FUNCTIONAL VERIFICATION FOR UML - BASED MODEL DRIVEN DESIGN OF EMBEDDED SYSTEMS

Martin Kardos, Norbert Fristacky

¹Heinz Nixdorf Institute, University of Paderborn, Germany; ²Faculty of Informatics and Information Technologies, Slovak University of Technology, Bratislava, Slovakia

Abstract: Modeling and formal specification in combination with formal verification can substantially contribute to the correctness and quality of design of embedded systems and consequently help reduce the development costs. This paper tackles the problem of providing a fully automated formal verification for UML-based design of embedded systems. For verification it employs the model checking technique. Unlike other approaches, the paper focuses on supporting a consistent subset of UML diagrams that cover all main aspects of an embedded system, i.e. the structure, communication and behavior. The whole approach is evaluated on a case study of a Manufacturing Flow System with distributed control.

Key words: Embedded System Design, UML, 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. Specification and modeling techniques, such as UML, offering facilities for structural decomposition, abstraction, refinement, concern separation, etc., can be employed at the *system level* of the design process to cope with the system complexity. However, the employment of these techniques into the design process of 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 design.

In this paper we present an approach for automated formal verification of UML-based design of embedded systems that employs for verification the model checking technique. The verification concept is presented in Section 3. Unlike other approaches, the main objective of this work is aimed on supporting a consistent subset of UML diagrams that cover all aspects of an embedded system, i.e. the structure, communication and behavior. The supported UML subset (the *Verifiable UML*) is discussed in Section 2. A prerequisite for employing formal verification into the design process is the presence of a rigorous formal specification of the UML semantics. The formalization of the *Verifiable UML* by means of the AsmL language is the subject of Section 4. The overall verification approach is evaluated on a case study of a *Material Flow System* that is discussed in detail in Section 5. The novelty of the presented verification approach and its relation to other research work is discussed in Section 6. Finally, in Section 7 the conclusion is drawn and the direction of the future work is presented.

2. VERIFIABLE UML

UML¹ [1] is a large-scale modeling language defining a set of diagrams to capture different aspects of modeled systems at different abstraction levels and under different points of view. It provides diagrams that can be seamlessly applied at the different stages of a system life-cycle like requirements specification and analysis, design, realization and deployment.

The work presented in this paper focuses mainly on the design phase. Therefore diagrams have been considered that are typically used during the design phase for modeling the main aspects of an embedded system. The diagrams considered are the *class* and *component diagram* for modeling system structure (architecture), the *state machine diagram* for modeling system behavior and the *action semantics*² for modeling operations inside the classes. We denote this subset of UML as *Verifiable UML*, i.e. a subset for which the formal verification is supported inside our approach. Although we support currently only the mentioned diagrams, in general, the *Verifiable UML* can be further extended with other UML diagrams, e.g. the activity diagram or the sequence diagram³.

¹ We consider the latest UML 2.0.

² Here the syntax of the AsmL language has been taken as textual notation.

³ An additional effort would be required to formalize these diagrams.

3. VERIFICATION CONCEPT

A concept of the proposed verification approach is shown in Figure 1. The picture contains three different levels covered by the verification approach. The *modeling level* represents an input level for the verification. Here, the UML model of a specific system (the *System model*) is created and the *system properties* to be verified are defined. In addition, the formal specification of the *Verifiable UML* described in Section 2 is considered to be a fixed immutable input for the verification⁴.

Figure 1. Verification concept for UML-based design

The models and transformations occurring at the *semantic level* form the foundation for the formal verification. At first, the semantics of the *Verifiable UML* is formalized. This step is depicted by the *Formalization* arrow. The formalization is briefly explained in Section 4. It results in a rigorous model of the *Verifiable UML* semantics described in the AsmL language, and in a set of transformation rules applied on the original *system model* to produce the *AsmL model of a concrete system*. The transformation works over an XMI representation of the *system model*. The *XMI export* functionality has to be supported by the selected modeling tool.

