Towards Scenario-Based Testing of UML Diagrams*

Petra Brosch[2], Uwe Egly[1], Sebastian Gabmeyer[2], Gerti Kappel[2], Martina Seidl[3], Hans Tompits[1], Magdalena Widl[1], and Manuel Wimmer[2]

[1] Institute for Information Systems, Vienna University of Technology, Austria
{uwe,tompits,widl}@kr.tuwien.ac.at
[2] Business Informatics Group, Vienna University of Technology, Austria
{brosch,gabmeyer,gerti,wimmer}@big.tuwien.ac.at
[3] Institute of Formal Models and Verification, Johannes Kepler University, Austria
martina.seidl@jku.at

Abstract. In model-driven engineering, models are not primarily developed for documentation and requirement specification purposes, but promoted to first-class artifacts, from which executable code is generated. As a consequence, typical development activities like testing must be performed on the model level. In this paper, we propose to use overlapping information inherent in multiple views of models for automatic testing. Using a prototype based on the model checker SPIN we show the feasibility of this approach and identify future challenges.

1 Introduction

Multi-view modeling languages like UML [6] offer different diagram types to lower the complexity of describing software systems. Each diagram provides a distinct view on the system, allowing for splitting a complex model into various areas of concern [4]. In that way, the diagrams complement one another, altogether providing a holistic representation of the system. The views are connected by information redundant in the different diagrams and consistency has to be assured [4]. In this paper, we investigate how this information can be used as test data.

Consider the following example modeled in Fig. 1. Two state machines show a typical behavior of a PhD student (PhD) and a coffee machine (CM). Both state machines change their states according to the messages they receive. Conditions for the state transitions are given in terms of transition labels. The transition labels consist of two parts: The left part denotes an action triggering the transition, and the right part indicates a set of actions performed during the transition. If no triggering action is defined ("$-$"), the transition is executed unconditionally. Starting in state $Tired$, the PhD student turns the coffee machine on and optimistically waits until it is ready. If she receives the $error()$ message, she becomes desperate, then tired and tries again. Otherwise, she is happy, demands coffee, and waits until it is completed. The sequence diagram in Fig. 1 models a forbidden scenario inside a neg fragment: After the coffee machine has sent an error, it

* This work was partially funded by the Vienna Science and Technology Fund (WWTF) through project ICT10-018, by the fFORTE WIT Program of the Vienna University of Technology and the Austrian Federal Ministry of Science and Research, and by the Austrian Science Fund (FWF) under grants P21698, J3159-N23, and S11409-N23.

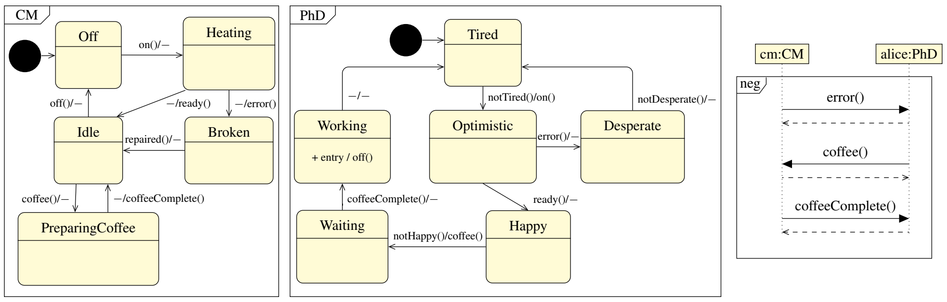

Fig. 1: Sequence diagram modeling a forbidden interaction for two state machines.

receives a coffee request and then sends a $coffeeComplete()$ message. Obviously, during the parallel execution of the state machines the forbidden sequence can occur.

In model-driven engineering, models are not only used as mere design documents but they serve as artifacts from which code is generated. It is important to detect faults in the models; otherwise they may propagate to the code. Designing test cases on the model level has been subject to extensive research, but often testing itself is transferred to the code level or requires a simulation engine. To circumvent this problem, we use communication scenarios modeled in sequence diagrams. Testing is thus shifted to model level. Hence, serious design and implementation errors in the model are detected at an early point in time by using the information available due to multi-view modeling.

