VERIFICATION FRAMEWORK FOR UML - BASED DESIGN OF EMBEDDED SYSTEMS*

Martin Kardos and Yuhong Zhao
Heinz Nixdorf Institute, University of Paderborn, Germany

Abstract: System level design incorporating system modeling and formal specification in combination with formal verification can substantially contribute to the correctness and quality of the embedded systems and consequently help reduce the development costs. Ensuring the correctness of the designed system is, of course, a crucial design criterion especially when complex distributed (real-time) embedded systems are considered. Therefore, this paper aims at presenting a verification framework designated for formal verification and validation of UML-based design of embedded systems. It first introduces an approach of using the AsmL language for acquiring formal models of the UML semantics and consequently presents an on-the-fly model checking technique designed to run the formal verification directly over those semantic models.

Key words: embedded system design, UML, formal semantics, ASMs, AsmL, formal verification,model-checking

1. INTRODUCTION

The increasing complexity of today's embedded systems imposes new demands on the overall design process and on the used design languages and verification techniques. The *system level design* has become a hot topic in the research area of embedded systems and is gradually gaining popularity in the designer community. Typical for system level design are specification and modeling techniques offering facilities for coping with the system complexity such as structural decomposition, abstraction, refinement, etc. However, employment of these techniques into the design process of embedded systems can not succeed without appropriate support for

*This work has been supported by the German National Science Foundation (DFG) embedded systems can not succeed without appropriate support for verification. Therefore, verification techniques are needed that are able to identify the design errors hidden in the abstract and often incomplete models at the earlier stages of the system level design.

The work presented in this paper deals with formal verification of UML-based design for embedded systems. The main objective resides in providing a unified verification framework based on a solid formal background that integrates formal verification techniques together with model-based validation techniques. In this way we believe that system designs of high complexity could be verified at early design phase of the system development lifecycle.

The remainder of the paper is organized as follows. Section 2 gives an overview of the proposed verification framework. Section 3 outlines the work on formalizing the UML semantics by means of the ASM-theory based specification language AsmL. Section 4 presents a model checking approach towards formal verification of the AsmL specifications. In this section, the focus is put on the description of an on-the-fly algorithm and its functional parts consequently followed by the introduction of possible enhancement towards the efficient model checking of distributed systems. In Section 5 the related work is discussed. Finally, the paper concludes with a brief outlook on the future work in Section 6.

2. FRAMEWORK OVERVIEW

The proposed verification framework is depicted by means of a process flow diagram shown in the Figure 1. The input to the verification process (dashed box) is represented by an UML model describing the specified system. The verification process is further divided into two parallel branches, namely the formal verification and the validation. The main goal of the formal verification consists in proving the correctness of the required properties that a given UML model has to fulfill. This is achieved by incorporating model checking techniques into the verification framework. The validation branch, on the other hand, comprises of the methods for conventional model simulation amended by the model-based testing. Both branches are built upon a common formal background based on the theory of Abstract State Machines (ASMs) and are implemented in the AsmL language.

In the rest of the paper we focus only on the formal verification, i.e. the model checking of AsmL specifications. The simulation and model-based testing approaches are based on the tool support coming together with AsmL and are out of the scope of this paper.

