Electronic Notes in Theoretical Computer Science 72 No. 3 (2002)
URL: http://www.elsevier.nl/locate/entcs/volume72.html 14 pages

Towards Symbolic Analysis of Visual Modeling Languages

Dániel Varró [1,2]

Department of Measurement and Information Systems
Budapest University for Technology and Economics
H-1117, Magyar tudósok körútja 2, Budapest, Hungary

Abstract

Graph transformation has recently become more and more popular as a general, rule-based visual specification paradigm to formally capture the operational semantics of modeling languages based on metamodeling techniques as demonstrated by benchmark applications focusing on the formal treatment of the Unified Modeling Language (UML). In the paper, we enable model checking-based symbolic verification for such modeling languages by providing a meta-level transformation of well-formed model instances into SAL specifications [4]. We also discuss several optimizations in the translation process that makes our approach efficient and independent of the SAL framework.

Keywords: graph transformation, metamodeling, formal verification, model checking

1 Introduction

Nowadays, the Unified Modeling Language (UML) has become the dominating object-oriented modeling language for the design process of IT systems. However, despite its industrial success as being a unified and visual notation, the impreciseness of UML (i.e., the lack of formal semantics) is still the major factor that hinders the general use of UML as a primary source language for (i) automated tools of formal verification and validation exploiting the results in the theory of formal methods, and (ii) automated code generators that would yield a provenly correct functional core of target application.

The abstract syntax of the UML modeling language has been defined visually by *metamodeling* techniques. A straightforward representation of such models can

[1] This work was carried out during my visit to Computer Science Laboratory at SRI International (333 Ravenswood Ave., Menlo Park, CA, U.S.A.) and supported by the National Science Foundation Grant (CCR-00-86096), and Hungarian Scientific Grants FKFP 0193/1999, OTKA T038027, and IKTA 00065/2000.

[2] Email: `varro@mit.bme.hu`

rely on the use of directed, typed, and attributed graphs as the underlying semantic domain. In this sense, graph transformation [12] has recently become very popular as being a general, rule-based visual specification paradigm to formally capture the operational semantics of modeling languages based on metamodeling techniques [11, 15, 17]. Similar ideas are applied directly on (i) integrating different views of the UML-based system model [6], and (ii) formalizing transformations from UML into various semantic domains (Petri nets, SOS rules, dataflow nets, etc.) [9, 18].

However, due to a huge abstraction gap between visual metamodel-based and formal mathematical descriptions, the specification and implementation of graph transformation systems in a metamodeling (or UML) environment are highly prone to human errors, which necessitates an *automated verification* method for such systems.

The theoretical basics of verifying graph transformation systems by model checking techniques have already been studied thoroughly in, e.g., [7, 8] (and subsequent papers); however, the framework the authors propose does not directly give further suggestions on concrete implementation or tool support how to verify formal specifications given in the form of graph transformation systems by existing model checking or theorem proving techniques.

In the current paper, we propose a *meta-level translation of models of high-level modeling languages with abstract syntax defined by metamodeling techniques and operational semantics captured by graph transformation systems into transition systems* serving as inputs for various model checking tools to enable automated verification for them. Our approach is demonstrated on transforming visual specifications into the SAL intermediate language [4] to provide access to a combination of symbolic analysis techniques. We also propose several tool-independent optimizations for this encoding in order to avoid the state explosion of model checking tools.

2 Defining Modeling Languages

Initially, we informally summarize below the major concepts for defining modeling languages by a traditional combination of *metamodeling* and *graph transformation* techniques that will serve as the input for our encoding.

2.1 Models and metamodels

The *abstract syntax* of domain specific modeling languages is defined by a corresponding metamodel, which conforms to the best engineering practices in visual specification techniques. Typically, models (denoted by M in the sequel) and metamodels are represented internally as typed, attributed and directed graphs. For instance, considering UML class (object) diagrams as a graphical representation of metamodels (models), each class (object) can be represented as a node while each navigable association (link) end may be described by a directed edge together with the corresponding typing homomorphisms [5].

