Using UML Environment Models for Test Case Generation

Maritta Heisel[1], Denis Hatebur[1,2], Thomas Santen[3], and Dirk Seifert[4]

[1] University Duisburg-Essen, Faculty of Engineering, Department of Computational and Cognitive Sciences, Working Group Software Engineering, Germany,{denis.hatebur, maritta.heisel}@uni-duisburg-essen.de
[2] Institut für technische Systeme GmbH, Germany, d.hatebur@itesys.de
[3] Department of Computer Science, Technische Universität Berlin, Berlin, Germany, santen@cs.tu-berlin.de
[4] LORIA – Université Nancy 2, Campus Scientifique BP 239, F-54506 Vandœuvre lès Nancy cedex, dirk.seifert@loria.fr

Abstract. We propose a new method for system validation by means of testing, which is based on environment models expressed as UML state machines. A sun blind control case study serves to illustrate the method.

1 Introduction

Model-based software development proceeds by setting up *models* of the software to be constructed. This approach has proven useful, because it allows developers to first elaborate the most important properties of the software before proceeding with the implementation. Often, software models are also used for code generation. In this case, however, a problem arises: it does not make sense any more to test the software against its models, because these were already used to generate it. We therefore propose to test the software not only against its *specification* (i.e., against the models), but also against its *requirements*, which describe the how the *environment* should behave in which the software will be operating (acceptance testing). For this purpose, we have to set up a model of the environment, too.

In this paper, we describe how UML state machines (with a corresponding support tool TEAGER [SS06]) can be used to realize the described approach in the area of reactive and/or embedded systems. For this kind of system, state machine models are particularly useful. We elaborate on two different testing approaches:

On-the-fly testing: Here, generating and executing test cases is intertwined. This has the advantage that state explosion is not a problem, but the disadvantage that for non-deterministic systems the tests may not be repeatable.

Batch testing: Here, test cases are generated and stored for later execution. This has the advantage that regression tests become possible but the disadvantage that all possible behavior variants must be computed.

In Sect. 2, we introduce Jackson's terminology [Jac01], which clarifies the notions used in the rest of the paper. The sunblind example is presented in Sect. 3. In Sect. 4, we describe our testing approach, presenting different test architectures. In Sect. 5, we

present state machine patterns that help to set up environment models. Some experimental results are given in Sect. 6. Sect. 7 discusses related work, and Sect. 8 summarizes our contributions.

2 Terminology

Jackson's [Jac01] terminology serves to clearly distinguish the different notions that have to be taken into account when developing software:

Machine is the thing we are going to build; it may consist of software and hardware.
Environment is the part of the real world where the machine will be integrated.
System consists of the machine *and* its environment.[5]
Requirements are *optative* statements; they describe how the *environment* should behave when the machine is in action.
Specifications are *implementable* requirements; they describe the machine and form the basis for its construction.
Domain knowledge is needed to transform requirements into specifications. It is expressed as *indicative* statements. We distinguish between facts and assumptions: **Facts** describe what holds in the environment, no matter how we build the machine. **Assumptions** describe things that cannot always be guaranteed, but which are needed to fulfill the requirements, e.g., rules for user behavior.

The domain knowledge D consists of both the facts F and the assumptions A: $D \equiv F \wedge A$. The relation between requirements and specifications is $S \wedge D \Rightarrow R$, i.e., we have to show that if we build the machine such that it satisfies the specification S and integrate it into an environment for which D holds, then the requirements R are satisfied.

3 Example

We illustrate our testing approach with the example of a sunblind control system. The task is to write software that controls a sunblind, taking into account user commands, wind, and sunshine: The sunblind can manually be lowered or pulled up. It is automatically lowered on sunshine for more than one minute. The sunblind can be destroyed by heavy wind, which should be avoided. The environment consists of user, sun and wind.

To illustrate the difference between requirements and specifications and to stress the importance of explicitly modeling the environment, we transform one requirement concerning the sunblind control problem into a specification, making use of domain knowledge.

R1 *The sunblind is not destroyed by wind.*

To make this requirement implementable, we must know when wind can destroy the sunblind, and how a destruction can be avoided:

[5] In the literature, the term "system" is used interchangeably for the software (i.e., the machine) as well as for the machine in its environment. Jackson's terminology establishes a clear distinction.

