The Analysis of UML State Machine Formal Checking Methods

Gundars Alksnis

Department of Applied Computer Science, Riga Technical University, Meza iela 1/3, Riga, LV-1048, Latvia
galksnis@cs.rtu.lv

Abstract. The purpose of this paper is to discuss current trends of such UML state machine model checking methods, where model checking is performed by translation into formal notations. Formal methods have shown their advantages for functional requirements' inconsistency checking and their elimination at early stages of development and for enriching graphical notations to reveal vagueness and inaccuracies as discussed in this paper.

Keywords: UML state machines, model checking, formal notations and methods.

1 Introduction

The design of system models and their modeling is widely used practice at the early stages of software development life cycle. Models provide better understanding of proposed system and allow making correct decisions concerning the implementation. For such purposes, the diagrams, which are based on some graphical notations, are used most common. To enable communication between developers, model diagrams have particular (standardized) underlying graphical notations. Among such graphical notations is Unified Modeling Language (UML) [1]. Additionally, the model driven development approach has been put forward to enable creation, validation and transfer of syntactically and semantically complete models, such that the source code generation can be automated and hidden from developers, thus, promoting models as the main artifact of software development.

However, currently available tools are mostly immature in a sense, that generated code must be manually refined to implement algorithms and system behavior conceived by the developer. The most essential barrier to implement model driven development vision is incomplete syntax and semantics of such graphical technologies, which can yield to inconsistent interpretations even at the initial stages of system design.

Even current UML 2.0 specification allows semantically incomplete model specifications for use in scenario "*UML-As-Sketch*", and trends show that this situation will remain such also in the future. This means, that UML models can and will be designed both formally and informally. The only requirement is that formal UML semantics must be a subset of the full UML semantics.

On the other hand, there are tools, which provide automated code generation (usually only framework) directly from UML models, and vice versa. However, code generated in this manner, either must be manually refined with algorithms that UML modeling tools cannot express explicitly or the generated code is not sufficiently effective for production.

Nevertheless, to make confidence about correctness of designed model and generated code (even partial), there must be done additional checks, which are in higher abstraction levels than code in high level programming languages. Namely, *model checking* must be carried out, to check whether particular model is semantically complete and self-consistent. Such checks, for example, can be done with the help of formal methods. Wherewith, there is a need for translation of graphical notations into some formal specification notation, which afterwards is analyzed and executed in the appropriate tools.

This is even more important for system's dynamic or *behavioral models*, when developer needs assurance, that the model expresses proposed system's behavior appropriately. For such cases, there are developed methods and their supporting tools, the brief survey and analysis of which is discussed in this paper.

The paper discusses current trends of such UML state machine model checking methods, where models are translated into formal notations. The comparative analysis of four such formal model checking methods is given. Finally, the paper concludes with the outline of current possibilities for formal model checking of UML state machine models with the help of formal methods.

2 Motivation

Formal model checking has proved to have a number of benefits, which gives detailed view about the system being implemented at the stages of system analysis and design. For example, model checking allows to:

- Validate behavior at the initial stages of system's design;
- Reveal and eliminate functional requrements errors before implementation;
- Obtain (automatically generate) formally "executable" model specification (i.e. system's prototype);
- Validate consistency and completeness of model specification, and more.

To make model checking effective and automated, model formal syntax and semantics must be defined. Otherwise, model can be viewed only as a *sketch*, and inconsistencies and contradictions must be validated manually by using developer's knowledge.

The most important thing in model checking is that models allow system analysis at high abstraction levels, where nondeterministic and parallel actions may take place. Thus, executable code generation from the model and checking of such code may not always give expected results. Developer must not only analyze each model's node interconnections with other nodes, but also, and more importantly, look at the model as a whole and even in connection with other models, where the same artifact is viewed from different aspects or viewpoints.

It is especially important in modeling system's behavior. Even if the model reflects system's behavior in time, usually it is shown in the form of static diagram. If such models are "executed" before they are transformed into code, it is possible to reveal some deficiencies, which may show up only in the run-time, but not in the compile-time.

The next section discusses in detail the possibilities of checking system's behavior from UML state machines [2], which are based on David Harel statecharts [3] and extended with object oriented properties.