A sample metamodel and a simple model of finite automata are depicted in Figure 1.

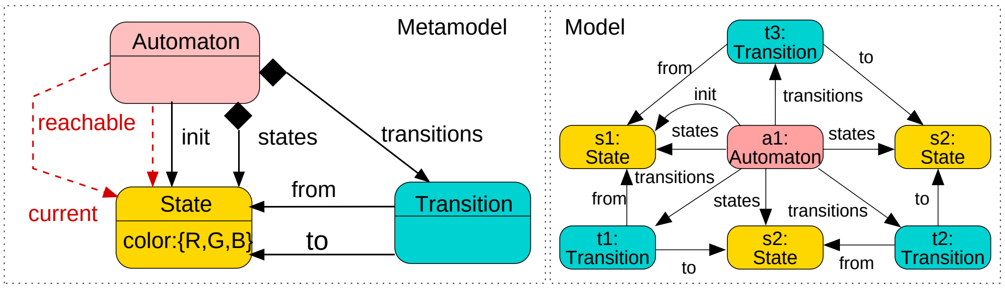


Fig. 1. A metamodel and model of finite automata

Example 2.1 According to the metamodel, a well–formed instance of a finite *automaton* is composed of *states* and *transitions*. A transition is leading between its *from* state and *to* state. The initial states of the automaton are marked with *init*, the active states are marked with *current*, while the reachable states starting from the initial states are modeled by *reachable* edges. Note that in the metamodel, we identified dynamic graph elements by dashed lines (by dynamic graph elements we mean objects that can be removed and added during the execution of models).

2.2 Transformation rules

The *dynamic operational semantics* of a modeling language is specified by graph transformation rules. A **graph transformation rule** is a 3-tuple $Rule = (Lhs, Neg, Rhs)$, where Lhs is the left-hand side graph, Rhs is the right-hand side graph, while Neg denote the (optional) negative application conditions.

The **application of a rule** to a model graph M (e.g., a UML model of the user) rewrites the user model by replacing the pattern defined by Lhs with the pattern of the Rhs. This is performed by

(i) *finding a match* of Lhs in M (graph pattern matching),

(ii) *checking the negative application conditions* Neg which prohibit the presence of certain nodes and edges

(iii) *removing* a part of the graph M that can be mapped to the Lhs but not the Rhs graph (yielding the context graph),

(iv) *gluing* Rhs and the context graph to obtain the derived model M'.

Example 2.2 A pair of rules describing how the reachability problem on finite automata can be formulated by graph rewriting rules is depicted in Figure 2. Rule *initR* states that all states of the automaton marked as initial are reachable (if the state has not been marked previously). Rule *reachR* expresses that if a reachable state S_1 of the automaton is connected by a transition T_1 to such a state S_2 that is not reachable yet then S_2 should also become reachable as a result of the rule application. Note that without the negative application condition (the crossed reachable edge in the left-hand side of the rule), the transformation would generate more than a single reachable edge between an automaton and a state, which contradicts our intuitive requirements.

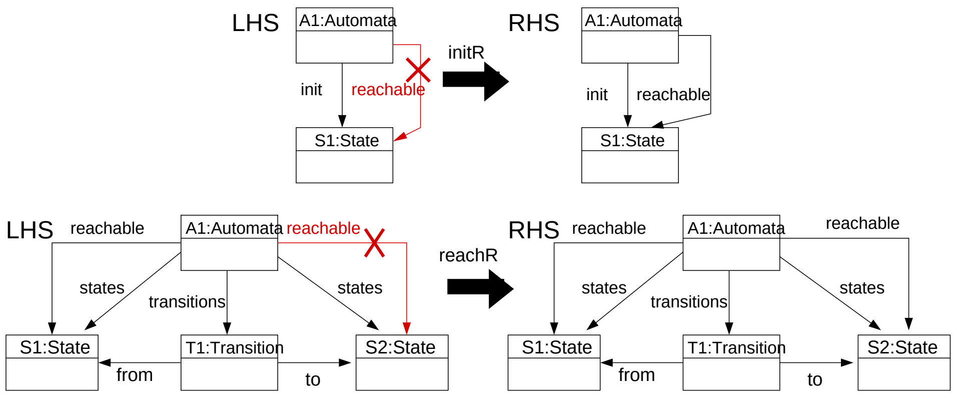


Fig. 2. Calculating reachable states by graph transformation

In many cases, modeling language specifications based on graph transformation systems also contain *control structures* to restrict rule application sequences. Due to space limitations, this extension is out of the scope of the current paper; the reader is referred to [14].

