TOWARDS FORMAL VERIFICATION OF MODEL TRANSFORMATIONS[*]

Dániel Varró
Dept. of Measurement and Information Systems
Budapest University of Technology and Economics
H – 1117, Budapest, Magyar tudósok körútja 2., Hungary
varro@mit.bme.hu

Abstract As the Unified Modeling Language is evolving into a family of languages with individually specified semantics, there is an increasing need for highly automated and provenly correct model transformations that would assure the integration of local views of the system (in the form of different diagrams) into a consistent global view. Graph transformation provides an easy-to-understand visual specification technique to formally capture the rules of such transformations. In the paper, we summarize the concepts of VIATRA, the general purpose model transformation system together with the major correctness requirements and a model checking based verification method for model transformations.

Keywords: model transformation, model transition systems, graph transformation, UML, formal verification,

1. Introduction

Nowadays, the Unified Modeling Language (UML) has become the dominating specification and modeling language for the design process of software. 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.

Recent initiatives (UML 2.0 Request for Proposal) of the Object Management Group (OMG) aim at to re-architecture the single and monolith language

[*]This work was supported by the Hungarian National Scientific Foundation Grant (OTKA 030804).

into a family of languages (built around a kernel metamodeling language) each having its own well-defined semantics.

Transformations of UML Models. However, as the formal semantics of different views of the system (i.e., separate diagrams like class diagrams, statecharts, sequence diagrams, etc.) might be defined in different semantic domain (e.g., by Petri nets, SOS rules, graph transformation systems etc.), the integration of such local views into a consistent global view of the system requires a precise specification and verification of transformations *within* and *between* UML models.

In practice, transformations are necessitated for several purposes: (i) *model transformations within a language* should control the correctness of consecutive model refinement steps, (ii) *model transformations between different languages* should provide precise means to project the semantic content of a diagram into another one, which is indispensable for a consistent global view of the system under design, and (iii) a visual UML model (i.e., a sentence in the UML language family) should be transformed into its semantic domain (called *model interpretation*).

As the abstract syntax of UML models is defined visually by a corresponding metamodel. A straightforward representation of such models can rely on the use of directed, typed, and attributed graphs as the underlying semantic domain. Therefore, the use of graph transformation [7] for capturing the semantics of UML models and their transformations is a natural choice which also fits well to engineering practices as a consequence of its visual expressiveness [3, 4, 9, 11].

However, due to a huge abstraction gap between visual UML models and formal mathematical descriptions, the specification and implementation of model transformation systems are highly prone to human errors, which necessitates a highly automated verification and program generation method for such systems.

The VIATRA Environment. VIATRA (VIsual Automated model TRAnsformations [11], see Figure 1) is a prototype tool being developed at the Budapest University of Technology and Economics, that provides a general means to specify, implement and verify various model transformations (tailored especially to UML-based transformations).

1 As the beginning of a typical interaction with VIATRA, the transformation designer constructs the metamodels of the source language (typically UML itself) and the target languages (formal models such as Petri Nets, Kripke automaton, etc.) in the form of UML class diagrams exported into the standard XMI format.

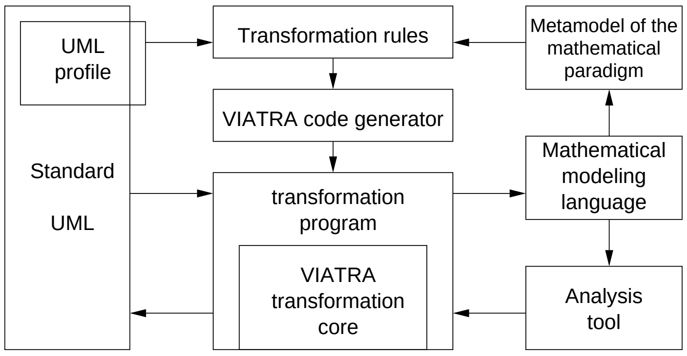

Figure 1. The VIATRA environment

2 Afterwards, the transformation rules are specified in a visual notation based on a UML profile tailored to graph transformation systems.

3 From this visual description, the model transformation program is generated automatically in the form of a Prolog program (see [10] for further details on automated program generation in VIATRA).

