on Graph-Theoretic Concepts in Computer Science*, pages 151–163, 1994.
15. Anna Gerber, Michael J. Lawley, Kerry Raymond, Jim Steel, and Andrew Wood. Transformation: The missing link of MDA. In *Proc. 1st International Conference on Graph Transformation, ICGT'02*, volume 2505 of *Lecture Notes in Computer Science*, pages 90–105. Springer Verlag, 2002.
16. CWM Partners. Common Warehouse Metamodel specification. OMG documents: ad/01-02-{01,02,03}, 2001.
17. W3C. The extendible stylesheet language transformations specification. http://www.w3.org/TR/1999/REC-xslt-19991116, 1999.
18. M. Peltier, Jean B´zevin, and G. Guillaume. MTRANS: A general framework, based on XSLT for model transformations. In *WTUML '01, Proceedings of the workshop on Transformations in UML, Genova, Italy*, 2001.
19. Object Management Group. Meta object facility (mof) specification, April 2002.
20. OMG. Interchange metamodel in xml. OMG document: formal/01-02-15, 2001.

21. Z. Somogyi, F. Henderson, and T. Conway. Mercury: an efficient purely declarative logic programming language. In *Proceedings of the Australian Computer Science Conference, Glenelg, Australia*, pages 499–512, 1995.
22. Michael Kifer, Georg Lausen, and James Wu. Logical foundations of object-oriented and frame-based languages. *Journal of the ACM*, 42(4):741–843, 1995.
23. OMG. UML profile for enterprise distributed computing (EDOC). OMG document: ptc/02-02-05, 2002.
24. DSTC. Breeze: workflow with ease. `http://www.dstc.edu.au/Research/Projects/Pegamento/Breeze/breeze.html`.
25. Jochen M. Küster, Shane Sendall, and Michael Wahler. Comparing two model transformation approaches. In *Proc. Workshop on OCL and Model Driven Engineering*, October 2004. Satelite event of the Seventh International Conference on UML.
26. Gregor Engels, Reiko Heckel, and Jochen M. Küster. The consistency workbench: A tool for consistency management in UML-based development. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *UML 2003 - The Unified Modeling Language. Model Languages and Applications. 6th International Conference, San Francisco, CA, USA, October 2003, Proceedings*, volume 2863 of *LNCS*, pages 356–359. Springer, 2003.
27. OMG (The Object Management Group). QVT-Merge Group. MOF 2.0 Query, Views, Transformations, Revised Submission., April 2004. OMG document ad/2004-04-01.
28. Heiko Dorr. *Efficient Graph Rewriting and Its Implementation*. Springer-Verlag New York, Inc., 1995.
29. Attila Vizhanyo, Aditya Agrawal, and Feng Shi. Towards generation of high-performance transformations. In *Proc. Generative Programming and Component Engineering*, Lecture Notes in Computer Science. Springer-Verlag, October 2004.
30. Gergely Varró, Katalin Friedl, and Dániel Varró. Graph transformations in relational databases. In *Proc. GraBaTs 2004: International Workshop on Graph Based Tools*, ENTCS. Elsevier, 2004. To appear.
31. Pieter Van Gorp, Dirk Janssens, and Tracy Gardner. Write Once, Deploy N: a performance oriented mda case study. In *Proceedings of the 8th IEEE International Enterprise Distributed Object Computing Conference*. IEEE, September 2004.
32. Gergely Varró and Dániel Varró. Graph transformation with incremental updates. In *Proc. GT-VMT 2004, International Workshop on Graph Transformation and Visual Modelling Techniques*, ENTCS. Elsevier, March 2004. To appear.
33. Leif Geiger and Albert Zündorf. Graph based debugging with Fujaba. In *Proce. Workshop on Graph Based Tools, International Conference on Graph Transformations*, 2002.
34. Gabi Taentzer. AGG: The attributed graph grammar system. http://tfs.cs.tu-berlin.de/agg/, 2003.
35. Progres Group. PROGRES: Programmed graph rewriting system. `http://www-i3.informatik.rwth-aachen.de/research/projects/progres/`, 2004.
36. Albert Zündorf. Graph pattern matching in PROGRES. In *Proc. 5th. Int. Workshop on Graph-Grammars and their Application to Computer Science*, volume 1073 of *LNCS*. Springer, 1996.

37. Leif Geiger and Albert Zündorf. Statechart modeling with fujaba. In *reviewed for Graph Based Tools (GraBaTs); workshop at ICGT 2004*, Rome, 2004.