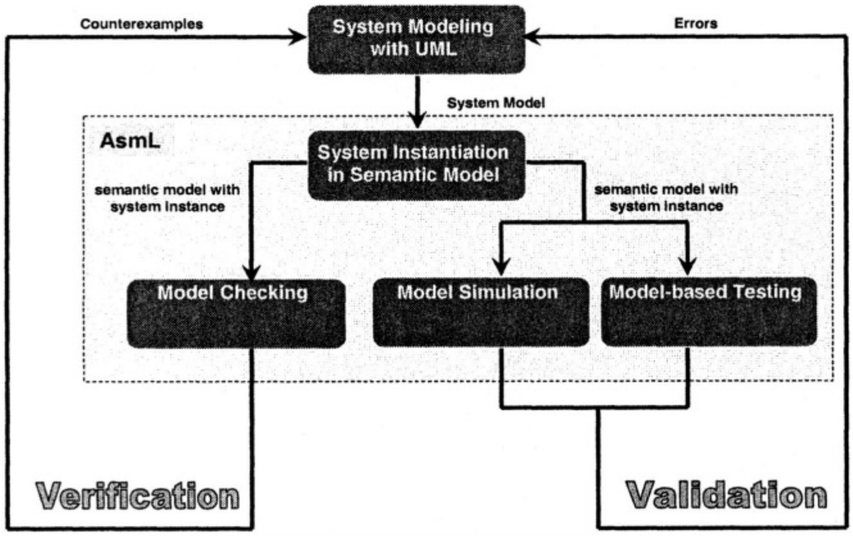

Figure 1. Verification framework for UML-based design

3. FORMALIZING UML SEMANTICS

The main prerequisite for integration of the proposed verification methods into the verification framework is the presence of a rigorous formal semantics of the modeling paradigm, in our case represented by the Unified Modeling Language (UML 2.0) [1]. Therefore, choosing the right formal method is one of the crucial decisions to be taken. In our approach the *Abstract State Machines* (ASMs) [2] has been chosen as a suitable formalism to define the formal semantics of UML. The ASMs have approved their strong modeling and specification abilities in various application domains [3] also including work on formalization of selected parts of the older version of UML [4,5]. In particular, we adopted the AsmL language [6], an executable specification language built upon the theory of Abstract State Machines, to formally describe the UML semantics.

Formalizing the UML semantics is a tedious task, especially when the complexity and vastness of the whole UML 2.0 is considered. Therefore, our aim is not to formalize the complete semantics of UML. Instead, we consider only those UML diagrams that have been adopted into our design methodologies focusing on the two main application domains we are active in: the design of distributed production control systems and the design of self-optimizing multi-agent systems with mechatronic components. In the former, UML is applied to model distributed software for controlling production lines. We use UML structure diagrams, collaboration diagrams and state machine diagrams combined with modeling of actions by means of so-called Story Diagrams [7]. In the latter, similar diagrams are employed except that the state machine diagrams are extended with discrete time semantics. Due to the fact that the formalization process is out of the scope of this paper we omit further details.

Although both application domains strongly overlap, there still exist specific semantic deviations that result in partially different semantic models of UML written in AsmL. However, the verification framework presented in this paper does not depend on any semantic deviations. The solution resides in using the AsmL as formal platform for all verification and validation methods of the framework that are designed in a way to support any AsmL specification regardless of what it describes.

4. MODEL CHECKING ASML MODELS

One of the qualities of AsmL is the high expressivity and richness of the language that allows us to keep the semantic models of UML in a readable and comprehensible form. This gives us flexibility in further maintenance of the semantic models and eases their modification and updating. However, in order to keep this advantage of AsmL we need to provide such a model checking approach that imposes least restrictions on the AsmL specification. Concretely, an AsmL specification should be allowed to fully exploit the robust data type system build in the AsmL, should allow dynamic object creation as well as usage of whole operational functionality provided by AsmL. The only constraint imposed on a specification is related to the size of its state space that has to be finite. The model checking approach presented in the next sections obeys all these requirements. It can be classified as an on-the-fly approach working over the explicit ASM state.

4.1 On-the-fly model checking

The intended model checking approach is depicted in the Figure 2. First of all, a particular AsmL specification and the property to be verified are provided as inputs. The property is specified in form of a temporal logic formula. In the first step, the temporal formula is transformed into a property automaton. As next, the AsmL specification is compiled and prepared for on-the-fly exploration. When both steps are successfully finished, the verification algorithm is started. During this process the state space exploration of a given AsmL specification is driven by the verification algorithm in an on-demand manner. The verification process may terminate in one of the following states: 1) in the OK state, after the whole state space has been explored and no contradiction of the property has been detected, 2) in the contradiction state, if a state of the system is found that does not satisfy the property and a counter example is produced 3) in the exception state, when an exception inside the specification is thrown during the state space exploration, and 4) in the user termination state, if the verification process was forced by the user to terminate.