3 SAL: Symbolic Analysis Laboratory

The SAL (Symbolic Analysis Laboratory) [4] framework aims at combining different tools for abstraction, program analysis, theorem proving, and model checking towards the evaluation of system properties. The SAL architecture can be described as a "tool–bus" where a collection of tools interact *through* a common intermediate language of transition systems. The individual analyzers (theorem provers, model checkers, static analyzers) are driven from this intermediate layer and the analysis results are fed back to this intermediate level.

In the SAL intermediate language, the unit of specification is a context, which contains declaration of types, constants, transition system modules, and assertions. A SAL module is a *transition system* unit formalized mathematically as *Kripke structures*. A basic SAL module is a state transition system where the state consists of *input*, *output*, *local*, and *global* variables, which refer to different access modes.

A basic module also specifies the initialization and transition steps. These can be given by a combination of definitions or guarded commands. A definition is of the form $x = expression$ or $x' = expression$, where x' refers to the new value of variable x in a transition. A guarded command is of the form $g \longrightarrow S$, where g is a boolean guard and S is a list of definitions of the form $x' = expression$.

SAL modules can be composed (i) *synchronously*, so that $M_1||M_2$ is a module that takes M_1 and M_2 transitions in a lockstep, or (ii) *asynchronously*, when $M_1[]M_2$ is a module that takes an interleaving of M_1 and M_2 transitions.

In the paper, we will use the SAL specification language for describing graph transformation systems as traditional state transition systems despite the fact that the SAL framework is not yet available for public. However, as SAL is aimed to provide a general front-end to many individual model checkers, we can achieve a high level of independence from concrete tools in exchange. We believe that the language itself is self-explanatory and common to many other well-known formalisms (such as SMV [2], Murϕ [1], SPIN/Promela [10], etc.).

4 From Graph Transformation Systems to Transitions Systems

In the current section, we outline a meta-level approach how to transform graph transformation systems into transition systems (with formal semantics defined as Kripke structures) in order to verify properties of user models by model checking tools (we used the SAL model checker for our experiences).

In other words, we propose a method that inputs (i) the *metamodel* of a visual modeling language, (ii) its operational semantics in the form of a *graph transformation system*, (iii) a *requirement* expressed by a combination of graph patterns and temporal logic formulae, and (iv) a concrete, well-formed *model instance* of the language, and generates a *transition system* (with guarded commands) as the output.

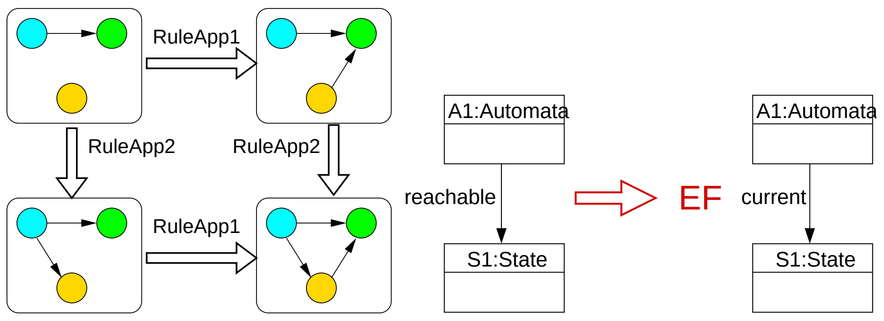
As a result, we do not reason about the language itself (as in the case of theorem provers); however, we can prove (automatically, with respect to the capabilities of model checkers) certain semantic correctness properties (like safety, liveness, etc.) for a well-formed concrete but arbitrary instance model of the language. It is essential to be pointed out that in practical cases, the user is only interested in the correctness of his or her model and not the correctness of the modeling language. Moreover, proving the correctness of a property for all valid model instances is often impossible.

