Modeling for Automated Test Generation – A Comparison

Hartmut Lackner, Bernd–Holger Schlingloff

Fraunhofer Institute for Computer Architecture and Software Technology FIRST
Berlin, Adlershof
{hartmut.lackner, holger.schlingloff}@first.fraunhofer.de

Abstract: In this contribution, we compare and analyze different methodologies of modeling for test generation. As an example, we use an industrial requirement specification given in natural language, which describes a safety function in a hybrid car. We model these requirements with three different paradigms and languages: as the specification imposes several timing constraints, we choose Abstract State Machines, Timed Automata and UML2 State Machines to formalize the given requirements. From these models, we employ different tools for generating test cases, and compare the resulting test suites with respect to coverage and fault detection capabilities. We discuss the process of designing the models and the implications for professional software testing.[1]

1 Introduction

Model-based software engineering, and, in particular, model-based testing (MBT) has received much attention as a state-of-the-art method for creating and testing embedded software. However, MBT is not yet a widespread standard method in industry. There are several reasons why model-based test generation is not as widely used as it could be. One of them is that initially it may be difficult to choose an adequate modeling notation and tool set. Even though UML2 claims to be *the* unified modeling language, it has several profiles and extensions. Moreover, there are other notations which are not included in this unification but still used for engineering embedded systems. A natural question is to investigate which methods are apt for various types of systems. However, to our knowledge there is no published result where different MBT approaches are compared and evaluated on a practical example.

In this contribution, we use three different modeling languages to formalize the same specification. This results in three different test models and, consequently, in different test suites. We compare the effectiveness of the various approaches. Even though such a comparison can never be "complete", it can provide assistance to choose an adequate method for a particular project, depending on the characteristics of the system under test.

[1]This work was partially supported by the BMBF Project SPES2020 - Software Platform Embedded Systems, FKZ 01|S08045J

2 Design of Test Models

Abstraction and automated generation of software artifacts are two key principles in software engineering. Research has produced various notations for programming, starting from assembler, high-level languages, to object-oriented and functional languages. Today, code can even be generated from system models, which forms another level of abstraction. However, the design of models currently cannot be automated. It is performed by specialists in modeling, which usually are proficient in a particular modeling paradigm. Modeling for test generation, however, is not as well-understood.

A test generator converts models in executable test cases. That is, the focus of test models is not the internal structure and workings, but the externally visible behavior of the SUT. It is often difficult here to find the right level of abstraction: For example, if the natural language specification states that "if the button is pushed, the light is switched on", then an implementation model will contain the inner workings. This may include writing internal data, making calls to the operating system (OS) (e.g. preparing device drivers), and setting up a listener for the button. Afterwards it may do some post processing (clean up) as well.

Depending on the test-target either all of the above mentioned activities shall be tested or just a subset of them. A very abstract test only checks whether the system turns on the light whenever the button is pressed, by e.g. observing the method call for turning on the light. A more in-depth test checks in addition to this whether the correct device drivers are requested from the OS, since the OS can be seen as one part of the system's environment. This example shows a general problem with abstract test models: it may not be possible to deduce how to observe the SUT in order to cast a verdict for a test run. Since there is only a limited amount of information about the system's internals, it can be hard to decide whether the observed behavior conforms to the intentions.

In this contribution, we focus on the design of test models. We design three models in different languages for a given requirements specification, which is written in natural language. The comparison of different modeling methods and languages is similar to a comparison of different programming languages. It may be difficult to find fair balanced criteria, since most metrics (execution time, memory footprint, lines of code, modularity, cohesion, programming efficiency, etc.) can be twisted in favor of any given tool for a particular language. Nevertheless a comparison makes sense as it can give a general flavor of the particularities, strengths, and advantages of each method under consideration.

3 ECU Case Study

In this section, we present the case study. The system under test (SUT) is a electronic control unit (ECU) implementing a safety function in a hybrid car. The ECU has one input sensor and one output channel to the engine. The input sensor reads the pressure of a pipeline connected to the engine. The ECU may read the following values from the sensor: "invalid", "low", "high", and "too high". Via the output channel, the ECU can stop the car's engine. Initially, the pressure is "invalid" and the engine is on.