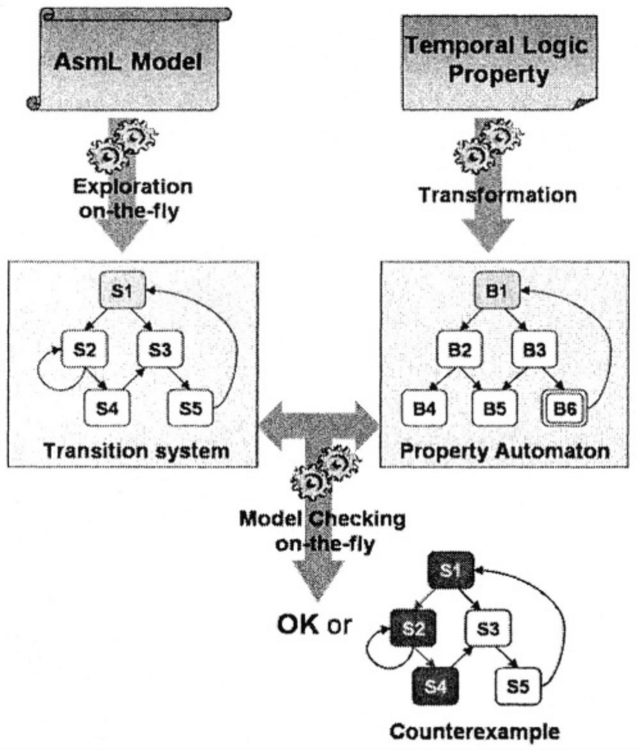

Figure 2. On-the-fly model checking of AsmL

4.1.1 Property specification and transformation

During model checking a system is verified against a property describing the desired system behavior. The property is expressed in form of a temporal logic formula. There exist several kinds of temporal logics, e.g. CTL, LTL, CTL* which usually differ in the set of expressible behaviors. In our approach we consider the CTL* logic that subsumes both CTL and LTL. The transformation of a CTL* formula into an automaton is done following the method introduced in [8]. This method uses a set of predefined goal-directed rules to derive the states of specialized tree automata called alternating Büchi tableau automata (ABTAs). An ABTA represents the property automaton showed in Figure 2.

4.1.2 Transition system construction

A transition system (a state transition graph) derived from an AsmL specification represents all possible runs of the specification. Obviously, the construction of such a transition system is, with respect to the needed time and resources, the most costly part of the overall model checking process. Therefore, we propose an on-the-fly construction approach that uses the exploration function built-in in the AsmL Toolkit. This function should allow us to drive the exploration of the system state space according to the demands of the verification algorithm. Additionally, the configurability of the exploration process gives us the apparatus to control how the state space is going to be explored. Thanks to this feature, even an infinite specification can be model checked within a fixed state space boundary (bounded model checking).

4.1.3 Verification algorithm

The model checking algorithm adopted in our approach originates in the work presented in [8]. It works over a product automaton, constructed from the produced property automaton and the transition system. Since, in our case, the transition system is generated in an on-the-fly manner, the original algorithm had to be adapted accordingly. In addition, the algorithm was redesigned in order to achieve a certain generics with respect to the implementations of transition system and property automaton. This gives us more freedom for experiments towards achievement of optimal implementations.

4.2 Incremental Model Checking

The presented model checking approach, similar to any other existing approaches, can show its weakness when it comes to verification of AsmL specifications that have a large state space. This is typical for example for distributed systems that consist of several interacting components running in parallel. In order to cope also with such distributed systems we propose a solution embedded into our verification framework. The main idea resides in defining an algorithm that is capable of executing the model checking in an incremental manner. The algorithm proposed here, depicted in Figure 3, can be seen as an enhancement of the on-the-fly algorithm presented above. It considers an AsmL specification consisting of several components (ASM agents) running in parallel and affecting each other only through their precisely defined communication. In addition, the properties to be verified are constrained to only ACTL formulas (the CTL formulas with only universal quantifiers).