F1 *Heavy wind for more than 30 sec is destructive.*
A1 *Heavy wind for less than 30 sec is not destructive.*
F2 *If the sunblind is up, it cannot be destroyed by wind.*

Using this domain knowledge, we can replace $R1$ by

R1' *The sunblind is up if there is heavy wind for more than 30 sec.*

because $F1 \wedge\ A1 \wedge F2 \wedge R1' \Rightarrow R1$. Next, we use

F3 *It takes less than 30 sec to pull up the sunblind.*

to obtain

R1" *If there is heavy wind and the sunblind is not up, it is pulled up.*

because $F3 \wedge R1'' \Rightarrow R1'$. Using the facts

F4 *There is heavy wind if and only if the wind sensor generates more than 75 pulses per sec.*
F5 *Turning the motor left pulls up the sunblind.*

we finally obtain the specification

S1 *If the wind sensor generates more than 75 pulses per sec and the last signals to the motor have not been* turn left*, followed by* motor left blocked *and* stop motor*, then the* turn left *signal is sent to the motor.*

(because $F4 \wedge F5 \wedge S1 \Rightarrow R1''$), which is quite different from the requirement we started out with. All in all, we have shown $F1 \wedge F2 \wedge F3 \wedge F4 \wedge F5 \wedge A1 \wedge S1 \Rightarrow R1$.

4 Test approach

How would the sunblind control software (SUT, system under test) be tested? Usually, conformance with the specification would be checked. In our example, we would have to verify that the machine generates the *turn left* signal. However, if the specification was not correctly derived from the requirements, the SUT passes the test nevertheless.

We therefore propose to test the SUT against the requirements. This means that we check whether the sunblind can enter a state where it would be destroyed. Besides detecting errors made in transforming requirements into specifications, testing against requirements allows us to verify that customer needs are satisfied (acceptance test).

In order to test the SUT against its requirements, we need a model of the environment, because the requirements refer to the environment and not to the machine. Much like the SUT, the environment can be modeled using UML state machines. The model explicitly contains the facts and the assumptions about the environment. The environment model consists of adapters and the input event generator: Adapters transform abstract events such as *pull up sun blind* into concrete ones, such as *turn motor left*. The input event generator produces abstract events. To capture stochastic properties of the environment probabilistic state machines of TEAGER can be used. This reduces the number of inadequate test cases.

The requirements are translated into state machines, too. These state machines serve to inform the tester whether a requirement is violated. They observe the stimuli and SUT outputs at an adequate level of abstraction. As shown in Fig. 1, the Test Case Generator component of the tool TEAGER can be used to simulate the environment model and to check the requirements. To calculate test cases, for each tick (1) an abstract stimulus is generated by the Input-Event-Generator in the environment model. Adapters transform

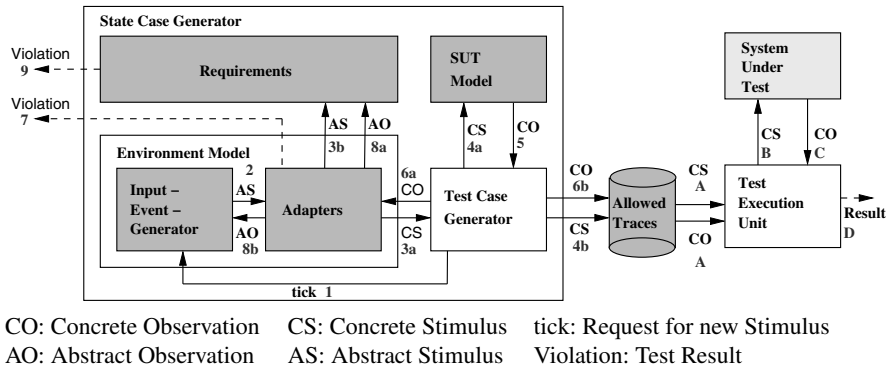

Fig. 1. Test architecture for batch testing.

the abstract stimuli into concrete stimuli for the Test Case Generator (3a) and send the abstract stimuli to the Requirements (3b). The Test Case Generator sends the concrete stimuli to the SUT Model, which determines suitable responses (4a, 5), and it stores the conrete stimuli and the determined concrete observations (4b, 6b). The Adapters transform the concrete observations (6a) into abstract observations that are checked by the Requirements (8a) and used to generate reasonable stimuli (8b, e.g., *isLowered* only after *LoweredSunBlind*). Violations can be detected by checking the requirements (9) and while transforming concrete stimuli into abstract stimuli (7). After the requirements are checked, a new tick (1) is generated. The generated Test Cases can be used to test the SUT with the Test Execution Unit. Concrete stimuli and observations in the allowed traces (A) are used to stimulate the SUT (B) and check the responses (C). Test results (D) are the output of the Test Execution Unit.