The behavior of the ECU is given by the following five rules defined by an industrial partner:[2]

(R1) If the pressure sensor is more than 5s (short delay) “too high” a quick stop occurs and the engine is shut off.

(R2) If the pressure sensor was invalid and switches to valid again and during the following 5s the pressure is not low a long delay of 20s is activated. In this state a “too high” triggers the quick-stop after 20s (long delay). (Long delay replaces the initial short delay).

(R3) If the pressure is “low” then the 5s (short delay) is valid again.

(R4) If the valid pressure switches to invalid the 5s (short delay) is valid again.

(R5) If during the delay the valid pressure is not “too high” for more than 0.3s the delay timer is reset to start a new delay period.

This specification uses a 5s “short delay” and 20s “long delay”. In the following, we refer to these delays as short and long timing periods.

4 Modeling

In order to model the above requirements, we use three different but common modeling languages. We chose one representative from each class: Abstract State Machines for programming-near specification, Timed Automata for real-time specification, and UML2 State Machines as a industry standard. For each model we give a short introduction to the modeling language and then highlight our design decisions.

4.1 Abstract State Machines

An abstract state machine (ASM) is a state machine operating on arbitrary data structures. Each operation may, but does not need to, have a guard and an effect [GRS05]. A guard limits the applicability of the operation to a subset of states. If no guard is specified, the operation is always applicable. An effect specifies how the operation changes the values in the data structure, when the operation is executed. As a change of the values denotes a different state, eventually an operation leads to a new state.

We chose Microsoft Visual Studio with the Spec Explorer plug-in for designing the model in C# [CGN$^+$05]. Spec Explorer offers the ability to explore the ASM, visualize the resulting state space as a finite “exploration graph”, and generate test cases.

[2]The rules have been very slightly modified for the purpose of this publication. However, we neither added nor clarified any ambiguities in this specification.

```
[Action("E(Delay.Short)")]
static void ExpireShortDelayTimer()
{
    Contracts.Requires(!timers[1].isExpired());
    Contracts.Requires(timers[1].getDelay() == Delay.Short);
    Contracts.Requires(timers[2].isExpired());
    timers[0].setExpired(true);
    timers[1].setExpired(true);
    isStopped = true;
    engineRunning = false;
}

[Action("E(Delay.Long)")]
static void ExpireLongDelayTimer()
{
    Contracts.Requires(!timers[1].isExpired());
    Contracts.Requires(timers[1].getDelay() == Delay.Long);
    timers[0].setExpired(true);
    timers[1].setExpired(true);
    isStopped = true;
    engineRunning = false;
}
```

Figure 1: The two operations for expiring the timer of rule (R1).

The state of the model is designed as follows: an enumeration holding the current pressure value, a boolean denoting the engine's status and a collection holding three timers. The timers correspond to the temporal statements in the rules (R1), (R2) and (R5). Since ASMs do not have built-in concepts for real time, we had to implement our own timer class.

```
[Action("T(newPressure)")]
static void SetPressureFromRunningTimer1(Pressure newPressure)
{
    Contracts.Requires(pressure != Pressure.Invalid);
    Contracts.Requires(!timers[1].isExpired());
    Contracts.Requires(!isStopped);
    Contracts.Requires(newPressure != pressure);
    if (newPressure == Pressure.Low || newPressure == Pressure.Invalid)
    {
        timers[1].setDelay(Delay.Short);
        timers[2].setExpired(true);
    }
    if (newPressure == Pressure.Low
            && timers[2].isExpired() && timers[1].getDelay() == Delay.Long)
        isStopped = true;
    timers[0].setExpired(false);
    if (newPressure == Pressure.TooHigh) timers[0].setExpired(true);
    if (newPressure == Pressure.Invalid)
    {
        timers[1].setExpired(true);
        timers[0].setExpired(true);
    }
    pressure = newPressure;
}
```

Figure 2: The operation for rule (R5).