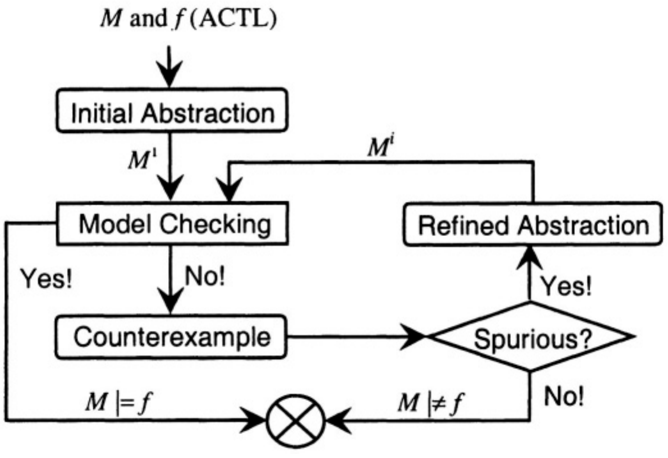

Figure 3. Control flow of incremental model checking

For an embedded system *M* with a finite set of variables $V = \{v_1, v_2, \ldots, v_n\}$ where each variable v_i has an associated finite domain D_i $(1 \leq i \leq n)$, the set of all possible states is $D = D_1 \times D_2 \times \ldots \times D_n$. Let *P* be the set of atomic propositions derived from the system. Then the system can be represented as a *Kripke* structure $M = (S, I, R, L)$ where $S = D$ is the set of states, $I \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation between states and L: $S \rightarrow 2^P$ is the labeling function. Given an ACTL property *f*, to avoid checking the satisfiability of *f* directly on *M* due to the state space explosion problem, we can obtain an abstract model (initial abstraction) from the original system by applying an appropriate abstraction function *h* to *M*. Intuitively, the abstraction function *h* induces an equivalence relation $\equiv_h$ on the domain *D*. That is, let *d, e* be states in *D*, then $d \equiv_h e$ iff $h(d) = h(e)$. It means that the equivalence relation $\equiv_h$ partitions *D* into a set of equivalence class denoted as $[D]_h = \{[d] \mid d \in D\}$ where $[d] = \{e \in D \mid h(e) = h(d)\}$. If we regard each equivalence class $[d]$ as a state from an abstract view, an abstract *Kripke* structure $M_h = (S_h, I_h, R_h, L_h)$ derived from *M* with respect to *h* can be defined as follows:

1. S_h is the abstract domain $[D]_h$;
2. $I_h = \{[d] \mid \exists e(e \in [d] \wedge e \in I)\}$;
3. $R_h = \{([d_1], [d_2]) \mid \exists e_1 \exists e_2(e_1 \in [d_1] \wedge e_2 \in [d_2] \wedge (e_1, e_2) \in R\}$;
4. $L_h([d]) = \cup_{e \in [d]} L(e)$.

Usually, the abstraction function *h* can be obtained by analyzing the dependency relationship between the variables in the system as well as the effect of these variables on the property to be verified. It is obvious that M_h

covers all possible behaviors of M but contains fewer states and fewer transitions than M. In this sense, M_h is an upper approximation to M, which means that an ACTL formula f *true* in M_h implies it's also *true* in M. However, in case that M_h falsifies f, the counterexample may be the result of some behavior in M_h which is not present in the original model M. Therefore, by refining M_h to a more precise model, i.e. far closer to M, it is possible to make the behavior which caused the erroneous counterexample disappear. For the refinement of M_h, we repeat the above procedure until a definite conclusion can be drawn. During this procedure, the initial abstraction M_h will be refined more and more close towards M. The refinement can be done based on the information derived from erroneous counterexamples [19]. As a result, the refined model is obtained by splitting the abstract state causing the erroneous counterexample into two subsets of the states, each of which represents a new abstract state. In this way, the erroneous counterexample does not exist in the refined model any more.

Given an abstraction function h, it is easy to know that the initial abstract model M_h can be constructed on-the-fly. Consequently, we can apply the on-the-fly model checking mentioned in section 4.1 to the abstract model of the original system M. If the abstract model satisfies f, then we can conclude the original system satisfies f. In case that a counterexample is found, we can locate the first abstract state which can cause the counterexample and then split the abstract state into two abstract states. Afterwards, we continue the on-the-fly model checking on this modified abstract model until a definite answer is obtained.

5. RELATED WORK

Many methods on model checking UML model [9,10,11,12,13] have been presented in recent years. The basic idea of all these methods is to transform the UML model to the input language of an existing model checking tool, say SMV, SPIN or UPPAAL for example. In other words, the semantics of the UML model is reflected through the input language of some model checker. The expressiveness of the model checker's input language usually limits the expressiveness of the checked UML model. Unlike these methods, our method presented in this paper uses the ASM-based executable specification language AsmL to define the semantics of the UML model. The expressive power of AsmL allows us to formalize the semantics of any complex UML model that implies no constraints on used UML diagrams at the user's side. In addition, the resulting AsmL specification can be executed or tested by the tools coming with AsmL.

Of course, AsmL can also be used to do model checking. Since AsmL is quite a new language, there are no published approaches aimed at model checking AsmL yet. However, a few papers can be found concerning model checking of Abstract State Machines [14,15,16]. Basically, we can identify two main approaches both based on translation of the selected subsets of ASMs into the input language of an existing model checking tool. In the [14,15] an ASM model is first simplified by flattening the data structure and the corresponding ASM rules, and then translated (by direct mapping) to the SMV [17] input language. The approach introduced in [16] follows similar strategy, but uses the SPIN [18] model checker and its PROMELA language. The main drawbacks of both approaches consist in the constraints imposed on the supported ASM models. On the other hand, imposing such constraints seemed to be an inevitable decision in order to bridge the gap between the different expressive power of ASMs and the model checker languages. Our method can avoid this problem by model checking AsmL specifications directly.

6. CONCLUSION AND FUTURE WORK

In this paper we have presented a verification framework designated for formal verification and validation of UML-based design of embedded systems. The main ideas consist in using the AsmL specification language to define the formal semantic model of the supported part of UML and consequently applying model checking technique directly on the resulting AsmL semantic model. In addition, we have introduced two model checking methods, on-the-fly model checking and incremental model checking that we hope, are suitable for verifying large complex system models.

