# TEAGER – Test Automation for UML State Machines

Thomas Santen Dirk Seifert
Technische Universität Berlin, Softwaretechnik
Sekr. FR 5-6, Franklinstr. 28/29, 10587 Berlin
santen,seifert@cs.tu-berlin.de

**Abstract:** TEAGER is a tool suite supporting test automation based on UML state machines, conforming to the UML semantics definition. For testing, the various sources of non-determinism in state machines pose major challenges. This article discusses those challenges and the approach for their solution taken in TEAGER. This includes probabilistic batch generation of test cases, which include expected observations, test execution and evaluation, as well as a probabilistic simulator for state machines, which is useful for model validation.

## 1 Introduction

One of the benefits of model-based development is the potential to support quality assurance. Models can support quality assurance activities from reviews to testing. In particular, software models can be a basis for test automation, increasing coverage and cost efficiency of the test process. The present article introduces TEAGER, *Test Execution And Generation Environment for Reactive Systems*, a tool suite automating tests based on UML state machines [UML04]. As a research prototype designed for easy adaption, TEAGER is particularly useful for investigating the characteristics of state machines as test specifications.

State machines are a variant of Statecharts [Har87, HN96, PS91] whose semantics has been adapted to the special requirements of object-oriented systems in general and the UML in particular. In the modeling process, state machines can be used for different purposes at different levels of abstraction. TEAGER interprets state machines primarily as *behavioral state machines*, which are used to model discrete system behavior.

TEAGER supports three tasks in testing: test case generation, model validation, and test execution including test result evaluation. The tool provides the following key features. It implements an interpretation of state machines *conforming to the UML semantics definition* [UML04]. It supports *non-determinism* in models as well as in systems under test (SUT). The *generation of test cases* takes place in a batch fashion according to several probabilistic strategies. Test case *execution includes evaluation* according to must and may testing [dNH84]. Finally, it includes a simulator for state machines as system under test, enabling easy validation of state machine models, in particular with respect to *modeling for testability*.

Section 2 introduces state machines by way of an example. Particular challenges in testing

based on state machines and their resolution in TEAGER are the topic of Section 3. The structure of a test case is discussed in Section 4. Section 5 explains the software architecture of TEAGER. Section 6 and 7 address test case generation and test execution. Section 8 puts the results presented here in the context of other work, and Section 9 summarizes and points out directions of future research.

## 2 State Machines – An Example

To provide an intuitive understanding of UML state machines, this section introduces state machines in general and presents an example state machine modeling a car audio system in Figure 1.

State machines communicate by exchanging events. The basic reaction of a state machine to external stimuli, i.e., events, is to perform transitions. While executing, a state machine generates events which are sent to other objects and receives events generated by other objects. The latter are stored for further processing within the state machine. Most practical applications use a *FIFO-Queue* for storing events, and we assume in the following that events are processed in a first-in-first-out fashion. Dispatching and processing of events take place one at a time in a *run-to-completion step*. Run-to-completion means that an event can only be processed if the processing of the previous event is fully completed.

When a state machine detects and dispatches an event, several transitions may become enabled for firing. If no transition is enabled, the event occurrence is discarded and the current run-to-completion step terminates. If several transitions are enabled they may be in *conflict* with each other. Some conflicts can be resolved based on a transition priority. This priority is defined by the relative position of the source states of the transitions in the state hierarchy. In a state machine, a transition originating from a substate has priority over a conflicting transition originating from any of its containing states. The *transition selection algorithm* determines which of the enabled transitions actually fires. Taking a transition comprises exiting the source state and entering the target state after executing all actions associated to the transition.

The selected set of transitions is one of the largest sets containing only mutually consistent transitions such that there is no transition outside the set which is consistent with all transitions in the set and with priority over a transition in the set. If there is more than one set fulfilling these constraints the state machine is non-deterministic.

The remainder of this section illustrates the workings of a state machine by way of the example in Figure 1. For any formal analysis, however, and for test case generation in particular, we need a precise semantics of UML state machines [SS06], which we cannot present in detail here. The semantics definition conforms to the UML 2.0 Superstructure specification [UML04] and also draws from other work on defining semantics for object-oriented state machines [BBK+04, LP99, LMM99]. It covers hierarchically and orthogonally structured state machines including multi-level transitions. It treats signal events in the trigger part, and sequences of internal and external signal events in the action part of a transition.