We start from a restricted subset of UML state machines and sequence diagrams for which we provide a formal description. This description is designed in such a way that it is extensible. The concept of model checking UML interactions has been described in [8]. We formulate an alternative encoding more natural for our use case with multiple communicating state machines. For experimental evaluation, we built a first prototype with PROMELA, the input language of SPIN, a highly configurable, state-of-the-art model checker. This allows us to derive challenges which have to be solved to put our vision of testing multi-view models into practice.

2 Preliminaries

We consider a subset of the UML state machine and sequence diagrams modeling only forbidden scenarios. Note that, to model forbidden scenarios, we consider only sequences that are enclosed in a neg fragment. The model is consistent if the sequences given in the sequence diagrams do not occur on any path of the state machines executing in parallel. The problem is formally defined as follows: A *software model* is a triple $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ where $\mathcal{M}$ is a set of state machines, $\mathcal{S}$ is a set of sequence diagrams, and $\mathcal{A}$ is a set of actions, including the empty action ϵ, necessary to model that a transition is triggered by a completion event (denoted by "–" in Fig. 1). Note that we omitted the actions within the states which cause the completion event, because they are not relevant for our purposes. For example, Fig. 1 shows a software model of two state machines and one sequence diagram. The set of actions comprises all method calls indicated on the transitions and entry or exit actions inside states.

Definition 1. *A* state machine *is a tuple $M = (S, \iota, A^T, A^P, T)$, where S is a set of* states, *$\iota \in S$ is a designated* initial state, *$A^T \subseteq \mathcal{A}$, $A^P \subseteq \mathcal{A}$, and $T \subseteq S \times A^T \times \mathcal{P}(A^P) \times S$ is a transition relation. Each transition contains a triggering action $a \in A^T$, called* event, *which triggers a state transition, and a set $B \in \mathcal{P}(A^P)$ of actions, called* effects, *which are performed when the transition is executed.*

Note that this definition also handles *entry* and *exit* actions defined inside states: An entry action in state s_i is included in the effects of each incoming transition to s_i, and an exit action in the effects of each outgoing transition from s_i.

Figure 1 shows six states for the state machine $M_{PhD} = (S, \iota, A^T, A^P, T)$. The initial state $\iota = \mathit{Tired}$ is denoted by an incoming transition from the black circle. The transition labels consist of two parts, separated by a backslash. The set A^T contains the string on the left side of the transition labels, and A^P the set of strings indicated on the right side of the transition labels or as entry or exit actions. For example, the transition from $\mathit{Waiting}$ to $\mathit{Working}$ is $(\mathit{Waiting}, \mathit{coffeeComplete}(), \{\mathit{off}()\}, \mathit{Working})$.

Definition 2. *A neg fragment in a* sequence diagram *$S \in \mathcal{S}$ is a triple (L, m, N), where L is a set of* lifelines, *$m : L \to \mathcal{M}$ is a bijective function assigning a state machine to each lifeline, and N is a forbidden sequence of triples $L \times \mathcal{A} \times L$.*

In the sequence diagram of our running example, there are two lifelines, cm and alice. The state machine assigned to alice is $m(\mathit{alice}) = \mathit{PhD}$. The sequence of messages is $N = (\langle cm, \mathit{error}(), \mathit{alice}\rangle, \langle \mathit{alice}, \mathit{coffee}(), cm\rangle, \langle cm, \mathit{coffeeComplete}(), \mathit{alice}\rangle)$.

The behavior of a set $\mathcal{M}$ of synchronously communicating parallel state machines is defined as the composition $\mathcal{M}_{||}$ as follows [2].