System model

The state space of a graph transformation system is constituted from attributed graphs created by elementary graph transformation steps (see Fig. 3(a) for an overview).

This state space has a special structure: while the graph representation of a user model is typically finite (for instance, infinite UML models are somewhat rare), graph attributes may result in potentially infinite state representations (e.g., in case of integers or reals). As current model checking tools can only traverse state spaces induced by state variables of finite domain, variables having infinite domains should be abstracted to boolean domains before model checking by a technique called *predicate abstraction* [13].

Thereafter, graph models having (either original or abstracted) attributes of finite domain will form the *state space of the system*, and they will be encoded as predicates over node identifiers. Applying a graph transformation rule for a single



(a) GTS as Kripke structures (b) Specifying properties

$$\forall a_1, s_1 : (automaton(a_1) \wedge state(s_1) \wedge reachable(a_1, s_1) \rightarrow \mathrm{EF}\ current(a_1, s_1))$$

Fig. 3. Outline of our model checking approach

match is represented as a *transition in the transition system*[3].

The major problem in such an encoding relies in the fact that while graph transformation is a meta-level specification technique transitions in a transition system are defined on the instance level. As a consequence, the application of *a single graph transformation rule* is encoded into *several transitions in the Kripke model*; moreover, *the same graph transformation rule may yield different enabled transitions* (even during the same execution) when applied to different instance graphs.

Properties to be verified

Properties of transformations to be verified are predicates composed of graph patterns that prescribe or prohibit the presence of certain situations (e.g., an unreachable state cannot be initial). From these atomic predicates, arbitrarily complex expressions can be constructed using the traditional operators of temporal logic. A sample property stating that a *reachable* state in a finite automaton should eventually become *current* on at least one execution path is depicted in Fig 3(b).

Naturally, these graphical pattern-based temporal specifications have to be translated into traditional temporal logic formulae to serve as a well-formed input for the SAL framework.

4.1 Encoding of graph models as Kripke states

In the sequel, we identify and discuss the key issues in the transformation of meta-level graph transformation-based system specification techniques into (instance-level) transition systems that serve as the the input language for many existing model checking tools (including SAL [4], SPIN [10], and Murϕ [1]).

Informally, a transition system is composed of a set of state variables together with their initialization, and a set of guarded commands (if-then like statements).

[3] In order to avoid confusion caused by the overloading of the words, the terms “graph” and “transformation (step)” refer to system states and transitions on the graph transformation level while the terms “state” and “transition” will refer to the corresponding notions of transition systems (Kripke structure)

Definition 4.1 A **transition system** $TS = (V, T, Init)$ is a three tuple where (i) $V = \{v_1, ...v_k\}$ is the set of *state variables* (with finite or infinite domains); (ii) $T = \{\tau_1, ..., \tau_n\}$ is the set of *transitions (guarded commands)* which is of the form $guard \longrightarrow v'_1 := e_1, ..., v'_n := e_n$ (where the $guard$ is a boolean condition and an action $v'_1 := e_1$ specifies state variable updates) inducing a *transition relation* $act_\tau(V, V')$ defined as $guard \wedge \bigwedge_{i=1}^{k} v'_i = e_i$; while (iii) $Init$ is a predicate defining the *initial state*.

For our convenience, we suppose that state variables can be stored in state variable arrays ranging on the set of object identifiers, and they can be referred as $v_j[i]$. The formal semantics of transitions systems are defined as Kripke structures.

Definition 4.2 A **Kripke structure** $KS = (\Sigma, N, I, \sigma)$ is a four tuple where (i) Σ is the set of *states* (induced by all possible evaluations of state variables); (ii) $N \subseteq \Sigma \times \Sigma$ is the *transition relation* (defined as $N = \bigcup_{i=1}^{n} Act_{\tau_i}$); (iii) $I \subseteq \Sigma$ is the set of *initial states*; and (iv) $\sigma : \Sigma \to 2^{AP}$ is a *labeling function* relating each state to a subset of atomic propositions AP that are valid in the given state.