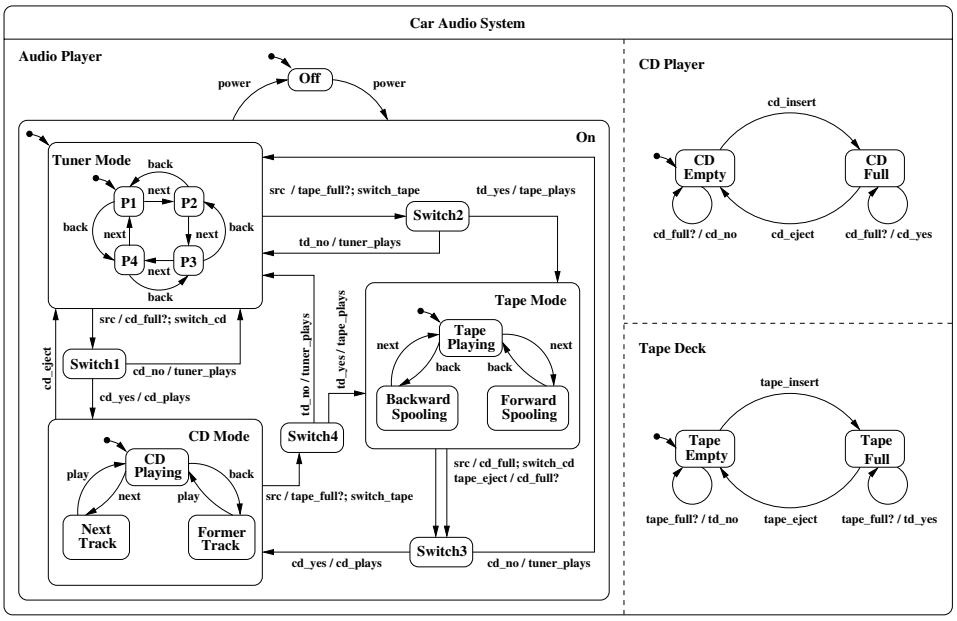


Figure 1: State Machine Specification of a Car Audio System.

The state machine in Figure 1 models a car audio system which has a minimal user interface to ensure maximal safety for the car driver. The system has three signal sources, namely a tuner, a tape deck, and a compact disc player.

At any time, the car audio system allows the driver to insert or eject a compact disc or an audio tape. A power button toggles the system between the on and off mode and, as one would expect, the system plays one of its sources only if it is in the on mode. If the system is switched on, a source button allows the driver to choose – in a round robin fashion – among the different signal sources depending on the audio media which are present in the system. Based on the source mode, two buttons, next and back, change the channel, spool the tape forward or backward, or select a track of the compact disc.

The state machine in Figure 1 consists of the root state `Car Audio System` which is refined into three orthogonal parts, namely `CD Player`, `Tape Deck` and `Audio Player`. Initially, the system contains neither a compact disc nor an audio tape and is switched off. The initial states of a state machine are marked by an arrow originating from a filled black circle.

Inserting a compact disc, i.e., *incoming* event `cd_insert` occurs, causes the system to change from state `CD Empty` to `CD Full`. Ejecting the compact disc (incoming event `cd_eject` occurs) causes the system to switch back to the state `CD Empty`. The two incoming events `tape_insert` and `tape_eject` have a similar effect on `Tape Deck`.

If the system is in state `On`, an incoming event `src` causes the system to change mode.

Then the two transitions originating from `Tuner Mode`, for example, generate the events `cd_full?` and `switch_cd` or `tape_full?` and `switch_tape` in response to `src`. The events `cd_full?` and `tape_full?` are *internal* events. Because it is possible to take each of the two – conflicting – transitions, the state machine is non-deterministic and the transition selection algorithm is free which one to choose.

Taking the transition to switch to `CD Mode`, the *observable* event `switch_cd` is sent to the environment, thus allowing it to react to the change from `Tuner Mode` to `CD Mode`. The internal event `cd_full?` is stored for further processing in the local event queue.

Once the system processes the event `cd_full?`, a self-transition in `CD Player` fires, depending on whether a compact disc is inserted or not. By sending the event `cd_no` or `cd_yes` this sub-automaton signals to the `Audio Player` whether a compact disc is available or not. The `Audio Player` in turn consumes that event and changes to the `CD Mode` or `Tuner Mode` subject to the received event. Again, the environment can observe this change of mode through the events `tuner_plays` and `cd_plays`. For the other events, the state machine works in a similar manner.