Definition 3. *Let $M_k = (S_k, \iota_k, A_k^T, A_k^P, T_k)$, $k \in \{1, \ldots, n\}$, be n state machines, let $A_k = A_k^T \cup A_k^P$, and let two state machines M_i, M_j, $i \neq j$, synchronize over all actions in $H_{ij} = ((A_i^T \cap A_j^P) \cup (A_j^T \cap A_i^P)) \backslash \{\epsilon\}$ such that communication is pairwise, i.e., $H_{ij} \cap A_l = \emptyset$ for $l \notin \{i, j\}$ (obviously, $H_{ij} = H_{ji}$). Then, the* composition *of a set $\mathcal{M}$ of state machines is given by $\mathcal{M}_{||} = (S_1 \times \ldots \times S_n, \langle \iota_1, \ldots, \iota_n\rangle, A_1 \cup \ldots \cup A_n, R)$ with*

1. *$(\langle s_1, \ldots, s_i, \ldots, s_n\rangle, a, \langle s_1, \ldots, s_i', \ldots, s_n\rangle) \in R$ iff $(s_i, a, E, s_i') \in T_i$ and $a \in (A_1 \cup \ldots \cup A_n) \setminus \bigcup_{0<j\leq n, i\neq j} H_{ij}$ with $1 \leq i \leq n$. The action a is called* local.
2. *$(\langle s_1, \ldots s_i, \ldots, s_j, \ldots, s_n\rangle, b, \langle s_1, \ldots, s_i', \ldots, s_j', \ldots, s_n\rangle) \in R$ iff $b \in H_{ij}$ and there exist transitions $(s_i, v, B, s_i') \in T_i$ and $(s_j, b, G, s_j') \in T_j$ with $b \in B$ and $1 \leq i, j \leq n$ and $i \neq j$. The action b is called* global.

Definition 4. *A sequence $\pi = \langle a_1, a_2, \ldots, a_l\rangle$ is a* path *in $\mathcal{M}_{||} = (S, \iota, A, R)$ iff there exist triples $(s, a_i, s'), (s', a_{i+1}, s'') \in R$ for all i where $1 \leq i < l$. A software model $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ is* consistent *iff for any neg fragment (L, m, N) in any sequence diagram with $N = \langle n_1, n_2, \ldots, n_k\rangle$ and $n_j = (l_j, a_j, l_j')$ for all j where $1 \leq j < k$, its sequence of actions $\langle a_1, a_2, \ldots, a_k\rangle$ does not occur as subsequence of any path in $\mathcal{M}_{||}$.*

3 Formulation of the Model Checking Problem

Inspired by previous work [8], we use the model checker SPIN [7] and its input language PROMELA to verify whether a set of state machines fulfills a safety property described as *neg* fragment of a sequence diagram. To this end, we encode the state machine as a set of active PROMELA processes and the *neg* fragment as `notrace` assertion. In verification mode, SPIN checks whether the behavior specified in the assertion occurs on any execution trace of the processes executing in parallel. If this is the case, SPIN returns the erroneous execution path on which the `notrace` behavior occurred. Otherwise, it returns no error. We evaluated this approach on several examples. The following elements of PROMELA are relevant for our encoding: `active proctype` (process behavior automatically instantiated at program start), `label` (identifier of a unique control state), `mtype` (declaration of symbolic names for constant values), `chan` (asynchronous or synchronous channel), and `notrace` (assertion defining unwanted sequences of channel activities). A software model $(\mathcal{M}, \mathcal{S}, \mathcal{A})$ is encoded in PROMELA as follows: Each action label $a \in \mathcal{A} \setminus \epsilon$ is encoded as an element of `mtype`. For each state machine $M \in \mathcal{M}$ we define an `active proctype` and a synchronous global channel `chan` of type `mtype`. Each `active proctype` representing a state machine $M = (S, \iota, A^T, A^P, T)$ contains a `label` for each state $s \in S$. The label representing ι is placed at the beginning of the process to be executed first. For each state machine, each transition $T = (s_i, a, B, s_j)$ is implemented within the PROMELA label representing state s_i: A transition consists of a receive statement for a if $a \neq \epsilon$ or nothing otherwise, a statement for each $b \in B \setminus \epsilon$ or nothing if $B = \{\epsilon\}$, and a `goto` statement directing to the label representing s_j. If s_i has more than one outgoing transition, the set of transitions is put inside an `if` statement. The sequence of messages on each lifeline is encoded as PROMELA `notrace` assertion. A `notrace` assertion is defined over some or all global channels and monitors all actions on these channels during program execution. When all channel actions defined by the assertion have been executed, an error is returned. Note that `notrace` assertions can contain `accept` labels to model forbidden infinite behavior. The encoding of Fig. 1 is available online at `http://www.modelevolution.org/media/scenario-based-testing/coffee.pml`.