Step 1: Type declarations, state variables

The encoding of graph models into state variables is driven by the metamodel. We define

- a **one-dimensional boolean state variable array** (a unary relation symbol) *for each node type* (such as *State, Transition,* and *Automaton* in our running example)
- a **two-dimensional boolean state variable array** (a binary relation symbol) *for each edge type* (e.g., *from, to, reachable,* etc.).
- a **one-dimensional state variable array with enumeration type** *for each attribute type* (see the *color* attribute of states).

For model checking purposes, we must restrict the dimension of each array and all the enumeration types to be finite during type declaration. For the corresponding graph transformation system, these restrictions imply that (i) there exists an *a priori* upper bound for the number of nodes in the graph for each node type (ii) multiple edges of the same type between two nodes are forbidden, and (iii) attributes of infinite type have been abstracted into some representative finite domain (e.g., by predicate abstraction), which is either carried out by the user, or, preferably, by the model checking tool itself.

Step 2: Initialization predicate

Supposing that each node in a concrete model has a unique identifier, a unary relation (boolean state variable array) p holds at n_i (denoted as $p[n_i] = \top$), if there exists a node identified by n_i of type p. Similarly, a binary relation $r[n_i, n_j] = \top$ if there exist an edge of type r between nodes n_i and n_j. Otherwise, a relation is false by definition: $p[n_i] = \bot$.

Thereafter, *a state in the transition system representation of a graph transformation system* is defined by the *current evaluation of the predicates*. The initial graph instance is therefore encoded into the initial predicate, which is the conjunction of basic assignments for elementary predicates (relations).

SAL specification

In the corresponding SAL specification, we introduce (i) *types* for the *set of identifiers*, and (ii) *boolean variable arrays* for representing the relations of nodes and edges. The SAL specification also allows the use of state variables of arbitrary type, which allows for straightforward encoding of metamodel attributes, as predicate abstraction is performed within the tool-kit. The initial model is set up during the initialization statement by mapping from graph objects to identifiers.

For example, the SAL encoding of a finite automaton would include (at least) the following lines.

```
 AutID : TYPE = {a1};
 StateID : TYPE = {s1, s2, s3};
 ColorType : TYPE = {R, G, B};
 GLOBAL automaton : ARRAY AutID OF Boolean % Nodes
 GLOBAL reachable : ARRAY AutID OF ARRAY StateID OF Boolean % Edges
 GLOBAL color:       ARRAY StateID OF ColorType % Attributes
INITIALIZATION
 automaton[a1] = TRUE; reachable[a1][s1] = FALSE; color[s1] = "R";
```

State space optimization in SAL

Our experiments (when the current approach was applied for encoding and verifying UML statecharts expressed by model transition systems in [16]) have revealed that the previous encoding consumes an unacceptable amount of space when model checking real applications. For instance, the encoding of an automaton having 20 states and 20 transitions requires more than 500 boolean state variables, which is typically far too many to be handled by state-of-the-art model checking tools (resulting in a state space having 2^{500} states).

The problem originates in the fact that in the previous naive approach, state variables were introduced "verbosely" for the static parts of a model. Supposing that the structure of finite automata remains unchanged during the life-time of the model, we need to *introduce state variables only for dynamic elements* (such as *current* or *reachable* in the metamodel of finite automata), while *static parts* can be *represented by boolean functions* defined at compile-time. As a summary, the major difference between state variables arrays and functions relies in the fact that boolean functions are statically defined (i.e., their value does not change during the execution of the model).