The timer of (R1) can be started with two different timing periods. Therefore, we designed two operations for this timer, each handling the expiration of one timing period. The first operation reflects the expiration with the short timing period and the second the expiration with the long timing period (see Fig. 1). The statements beginning with `Contracts.Requires` denote the preconditions of the operation. The preconditions for operation `ExpireShort DelayTimer` demand that the timer for (R1) (`timer[1]`) is running, the timing period (`getDelay()`) of the timer is set `short` and the timer of

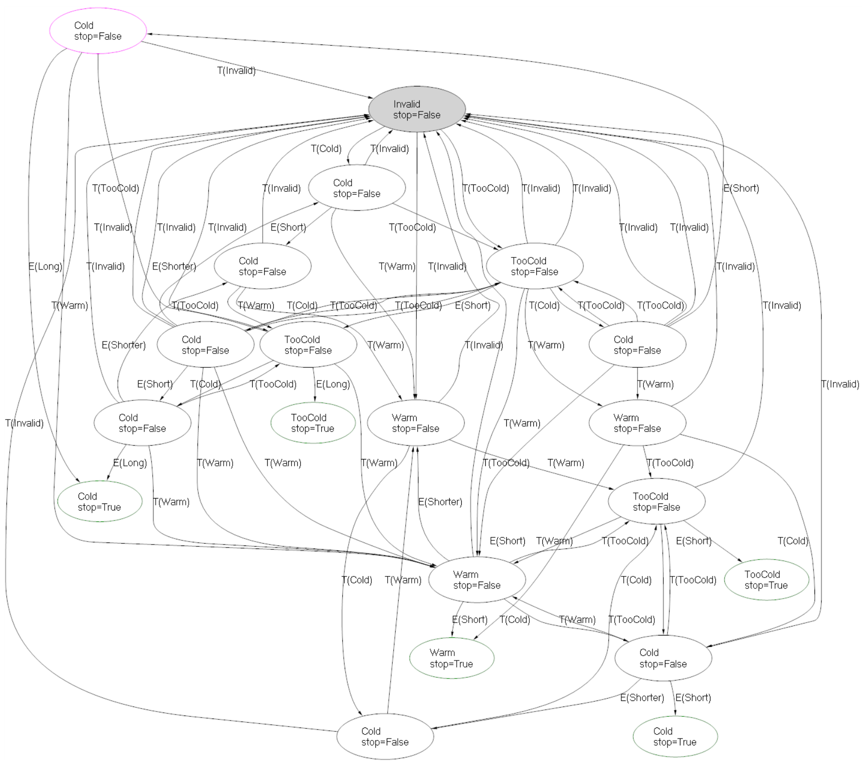


Figure 3: Finite graphical representation of the ASM's state space.

(R2) is not running. If the timer of (R2) would be running, the two timers would expire in the very same moment or the timer of (R1) would expire later than the timer of (R2). We decided to prioritize the timer of (R2) if both timer expire at the same moment and hence, block the timer of (R1) in this case.

The precondition for the operation `ExpireLongDelayTimer` is defined similar to that of operation `ExpireShortDelayTimer`. The precondition for blocking the operation when the timer of (R2) runs is obsolete here, since in this situation the two timers never run in parallel. The effect of both operations is the same: when executed, the operation sets the timers to expired and then stops the engine. The design of the operations for expiring the other two timers in (R2) and (R5) is similar.

To complete the model, we then designed three operations for setting the pressure. The first operation handles all pressure changes from any value but "invalid" to any other value while no timer is running (rule (R1), (R3), and (R4)). The second operation handles the case when the pressure is "invalid" or the timer for (R2) is running (rule (R2)).

The last operation (`SetPressureFromRunningTimer1`) is applicable when the pressure changes while the timer of (R1) is running (rules (R1), (R3), (R4), and (R5)) (see Fig. 2). This operation is only executable when the current pressure value is not "invalid",