⁴ The Verifiable UML can be further modified or extended but this implies a consequent adaptation of the semantic metamodel and the transformation rules.

The *verification level* represents the defined verification method and its algorithms. The applied verification method is based on the model checking technique and was specifically designed for verification of AsmL specifications [2]. Basically, the verification takes on input an AsmL specification, in our case the *AsmL model of a concrete system*, and the specified *system property* (a CTL* formula). First, the *system property* is transformed to a specific automaton denoted as the *property automaton*. Subsequently, an on-the-fly model checking algorithm is started that tries to prove the soundness of the given property. During the run-time the algorithm drives the *model exploration* and builds a corresponding *Kripke structure*. As verification result the algorithm returns “ok”, if the verified property holds, otherwise it provides a counterexample that violates the property.

4. FORMALIZING UML SEMANTICS

The main prerequisite for the application of a formal verification is the presence of a rigorous formal semantics of the modeling paradigm, in our case represented by the *Verifiable UML*. In our approach the *Abstract State Machines* (ASMs) [3][4] 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 [5] also including work on formalization of selected parts of the older version of UML [6][7]. In particular, we adopted the AsmL language [8][9], an executable specification language built upon the theory of Abstract State Machines, to formally describe the semantics of the *Verifiable UML*. Basically, the formalization consists of the following steps:

1. Defining transformation rules to transform a UML class diagram⁵ into the corresponding AsmL model.

The transformation of a UML Class diagram to a corresponding AsmL model consists in defining a mapping of the graphical notation (syntax) used in UML to the textual syntax of AsmL. Such a transformation results in an AsmL model whose structure definition directly reflects the transformed diagram. This approach is widely used by the existing UML tools for the source code generation in the target programming language. However, unlike the usual programming languages, the AsmL expressiveness allows to cover also enhanced modeling concepts incorporated into the UML 2.0 Class diagram, e.g. the pre and postconditions inside the methods, derived attributes, or enhanced association types like sets, bags, sequences, etc.

⁵ It includes also the component diagram that is just a specialization of the class diagram

2. Defining a semantic model for the UML state machine in AsmL
The informal operational semantics of a state machine defined in the UML specification is converted into a precise AsmL specification. The resulting AsmL model covers the whole UML state machine (except the time aspect) including complex and history states, inter-level transitions, deferred events, etc. This work has been partially published in [10].
3. Defining a mapping of the UML action semantics to AsmL constructs
Since AsmL is a high-level specification language with strong expressivity, it is a good candidate for modeling the class operations (methods) and the actions contained inside the state machine diagram. To show a conformance to the action semantics proposed by UML, a mapping of the action semantics to AsmL should be provided.

5. MATERIAL FLOW SYSTEM CASE STUDY

This section elaborates the application of the proposed verification approach to the design of embedded control software for the *Material Flow System* (MFS) case study shown in Figure 2.

Figure 2. Material flow system - schematic view

The MFS is based on a modern modular material flow technology that allows to build complex transportation topologies by composing standard

modules such as straight and curved tracks, switches (joiner and brancher), and stations. In the current configuration the MFS is used for producing bottle openers. Special vehicles called shuttles are employed to transport material pieces over the railway between stations where some processing takes place. While operating, tracks are permanently supplied with current such that shuttles keep moving over the railway in a fixed direction until they are arrested by means of stopping cams placed at certain points along the tracks (see Figure 2). Some tracks are additionally equipped with inductive sensors to detect the arrival or departure of a shuttle. Finally, each shuttle has a built-in device that avoids collisions by enforcing a minimum distance from the foregoing shuttle. The control of the MFS is realized by a network of concurrently operating and asynchronously communicating control devices attached to active tracks (e.g. stations, switches) and robots.