As a consequence, the following SAL model provides a much more efficient state space representation for our finite automaton.

```
% SAL notation: function: f(x), state variable array: f[x]
 AutID : TYPE = {a1};
 StateID : TYPE = {s1, s2, s3};
 automaton(a: AutID) : Boolean =
    IF (a=a1) THEN TRUE ELSE FALSE
 states(a: AutID, s: StateID) : Boolean =
    IF (a=a1) AND
       (s=s1 OR s=s2 OR s=s3)
    THEN TRUE ELSE FALSE
 GLOBAL reachable : ARRAY AutID OF ARRAY StateID OF Boolean
INITIALIZATION
 reachable[a1][s1] = FALSE;
 reachable[a1][s2] = FALSE;
 reachable[a1][s3] = FALSE;
```

Step 3: State space optimization in transition systems

Unfortunately, model checkers, do not always support the definition of functions (for instance, SAL does support functions; in Murϕ, functions can be simulated with a hack; while SMV does not support functions). Therefore, for transition systems in general, the transformation process only generates state variables for nodes, edges and attributes that are dynamic in the metamodel. A metamodel element is considered to be *dynamic* if there exists at least one graph transformation rule of the model that prescribes to create, remove or update an instance of the metamodel element.

Naturally, as a rule may never be applied during an execution of a specific model, the transformation may still introduce state variables for metamodel elements that are never changed. However, as our translation from graph transformation systems to transition systems is defined on the metalevel, the only possibility to eliminate such a model instance specific overhead is the use of sophisticated static analysis techniques of graph transformation rules (e.g., [3]).

4.2 Encoding transformation steps as transitions

The main task in encoding transformation steps (potential applications of graph transformation rules) into transitions of transition systems is to simulate efficiently the graph pattern matching process in a low-level structure. As graph transformation is a meta-level specification technique, a single graph transformation rule will be encoded into several transitions. In fact, all potential occurrences of a pattern (application of a rule) have to be enumerated explicitly as different guarded commands.

Formally, we may define a transition function for each transformation rule as follows. Let us assign first a variable (not state variable!) for each node in the rule. Then the *guard of the transition* is constructed from the left-hand side and the negative application condition graphs (following the previous encoding of graph models), while the *state variable updates* are specified by the objects that are created or removed by the rule. For instance, rule *initR* of the running example is encoded as

$initR(A_1, S_1) = \textbf{if}\ automaton(A_1) \wedge state(S_1) \wedge init(A_1, S_1) \wedge$

$$\neg reachable(A_1, S_1)$$
$$\textbf{then } reachable'(A_1, S_1) := \top.$$

This example can be read as follows: in the next state, the $reachable$ relation should become true for all assignments for variables A_1 and S_1 whenever (according to the function defined at compile-time) the $automaton$ function holds at A_1, $state$ holds at S_1 and $init$ holds at A_1, S_1.

Step 4: Guarded commands

Unfortunately, such a transition function requires a sophisticated unification algorithm (for correctly instantiating variables with identifiers), which is not supported by existing model checking tools. Therefore, we have to generate (at compile-time) all the potential transitions corresponding to a rule application on a specific match. The number of potential transitions (according to a naive first estimation) are determined by the *complexity of the LHS* of a rule (i.e., the number of nodes), and *the size of the model* (i.e., the cardinality of the sets of identifiers). In case of rule *initR*, the corresponding SAL specification in a naive encoding is as follows.

```
TRANSITION % guarded commands for initR
% first potential match
 automaton(a1) AND init(a1,s1) AND state(s1) AND
 NOT (reachable[a1][s1]) --> % guard
    reachable'[a1][s1] = TRUE;  % assignment
[] % asynchronous composition
% second potential match
 automaton(a1) AND init(a1,s2) AND state(s2) AND
 NOT (reachable[a1][s2]) -->
    reachable'[a1][s2] = TRUE;
[]
% third potential match
 automaton(a1) AND init(a1,s3) AND state(s3) AND
 NOT (reachable[a1][s3]) -->
    reachable'[a1][s3] = TRUE;
```

It is obvious that such a naive approach generates redundant transitions with guards that can never be satisfied. However, each one is investigated and tested at each step over and over again causing an unacceptable decline in performance. For instance, the finite automaton model in our running example (Fig. 1) has a single initial state s_1, thus the second and third potential matches are superfluous as the guards are constantly false.