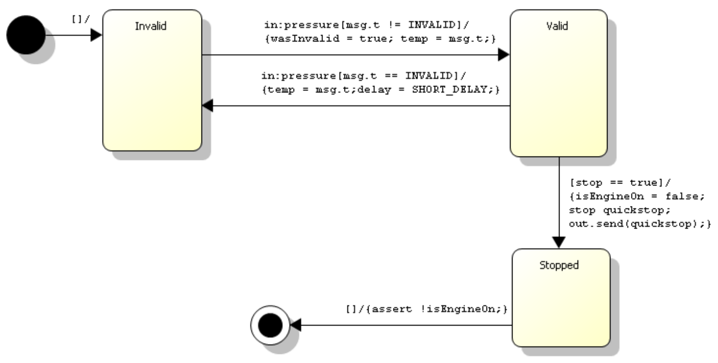

[]/
Invalid
in:pressure[msg.t != INVALID]/
{wasInvalid = true; temp = msg.t;}
Valid
in:pressure[msg.t == INVALID]/
{temp = msg.t;delay = SHORT_DELAY;}
[stop == true]/
{isEngineOn = false;
stop quickstop;
out.send(quickstop);}
Stopped
[]/{assert !isEngineOn;}

Figure 4: Top level of the UML2 State Machine.

the timer of (R1) as well as the engine are running, and the new pressure differs from the current value. The operation has multiple effects, which occur under different conditions. The operation changes the timing period of timer 1 to “short” according to (R3) and (R4), if the new pressure value is “low” or “invalid”. Then the operations starts the timer of (R5), just to stop it right after if the pressure is “too high”. If the new pressure value is “invalid”, the timers for (R1) and (R5) are stopped. Finally, the new pressure value is stored.

The complete model has about 150 line of code. The result of the state space exploration is given in figure 3.

4.2 UML2 State Machines

A UML2 State Machine Diagram is the graphical representation of a State Machine. It essentially consists of a finite number of states and transitions [UML05]. In contrast to ASMs, UML2 State Machines have a more complex notion of a state. A state may contain several submachines, which again may contain states. A transition may contain a trigger, a guard and an effect. In particular, a transition can be triggered after a certain time.

UML2 State Machines describe the behavior of a class of objects, usually modeled as a class diagram. The class diagram declares attributes and operations, which are referenced by the State Machine. It is difficult to identify the right balance between variables and states, because states can be implemented as variable valuations and vice versa.

The states of our model reflect the values of the pressure sensor. We model the specification by distinguishing two states: one for the “invalid” pressure value and the other for all valid pressure values (“low”, “high”, “too high”). This step facilitates the modeling of (R2), (R3), and (R4), because in this case a single transition is sufficient to model the change from a valid pressure value to “invalid” (see Fig. 4).

The state `Valid` is refined to a submachine containing one state for each pressure value plus an intermediate state (see Fig. 5). In the intermediate state, the long timing period

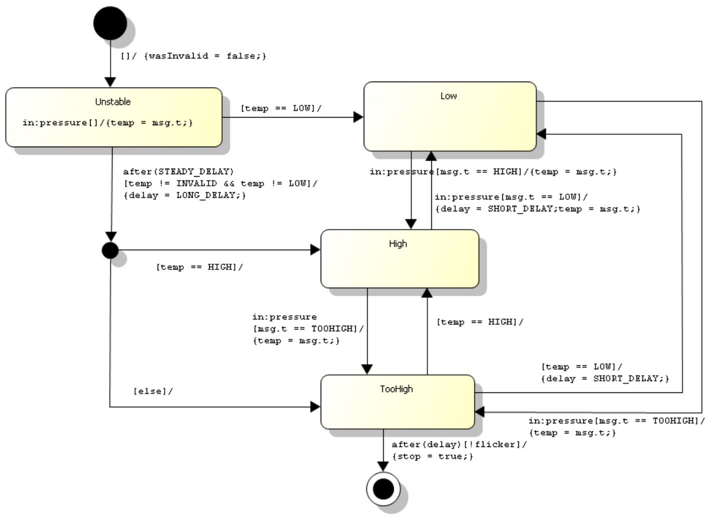
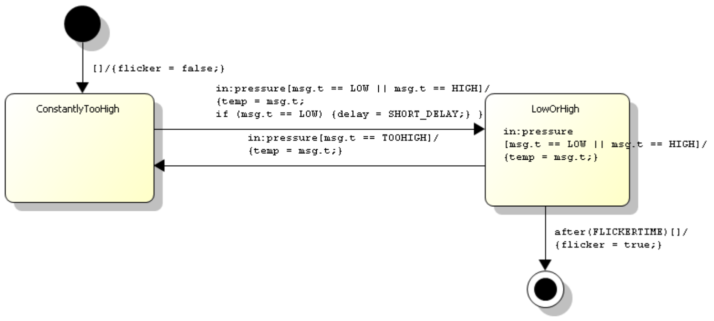


Figure 5: Refined “Valid”-State: States for pressure values and an intermediate state.

is activated, if the pressure does not switch to “low”, according to (R2). The timer for (R1) is modeled by adding a final node to the submachine and connect it via a transition with a parameterizable timer-trigger to the state “too high”. When the timer expires, the submachine terminates, and the upper level State Machine will stop the engine before it also terminates.