Alternatively, the environment model can be directly connected to the SUT, and within the simulated environment the requirements are checked at runtime. In this case no SUT model is necessary. This scenario is especially useful for acceptance tests. Here, regression tests can be performed by saving the generated event sequences. Then, the State Machine Executor recalculates the test events by simulating the environment and checks the observed behavior according to the requirements. The test system architecture – annotated with sample observations and stimuli for the sunblind example and with the execution order – for this "on the fly"-testing approach is shown in Figure 2.

5 Patterns for environment models

Setting up the state machines for the environment model is not a trivial task. However, we can identify different patterns for setting up environment models, especially for expressing requirements as state machines. The overall structure of the state machine consists of parallel regions. That is, the environment model is in all of the parallel machines R_i, *Input-Event-Generator* and *Adapter* at the same time, and the different sub-machines communicate with each other via common events. The left-hand side of Figure 4 shows an example of an input event generator. Note that assumption *A1* (namely, that heavy wind for less than 30 sec is not destructive) is modeled explicitly.

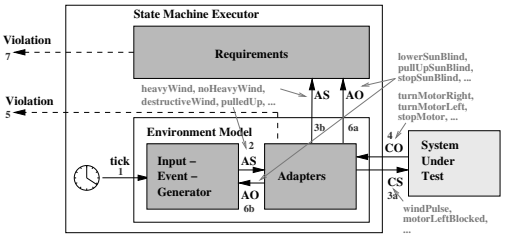

Fig. 2. Test architecture for on-the-fly testing.

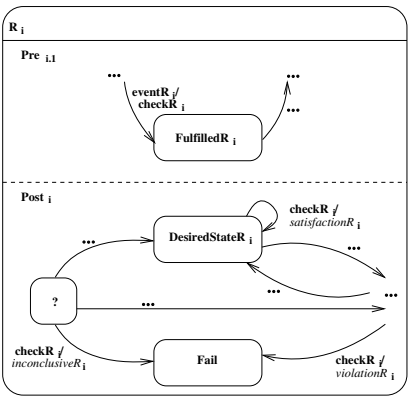

Fig. 3. Patterns for environment models

Moreover, probabilities for the different transitions are given. These can be processed by the TEAGER tool. As an example of an adapter, we present the motor adapter, which transforms concrete observations into abstract ones in the right-hand side of Fig. 4. It specifies how motor commands correspond to events that are visible in the environment.

For modeling requirements, we have developed different patterns, of which we can present only one for reasons of space. The pattern is usable when the requirement has the form "When [$eventR_i$] happens, [controlled domain] should be in [*desired-State*R_i]". Its representation as a state machine is shown in Fig. 3. When the event of interest happens, then the precondition of the requirement is fulfilled, and the event $checkR_i$ is generated. The state machine representing the postcondition contains the desired state and may also contain other states. Only if it is in the desired state, the test passes; otherwise, a violation is determined, or the test is inconclusive. The latter happens, for example, if the actual state of the system is not known. Then, the result of checking a requirement should neither be pass nor fail. In our example, we do not initially know the (physical) state of the sunblind. Hence, we introduced an "unknown state" (denoted by "?") expressing this situation. Checking requirement R_1 in this state yields an inconclusive result.