4 Related Work

In the following, we focus on works which present results on the successful application of verification techniques for multi-view system specifications. Cimatti et al. [5] use Hybrid Automata (HA) to describe a system of message exchanging components and verify the system against a scenario-based specification modeled with a Message Sequence Chart (MSC). They present an extension to bounded model checking using k-induction to prove that there exists no trace which satisfies a given scenario. Li et al. [9] use MSCs as scenario-based specifications for concurrent systems modeled with Petri nets and discuss an approach to check if a Petri net either satisfies a mandatory scenario on all of its traces, a forbidden scenario on none of its traces, or a dependent scenario on all traces once a given, other scenario is satisfied. The CHARMY tool suite [10] offers a modeling, simulation, and verification environment for software architectures (SA). SAs describe the static and behavioral structures of systems with component, state transition, and sequence diagrams. CHARMY employs SPIN and translates the SA to PROMELA to detect deadlocks and unreachable states. The work most closely related to ours is the one by Schäfer et al. [11]. They propose to verify a set of message-exchanging state machines against a specification described by UML collaboration diagrams. They implement their approach in HUGO, which automatically translates the state machine diagrams to PROMELA and generates Büchi automata, so-called "never claims", from the collaboration diagrams. The generated artifacts form the input for SPIN, which performs the verification. Knapp and Wuttke [8] extend the approach of Schäfer et al. [11] to accommodate UML 2.0 sequence diagrams. Their encoding focuses on integrating many language concepts, while we present an encoding suitable for our testing use case.

Another, more widely related research area is the synthesis of state machines from sequence diagrams. Synthesis aims at automatically deriving design models from requirements given as scenarios, as described by Whittle and Schumann [14]. An extension of the latter synthesis algorithm is proposed by Grønmo and Møller-Pedersen [13] by considering also combined fragments in sequence diagrams. The synthesis of model transition systems from scenarios is discussed by Uchitel et al. [12] who also consider safety properties besides scenarios. Common to all these approaches is that the consistency between the scenarios and the state machines are given by construction. However, the synthesis rules may form an important input for the extension of our approach.

5 Discussion and Future Challenges

In this paper, we discussed the use of model checking to detect errors in multi-view system specifications expressed with UML diagrams. We employ sequence diagrams to model test cases, which express forbidden scenarios with neg fragments. As this allows us to perform testing on the level of models, modelers remain within one level of abstraction. Our current prototype is a proof of concept and restricted to the modeling language elements discussed in this paper. Yet, it serves as test bed for various interesting application scenarios. In future work, we plan to integrate positive scenarios in sequence diagrams and additional constructs of state machines, like hierarchies, asynchronous communication, or transition guards, into our framework. Also, other techniques to assemble the information of the sequence diagram and more advanced encodings (including model checkers other than SPIN) will be considered. Further, we intend to compare our encoding with the one of Knapp and Wuttke [8] with respect to scalability and ease of information extraction. We conclude with lessons learned from building our prototype.