3 Comparative Analysis of Formal Model Checking Methods

As the subject for comparison and analysis, we chose four formal model checking methods (approaches), which provide means of translating state machine models into formal notations for actual model checking. The methods are the following:

- Fernández-Toval rewrite logic;
- Zhao graph transformation;
- Knapp-Merz *Hugo/RT* tool;
- Sekerinski-Zurob abstract machine notation and B;

This decision is based solely on a variety of implementation forms. The only common thing among chosen methods is that they formalize UML state machine notation by transforming models into formal specifications with which formal model checking is performed. From the implementation viewpoint, these methods are completely independent from each other. There are many other methods available, but the selection for this research was based on the degree of involved formal methods and notations.

José Luis Fernández Alemán and Ambrosio Toval Álvarez propose UML metamodel based method, which formalizes UML state machine diagrams in rewrite logic [4]. To accomplish this, they apply rewrite logic and the specification language Maude.

Yu Zhao with colleagues [5] suggest to perform transformation and checking of models in Petri nets. The motivation for selecting Petri nets formalism is that it allows comprehensive and automated model checking.

Alexander Knapp and Stephan Merz propose model checking approach based on relations between different UML diagram types [6]. Namely, they take into account connections between UML state machine diagrams and UML interaction (i.e., collaboration and sequence) diagrams. They have developed tool *Hugo/RT*, which allows verifying if two models are consistent.

The last of the reviewed methods is proposed by Emil Sekerinski and Rafik Zurob. In it, the statechart diagrams are translated into Abstract Machine Notation (AMN) of B method [7]. Although they are speaking about statecharts

Table 1. Summary of reviewed UML state machine formal model checking methods

Method	**Fernandez-Toval**	**Zhao**	**Knapp-Merz**	**Sekerinski-Zurob**
Underlying theory / *tool*	Rewrite logic / *Maude*	Graph transformations and Petri nets	*Hugo/RT*	AMN and B method / *iState*
Has support for static semantic	Yes	Yes	Yes	Yes
Has support for dynamic semantic	No	Yes	Partly (by translating into Java code)	Partly (by translating into code)
Model can be executed	Only for checking of static semantics	In any tool which supports execution of Petri net models	Yes (for demonstration purposes only)	Yes (as final product code)
Uses UML metamodel	Yes	Yes	Yes	Yes
Automatic output of model errors	Partly (only for static semantics)	Partly (must use Petri net debugging)	Partly (only for static semantic)	Partly (only for static semantic)
Can generate source code	No	No	Yes (for demonstration purposes only)	Yes

in Harel's notation, but taking into account the origins of UML state machine diagrams, their method can be modified for application to UML state machines.

Summary of reviewed methods is given in Table 1. First column lists comparative properties and the rest of columns contain property values for each method. Chosen comparison characteristics are based on semantics (both, static and dynamic) handling and analysis of model checking results.

Review of the methods revealed, that the main problem to handle, is the formalization level of UML state machine operational semantics (mathematical foundation for interpretation and execution sequence), which is not fully defined by UML designers. This means, that formalization may be different from one method to another and thus made incompatible, and lead to different interpretations, especially in the aspect of execution.

Possible solution to this problem could be the translation of UML state machines into hierarchical finite state automata. However, it does not provide means for description of some essential UML state machine properties, for example, activities, input and output actions, finishing events and transitions, historical and branching pseudostates, and adaptation of them are not trivial.

All reviewed methods supports checking of static semantics—they can give answer to whether model is complete—with entry, initial state, intermediate states and final state and exit, but cannot give answer whether it is logical. Checking of dynamic semantics is allowed in methods with executable formal specifications.

The purpose of checking of dynamic semantics is to clarify, whether model elements are well connected. This is accomplished by selection of appropriate formal notation, as in Zhao's method, where formalization in Petri nets and tool availability enables more throughout model checking. Model execution is exploration of model state space. Because state space can be infinite, model execution strategies (especially for nondeterministic and concurrent states) determine whether results will be adequate.

However, all reviewed methods have common characteristic. Namely, they all are based on UML metamodel, which enables easer refinements of the method, when UML state machine metamodel definition change. Additionally, metamodel inconsistencies and missing definitions must be corrected.