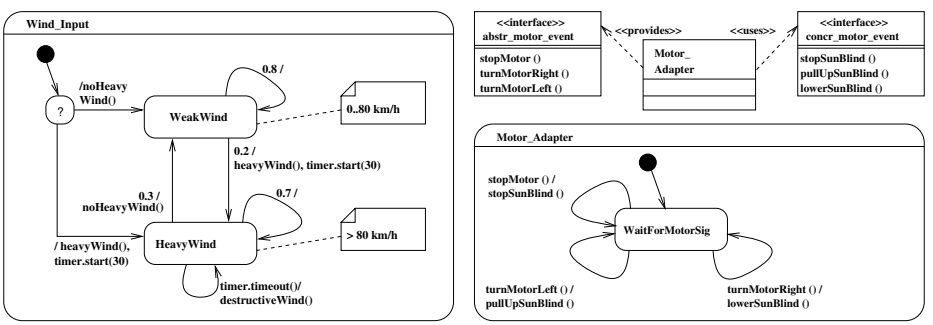

Fig. 4. Input generator and adapter for the sunblind

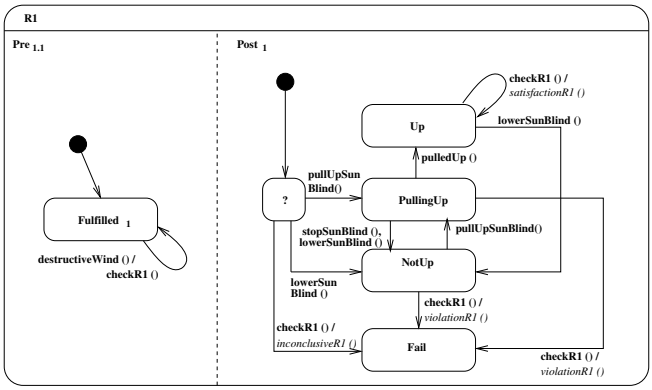

Fig. 5. State machine for requirement R_1

Requirement R_1 of Sect. 3 is an instance of this pattern: whenever there is destructive wind, the sunblind must be up. Figure 5 shows the instantiated pattern. Whenever the event *destructive Wind* occurs, the event *checkR1* is generated. If the sunblind is in state *up*, the requirement is satisfied. Otherwise, it is violated. The *Fail* state corresponds to a state where the sunblind would be destroyed.

All in all, to completely model the sunblind control problem, 4 input event generators, 4 adapters, and 7 requirement state machines have to be set up. All the requirements are instances of patterns.

6 Experimental results

To evaluate our approach, we used the tool suite TEAGER [SS06, Sei08]. The tool suite allows its users to generate and execute test cases or to directly stimulate the SUT. TEAGER logs the stimuli it sends to the SUT and the reactions of the SUT. During execution, these reactions are compared to the pre-calculated possible correct reactions to evaluate the test execution process. Table 1 shows some generation times for several experiments. Here, the number of triggers controls the length of test cases, while the search depth controls the exploration of the model's state space. The results show that no state explosion occurs, which makes the approach applicable in practice.

No. of Test Cases			No. of generated Triggers			Search Depth			Generation Time		
50	100	500	10	10	10	5	5	5	4 sec	5 sec	18 sec
50	100	500	100	100	100	5	5	5	15 sec	32 sec	150 sec
50	100	500	100	100	100	10	10	10	105 sec	180 sec	943 sec

Table 1. Test case generation times for several experiments.

7 Related work

To our knowledge, there neither exist approaches for testing requirements expressed as UML state machines, nor approaches for combining conformance testing on unit testing level with testing requirements on acceptance testing level.

A detailed overview of the fundamental literature for classical formal testing can be found in Brinksma's and Tretmans' annotated bibliography [BT01]. De Nicola and Hennessy [DH84] introduce a formal theory of testing on which Brinksma [Bri88] and Tretmans [Tre96] build approaches to derive test cases from a formal specification. In contrast to our work, these approaches assume that a testing process can communicate synchronously with the system under test. The developed tool TorX (`fmt.cs.utwente.nl/tools/torx`) also allows conformance testing of reactive systems.

Belli at al. (see [BH07] and the work cited there) base their testing methodology on a variant of state machines. In contrast to our approach, they do not test against requirements, but against a fault model that has to be set up explicitly. Moreover, they do not execute the state machines directly, but represent them as event sequence graphs.

Auguston et al. [AMS05] use environment models for test case generation. In contrast to our approach, they do not use state machines, but attributed event grammars.

While these works have their merits, we think that the combination of environment models and UML state machines for testing is a particularly attractive one.