Variations in Semantics. The UML standard's informal definition of its diagrams' semantics leaves much room for varying and even contradicting interpretations. For example, a scenario modeled by a sequence diagram describing the interaction of a set of parallel state machines may be interpreted such that either (i) at least one execution path over the set of state machines must satisfy the scenario, (ii) all possible execution paths must satisfy the scenario, or (iii) the occurrence of the scenario's first element implies the occurrence of all subsequent elements on *all* execution paths. By its very nature, the encoding provides one such interpretation that has to eliminate all semantic variation points. This in turn requires a rigorous formalization of the UML standard which should incorporate the smallest set of unambiguous constructs that retain a maximum of the UML's expressiveness. Presently, we started from a simplified version of UML with a concise semantics, but for more language features we will consider works on UML formalization [1, 3].

Incomplete Information. In general, models do not describe a system in full detail, but capture only certain aspects. This way, the modeler is not distracted by temporarily irrelevant details. For building an executable system, the missing information is then gathered in multiple refinement steps, eventually at the code level. For automated testing, this kind of information may be necessary and has therefore to be collected.

State-Space Explosion. The most significant problem in model checking is the large state space to be searched. To shrink the state space, techniques like partial order reduction have been proposed, where equivalent traces are considered only once. Although implemented in model checkers like SPIN, we assume that such optimizations may be performed at the encoding level by exploiting particularities of the modeling language.

Co-Evolution of Code. So far, we have treated sequence diagrams as a visualization of safety properties. Alternatively, sequence diagrams may be used as visualizations of excerpts of a program. Then, the role of sequence diagrams and state machines is inverted, and sequence diagrams are verified against the state machine. In this manner, we shift the focus to the detection of inconsistencies between the model and the code, which may be introduced due to the evolution of the software system.

Presentation Issues. When a model checker determines that a specification is not satisfied, it returns a counterexample, which explains the cause of the problem. Providing an adequate representation of the counterexample, e.g., in the concrete syntax of the employed modeling language, is indispensable for user-friendliness.

References

1. F.S. de Boer, M.M. Bonsangue, M. Steffen, E. Brahm. A Fully Abstract Semantics for UML Components. *FMCO*, 2004.
2. C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2008.
3. M. Broy and M. Cengarle. UML Formal Semantics: Lessons Learned. *SoSyM*, 10(4), 2011.
4. J. Rivera, J. Romero, A. Vallecillo. Behavior, Time and Viewpoint Consistency: Three Challenges for MDE. *MoDELS 2008 Workshops*, pages 60–65, Springer, LNCS 5421, 2009.
5. A. Cimatti, S. Mover, and S. Tonetta. Proving and Explaining the Unfeasibility of Message Sequence Charts for Hybrid Systems. *FMCAD*. 2011
6. OMG. Unified Modeling Language (UML), Superstructure V2.4.1. `http://www.omg.org/spec/UML/2.4.1/`, August 2011.
7. G. J. Holzmann. The Model Checker SPIN. *TSE*, 23(5):279–295, 1997.
8. A. Knapp and J. Wuttke. Model Checking of UML 2.0 Interactions. *Models in Software Engineering*, pages 42–51, Springer, LNCS 4364, 2006.
9. X. Li, J. Hu, L. Bu, J. Zhao, and G. Zheng. Consistency Checking of Concurrent Models for Scenario-Based Specifications. *SDL*, pages 298–312, Springer, LNCS 3530, 2005.
10. P. Pelliccione, P. Inverardi, and H. Muccini. CHARMY: A Framework for Designing and Verifying Architectural Specifications. *TSE*, 35(3):325–346, 2008.
11. T. Schäfer, A. Knapp, and S. Merz. Model Checking UML State Machines and Collaborations. *ENTCS*, 55(3):357–369, 2001.
12. S. Uchitel, J. Kramer, and J. Magee. Synthesis of Behavioral Models from Scenarios. *TSE*, 29(2):99–115, 2003.
13. R. Grønmo, and B. Møller-Pedersen. From UML 2 Sequence Diagrams to State Machines by Graph Transformation. *JOT*, 10(8):1–22, 2011.
14. J. Whittle, and J. Schumann. Generating Statechart Designs from Scenarios. *ICSE*, pages 314–323, ACM, 2000.