## 3 Challenges for Test Automation

There is a well-established theory for generating test cases from finite automata or labeled transition systems [Tre96, BJK+05], that has been implemented in tools like TorX [Tor05] (cf. Section 8). The standard theory relies on two assumptions: synchronous communication and *quiescence*, which means that after each trigger the system under test will eventually reach a stable state and the events the system generates up to that state can be considered the reaction to the immediately preceding trigger.

State machines do *not* satisfy those assumptions. They communicate asynchronously with their environment by means of the event queue. For a state machine model, it also is unrealistic to assume that the environment will wait for the system to become quiescent before it will trigger the system again.

The fact that events are stored in the event queue makes state machines truly more powerful than finite automata or labeled transition systems, and many interesting properties of fifo-automata are undecidable [HCF+02]. The event queue is not directly observable by the environment and it induces non-deterministic behavior.

Non-deterministic specifications pose a notoriously difficult problem for test automation. State machines are particularly challenging because their semantics [UML04] contains three major sources of non-determinism:

1. *Several firing transition sets.* If transitions are in conflict with each other, i.e., firing them all would lead to an ill-formed configuration, then the transition selection algorithm produces several consistent firing transition sets. Only one of them is selected for firing, but the semantics does not prescribe which one.

2. *Order of firing transitions.* The order of firing transitions from the selected transition

set is arbitrary, too. The particular order of firing transitions determines the order of observable events produced by the actions of those transitions, and therefore it influences the observable behavior of the state machine.

3. *Order of event processing.* The actions associated with transitions produce internal events that can enable transitions in subsequent steps. Those events and the ones incoming from the environment are processed asynchronously. They are queued into the event queue that also stores incoming events. That queue is part of the "state" of the state machine, but it is not directly observable. Because the order of firing transitions is chosen non-deterministically and may not be observable, the environment can only speculate about the state of the event queue. Theoretically valid techniques to determine the state of a finite automaton from sequences of observations are not applicable because the event queue is unbounded and it is undecidable whether there is a bound for a given state machine [HCF+02]. The state of the queue, however, crucially influences the subsequent behavior of the state machine.

Although the boundedness of the event queue is undecidable, earlier work [SHS04] tried to reduce state machines to labeled transition systems, considering each different configuration and state of the event queue a different state of the labeled transition system. The idea was to assume boundedness (for an unknown bound) and reduce the number of states by exploiting symmetries. It turned out, however, that treating all possible reactions of a state machine to a sequence of stimuli in a single test case is practically intractable even under that assumption. Even for (unrealistically) small state machines, the resulting branching factors are so large that the resulting data objects become huge after only very few incoming events. Therefore, TEAGER uses a different approach than the one discussed in [SHS04]: it *linearizes* test cases as explained in the following section.

The third source of non-determinism, the ordering of events in the event queue, poses a severe problem for the evaluation of test results, too. *False negative* verdicts seem to be unavoidable, in general. For example, let us stimulate an implementation of the state machine for the car audio system with the input sequence `power` · `src` · `power`. Which observations do we expect? The critical event is `src`. Processing `src` may yield the internal event `cd_full?` and, depending on the processing of this event, one of the observations `cd_plays` or `tuner_plays`. Additionally, the actual observation also depends on the order in which the events `cd_full?` and `power` are stored in the event queue. If `cd_full?` is processed first, we observe one of the two expected observations but if `power` is processed before `cd_full?` no observable event will be processed at all, because the "answer" `cd_yes` or `cd_no` of `CD Player` will be ignored by `Audio Player`, which will be in the state `Off`.

## 4 Structure of a Test Case

We put two major requirements on the data structure of test cases. First, we wish to keep both test execution and evaluation as simple as possible. Therefore, we require test cases to include the expected observable behavior of the SUT. Second, we need to address the

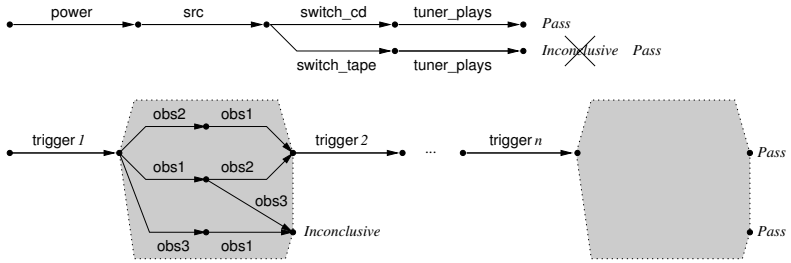


Figure 2: Test Cases.

problems discussed in the previous Section 3. Our approach is to require test cases to be basically linear.