4 This Prolog program takes a source UML model as input and generates the target mathematical model as the output (since all models are represented in VIATRA using the XMI standard the underlying Prolog engine is hidden from the user).

5 The results of the mathematical analysis can be back–annotated into the source language by additional model transformations.

The main contribution of the VIATRA approach is its flexibility and preciseness for automated model transformations aiming at to investigate and formally verify UML models from different aspects (such as logical correctness, dependability and performability analysis). However, such a transformation based automatic model generation approach might not assure a higher quality of system verification, unless the faithfulness and consistency of UML models and the abstract mathematical structures of formal analysis can be guaranteed (preferably, formally proven). In the current paper, after a brief summary of concepts we discuss the major correctness requirements and a model checking based verification approach of model transformation systems.

2. The Concepts of Model Transformation

We informally summarize below the major model transformation concepts of VIATRA. Models (and metamodels, which describe the abstract syntax of models) are represented as typed, attributed and directed graphs (denoted M in the sequel). The *static semantics* of models are defined by visual and graph pattern-based well–formedness constraints, an alternate approach to the traditional solutions using the Object Constraint Language.

Transformation rules. The *dynamic semantics* of a model as well as the *transformations between different models* are specified operationally 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 is (an optional) negative application condition.

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

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

2 *checking the negative application conditions* Neg which prohibits the presence of certain nodes and edges

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

4 *gluing* Rhs and the context graph to obtain the derived model M'.

A pair of rules describing how the reachability problem on finite automatons can be formulated by graph rewriting rules is depicted in Figure 2. Rule *initR* states that all a state of the automaton marked as initial is reachable. Rule *reachR* says 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 intuitional requirements.

Model transition system. The entire operational semantics of a VIATRA model or its transformation is defined by a **model transition system**, where the allowed transformation sequences are constrained by **control flow graph** (CFG) applying a transformation rule in a specific **rule application mode** at each node. As the majority of rules perform local modifications on models they can often be executed parallelly for all matches (in **forall** mode). Alternatively, a rule in **try** mode is applied on a (non-deterministically chosen) single matching (if it is possible and fails otherwise), or, it is executed as long as it is applicable (**loop** mode). Note that the specification of transformations can still contain non-determinism when a rule is applied in a *try* or *loop* mode.

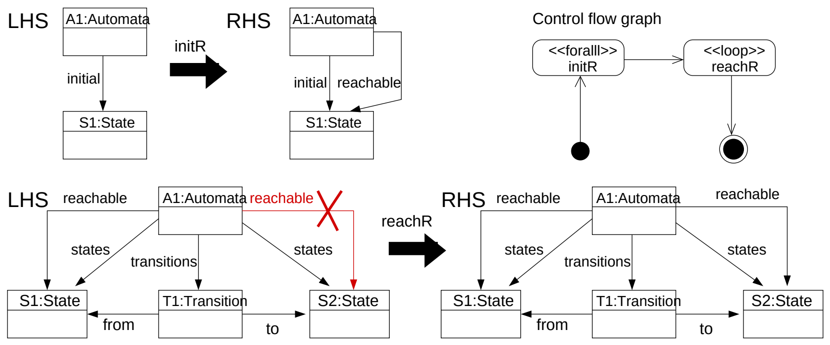

Figure 2. Calculating reachable states by graph transformation

As for the model transition system of the reachability problem (the control flow graph is depicted in the upper right corner of Figure 2), a possible solution is to apply rule *initR* in *forall* mode – thus parallelly for all matchings (as all the initial states can be marked as reachable independently from each other), and then to execute rule *reachR* as long as possible (therefore in *loop* mode) since we need to calculate the transitive closure of the *reachable* relation, which cannot be performed in a single *forall* application.

Our running example demonstrated the use of model transition systems to describe the behavior of transformations *within* models. In case of transformations *between* models (called *model transformation* in the following), the transformation rules are structured specially. A **model transformation rule** is operating on a so-called **reference graph** which interrelates the objects of the source and target language to form a single graph by reference nodes and edges.

