VIATRA - Visual Automated Transformations for Formal Verification and Validation of UML Models

György Csertán Gábor Huszerl István Majzik Zsigmond Pap András Pataricza
Dániel Varró
Budapest University of Technology and Economics
Department of Measurement and Information Systems
{varro,pataric}@mit.bme.hu

Abstract

The VIATRA (VIsual Automated model TRAnsformations) framework is the core of a transformation-based verification and validation environment for improving the quality of systems designed using the Unified Modeling Language by automatically checking consistency, completeness, and dependability requirements. In the current paper, we present an overview of (i) the major design goals and decisions, (ii) the underlying formal methodology based on metamodeling and graph transformation (iii) the software architecture based upon the XMI standard, (iv) and several benchmark applications of the VIATRA framework.

1. Introduction

The advent of visual design languages promises not only a better requirement capture and easier software architecture process, but a radical increase in software productivity, as well. The rapid spread of UML, the Unified Modeling Language [9], as the dominant object-oriented CASE technology clearly indicates the market need for effective visual design technologies.

However, the use of visual CASE methodologies does assure neither the correctness of the design, nor the dependability of the target application. The designer can still construct syntactically correct but semantically incorrect models. The assurance of the qualitative correctness of a design necessitates the checking of (i) the *completeness and consistency* of the system specification, (ii) *global correctness attributes*, like the deadlock freedom of the design, and (iii) *application-specific requirements*, like safety requirements. The dependability of the target application necessitates the fulfillment of several quantitative requirements, as well. For instance, (i) *timeliness* of the application is one of the major criteria in real-time system design, or (ii) *reliability and availability measures* are crucial in the system design phase.

During the last decades computer science has successfully attacked the majority of these problems by providing mathematic methods and tools for the modeling and analysis of dependability attributes. However, these methods are not widely used in the industry primarily due to the high level of abstractness of the mathematical modeling and analysis techniques.

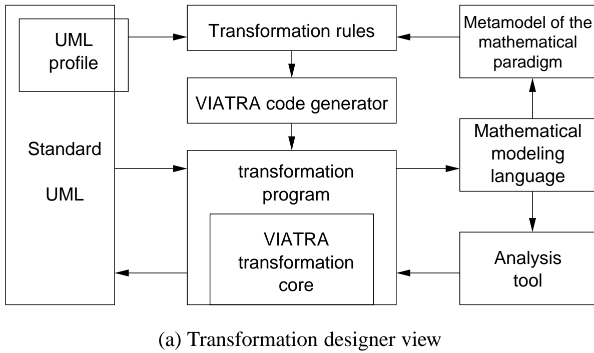
A former ESPRIT project under the acronym HIDE (carried out together with FAU Erlangen, CNUCE Pisa and two industrial partners [2]) has shown the feasibility of an automated, multi-aspect dependability evaluation of UML designs. In HIDE, the UML model of the target design was enriched by dependability requirements and local dependability attributes associated to the individual components. The mathematical models (like timed Petri-nets for the quantitative evaluation of dependability) were derived from this model automatically by custom-built model transformations. The results of the mathematical analysis were back-annotated to the UML model for presentation to the designer.

2. Design guidelines

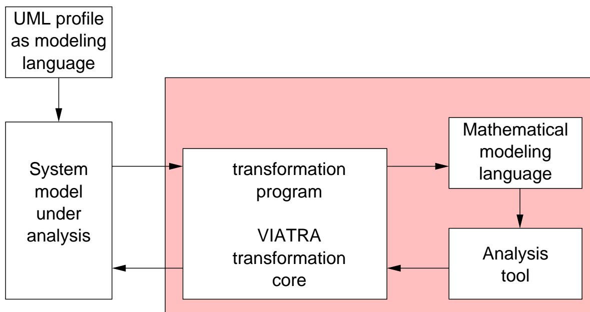
An important experience gained from HIDE was that an ad-hoc implementation of transformations lacks the necessary preciseness, thus implementation errors can make the transformations be the weakest point in the chain of tools serving for dependability evaluation. This way a mathematically precise paradigm was searched for leading to a general-purpose framework for the definition and implementation of transformations. Another argument towards the use of such a framework was the need for a high-degree of flexibility due to the changing and extensible UML standard, the problems revealed in UML and the implementation-related aspects of the target application to be included into the transformation.

As it turned out, an open and flexible transformation based V & V framework necessitates (i) a general purpose, mathematically solid paradigm and a user friendly methodology for the definition of UML notations (dialects) and transformations towards a variety of mathematical analysis tools; (ii) an efficient and mechanized methodology to derive model transformation and back-annotation programs from these definitions.