5.1 Architecture

The MFS architecture is modeled using the UML class diagram shown in Figure 3. The topology of the MFS is modeled by a set of interconnected tracks. Therefore, we defined the abstract class *Track* that models a single track. A track must be connected to its neighboring track. This connection is modeled by the *direction* association. The reference to the neighboring track (in the main loop direction) is stored in the attribute *next*. To detect the arrival or departure of a shuttle, a track can own an arriving or leaving sensor respectively, modeled via associations with the *Sensor* class. The signal communication is initiated by the sensor and realized by sending a signal instance to all tracks contained in the *signaledTracks* attribute (the *communicates* association). In some cases a track has to be able to stop the shuttle, e.g. when some processing of the transported material is required. This capability is modeled through the association with the *StartAndStopUnit* class that models a stopping clam. Finally, a track owns the *at* association with the *Shuttle* class used to detect when a shuttle is positioned on a track. In addition, *Track* implements the *ITrack* interface that is used to simulate the movement of a shuttle along the track. The *Track* class is an abstract class. For modeling specific kinds of tracks we further specialize the *Track* class and define the classes *Station*, *TransferGate*, *Straight* and *Curve*. The last two classes are leaf classes that model the straight and curved tracks, respectively. The *Station* class is again an abstract class that models tracks, at which a shuttle has to be stopped and some processing is executed. A station owns a *name* and provides methods for stopping (*setStopper()*) and releasing (*releaseStopper()*) a shuttle. The concrete stations are modeled by following specialized classes: *Stock*, *Workstation*, *Portal* and *Gateway*. The *TransferGate* class models a type of

track that can switch its position in order to change the moving direction of a shuttle. There are two allowed directions the straight or the fork direction.

Figure 3. MFS architecture – class diagram

The current position of a *TransferGate* is indicated by the *isStraight* attribute. The methods *switchToStraight()* and *switchToFork()* are used to force the switching. Depending on the usage we recognize two different kinds of a transfer gate: the *Brancher* that forks the direction and the *Joiner* that joins two directions (see Figure 2). Finally, we have to mention the standalone classes *ScaraRobot*, *PortalRobot*, *Shuttle* and *ShuttleSimulator*.

The first two model the robot machines placed along the *Stock* and the *Portal* stations. The *Shuttle* class models a shuttle. A shuttle is identified by its *id* and owns a reference to the *destination* station it has to go to. The *ShuttleSimulator* class is an auxiliary class used to simulate the movement of shuttles.

5.2 Behavior

According to the UML specification a class can have a behavior. Such a class is called an active class. In MFS following classes are active: stations *Stock*, *Workstation*, *Portal* and *Gateway*, transfer gates *Brancher* and *Joiner*, robots *ScaraRobot* and *PortalRobot*, and the auxiliary class *ShuttleSimulator*.

The *Stock* station serves for unloading of finished goods (in our case bottle openers) from the shuttle and for storing them in the storage unit. The action of unloading and storing initiated by the *Stock* station is realized by the *ScaraRobot*. Figure 4 shows the state machine modeling the life-cycle of the *Stock* station.

Figure 4. UML State machine of the Stock station

At the beginning of the life-cycle the *Stock* station fires the transition from the *initial* state to the *Idle* state and sets the stopping cam. When the *shuttle_arrived* event occurs (sent by the associated arriving sensor) the state machine enters the *Unloading* state and initiates the unloading of the good by notifying the *ScaraRobot*. Then it waits in the *Wait* state until the unloading is confirmed by the *unloading_finished* event sent by the *ScaraRobot*. After the confirmation, the stopping cam is released in the *Releasing* state and the shuttle moves on. Subsequently, the stopping cam is set again to stop a possibly approaching next shuttle. After reaching the state *WaitingForShuttleGone* the state machine waits until the *shuttle_left* event occurs (sent by the leaving sensor) and then moves further to the *CurveGuarding* state. It stays in the *CurveGuarding* state until the shuttle reaches the *Workstation*. Finally, the state machine moves to the *Idle* state.