For modeling (R5), the state “too high” is refined into the submachine shown in Fig. 6. The submachine allows the pressure to alternate between all the valid pressure values, as long as the value is not “low” or “high” for more than 0.3 seconds. Therefore, the submachine consists of the states “too high” and “not too high” as these two situations can be distinguished in (R5). If the pressure is “low” or “high” for more than 0.3 seconds, the submachine terminates and returns the control to the enclosing state `Valid`.



[]/{flicker = false;}
ConstantlyTooHigh
in:pressure[msg.t == LOW || msg.t == HIGH]/
{temp = msg.t;
if (msg.t == LOW) {delay = SHORT_DELAY;} }
in:pressure[msg.t == TOOHIGH]/
{temp = msg.t;}
LowOrHigh
in:pressure
[msg.t == LOW || msg.t == HIGH]/
{temp = msg.t;}
after(FLICKERTIME)[]/
{flicker = true;}

Figure 6: Refined “Too High”-State.

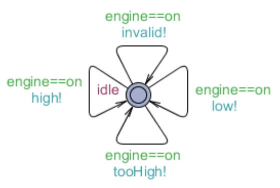


Figure 7: The sensor TA.

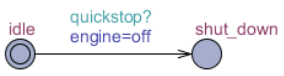


Figure 8: The engine TA.

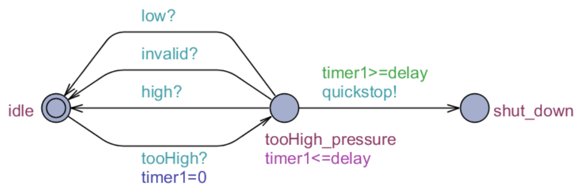


Figure 9: Rule (R1) for stopping the engine.

4.3 Timed Automata

Timed Automata (TA) are a concept from theoretical computer science where classical finite automata are extended by clocks [AD94]. In existing tools, one can also declare variables over certain data structures (bounded integers, bounded arrays and boolean types). A state in a timed automaton is defined by the currently active location(s), the valuation of clocks, and the variable valuation. TA running in parallel may communicate with each other via channels, which either provide synchronized peer-to-peer communication or asynchronous broadcasts to multiple peers.

In the modeling, we tried to stick as close to the specification as possible. Therefore, we decided to model each of the rules (R1) – (R5) as a separate automaton. The environment is modeled by two additional automata: one for the pressure sensor and one for the engine's status (see Fig. 7 and 8). Since the range of the pressure sensor comprises only four values, it can be modeled by one broadcast channel for each value. Additionally, we use a peer-to-peer channel for sending the quickstop command to the engine.

The variable declarations on the top level are as follows: The only variable the TA have to share is the timing period for the timer in (R1). The behavior of the system is modeled as follows: (R1) is formalized by three locations, denoting an idle timer, a running timer and an expired timer (Fig. 9). The system reacts on changing pressure values by switching between the idle and the running timer. If the timer expires the stop message is send.

(R2) is formalized by three locations, which denote again an idle condition, a condition where the pressure value is "invalid" and a condition for the pressure being not "low" when the pressure was invalid beforehand (Fig. 10). In the location `not low pressure` the decision is made whether the timer of (R1) is untouched or set to a long timing period.

The TA for (R3) (in Fig. 11) and (R4) (in Fig. 12) consist of one location and one self-loop only. They set the timer from (R1) to its short timing period.