38. Pat Niemeyer. Beanshell. http://www.beanshell.org/, 2004.
39. IBM. Rational Rose XDE Developer. `http://www-306.ibm.com/software/awdtools/developer/rosexde/`, 2004.
40. Formal Techniques in Software Engineering. Model driven, Template based, Model Transformer (MoTMoT). http://sourceforge.net/projects/motmot/, 2004.
41. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. In *Proc. Int'l Workshop Software Evolution through Transformations (SETra)*, 2004. To appear in ENTCS.
42. Audris Kalnins, Janis Barzdins, and Edgars Celms. Model transformation language MOLA. In *Proceedings of MDAFA 2004 (Model-Driven Architecture: Foundations and Applications 2004)*, pages 14–28, 2004.
43. Gabor Karsai, Aditya Agrawal, Feng Shi, and Jonathan Sprinkle. On the use of graph transformation in the formal specification of model interpreters. In *Journal of Universal Computer Science 9 (2003)*, pages 1296–1321, 2003.
44. György Csertán, Gábor Huszerl, István Majzik, Zsigmond Pap, András Pataricza, and Dániel Varró. VIATRA: Visual automated transformations for formal verification and validation of UML models. In Julian Richardson, Wolfgang Emmerich, and Dave Wile, editors, *Proc. ASE 2002: 17th IEEE International Conference on Automated Software Engineering*, pages 267–270, Edinburgh, UK, September 23–27 2002. IEEE Press.
45. Egon Börger and Robert F. Stärk. *Abstract State Machines. A method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
46. Juan de Lara and Hans Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In Ralf-Detlef Kutsche and Herbert Weber, editors, *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, volume 2306 of *LNCS*, pages 174–188. Springer, 2002.
47. Krzysztof Czarnecki and Ulrich W. Eisenecker. *Generative programming: methods, tools, and applications*. ACM Press/Addison-Wesley Publishing Co., 2000.
48. Eelco Visser. A survey of rewriting strategies in program transformation systems. In B. Gramlich and S. Lucas, editors, *Proc. Workshop on Reduction Strategies in Rewriting and Programming*, volume 57 of *Electronic Notes in Theoretical Computer Science*. Elsevier, May 2001.
49. Martin Fowler. *Refactoring - Improving the Design of Existing Code*. Addison Wesley, July 1999.
50. Tom Mens, Serge Demeyer, and Dirk Janssens. Formalising behaviour preserving program transformations. In *Graph Transformation*, volume 2505 of *Lecture Notes in Computer Science*, pages 286–301. Springer-Verlag, 2002. Proc. 1st Int'l Conf. Graph Transformation 2002, Barcelona, Spain.
51. Pieter Van Gorp, Niels Van Eetvelde, and Dirk Janssens. Implementing refactorings as graph rewrite rules on a platform independent meta model. In *Proceedings of Fujaba Days 2003*, 2003.
52. Hoda Fahmy and Richard C. Holt. Using graph rewriting to specify software architectural transformations. *Proceedings of Automated Software Engineering Conference*, pages 187–196, 2000.

53. Katja Cremer, André Marburger, and Bernhard Westfechtel. Graph-based tools for re-engineering. *Journal of Software Maintenance and Evolution: Research and Practice*, 14:25–292, 2002.
54. Andy Schürr, Andreas J. Winter, and Albert Zündorf. Graph grammar engineering with PROGRES. In *Proceedings 5th European Software Engineering Conference ESEC*, volume LNCS 989, pages 219–234. Springer, 1995.
55. Dan Hirsh, Paola Inverardi, and Ugo Montanari. Modeling software architecutes and styles with graph grammars and constraint solving. *Proceedings of First Working IFIP Conference on Software Architecture (WICSA1)*, pages 127–144, 2000.
56. Lare Grunske. Automated software architecture evolution with hypergraph transformation. In *7th International IASTED on Conference Software Engineering and Application (SEA 03)*, IASTED Proceedings, pages 613–621, Marina del Ray, CA, USA, November 3-5 2003.
57. Pieter Van Gorp, Hans Stenten, Tom Mens, and Serge Demeyer. Towards automating source-consistent UML refactorings. In *Proceedings of the 6th International Conference on UML The Unified Modeling Language.*, 2003.
58. Serge Demeyer, Sander Tichelaar, and Patrick Steyaert. FAMIX 2.0 – the FAMOOS information exchange model. `http://www.iam.unibe.ch/~famoos/FAMIX/`, 09 1999.
59. J. Ebert, K. Kontogiannis, and J. Mylopoulos. Interoperability of reengineering tools, January 2001. Seminar Number 01041, Report Number 296.