The *Workstation* serves for loading the material onto the shuttle and for assigning a task to the shuttle (method *LoadComponents()*). The task assignment is realized by setting or resetting the destination attribute of the shuttle instance. If the destination is set to *Portal* then a shuttle has been assigned a task to produce a bottle opener. The behavior of the *Workstation* class is shown in Figure 5. At the beginning of the life-cycle the *Workstation* moves from the *initial* state to the *Idle* state and sets the stopping cam. After a shuttle has arrived to *Workstation* it is loaded with components and gets assigned a task (the *Loading* state). When a shuttle is loaded the state machine moves to the *SwitchingBrancher* state. In this state the switching of the *Brancher* is initiated with respect to the shuttle's task (action *notifyBrancher()*). If the shuttle is assigned a task the *Brancher* is asked to switch to fork direction otherwise the straight direction is required. After receiving the *brancher_ready* event from the *Brancher*, the stopping cam is released and the shuttle moves on. After the shuttle has left, the state machine moves to the *Idle* state and sets the stopping cam.

Figure 5. UML State machine of the Workstation

Because of space limitations we omit the description of behaviors for the remaining active classes.

5.3 Verification

Before the verification can take place the properties to verify have to be specified. In the MFS case study we wanted to check following two properties: 1) every shuttle has to fulfill its task and reach its starting location, and 2) there must not appear a situation where two shuttles are located within one curve area. The second property represents a safety property in order to avoid a possible collision of shuttles. The collision may happen when two shuttles are located one after another in the same curve area and the front shuttle is out of the reach area of the collision detection sensor of the back shuttle. We write the informal properties formally in form

of temporal logic formulas “**AF**(AllFinished)” and “**AG**(!CollisionEmergency)”. The *AllFinished* and *CollisionEmergency* represent propositions that are evaluated in every state of the verified system. They are expressed as AsmL expressions shown in Table 1.

Table 1. Proposition expressions in AsmL

AllFinished	forall s in Shuttle_Instances holds s.cycleCounter >= 1
CollisionEmergency	exists c in Curve_Instances where c.shuttle $\diamond$ null and c.next is Curve and c.next.shuttle $\diamond$ null

The *AllFinished* evaluates to true when every shuttle has passed its starting locations, i.e. the counter of shuttle cycles is greater than 1. The *CollisionEmergency* evaluates to true when two curves, such that one is the successor of the other, both contain a shuttle.

At first, we transformed the UML model of the *MFS* to a corresponding AsmL model following the transformations on the *semantic level* as depicted in Figure 1. Subsequently we conducted the verification of the MFS model and discovered several design errors that caused the violation of the given properties. The two most crucial errors are covered by following scenarios:

1. The *setStopper()* action of the *Stock* station behavior (Figure 4) placed at the transition leaving the *Releasing* state was in the old version of the *Stock* station behavior placed as the entry action of state *WaitingForShuttleGone*. In case of three shuttles *Shuttle(1-3)* placed initially at tracks *Straight(6-4)* (Figure 2) a situation happened where *Shuttle1* arrived to the *Workstation* and *Shuttle2* was located at *Curve2* followed by *Shuttle3* moving on *Curve1*. This was a violation of the second property.
2. When a shuttle arrived at a station while the station's state machine was in another state than *Idle*, the occurred *shuttle_arrived* event has been dispatched in wrong state and consequently discarded because it did not fire any transition. The arrived shuttle stayed at the station without being ever served. This violated the first property. The problem was solved by making the *shuttle_arrived* event deferrable.

6. RELATED WORK

Since UML is a complex modeling language, there exist no verification approaches that would cover the whole UML. Basically, only a very restricted subset of UML is supported. For example, a plethora of approaches towards model checking different variations of statecharts such as *Harel* statecharts [11], *STATEMATE* statecharts [12][13] and *UML* statecharts [14,15,16,17,18,19,20] have been presented in recent years. The basic idea of all these methods is to transform the considered statechart

model to the input language of an existing model checking tool, say SMV, SPIN or UPPAAL for example. The semantics of the statechart is reflected through the input language of some model checking tool or through an intermediate representation that is further mapped into the model checker's language. Since the expressiveness of the model checker's input language is very restricted, it imposes strong limitations on the supported semantics.