Besides the work from academia, an increasing number of CASE Tool manufacturers offer components for model based testing. I-Logix Rhapsody, for example, offers two add-on products, Test Conductor and Testing and Validation, for testing state machines (`www.telelogic.com`). Simulink Verification and Validation generates test cases in Simulink and Stateflow, and measures test coverage for Statecharts (`www.mathworks.com`). Conformiq Software Ltd. offers a *Test Generator* which accepts "extended UML state charts" as the model of the system under test for dynamic testing (`www.conformiq.com`). *AsmL 2* by Microsoft provides an executable specification language based on the theory of Abstract State Machines (`research.microsoft.com/fse/asml`). The AsmL Test Tool supports parameter generation and test sequence generation based on interface automata.

8 Conclusion

We have developed a novel approach to testing reactive and embedded systems, based on environment models and using UML state machines. The approach is supported by the TEAGER tool. Using Jackson's terminology, we have defined uniform architectures and procedures for on-the-fly as well as batch testing that have the following characteristics:

- Requirements, facts, and assumptions are modeled explicitly.
- We have defined patterns for the different state machines: For requirements, a parallel state machine is set up for each precondition. When all preconditions are fulfilled, the postcondition is checked. Input generators and adapters also consist of parallel state machines, one for each item of the environment that generates stimuli or receives observations, respectively.

– Once these models have been set up manually (but systematically), the tests are performed automatically, using the tool TEAGER.

Our approach has the following advantages: When **requirements change**, in the test case generator only the state machine describing those requirements must be changed. On the other hand, changed requirements will lead to a new SUT model. The new SUT model can be validated while the test cases are generated. Modeling the facts and assumptions about the environment supports the **validation of requirements**. For example, it can be discussed if heavy wind can be destructive to the sunblind within the time that a sunblind needs to be pulled up. Although the model of the environment has nearly the same complexity as the model of the machine, a structured approach to develop the environment model helps to **identify subproblems** that can be treated separately. Sometimes, states like "sunblind destroyed" are not modeled in the machine, but must be modeled in the environment to verify that this state cannot be reached. On the other hand, states can be left out in the environment model if the machine implements features that are not part of the requirements. The same environment model can be **(re-)used** for a sunblind control that can stop at an arbitrary height and a sunblind control that can only open or close the sunblind completely. Modeling the environment adds **diversity** to the development process and thus helps to avoid that the same mistake occurs for test development and SUT development. This is because the test developers, who model the environment, must think in terms of the environment rather than the SUT behavior. In the environment model, a reasonable test case selection strategy can be defined, so that **no inadequate test cases** are generated. Atypical behavior can be identified and tested using a dedicated environment model.

References

[AMS05] M. Auguston, J. B. Michael, and M.-T. Shing. Environment behavior models for scenario generation and testing automation. In *Proc. First International Workshop on Advances in Model-Based Testing, ICSE 2005*, pages 1–6. ACM, 2005.

[BH07] Fevzi Belli and Axel Hollmann. Holistic testing with basic statecharts. In W.-G. Bleek, H. Schwentner, and H. Züllighoven, editors, *Software Engineering 2007 – Beiträge zu den Workshops*, LNI 106, pages 91–100. Ges. f. Informatik, 2007.

[Bri88] Ed Brinksma. A Theory for the Derivation of Tests. In *Protocol Specification, Testing and Verification*. North-Holland, 1988.

[BT01] Ed Brinksma and Jan Tretmans. Testing Transition Systems: An Annotated Bibliography. *Lecture Notes in Computer Science*, pages 187–195, 2001.

[DH84] Rocco De Nicola and M. C. B. Hennessy. Testing Equivalences for Processes. *Theoretical Computer Science*, pages 83–133, 1984.

[Jac01] Michael Jackson. *Problem Frames. Analyzing and structuring software development problems*. Addison-Wesley, 2001.

[Sei08] Dirk Seifert. The TEAGER Tool Suite. test execution and generation framework for reactive systems, 2008. `swt.cs.tu-berlin.de/~seifert/teager.html`.

[SS06] Thomas Santen and Dirk Seifert. Teager - Test Automation for UML State Machines. In B. Biel, M. Book, and V. Gruhn, editors, *Software Engineering 2006*, LNI P-79, pages 73–83. Gesellschaft für Informatik, 2006.

[Tre96] Jan Tretmans. Test Generation with Inputs, Outputs and Repetitive Quiescence. *Software–Concepts and Tools*, 17(3):103–120, 1996.