60. Timothy C. Lethbridge, Sander Tichelaar, and Erhard Ploedereder. The Dagstuhl Middle Metamodel: A schema for reverse engineering. In *Proceedings of the International Workshop on Meta-Models and Schemas for Reverse Engineering (ateM 2003)*, volume 94 of *Electronic Notes in Theoretical Computer Science*, pages 7–18. Elsevier, May 2004.
61. Richard C. Holt, Ahmed E. Hassan, Bruno Laguë, Sébastien Lapierre, and Charles Leduc. E/R schema for the datrix C/C++/java exchange format. In *Proceedings of the Seventh Working Conference on Reverse Engineering*, page 284. IEEE Computer Society, 2000.
62. Gabriele Taentzer Paolo Bottoni, Francesco Parisi Presicce. Specifying integrated refactoring with distributed graph transformations. In *Proceedings of AGTIVE 2003*, pages 227–242, 2003.
63. Dániel Varró and András Pataricza. Automated formal verification of model transformations. In Jan Jürjens, Bernhard Rumpe, Robert France, and Eduardo B. Fernandez, editors, *CSDUML 2003: Critical Systems Development in UML; Proceedings of the UML'03 Workshop*, number TUM-I0323 in Technical Report, pages 63–78. Technische Universität München, September 2003.
64. Dániel Varró. Towards symbolic analysis of visual modelling languages. In Paolo Bottoni and Mark Minas, editors, *Proc. GT-VMT 2002: International Workshop on Graph Transformation and Visual Modelling Techniques*, volume 72 (3) of *ENTCS*, pages 57–70, Barcelona, Spain, October 11-12 2002. Elsevier.
65. Dániel Varró. Automated formal verification of visual modeling languages by model checking. *Journal of Software and Systems Modeling*, 3(2):85–113, May 2004.
66. Ákos Schmidt and Dániel Varró. CheckVML: A tool for model checking visual modeling languages. In Perdita Stevens, Jon Whittle, and Grady Booch, editors, *Proc. UML 2003: 6th International Conference on the Unified Model-*

ing Language, volume 2863 of *LNCS*, pages 92–95, San Francisco, CA, USA, October 20-24 2003. Springer.
67. Arend Rensink, Ákos Schmidt, and Dániel Varró. Model checking graph transformations: A comparison of two approaches. In *Proc. ICGT 2004: Second International Conference on Graph Transformation*, volume 3256 of *LNCS*, pages 226–241, Rome, Italy, 2004. Springer.
68. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.
69. R. Iosif and R. Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer, September 1999.
70. Arend Rensink. Canonical graph shapes. In D. A. Schmidt, editor, *Programming Languages and Systems — European Symposium on Programming (ESOP)*, volume 2986 of *LNCS*, pages 401–415. Springer-Verlag, 2004.
71. Saddek Bensalem, Vijay Ganesh, Yassine Lakhnech, César Munoz, Sam Owre, Harald Rueß, John Rushby, Vlad Rusu, Hassen Saïdi, N. Shankar, Eli Singerman, and Ashish Tiwari. An overview of SAL. In C. Michael Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, pages 187–196, 2000.
72. *The Murphi Model Checker*. `http://verify.stanford.edu/dill/murphi.html`.
73. Arend Rensink. The GROOVE simulator: A tool for state space generation. In M. Nagl and J. Pfalz, editors, *Applications of Graph Transformations with Industrial Relevance (AGTIVE)*, volume ?? of *LNCS*. Springer-Verlag, 2003. To be published.
74. Reiko Heckel. Compositional verification of reactive systems specified by graph transformation. In *Proc. FASE: Fundamental Approaches to Software Engineering*, volume 1382 of *LNCS*, pages 138–153. Springer, 1998.
75. R. Heckel, H. Ehrig, U. Wolter, and A. Corradini. Integrating the specification techniques of graph transformation and temporal logic. In *Proc. Mathematical Foundations of Computer Science (MFCS'97), Bratislava*, volume 1295 of *LNCS*, pages 219–228. Springer, 1997.
76. Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In Andrea Corradini, Hartmut Ehrig, Hans-Jörg Kreowski, and Grzegorz Rozenberg, editors, *Proc. ICGT 2002: First International Conference on Graph Transformation*, volume 2505 of *LNCS*, pages 14–29, Barcelona, Spain, October 7–12 2002. Springer.
77. F. L. Dotti, L. Foss, L. Ribeiro, and O. M. Santos. Verification of object-based distributed systems. In *Proc. 6th International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 261–275, 2003.
78. Hans Schippers, Pieter Van Gorp, and Dirk Janssens. Leveraging UML profiles to generate plugins from visual model transformations. In *Proc. Int'l Workshop Software Evolution through Transformations (SETra)*, Electronic Notes in Theoretical Computer Science, 2004.