A test case is a linear sequence of input events to the SUT and *acceptance graphs* for observations from the SUT. We implement linearity in the acceptance graphs by tracking only one correct behavior in a test case and marking other correct behaviors as "correct but not to pursue further" in that test case. We assign those continuations an inconclusive verdict except for the end of a test case, where we replace the inconclusive verdict with a pass verdict, because the test case will not take any continuation, anyway. By executing a test case several times, we try to verify the expected correct behavior (may testing).

The structure of acceptance graphs results from firing several transitions in a step. The UML does not define in which order these transitions will be fired whereas the order of associated actions of a transition is fixed. Generally, the implementation is free to choose one specific order. Therefore, the test case must cover all permitted sequences of actions associated with the firing transitions. Additionally, if there is more than one firing transition set, the test case must cover not only the permitted observations (leading to a pass verdict) but also the permutations of those observations (leading to inconclusive verdicts).

In general, those sequences of events have common prefixes. They make up a (small) regular language whose terminal symbols are observable events. Consequently the acceptance graph is a deterministic acceptor for that language with two accepting states and evaluating the response of the SUT means deciding whether observations form a word of that language.

The top of Figure 2 shows a test case for the car audio system. Beginning at the start node, the test case produces two inputs, `power` and `src`, to the SUT and expects `switch_cd` followed by `tuner_plays` as an reaction from the SUT, i.e., as an reaction to `src` the system tries to switch to `CDMode` but does not succeed because the CD player is empty. If the SUT produces those observations, the test reaches the termination node *Pass* indicating successful execution of the test case. But the state machine is non-deterministic and may also react with the observation `switch_tape` followed by `tuner_plays` in response to the input `src`. Although this response is correct, it is not the behavior that the particular test case expects and so the test reaches the termination node *Inconclusive*. Since the dis-

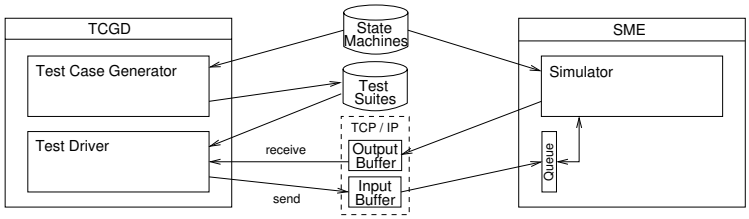


Figure 3: Architecture of TCGD and SME.

tinction between inconclusive and pass verdicts is unnecessary here, the verdict assigned to that branch is *Pass*.

The bottom of Figure 2 shows the general structure of a test case. Sequences of trigger events and acceptance graphs alternate until the test case terminates with the verdict *Pass* or *Inconclusive*, where the acceptance graphs, shaded gray in the figure, treat the possible correct observations. For example, accepting the two observation sets {`obs1,obs2`} and {`obs1,obs3`} generated when firing each of two possible transition sets, respecting any possible order and treating only the first set as correct behavior we yield the first acceptance graph in Figure 2.

In summary, a test case considers one particular behavior of the state machine. Alternative correct responses of the SUT due to non-determinism yield the verdict *Inconclusive*, whereas responses that are not consistent with the state machine (including failure of the system to react at all) make the test case fail.

## 5 Tool Architecture

TEAGER is a tool set for test case generation, model validation, and test execution including test result evaluation. Figure 3 sketches its architecture. TEAGER consists of two main components: the test case generator and test driver, TCGD, and the state machine executor, SME. The TCGD has mainly two tasks, to generate test cases from a state machine and to execute test cases by triggering an SUT and matching the responses of the SUT with the results the test cases prescribe.

According to that general picture, the TCGD works on two repositories, storing state machines and test suites. The former also are input to the SME, which simulates a state machine and acts as an SUT for model validation. The communication between the test driver (in the TCGD) and the SUT takes place asynchronously over a TCP/IP connection. It uses two buffers for incoming and outgoing events.

Both components offer graphical user interfaces to edit state machines, generate and execute test cases, to simulate a SUT, and to control parameters related to these activities. The following sections discuss the test case generation and test execution processes in more detail.