Our new framework (called VIATRA: VIsual Automated model TRAnsformations; depicted in Fig. 1) for UML-based system verification has the following main attributes:



(a) Transformation designer view



(b) User view

Figure 1. The architecture of VIATRA

- Both the UML dialect to be used by the modeler and the input notation of the target mathematical analysis tool are defined by their respective *metamodels*. This offers flexibility.
- Transformations can be defined in the form of a set of simple transformation rules correlating individual UML notational elements with the target mathematical notation. These transformation rules themselves can be designed visually in UML.
- The transformers are automatically derived from the rules by using the mathematically well-defined and widely used principle of graph transformations [8].
- A back-annotation engine (based directly upon the automatically generated transformer) provides the user with the analysis results integrated into the original UML model. The entire transformation framework is hidden from the end user.

3. Modeling concepts in VIATRA

A model transformation-based verification approach for UML models requires the precise definition of models from various application domains, which are specified uniformly by visual metamodeling techniques in VIATRA. A precise metamodeling method includes the formal definition of the abstract syntax, the static and dynamic semantics of a language.

In VIATRA, the *static syntax* of a modeling language is specified in the form of UML class diagrams (following basically the concepts of MOF metamodeling [6]) and formalized by typed, attributed and directed graphs. *Metamodels* are interpreted as type graphs, and *models* are valid instances of their type graphs [11]. Our experiments showed that mathematical notations described by a corresponding metamodel are much more expressive for engineers than pure mathematical formulae.

A sample metamodel of finite automata (taken from a mathematical domain) is depicted in Fig. 2.

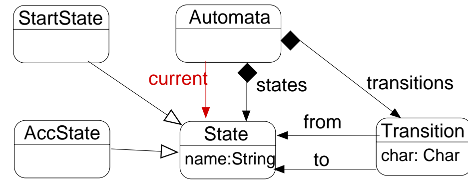


Figure 2. The metamodel of finite automaton

VIATRA uses a declarative and pattern-based description technique for *static semantics* where the majority of well–formedness constraints is provided by graph patterns that preserve visuality.

Graph patterns also play a major role in the definition of the *dynamic semantics* of a model as the evolution of a model is described by graph transformation rules [8]. A *graph transformation rule* is a 3-tuple (LHS, N, RHS), where LHS is the left-hand side graph, RHS is the right-hand side graph, while N is (an optional) negative application condition graph. The application of the rule to a model graph (UML model of the user) rewrites the user model by replacing the pattern defined by LHS with the pattern of the RHS. A sample graph transformation rule is depicted in Figure 3. This rule describes the dynamic operational semantics of finite automata (i.e., how a transition can be fired).

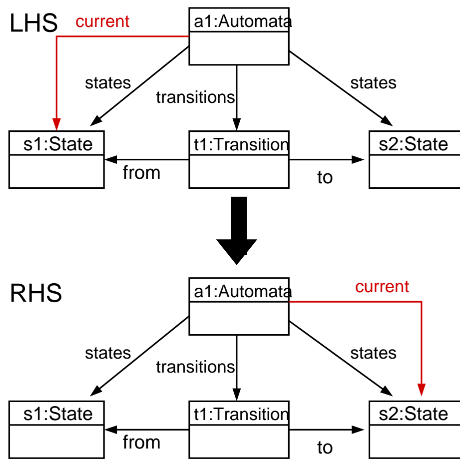


Figure 3. A sample transformation rule

Transformations *within* and *between* models are uniformly specified by corresponding graph transformation rules thus providing an easy-to-understand visual way for semantic definitions.

As the main goal of model transformation is to derive a target model from a given source model, source and target objects are linked together to form a single graph. For this reason, the concepts of *reference graphs* are introduced. The structure of a reference graph is also constrained by a corresponding reference metamodel, which contains (i) references of existing source and target metamodel nodes; (ii) novel reference nodes that provide a typed coupling of source and target objects, and (iii) reference edges interrelating nodes. Reference graphs also provide the primary basis for the back–annotation of analysis results.

The entire operational semantics of a VIATRA model or its transformation is defined by a *model transition system*, where the next graph transformation rule to be applied in a specific mode is constrained by a *control flow graph*. As the majority of rules perform local modifications on the models they can be executed in parallel for all occurrences (*forall* mode). Alternatively, a rule is applied on a (non-deterministically chosen) single matching (*try* mode), or, it is applied as long as possible (*loop* mode).

4. A technological overview of VIATRA

The general technological concept of the VIATRA framework is the use of the XMI standard for arbitrary MOF-based metamodel (simultaneously including UML and mathematical models like Petri nets, dataflow networks, hierarchical automata, etc.) in order to obtain an open, tool-independent architecture. A typical scenario of model transformations in a UML environment is as follows:

Design phase. The interaction with VIATRA commences with a design phase performed within a traditional UML CASE tool that has XMI export facilities for MOF. We create the MOF metamodels of the source and target modeling language. Afterwards, we relate the source and target objects to each other by constructing the reference metamodel. Then we export our metamodels in an XMI format that conforms to the MOF Model [6]. These files serve as primary inputs for VIATRA. For the next step, the transformation rules and the control structures are created by a special UML profile tailored to graph transformation systems, and exported into the UML XMI format.