3. Correctness Criteria of Transformations

Model transition systems provide a precise means to specify transformations in various domains. An elementary requirement for such transformations is to assure *syntactic correctness*, i.e., to guarantee that the generated model is syntactically well–formed. However, in order to assure a higher quality of model transformations, at least the following *semantic requirements* are aimed to be verified in VIATRA.

- **Correctness:** 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 define correctness requirements, which are typically transformation specific. For instance, in our running example one may prescribe the natural criteria for correctness that each state that is reachable from one of the initial states has to be marked by a reachable edge but no other state is allowed to be marked.

- **Termination:** We must also guarantee that the transformation will terminate. A non-terminating transformation can be caused by a cycle in the control flow graph with unsatisfiable termination condition or an erroneous use of the *loop* mode for rules.

- **Completeness:** In the case of model transformations (i.e., transformations *between* models), an additional requirement is to completely cover the source language by transformation rules, i.e., to prove that there exist a corresponding element in the target model for each construct in the source language. For proving completeness, we have to show that such elements exist when the transformation terminates.

- **Uniqueness:** As non-determinism is used in the specification of model transition systems (selecting an appropriate matching for rule applications) we must also prove that the transformation yields a unique result. For this reason, we have to show that whenever a transformation terminates, this final state is unique, which can be interpreted as a forward reachability problem.

4. Verifying Transformations by Model Checking

In the current section, we outline how to transform model transition systems into Kripke structures in order to verify properties of model transformations by traditional model checking tools.

- **System model:** The state space of a model transformation is constituted from graphs created by elementary graph transformation steps. This state space is typically finite, which makes *model checking* techniques eligible for model transition systems *to verify the properties of a single transformation starting from a single model.* For this reason, graph models (as *system states*) will be encoded as predicates over node and edge identifiers (as domains). Applying a rule for a single match is represented as a *single transition* (a micro step) in the Kripke structure, while the *macro step* semantics (of rule application modes) assigned to a *node* in the CFG is composed of these basic transitions in order to reduce the state space to be traversed.

- **Properties to be verified:** Properties of transformations to be verified are predicates composed of (i) statements that the control in the CFG is at a specific node (e.g., the *exit* node is reached in the CFG thus the transformation terminates), and (ii) 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 linear temporal logic.

System states. For the encoding of graph models, let us define a unary relation symbol for each node type (such as *State, Transition,* and *Automaton* in our running example) and binary relation symbol for each edge type (e.g., *from, to, reachable,* etc.). Supposing that each node in a concrete model has a unique identifier, a unary relation p holds at n_i (denoted as $p(n_i) = \top$), if the node identified by n_i is 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.

Thereafter, a state in the Kripke structure is defined by the current evaluation of the predicates. The states will be changed by applying graph transformation rules as transitions.

Elementary transitions. We define a transition function (a guarded command) for each transformation rule as follows. Let us assign first a variable for each node in the rule. Then the *guard of the transition* is constructed from the LHS and the negative application condition graphs (following the encoding rules of the previous paragraph), while the *state variable updates* are specified by the objects appearing only on the RHS of the rule.

For instance, rule *initR* of the running example is encoded as

$$\begin{aligned} initR(A_1, S_1) \quad = \quad & \textbf{if}\ automaton(A_1) \wedge state(S_1) \wedge initial(A_1, S_1) \\ & \textbf{then}\ reachable(A_1, S_1)' := \top. \end{aligned}$$

Rule application modes. When a rule is applied in a certain mode, the elementary transitions are modified by quantification (existential quantification for *try* mode and universal quantification for *forall* model), or by fixpoint operators (*loop* is the least fixpoint of the basic transition function) in order to obtain the proper semantics of the macro step. As (up to our knowledge) no model checking tools directly support these operations, they have to be modeled manually in the Kripke automata of our model transition system by introducing additional state variables to serving as a program counter for the control flow graph. Although the properties of model transformation system could be verified directly on the elementary transition (micro step) level, the use of rule application modes can drastically decreased the state space to be traversed during verification.

As a result of the previous encoding of model transition systems into Kripke structures, traditional model checking tools are available for the verification of correctness properties.