However formal or automated model checking may be, inevitably, there are situations in which decisions must be made by the model developer. Nevertheless, his or her decision may largely depend on information about current model and further refinement. For such cases, the tool must output appropriate model checking results. All reviewed methods output inconsistencies of model static semantics, nonetheless, model behavior correctness and inconsistency elimination is solely on developers' competence and intuition.

Finally, in the context of model driven development, comprehensive code generation from the model is essential step, which gives confidence that transformation is performed with strict rules and that code indeed reflects developed model. Unfortunately, in reviewed methods this step is still immature, if supported at all. The result is whether inefficient from run-time viewpoint (Knapp-Merz's method) or define restrictions against model properties and programming languages (Sekerinski-Zurob's method).

4 Conclusions

In this paper we outlined selection of formal model checking methods, the main distinction of which is their use of model translations into formal notations and formal specification languages particularly, with the aim to utilize advantages of formal specifications in recovery of inconsistencies and their eliminations. The paper outlined main problems developer must deal with when UML state machine diagram model checking is performed.

The common conclusion is that there are many different methods available, but none of them covers all aspects of behavioral model checking—each has its own advantages and disadvantages. However, the most notable shortcoming is that current UML specification is not fully formalized, which, in turn, makes method creators do their own formalization, which may differ from other formalization approaches, though slightly.

This topic is also important in the context of model driven development, where models are supposed to be the main artifact of system's design. In fact, the more elaborated tools developers will have the more high-level abstraction checks they will be able to perform, the more precise models will be developed and generated code will be more effective.

This research was conducted as part of ongoing research for implementation of framework for Model Driven Architecture extension with formal methods [8]. In this UML metamodel based framework the formal specification languages are used for Platform Independent Model transformations and refinements into Platform Specific Models.

The common conclusion is that the model checking still is mainly guided by the developer, and for automatically generated code to be fully optimized for particular model, there is the need for further research. In addition, formal methods are subject to critique for their large learning curve and missing effective tool support; however, formal methods applications has proved as advantageous in functional requirement inconsistency checking and their elimination at early stages of system's development, and for enriching graphical notations to reveal vagueness and inaccuracies, as was discussed in this paper.

This work has been partly supported by the European Social Fund within the National Programme "Support for the carrying out doctoral study program's and post-doctoral researches" project "Support for the development of doctoral studies at Riga Technical University".

References

1. OMG: *Unified Modeling Language.* //Internet: http://www.uml.org/
2. Dan Pilone, Neil Pitman: *UML 2.0 in a Nutshell. A Desktop Quick Reference.* O'Reilly Media, Inc., USA, California, 2005.
3. David Harel: *On Visual Formalisms.* Communications of the ACM, volume 31, number 5, May 1988, pp. 514–530. //Internet: http://doi.acm.org/10.1145/42411.42414
4. José Luis Fernández, Ambrosio Toval: *Can Intuition Become Rigorous? Foundations for UML Model Verification Tools.* Proceedings of the 11th International Symposium on Software Reliability Engineering (ISSRE 2000), IEEE Press, 2000, pp. 344–355.
5. Yu Zhao et al.: *Towards Formal Verification of UML Diagrams Based on Graph Transformation.* Proceedings of the E-Commerce Technology For Dynamic E-Business International Conference, Volume 00, IEEE Computer Society, Washington, DC, USA, 2004, pp. 180–187. //Internet: http://dx.doi.org/10.1109/CEC-EAST.2004.70
6. Alexander Knapp, Stephan Merz: *Model checking and code generation for UML state machines and collaborations.* D. Haneberg, G. Schellhorn, and W. Reif, editors, 5th Workshop on Tools for System Design and Verification, Technical Report 2002-11, Institut fur Informatik, Universitat Augsburg, 2002, pp. 59–64.
7. Emil Sekerinski, Rafik Zurob: *Translating statecharts to B.* M. J. Butler, L. Petre, and K. Sere, editors, 3rd International Conference on Integrated Formal Methods (IFM), Lecture Notes in Computer Science, Springer-Verlag, May 2002, pp. 128–144.
8. Gundars Alksnis: *Formal Methods and Model Transformation Framework for MDA.* Proceedings of the 1st International Workshop on Formal Models (WFM'06), Dusan Kolar and Alexander Meduna (Eds.), Ostrava: MARQ, 2006, pp. 87–97.