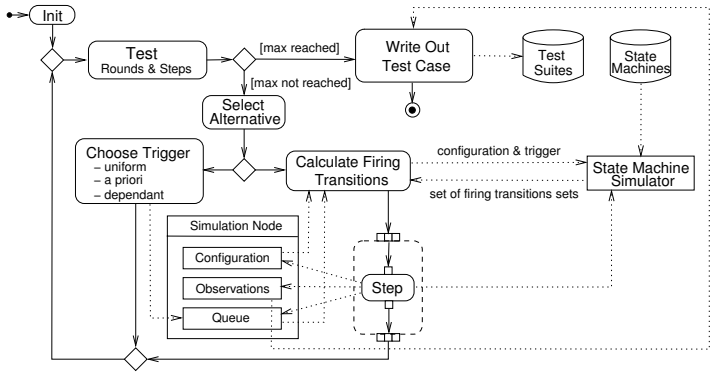


Figure 4: Test Case Generation Process.

## 6 Test Case Generation

The main task of the TCGD is to derive test cases from state machine specifications. It translates a given state machine into an intermediate format which it uses for stepwise simulation. The results of this simulation constitute the basis to generate test cases, as shown in Figure 4.

The first step initializes all required data structures. That includes reading and parsing the state machine and setting up the initial configuration of the simulation which usually includes an empty event queue. The overall generation process is controlled by two parameters, the number of test cases to be generated (`Generation Rounds`) and the length of a test case (`Simulation Length`). The "length" of a test case is determined by the number of simulation steps needed to generate it. When the simulation reaches that bound, TCGD writes out the test case to the test suite.

In each round, there is a choice between two alternatives: to stimulate the SUT with a new trigger, or to execute the state machine, i.e., to execute the next run-to-completion step. The first alternative is taken with a certain probability, which is a parameter of TCGD (`Trigger Probability`).

This and the following choices in the test case generation process are resolved probabilistically to influence the way test cases are selected. The probability functions model assumptions about the environment behavior and about the relative speed of the SUT and the environment. They can derived, e.g., from operational profiles of system usage.

If the SUT is to be triggered, TCGD probabilistically chooses an incoming event[1] (trigger for the SUT) based on one of three possible distributions, namely a *uniform*, an *a priori* or a *dependant* one (`Trigger Distribution`). With a uniform distribution all incoming events have the same probability of being selected. An a priori distribution (a prior), which

[1] State machines are *input enabled* by default and hence it is allowed to choose any incoming event as the next trigger.

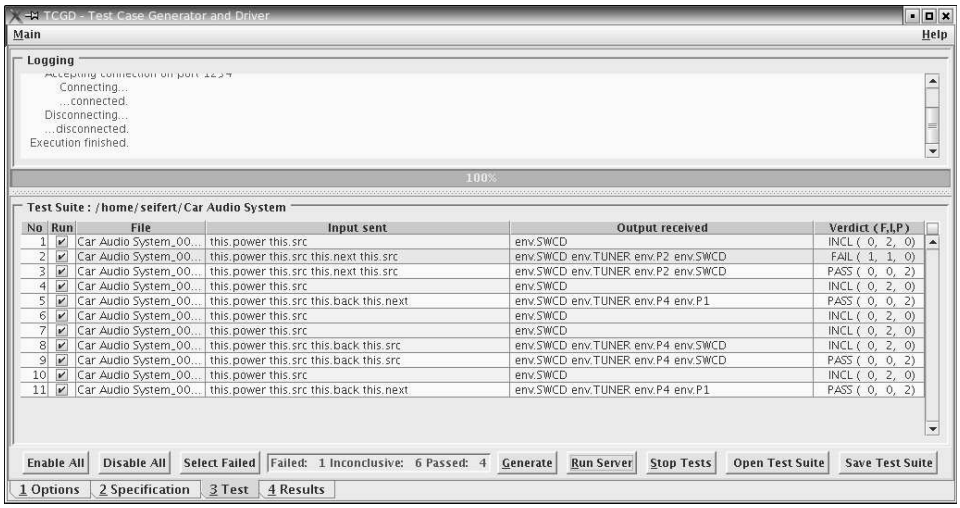


Figure 5: Test View of TCGD.

is annotated in the event declaration part of the specification, assigns fixed but possibly different probabilities to the input events. A dependant distribution starts with the same prior. But taking an event $a$ at time $t_n$ decreases the probability for $a$ for the next choice (at time $t_{n+1}$) and increases the probability for all other events accordingly. Thus TCGD ensures that the frequency of an event in a test case is its *a priori* probability multiplied by the maximal number $N$ of triggers in a test case. In any case, the chosen event is added to the simulation queue.