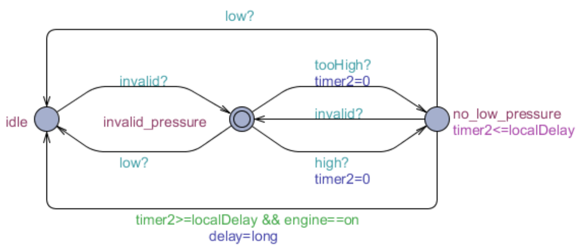


Figure 10: Rule 2 for waiting for warm.

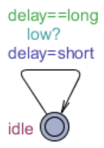


Figure 11: (R3)

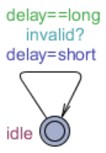


Figure 12: (R4)

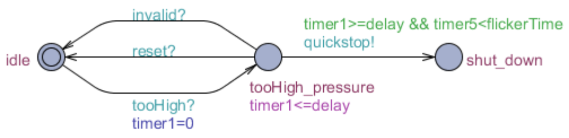


Figure 13: Rule (R1): Updated timed automaton.

Modeling (R5) is more intricate, since it non-trivially interferes with (R1): the timer of (R1) must not be reset, though the pressure is not "too high" anymore. Rule (R5) allows the pressure to change from "too high" to any other value but "invalid", if it changes back to "too high" within 0.3 seconds. Thus, we have to find a way to inhibit the TA for (R1) interrupting this timer if the change occurs. The solution is to synchronize both by a newly introduced transition, which actively stops the timer of (R1) (see Fig. 13). The existing transitions for "low" and "high" have to be removed. Now the TA for (R1) switches to the idle location only if the pressure is "invalid" or it receives a "reset" message.

With this modification of the automaton for (R1), rule (R5) can be modeled with three locations, one of them being an idle location (see Fig. 14). The others denote that the pressure is "too high" and that the pressure has switched from "too high" to "high" or "low". The decision whether the timer of (R1) is stopped and reset is made in the latter location and depends on the expiration of the local timer. If the local timer expires, the timer of (R1) is stopped and reset. Otherwise, if the pressure changes back to "too high" prior to the local timer's expiration, the timer of (R1) is not interrupted.

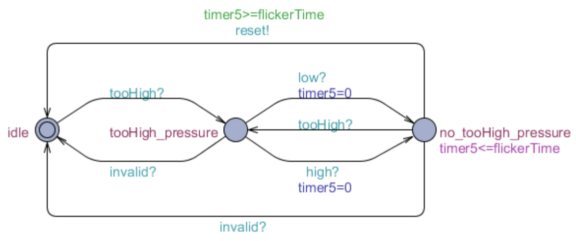


Figure 14: Rule (R5) for enabling flicker without interrupting the timer of (R1).

5 Evaluation

To compare and evaluate the models, we used three different tools to generated test cases:

- For the ASM models we used Spec Explorer, which is a Visual Studio Power Tool and has been extensively used in internal applications at Microsoft [CGN$^+$05].
- For UML2 State Machines, there are a number test generators, see [GNRS09]. We used Conformiq Designer, which is widely used in the telecommunications domain.
- For timed automata, UPPAAL [BLL$^+$96] is a well-known simulation and verification environment; we used CoVer as an offline test generator for UPPAAL [HP07].

Test suites can be compared statically and dynamically. As a static comparison, we looked at statistical information on the number of test cases and test steps for each test suite. However, these figures are *per se* not very meaningful, since the size of a test suite is not necessarily related to its fault detection capabilities. Therefore, we built an implementation to be able to execute and compare the dynamic effectiveness of the test suites.

We mutated the SUT with the tool JUMBLE [IPT$^+$07]. With all its mutation capabilities turned on, JUMBLE generates 68 mutants for our SUT. These mutants include so-called "masked mutants", which show no observable difference in behavior from the original SUT and therefore can never be detected by any test suite. Each test suite was executed with the original implementation and with all of its mutants. From the test results, JUMBLE calculated the mutation score, which is the percentage of detected mutants.

As a further evaluation criterion, we used the tool CodeCover to measure the code coverage of the test suites in the SUT [Cod12]. In particular, we measured the Ludewig term coverage, which is similar to MC/DC, but has some advantages: the Ludewig term coverage is defined for partly coverage, and for each individual term value set the term coverage can be determined - both is not possible for MC/DC.