Optimizations in guarded commands

To reduce such an overhead, we eliminate guard conditions that can never be satisfied by further preprocessing on the graph transformation level.

(i) The dynamic and static parts of a graph transformation rule are identified and separated.

(ii) A graph pattern matching step is executed to find all the occurrences defined by the *static* parts of the LHS of the rule. The aim of this step is to limit the potential matches to those that certainly satisfy the static parts of the guard.

(iii) 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 eliminated from (better to say, not generated in) the transition system. In fact, this step prevents the creation of the second and third transitions in the previous example.

(iv) All conditions appearing in guards that refer to static parts of the model are eliminated afterwards (since they are guaranteed to be fulfilled at compile-time).

(v) The resulting transition system has guarded commands using only state variables as conditions (and assignments) without the previously introduced functions. Note that this approach can be applied to generate a more optimal set of guarded commands for transitions systems in general (thus, it is totally independent of SAL as target language).

This preprocessing step would yield the following SAL specification for our sample automaton model (in Fig. 1).

```
AutID : TYPE = a1;
StateID : TYPE = s1, s2, s3;
fa1 : MODULE =
BEGIN
  GLOBAL reachable: ARRAY AutID OF ARRAY StateID OF BOOLEAN
INITIALIZATION % reachable is equal to FALSE
  reachable[a1][s1] = FALSE; reachable[a1][s2] = FALSE; ...
TRANSITION % guarded commands for initR and reachableR
  NOT reachable[a1][s1]  --> reachable’[a1][s1] = TRUE; [] % s1 is init
  reachable[a1][s1] AND NOT reachable [a1][s2] -->
    reachable’[a1][s2] = TRUE; []  % s1 -> s2
  reachable[a1][s1] AND NOT reachable [a1][s3] -->
    reachable’[a1][s3] = TRUE;  [] % s1 -> s3
  reachable[a1][s2] AND NOT reachable [a1][s3] -->
    reachable’[a1][s3] = TRUE; % s2 -> s3
END;
```

Finally, the entire (meta)encoding of typed and attributed graph transformation systems into SAL specifications is summarized in Table 1.

Graph transformation systems	transition systems
static parts of the model	—
dynamic parts of the model	state variables
potential rule applications	guarded commands

Table 1
A summary of encoding graph transformation systems into SAL

5 Conclusions

In the paper, we presented an automatic transformation of modeling languages defined by metamodeling techniques (static structure) and graph transformation rules (dynamic behavior) into the SAL framework. As a result, we are able to investigate certain *semantic properties of any specific well-formed model instance* of the language by various model checking-based symbolic analysis techniques provided by the SAL environment. After several optimizations, the presented approach has become independent of the SAL language, thus it can be directly applied to any model checking tool with a specification language based on guarded commands.

We are currently building a tool that is capable of automatically translating models of arbitrary visual modeling languages (defined by metamodeling and graph transformation) into the corresponding SAL specifications. However, prior to that, we carried out (with partially automated translations) several benchmark experiments of our approach in different domains.

For an industrial strength case study, we formalized the semantics of UML Statecharts by means of model transition systems (see [16] for the specification of UML statecharts by metamodeling and graph transformation techniques), which was encoded afterwards as a SAL specification. Afterwards, we carried out simple verification tasks by the SAL model checker. In fact, the need for the optimizations described in Sec. 4 were triggered by unsuccessful preliminary verification attempts. The automatic transformation into SAL specifications (for this specific modeling language, namely, UML) was carried out within the VIATRA environment [18].

In another case study [14], we captured the operational semantics of Petri nets by graph transformation systems and translated them into the specification language of the Murϕ model checker. In case of bounded Petri nets (where the number of tokens in Petri nets has an *a priori* upper bound), our approach was directly applicable. We also exploited the use of predicate abstraction by abstracting away from the concrete number of tokens, which resulted in a semi-decision procedure for proving liveness and safety properties of Petri nets.

Acknowledgement

I would like to thank András Pataricza (Budapest University of Technology), John Rushby and many of his colleagues (at SRI International) for their encouragement and support. I also very much appreciate the valuable comments of the anonymous reviewers of the paper.