The work presented here is still an ongoing research work that needs to be evaluated in order to approve its practical utilization. Therefore, after finishing the implementation of the discussed methods we plan to focus on their evaluation by taking real system examples from the already mentioned application domains. Consequently, we plan to integrate the formal verification into our verification framework together with the simulation and model based-testing functions provided by the AsmL tools.

REFERENCES

[1] Object Management Group. UML Superstructure Submission V2.0. OMG Document ptc/02-03-02, January 2003. URL: http://www.omg.org/cgi-bin/doc?ad/2003-01-06.

[2] Y. Gurevich: Evolving Algebras 1993: Lipari Guide; E. Börger (Eds.): Specification and Validation Methods, Oxford University Press, 1995.

[3] Abstract State Machines web page: http://www.eecs.umich.edu/gasm/

[4] E. Börger, A. Cavarra, and E. Riccobene: An ASM Semantics for UML Activity Diagrams, in Teodor Rus, ed., Algebraic Methodology and Software Technology, 8th International Conference, AMAST 2000, Iowa City, Iowa, USA, May 20-27, 2000, Proceedings, Springer LNCS 1816, 2000, 293--308.

[5] K. Compton, J. K. Huggins, and W. Shen: A Semantic Model for the State Machine in the Unified Modeling Language. In Gianna Reggio, Alexander Knapp, Bernhard Rumpe, Bran Selic, and Roel Wieringa, eds., "Dynamic Behaviour in UML Models: Semantic Questions", Workshop Proceedings, UML 2000 Workshop, Ludwig-Maximilians-Universität München, Institut für Informatik, Bericht 0006, October 2000, 25-31.

[6] Y. Gurevich,W. Schulte,C. Campbell,W. Grieskamp. AsmL: The Abstract State Machine Language Version 2.0. http://research.microsoft.com/foundations/AsmL/default.html

[7] T. Fischer, J. Niere, L. Torunski, A. Zündorf: Story Diagrams: A new Graph Rewrite Language based on the Unified Modelling Language and Java; in Proc. of the 6th International Workshop on Theory and Application of Graph Transformation (TAGT), Paderborn, November 1998, LNCS, Springer Verlag.

[8] G. Bhat, R. Cleaveland, and A. Groce: Efficient model checking via Buchi tableau automata. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000

[9] T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. In *Proc. Wsh. Software Model Checking,* Volume 55(3) of *Elect. Notes Theo. Comp. Sci.,* Paries, 2001.

[10] A. Knapp, S. Merz, and C. Rauh. Model Checking Timed UML State Machines and Collaborations. *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault Tolerant Systems,* LNCS 2469, pages 395-416. ©Springer, Berlin, 2002

[11] K. Diethers, U. Goltz and M. Huhn. Model Checking UML Statecharts with Time. In Proc. of the Workshop on Critical Systems Development with UML, 2002.

[12] A. David, M.Möller, and W. Yi. Formal Verification of UML Statecharts with Real-Time Extensions. In Proc. of FASE 2002 (ETAPS 2002). LNCS 2306, p218-232, 2002.

[13] S. Gnesi and D. Latella. Model Checking UML Statechart Diagrams using JACK. In *Proc. Fourth IEEE International Symposium on High Assuarance Systems Enginering,* IEEE Press, 1999.

[14] G. del Castillo and K. Winter: Model checking support for the ASM high-level language. In S. Graf and M. Schwartzbach, editors, Proc. on 6th Int. Conf. TACAS 2000, volume 1785 of LNCS, pages 331-346, 2000.

[15] Kirsten Winter: Model Checking Abstract State Machines, Ph.D. thesis, Technical University of Berlin, Germany, 2001.

[16] A. Gargantini, E. Riccobene, S. Rinzivillo: Using Spin to Generate Tests from ASM Specifications, In E. Börger, A. Gargantini, E. Riccobene, editors, Proc. of 10th International Workshop on Abstract State Machines 2003, Taormina, Italy, March 3-7, 2003

[17] K. McMillan: Symbolic Model Checking, Kluwer Academic Publishers, Boston (1993).

[18] G. J. Holzmann: The model checker SPIN. IEEE Transactions on Software Engineering, May 1997.

[19] E. Clarke, O. Grumberg, S. Jha, Y. Lu and H. Veith. Counterexample-guided abstraction refinement. In Computer Aided Verification, volume 1855 of LNCS, pp. 154-169, 2000.