Spec Explorer was configured to generate "long" test cases and to accept any state as an end state. We used the standard built-in traversal algorithm for test generation, which covers all transitions on the exploration graph. With this configuration, Spec Explorer generated 101 test steps in seven test cases, which achieved a mutation score of 50 % and 100 % term coverage.

For Conformiq Designer we applied two configurations. For the first test suite we used the standard configuration which covers all transitions on the UML2 State Machine level. Only test cases ending in the final state are accepted. With theses settings Conformiq Designer generated a test suite with nine test case and a total of 30 test steps. This test suite is the shortest by means of test steps and yields a mutations score of 35 % and 78.1 % term coverage in this setting.

For the second configuration of Conformiq Designer, we choose the advanced coverage criteria "2-Transition Coverage" and "Implicit Consumption". 2-Transition Coverage covers every pair of two subsequent transitions at least once. Implicit Consumption tests if the system correctly ignores messages that are not handled on any transitions going out from a state. Given this configuration, Conformiq Designer generated a test suite consisting of 34 test cases with 141 test steps, that achieved 58 % on the mutation score and 100 % term coverage.

Finally, we used the tool CoVer to generate a test suite from the UPPAAL timed automata. For CoVer one has to design one's own coverage criterion by defining a so-called "coverage automaton". We constructed such a coverage automaton that equals to transition coverage on the TA and accepts any state as an end state. With these parameters, CoVer generated 25 test cases with 70 test steps altogether, which achieve a mutation score of 36 % and covered 100 % of the terms.

6 Results and Lessons Learned

In this section, we present the results and lessons learned during the study. First, we discuss similarities in the approach of designing the models. Then we compare the test generation capabilities of the tools and their impact on a professional software engineering process.

6.1 Design

An obvious difference in the three modeling approaches is that Spec Explorer uses a textual language (C#), whereas Conformiq Designer and UPPAAL work with (the XML representation of) graphical objects. Thus, Spec Explorer may be more comfortable for experienced programmers, and the other two may be more suited for application engineers. However, with the coming of age of model transformation technologies, these differences might become less and less important.

A more profound difference is in the structuring concepts which are provided by the language. In UPPAAL, currently only a parallel composition of automata is possible, whereas UML2 State Machines can also be structured in a hierarchical way. Spec Explorer offers the full modularity features of the C# language. However, these structuring mechanisms easily lead to complex models, where the test generation may suffer from the state explosion problem. Thus, it is important to be aware of the complexities during the modeling.

The modeling languages differ also in their capabilities for modeling real-time aspects. Timed automata were conceived as a means for specifying real-time systems, they offer an intuitive concept of clocks. However, the complexity of analyzing a timed automaton increases exponentially with the number of clocks it contains. Basic UML2 defines the class `TimeEvent` of delays, which can be used as a trigger in a transition. There are several extensions (e.g., MARTE [Gro08]) offering more elaborate modeling elements for real-time. In Spec Explorer, there are no predefined constructs for timing. As we have shown above, timers can be easily defined in C# with an appropriate library module.

We tried to develop the different models as independently from one another as possible. However, there are remarkable similarities in the three models: In the UML2 approach we designed three State Machines, which map directly to the three operations for altering the pressure value in the ASM. Furthermore, the UML2 submachine for R5 directly corresponds to the timed automaton for R5, as the states can be mapped to the locations. Of course, we cannot guarantee that design decisions for one model did not have an impact on models which were conceived later on. However, we presume that the similarities point at an underlying "common core" of the models. Similar as algorithms can be explained in a "pseudo programming language" and implemented in various programming languages, there might be an upcoming "pseudo modelling language" to describe the common core of a model.

The timed automata modeling follows the structure of the requirements more explicitly than the other two models: for each rule, we designed an automaton realizing this rule. This procedure could have been mimicked with the other modeling formalisms as well. It supports an incremental development process of models. However, as we have seen with (R5), it is not always possible to follow such an incremental paradigm. In order to build an automaton for (R5) we were forced to modify the previously defined automaton for (R1). A reason for this is that the natural language requirement (R5) nontrivially interferes with (R1). Thus, it is advisable to study and explicate the cross-references within the requirements before starting to model a system.