References

[1] "The Murϕ Model Checker," .
URL `http://verify.stanford.edu/dill/murphi.html`

[2] "The SMV Model Checker," .
URL `http://www-2.cs.cmu.edu/~modelcheck/smv.html`

[3] Baldan, P., A. Corradini and B. König, *A static analysis technique for graph transformation systems*, in: K. G. Larsen and M. Nielsen, editors, *CONCUR 2001 - Concurrency Theory, 12th International Conference*, LNCS **2154** (2001), pp. 381–395.

[4] Bensalem, S., V. Ganesh, Y. Lakhnech, C. Munoz, S. Owre, H. Rueß, J. Rushby, V. Rusu, H. Saïdi, N. Shankar, E. Singerman and A. Tiwari, *An overview of SAL*, in: C. M. Holloway, editor, *LFM 2000: Fifth NASA Langley Formal Methods Workshop*, 2000, pp. 187–196.

[5] Corradini, A., U. Montanari and F. Rossi, *Graph processes*, Fundamenta Informaticae **26** (1996), pp. 241–265.

[6] Engels, G., R. Heckel and J. M. Küster, *Rule-based specification of behavioral consistency based on the UML meta-model*, in: M. Gogolla and C. Kobryn, editors, *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, LNCS **2185** (2001), pp. 272–286.

[7] Heckel, R., *Compositional verification of reactive systems specified by graph transformation*, in: *Proc. FASE: Fundamental Approaches to Software Engineering*, LNCS **1382** (1998), pp. 138–153.

[8] Heckel, R., 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*, LNCS **1295** (1997), pp. 219–228.

[9] Heckel, R., J. Küster and G. Taentzer, *Towards automatic translation of UML models into semantic domains*, in: *Proc. AGT 2002: Workshop on Applied Graph Transformation*, Grenoble, France, 2002, pp. 11–21.

[10] Holzmann, G., *The model checker SPIN*, IEEE Transactions on Software Engineering **23** (1997), pp. 279–295.

[11] Juan de Lara, H., Vangheluwe, *Atom3: A tool for multi-formalism and meta-modelling*, in: R.-D. Kutsche and H. Weber, editors, *5th International Conference, FASE 2002: Fundamental Approaches to Software Engineering, Grenoble, France, April 8-12, 2002, Proceedings*, LNCS **2306** (2002), pp. 174–188.

[12] Rozenberg, G., editor, "Handbook of Graph Grammars and Computing by Graph Transformations: Foundations," World Scientific, 1997.

[13] Saïdi, H., *Model checking guided abstraction and analysis*, in: J. Palsberg, editor, *Seventh International Static Analysis Symposium (SAS'00)*, LNCS **1824** (2000), pp. 377–339.
URL `http://www.sdl.sri.com/papers/saidi_sas00/`

[14] Salamon, G., "Formal Verification of Model Transformation Systems," Master's thesis, Budapest University of Technology and Economics (2002).

[15] Sprinkle, J. and G. Karsai, *Defining a basis for metamodel driven model migration*, in: *Proceedings of 9th Annual IEEE Internation Conference and Workshop on the Engineering of Computer-Based Systems, Lund, Sweden*, 2002.

[16] Varró, D., *A formal semantics of UML Statecharts by model transition systems*, in: A. Corradini, H. Ehrig, H.-J. Kreowski and G. Rozenberg, editors, *Proc. ICGT 2002: 1st International Conference on Graph Transformation*, LNCS **2505** (2002), pp. 378–392.

[17] Varró, D. and A. Pataricza, *Metamodeling mathematics: A precise and visual framework for describing semantics domains of UML models*, in: J.-M. Jézéquel, H. Hussmann and S. Cook, editors, *Proc. Fifth International Conference on the Unified Modeling Language – The Language and its Applications*, LNCS **2460** (2002), pp. 18–33.

[18] Varró, D., G. Varró and A. Pataricza, *Designing the automatic transformation of visual languages*, Science of Computer Programming **44** (2002), pp. 205–227.