Automated program generation. In the next phase, a Prolog implementation of the transformation program is generated automatically. This automated program generation method itself, which is the semantic core of the VIATRA framework, was also designed and implemented by consecutive model transformations in a reflective way [10].

Automated transformations. Finally, the previously generated transformation programs can be applied to the transformation of various source models. As for the typical case, the UML model created by a software engineer will serve as the input of a transformation thus it is exported into an XMI format. The outputs of the transformation, i.e. the reference and the target models, are also exported in an XMI format. The concrete input language of a specific analysis tool can typically be generated from this format by either (i) simple Prolog programs, (ii) XSLT transformations, or (iii) Java programs.

Verification of transformations. As an ongoing activity, we also aim at the formal verification of model transformations in order to provide a higher level of quality and faithfulness for such transformations. Syntactic correctness and completeness can be verified by planner algorithms [11]. Semantic correctness of transformations is being verified by projecting model transformation rules into the SAL intermediate language [1], which provides access to an automated combination of symbolic analysis tools (like model checkers and theorem provers).

5. Pilot transformations

Formal verification. The formal verification of logic correctness of concurrent object-based systems designed in UML necessitates the transformation of the statechart diagrams (describing the behavior of the objects) to mathematical models amenable to formal verification. In [4] a transformation from a subset of UML statecharts (covering

all aspects of concurrent behavior) to Promela, input language of the model checker SPIN [3], was presented. We extended this approach to multiple statecharts (i.e. objects) communicating through event queues and implemented the transformation in the VIATRA framework.

The UML model is transformed by about 40 graph transformation rules to a semantically equivalent formal model called Extended Hierarchical Automata (EHA). The EHA format has the advantage that the interlevel and compound transitions are resolved and the state refinement is expressed in a strict tree structure. A single EHA is composed of simple sequential automata related by a state refinement function, while the individual objects specified in the UML model are represented by a set of communicating EHAs. The resulting XML representation of the EHAs is postprocessed by a Java application to generate the corresponding Promela code. The results of the verification (counterexamples) are available in the form of message sequence charts and execution traces to be back–annotated to the UML CASE environment.

Checking general safety criteria. Most of the accidents caused by computer programs occur due to flaws in the specification; mainly because of its incompleteness, inconsistency, or non–determinism. For this reason, N. Leveson has specified 47 general criteria for the specification of safety-critical software [5]. When using UML as the specification language for such software, the majority of these criteria should be verified on statecharts.

The standard metamodel of statecharts is not directly appropriate for an automated analysis due to its complex state hierarchy (composite and concurrent states etc.). In order to automate the verification process, we introduced a *reduced form* of statecharts which is a flat model having only basic elements like states, events, transitions and actions [7]. The transformation process to the reduced form of statecharts has been implemented by about 60 (relatively simple) rules in VIATRA. The main safety criteria (as being static well–formedness constraints) are also stated in the form of graph patterns, and required to introduce some additional 40 rules having a criterion on the left-hand side and an error message object on the right. The user is informed about the results of analysis via an XML file generated by VIATRA.

Benchmark applications. We used the UML models of two industrial dependability-critical applications in order to validate our approach. One pilot design is a safety-critical part of an *artificial kidney machine*. The second application is the core part of a *railway supervisory traffic control and optimization system* which provides a real-time global view of the traffic and delivers information for operator decisions. Both example systems necessitate the analysis of their models for correctness, completeness and consistency.

6. Conclusions

The first experiences with VIATRA, which includes more than 10 complex model transformations (manipulating source models having more than 50,000 graph objects), are promising. The generation time of a new transformation program lies in the range of several minutes on a usual desktop PC. The generation of the mathematical model itself takes typically less than a few tens of seconds for the models of small and medium size evaluated so far.

The combination of visual design of transformation rules as definition language and graph transformation based generation of transformers seem to be an effective way to conquer the problems related to the implementation of complex mathematical software.

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] A. Bondavalli, M. Dal Cin, D. Latella, I. Majzik, A. Pataricza, and G. Savoia. Dependability analysis in the early phases of UML based system design. *International Journal of Computer Systems - Science & Engineering*, 16(5):265–275, 2001.

[3] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[4] 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.

[5] N. G. Leveson. *SAFEWARE: System Safety and Computers*. Addison Wesley, 1995.

[6] Object Management Group. *Meta Object Facility Version 1.3*, September 1999.

[7] 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. Conf., SAFECOMP-2001)*, volume 2187 of *LNCS*, pages 46–55. Springer, 2001.

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

[9] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, 1999.

[10] D. Varró. Automatic program generation for and by model transformation systems. In H.-J. Kreowski, editor, *Proc. AGT 2002: Workshop on Applied Graph Transformation*, 2002.

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