6.2 Test Generation

The results of executing the test suites show no significant correlation between the size of the generated test suites and their fault-detection capabilities. Even with a low number of steps one test suite achieved a decent mutation score, whereas another one which was more than twice as large had almost the same score. Though this was to be expected, it shows that the size of a test suite is no indicator of its quality, and that it is important to manage the test generation process.

With appropriate tuning all three employed tools were able to produce good results. Configuration facilities, however, greatly differ between the various tools. The options range from predefined coverage criteria, model annotations, to the definition of custom generation algorithms. The most convenient solution to configure a test generation algorithm is to choose from a list of predefined coverage criteria. Conformiq Designer offers various criteria to choose from, including transition coverage, boundary value analysis, and all-paths. Spec Explorer offers only one built-in coverage criterion, namely transition coverage on the exploration graph. For advanced users, Spec Explorer provides the interface `IDynamicTraversal`, which can be implemented by the user to influence the test generation process. CoVer allows to use the "Hessel Observer Framework" for defining observer automata to configure the coverage. Usually, it is necessary to define such automata for each project. The initial coverage results (without configuration) differ significantly. However, in any professional software testing process, the goals are given by company or even regulatory standards. Therefore, all employed tools must be optimized for their intended purpose, e.g., to reach a certain predefined coverage or mutation score. Currently, this process requires some experience and proficiency with the given tool.

Test coverage can also be measured on the requirements level. Most modeling tools allow to annotate the models with requirements to enable traceability. Conformiq Designer offers the option to use these annotations in the test generation process. Spec Explorer copies the annotations into the generated test script; thus it is possible to observe whether a requirement has been covered. Currently, CoVer does not offer processing of annotations for traceability. In any serious testing process, traceability of requirements to test cases is essential. Integrated test environments provide traceability of artifacts by additional test management tools. However, currently the integration is mostly done by hyperlinks, i.e., on a syntactic level only. It would be desirable to have a semantical integration where the semantics of the requirements is used to generate models and control the generation of test cases. Thus our future work includes semantical analysis of informal requirements with the prospect of test generation.

References

[AD94] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126:183–235, 1994.

[BLL+96] Johan Bengtsson, Kim Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a Tool Suite for Automatic Verification of Real-Time Systems, 1996.

[CGN+05] Colin Campbell, Wolfgang Grieskamp, Lev Nachmanson, Wolfram Schulte, Nikolai Tillmann, and Margus Veanes. Testing concurrent object-oriented systems with Spec Explorer. In *Proc. Intl. Symposium of Formal Methods Europe*, pages 542–547. Springer, 2005.

[Cod12] CodeCover Website: http://www.codecover.org/, January 2012.

[GNRS09] Helmut Goetz, Markus Nickolaus, Thomas Rossner, and Knut Salomon. *Modellbasiertes Testen: Modellierung und Generierung von Tests - Grundlagen, Kriterien fr Werkzeugeinsatz, Werkzeuge in der bersicht*, volume 01/2009 of *iX Studie*. Heise Verlag, 2009.

[Gro08] Object Management Group. The UML MARTE Standardized Profile. In *Proceedings of the 17th IFAC World Congress*, 2008.

[GRS05] Yuri Gurevich, Benjamin Rossman, and Wolfram Schulte. Semantic essence of AsmL. *Theor. Comput. Sci.*, 343:370–412, October 2005.

[HP07] Anders Hessel and Paul Pettersson. Cover - A Test-Case Generation Tool for Timed Systems. In Jan Tretmans Alexandre Petrenko, Margus Veanes and Wolfgang Grieskamp, editors, *Testing of Software and Communicating Systems*, pages 31–34, June 2007.

[IPT+07] Sean A. Irvine, Tin Pavlinic, Leonard Trigg, John Gerald Cleary, Stuart J. Inglis, and Mark Utting. Jumble Java Byte Code to Measure the Effectiveness of Unit Tests. In *Proceedings of the 3rd Workshop on Mutation Analysis (MUTATION'07)*, pages 169–175, Windsor, UK, 10-14 September 2007.

[UML05] UML 2.0 Superstructure Specification. Technical report, Object Management Group (OMG), August 2005.