If the state machine is to be executed, TCGD dequeues an event from the simulation queue of the current simulation node and initializes the state machine simulator with the appropriate configuration. The simulator determines all consistent *firing transition sets*. For each such set, TCGD creates a new simulation node and fires all transitions in the set. The results are new state machine configurations with internal events enqueued to the simulation queue and observable events added to the list of observations. The internal and observable events stem from the action parts of firing transitions. In this way, TCGD starts to generate a new test case for each possible continuation.

## 7 Test Execution

Figure 5 shows the *test* view of TCGD for the car audio system after execution of a test suite. This view allows the user to generate new test suites and to control test execution. All test cases from a test suite are listed in a table, which also displays triggered and observed events and test results from the execution rounds. At the top and the bottom of the test view, information concerning logging, test progress, and the overall result of a test

are displayed.

Executing a test case consists of traversing it and, depending on the type, triggering the SUT or comparing observed outputs from the SUT with expected results. A parameter controls the time TCGD waits for a response by the SUT (`Timeout`). The input and output buffers in Figure 3 make up the interface to an SUT. Technically, TCGD initiates the execution process by accepting connections from an SUT at a specified port (`SUT Port`). After the connection is established, the TCGD asks the SUT to initialize itself and waits until this is acknowledged. That initialization takes place for every test case.

While executing, TCGD sends trigger events with a specified rate (`Trigger Rate`) and queries the output buffer of the SUT periodically (`Observation Sampling Rate`). If SME acts as the SUT, it simulates real-time properties by sending observations only after a specified time (`Response Time`). In this way, the user can control and test the effects of different timings in the communication.

Each test case is repeated several times (`Execution Rounds`) to test for unexpected non-determinism in the SUT or, in the case of a non-deterministic specification, to test for the expected continuation. The result of each execution is saved and contributes to the final verdict. Depending on the test evaluation method (`Pass Condition`) we require for *must* testing that all test case execution rounds finish with a pass verdict to yield a *pass* for the test case. For *may* testing, we require at least one test case execution to finish with a pass verdict.

To validate a state machine using SME, we need to approximate a realistic execution behavior. In addition to the response time, which models real time behavior, the SME probabilistically decides whether or not to perform the next execution step (`Step Probability`). Executing a step may yield more than one firing transition set. The selection which SME takes is also controlled in two different ways (`FTS Selection`). Either, SME chooses the transition set probabilistically or it resolves that choice deterministically.

## 8 Related Tools

TorX is a testing tool for conformance testing of reactive systems developed at the university of Twente [Tor05]. Accepted specification languages are LOTOS, PROMELA and SDL. The internal representation bases on (input/output) labeled transition systems. TorX supports on the fly test case generation, in contrast to TEAGER, which produces test suites in a batch fashion.

*AsmL 2* by Microsoft provides an executable specification language based on the theory of Abstract State Machines [Asm05]. The AsmL Test Tool supports parameter generation and test sequence generation based on interface automata. It is possible to run an AsmL model in parallel with the SUT in order to check conformance of the SUT to the model.

Conformiq Software Ltd. offers a *Test Generator* which accepts UML state diagrams as the model of the system under test for dynamic testing [Con05]. The tool provides online and off-line test execution with test coverage measuring and report generation. Test

Generator uses “extended UML state charts”. Due to a lack of technical documentation, the relation to UML state machines is unclear.

*Reactis* from Reactive Systems Inc. generates test data from Simulink and Stateflow models [Rea05]. The test generation component focuses on data and uses several coverage criteria (decision, condition, and MC/DC; Simulink-specific measures such as branch and subsystem; and Stateflow-specific metrics such as state, condition-action and transition-action). The Validator uses a *test and check* approach for verifying models but an explicit instrumentation of the model is necessary.

An increasing number of CASE Tool manufactures includes 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 [Rha05]. Simulink Verification and Validation generates test cases in Simulink and Stateflow, and measures test coverage for Statecharts [Mat05].

## 9 Conclusion

TEAGER conforms to the UML semantics and therefore is particularly useful for investigating test automation strategies based on UML state machines. Designing the tool suite and experimenting with state machine specifications has already provided insight into the particular challenges state machines pose for testing.