Unlike these methods, our method presented in this paper uses the ASM-based specification language AsmL to define the semantics of the UML model. The expressive power of AsmL allows us to formalize the UML semantics without imposing any constraints on the supported UML diagrams. Therefore, in our formalization of UML we consider not only the complete semantics of UML state machine but also the whole class diagram for modeling the system structure and the UML action semantics for modeling operations inside the classes. Moreover, this UML subset (the *VerifiableUML*) can be fully executed and tested by the tools coming with AsmL. Further advantage of our approach in comparison to the conventional approaches is the employment of the model checking tool specially developed for the verification of AsmL specifications [2]. This eliminates the necessity of additional model transformations and additionally provides greater capabilities for properties definitions.

7. CONCLUSION AND FUTURE WORK

In this paper we have presented an approach towards the formal verification 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 executable part of UML (the *Verifiable UML*) and consequently employing model checking technique specifically designed for AsmL (the novelty of this approach is discussed in the Related Work section). As a future work the main intension is to evaluate the presented verification approach on more complex systems in order to test its scalability and to approve its practical utilization.

REFERENCES

- [1] Object Management Group. UML 2.0 Superstructure Specification. OMG Final Adopted Specification, February 2004. URL: <http://www.omg.org/cgi-bin/doc?ptc/2004-10-02>.
- [2] M. Kardos. An approach to model checking AsmL specifications, In Proceedings of the 12th International Workshop on Abstract State Machines, ASM2005, Paris, France, March 8-11, 2005

- [3] Y. Gurevich: *Evolving Algebras 1993: Lipari Guide*; E. Börger (Eds.): *Specification and Validation Methods*, Oxford University Press, 1995.
- [4] E. Börger and R. Stärk: *Abstract State Machines: A Method for High-Level System Design and Analysis*. Springer-Verlag, 2003.
- [5] Abstract State Machines web page: <http://www.eecs.umich.edu/gasm/>
- [6] 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.
- [7] 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.
- [8] Y. Gurevich, B. Rossman and W. Schulte: Semantic Essence of AsmL. To appear in the *Journal of Theoretical Computer Science* (in special issue dedicated to FMCO 2003), 2005
- [9] 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>
- [10] M. Kardos, F. Rammig: Model Based Formal Verification of Distributed Production Control Systems. Integration of Software Specification Techniques for Applications in Engineering. Springer Verlag, 2004 LNCS 3147, S. 451-473
- [11] W. Chan, R.J. Anderson, P. Beame, S. Burns, F. Modugno, D. Notkin, and J.D. Reese: Model checking large software specifications. *IEEE Transactions on Software Engineering*, 24(7), 1998.
- [12] E. M. Clarke and W. Heinle. Modular translation of statecharts to SMV. Technical Report CMU-CS-00-XXX, Carnegie-Mellon University School of Computer Science, 2000.
- [13] E. Mikš, Y. Lakhnech, M. Siegel, and G.J. Holzmann. Implementing statecharts in PROMELA/SPIN. In *Proceedings of the 2nd IEEE Workshop on Industrial-Strength Formal Specification Techniques*, pages 90--101. IEEE Computer Society, October 1998.
- [14] 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.*, Paris, 2001.
- [15] A. Knapp, S. Merz, and C. Rau. 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
- [16] 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*.
- [17] 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.
- [18] S. Gnesi and D. Latella. Model Checking UML Statechart Diagrams using JACK. In *Proc. Fourth IEEE International Symposium on High Assurance Systems Engineering*, IEEE Press, 1999.
- [19] D. Latella, I. Majzik, and M. Massink. Automatic verification of a behavioural subset of UML statechart diagrams using the SPIN model-checker. *Formal Aspects of Computing*, 11(6):637--664, 1999.
- [20] I. Paltor and J. Lilius. vUML: A tool for verifying UML models. In R.J. Hall and E. Tygu, editors, *Proc. of the 14th IEEE International Conference on Automated Software Engineering, ASE'99*. IEEE, 1999.