5. Conclusions

In this paper, we outlined how to provide model checking facilities for model transition systems. Model transition systems play a major role in the precise formalization of automated transformations from UML-based systems models into different semantic domains (such as Petri nets, dataflow networks, Kripke automaton) in order to provide a variety of quantitative and qualitative analysis techniques.

In order to increase the faithfulness of transformations, not only the UML models of the designer but the model transformation itself needs to be verified, otherwise we cannot guarantee that mathematical models used for formal analysis represent the intended semantic content of the UML design. For this reason, we should verify that automated model transformations are correct, terminating, complete, and unique. For a specific source model, we used model checking techniques to answer these questions.

5.1. Practical Applications of VIATRA

VIATRA has already been applied successfully to provide an automated implementation of transformations specified by means of model transformation systems. An industrial strength transformation is the automated implementation of [5] where a mapping is presented from UML Statecharts to Extended Hierarchical Automata (EHA) that provides formal operational semantics for statecharts. We re-formalized the formal operational EHA semantics by means of model transition systems, which was encoded afterwards as a SAL (Symbolic Analysis Laboratory [1]) specification, and verified by the SAL model checker.

Additional applications of VIATRA in joint industrial projects include the verification of completeness of UML statechart specifications in a dependable environment [6], and a transformation from UML statecharts to Stochastic Reward Nets [2] is also under implementation to provide access to Petri Net based analysis techniques.

5.2. Future work

Our experiments demonstrated that the model checking based verification process of industrial strength UML statechart models formalized as model transition systems must also face the state space explosion problem. Without exploiting the compositionality and symmetry of statecharts (for instance, objects of the same class have identical behavior), UML models of practical size and relevance cannot be verified by “brute force” model checking. An alternate way to attack state space explosion is to introduce (a controlled level of) imprecision by abstraction. For instance, property preserving abstractions are powerful techniques exploiting and combining the advantages model checking and theorem proving [8].

Further semantic questions concern the use of metamodeling techniques for mathematical models. On one hand, such *visual metamodel definitions significantly increase the "legibility of mathematics" for engineers*. However, on the other hand, metamodels definitions might introduce undesirable imprecision into mathematical models. An ongoing research activity is to *metamodel mathematics*, i.e., to provide a precise way to represent mathematical models in UML based metamodeling framework.

Acknowledgments

I would like to thank my supervisor András Pataricza (Budapest University of Technology), John Rushby and many of his colleagues (at SRI International) their encouragement and support.

References

[1] S. Bensalem, 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*, pages 187–196, 2000.

[2] M. Dal Cin, G. Huszerl, and K. Kosmidis. Evaluation of safety-critical systems based on guarded statecharts. In R. Paul and C. Meadows, editors, *Proc. of the 4th IEEE International Symposium on High Assurance Systems Engineering*, 1999.

[3] G. Engels, 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*, volume 2185 of *LNCS*, pages 272–286. Springer, 2001.

[4] S. Kuske. A formal semantics of UML state machines based on structured graph transformation. In M. Gogolla and C. Kobryn, editors, *UML 2001: The Unified Modeling Language. Modeling Languages, Concepts and Tools*, volume 2185 of *LNCS*, pages 241–256. Springer, 2001.

[5] D. Latella, I. Majzik, and M. Massink. Automatic verification of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637–664, 1999.

[6] Z. Pap, I. Majzik, and A. Pataricza. Checking general safety criteria on UML statecharts. In U. Voges, editor, *Computer Safety, Reliability and Security (Proc. 20th Int. Conference, SAFECOMP-2001)*, volume 2187 of *LNCS*, pages 46–55. Springer Verlag, 2001.

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

[8] H. Saïdi and N. Shankar. Abstract and model check while you prove. In N. Halbwachs and D. Peled, editors, *Computer-Aided Verification (CAV'99)*, number 1633 in Lecture Notes in Computer Science, pages 443–454, Trento, Italy, jul 1999. Springer-Verlag.

[9] J. Sprinkle 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*, April 2002.

[10] D. Varró and A. Pataricza. Mathematical model transformations for system verification. Technical report, Budapest University of Technology and Economics, May 2001.

[11] D. Varró, G. Varró, and A. Pataricza. Designing the automatic transformation of visual languages. *Science of Computer Programming*. In print.