The non-determinism induced by asynchronous event processing (cf., Section 3) not only is a theoretical possibility but also a practical problem. Even for a small state machine like the one in Figure 1, the actual state of the event queue can hardly be inferred from the observable reactions of the state machine to a (short) sequence of triggers. It is hard to avoid false negatives when testing against such a state machine specification. Consequently, not only design for testability but also *modeling for testability* is an important issue. TEAGER helps to evaluate state machines in this respect because it allows testers to execute test cases generated from a state machine against a simulation of that very state machine. Executing manually produced test cases against a state machine specification allows for additional model validation.

It has turned out instrumental for producing interesting test cases to resolve the other forms of non-determinism probabilistically, and to specify a fixed start sequence for test cases, which puts the SUT in a desired start configuration for testing.

Further research must address more elaborate test case generation strategies and associated coverage criteria. For test evaluation, must-testing will rarely be an option. Therefore, adequate acceptance criteria for may-testing are needed. In this context, generating a test suite as a batch process has advantages over on-the-fly testing. A stored test suite makes it easy to implement various repetition strategies for may-testing. It also supports regression testing well.

TEAGER is a flexible basis to augment theoretical investigations of these issues by practical experiments.

## References


[Asm05] AsmL 2. Microsoft Research - Foundations of Software Engineering Group, 2005. `research.microsoft.com/fse/asml`.

[BBK+04] M. Balser, S. Bäumler, A. Knapp, W. Reif, and A. Thums. Interactive Verification of UML State Machines. In J. Davies, W. Schulte, and M. Barnett, editors, *Formal Engineering Methods (ICFEM'04)*, LNCS 3308, pages 434–448. Springer, 2004.

[BJK+05] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors. *Model-Based Testing of Reactive Systems*. LNCS 3472. Springer, 2005.

[Con05] Conformiq Test Generator. Conformiq Software Ltd., 2005. `www.conformiq.com`.

[dNH84] R. de Nicola and M. C. B. Hennessy. Testing Equivalences For Processes. *Theoretical Computer Science*, 34:83–133, 1984.

[Har87] D. Harel. Statecharts: A Visual Formulation for Complex Systems. *Science of Computer Programming*, 8(3):231–274, 1987.

[HCF+02] F. Herbreteau, F. Cassez, A. Finkel, O. Roux, and G. Sutre. Verification of Embedded Reactive Fiffo Systems. In *Latin American Theoretical Informatics Conference (LATIN'02)*, LNCS 2286, pages 400–414. Springer, 2002.

[HN96] D. Harel and A. Naamad. The STATEMATE Semantics of Statecharts. *ACM Transactions on Software Engineering and Methodology*, 5(4):293–333, 1996.

[LMM99] D. Latella, I. Majzik, and M. Massink. Towards a Formal Operational Semantics of UML Statechart Diagrams. In *Formal Methods for Open Object-Based Distributed Systems (FMOODS'99)*, page 465. Kluwer, 1999.

[LP99] J. Lilius and I. P. Paltor. Formalising UML State Machines for Model Checking. In R. France and B. Rumpe, editors, *The Unified Modeling Language (UML'99)*, LNCS 1723, pages 430–445. Springer, 1999.

[Mat05] Matlab/Simulink. The MathWorks Inc., 2005. `www.mathworks.com`.

[PS91] A. Pnueli and M. Shalev. What is in a Step: On the Semantics of Statecharts. In *Theoretical Aspects of Computer Software (TACS'91)*, pages 244–264. Springer, 1991.

[Rea05] Reactis Tester. Reactive Systems Inc., 2005. `www.reactive-systems.com`.

[Rha05] Rhapsody. I-Logix Inc., 2005. `www.ilogix.com`.

[SHS04] D. Seifert, S. Helke, and T. Santen. Test Case Generation from UML Statecharts. In M. Broy and A. V. Zamulin, editors, *Perspectives of System Informatics (PSI'03)*, LNCS 2890, pages 462–468. Springer, 2004.

[SS06] T. Santen and D. Seifert. Executing UML State Machines. Technical report, Technische Universität Berlin, 2006. to appear.

[Tor05] TorX. University of Twente, 2005. `fmt.cs.utwente.nl/tools/torx`.

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

[UML04] The Unified Modeling Language 2.0: Superstructure FTF convenience document (ptc/04-10-02). Object Management Group, 2004. `www.uml.org`.