inria-00268864, version 2 - 23 Apr 2008

Test Case Generation from UML State Machines*

Dirk Seifert
Loria – Université Nancy 2
Campus Scientifique, BP 239
F-54506 Vandoeuvre lès Nancy cedex
Dirk.Seifert@Loria.fr

ABSTRACT

In this paper we describe a comprehensive approach for conformance testing of embedded reactive systems. Based on a formal specification, namely UML state machines, we automatically generate test cases and use them to check the functional conformance of a system under test. Our test cases include not only stimuli to trigger the system under test, they also include possible correct observations to automatically evaluate the test case execution. In contrast to classical Harel Statecharts, state machines behave asynchronously, which makes automatic test case generation a challenge. The TEAGER Tool Suite implements the automatic generation, execution and evaluation of test cases and proves the applicability of our test approach.

1. INTRODUCTION

The impact of embedded systems in our everyday life is steadily growing. They are present not only in very specific contexts but also in nearly every electrical device we use. In general, embedded systems comprises of hardware and software components interacting with a specialized technical environment via sensors and actors. The main reason for their success is the combination of specific or high-performance hardware with the flexibility of software. The software is responsible for controlling the hardware and software components and for calculating reactions as responses to received events. It is remarkable that users unconditionally trust in the correct functioning of such systems. This is true not only for safety critical systems like an anti-lock brake system in a car but also for comparatively simpler systems like a cellular phone. The development should satisfy this confidence. Erroneous systems annoy the costumers and are a high commercial risk in mass customization. Moreover, size and complexity of nowadays systems which have to be developed, demand for improved and automated processes: for development as well as for quality assurance.

Model-based software development bases on setting up models of the system to be constructed. This approach has proved to be useful, because it allows developers to first elaborate the most important properties of the software before proceeding with the implementation. Nowadays the Unified Modeling Language (UML) [24] is widely used to model systems and to guide development processes. The UML comprises of several diagram types to specify the structure and the behavior of a system or system component. State machines are used to either describe the discrete reactive behavior (behavioral state machines) or to describe the usage protocol (protocol state machines). In this paper we refer to behavioral state machines. We use them to specify the states a system can take and actions it can execute during its lifetime in response to external and internal events. Due to the discrete reactive character of state machines and the possibility to completely describe the discrete behavior of a system, state machines are appropriate to model embedded systems.

We intend to use state machine models not only for development but also to support an automated quality assurance process. In Section 2 we introduce the syntax and semantics of state machines we need in this paper by means of an example. In Section 3 we present our test approach. In this approach we automatically generate test cases out of a state machine specification. The test cases include not only the stimuli to trigger the system under test, they also include the possible correct reactions. The latter allows an automatic evaluation of the test case execution. We describe the underlying theory, the developed test case generation algorithm, how approximation techniques are used to develop a practical approach and how the test case generation and execution can be controlled and evaluated. In Section 4 we describe the TEAGER Tool Suite which completely implements the introduced approach. Finally, in Section 5 we conclude our work, discuss related work and give an outlook to ongoing and future research.

2. STATE MACHINES

UML state machines [24] are an object-oriented extension of the classical Harel-Statecharts [15]. In this paper we use *behavioral state machines* to describe the sequence of states a system or system component can take and the actions it executes when changing these states. State machines are mathematical models with a graphical representation: the nodes depict simple or composed states of the system and the labeled edges depict transitions between these states. Composite states allow to hierarchically and orthogonally structure the model, thus reducing the graphical complexity. Labels express conditions under which transitions can be taken and the actions which will be executed when the transition is taken. Events are used as triggers to activate transitions and can be parameterized to exchange data. Optional, every state machine has a data space which can be read and manipulated by the state machine during

*This work has partly been done in the research group Softwaretechnik at the Technische Universität Berlin.

inria-00268864, version 2 - 23 Apr 2008

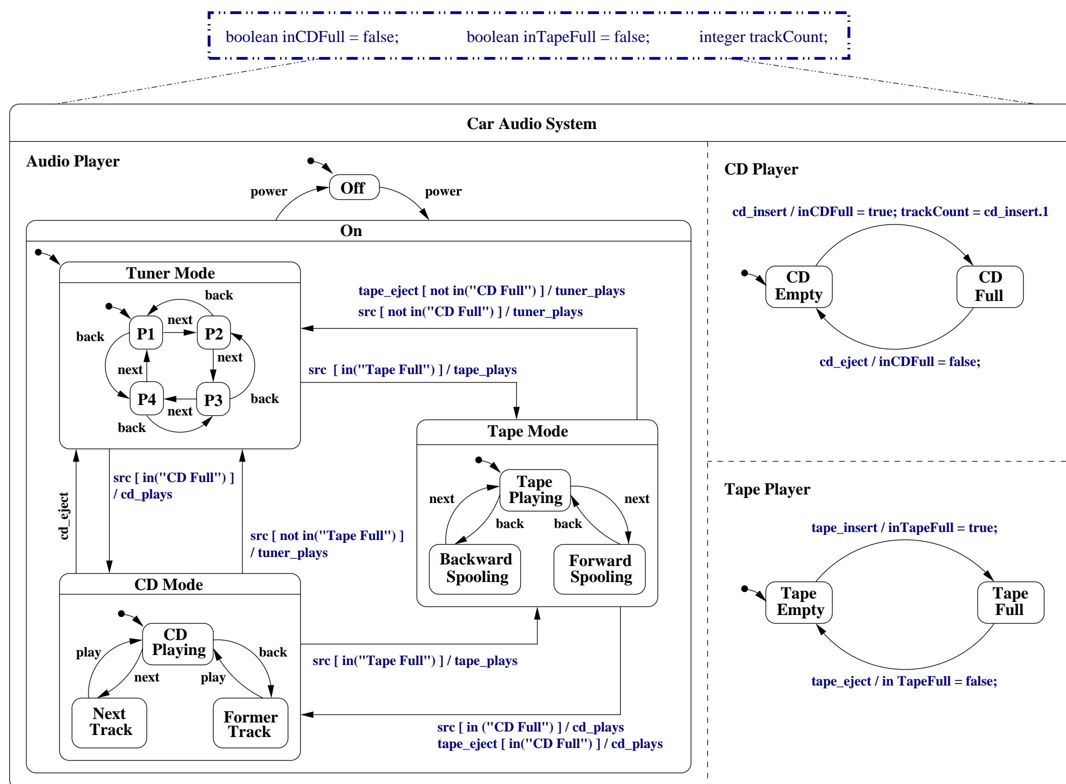

Figure 1: State machine specification for the Car Audio System.

execution. More precisely, it is possible to read the data to describe specific conditions when a transition can be taken or to manipulate the data and exchange information within the actions.

The general structure of a transition consists of a *source* state, a *trigger* event, an optional *guard* in square brackets, an optional *action sequence* separated from the previous elements by a slash, and a *target* state. With the optional guard a fine-grained condition to enable the transition can be described depending on the system's state. Hence the activation of the source state, the trigger event and the fulfilled guard condition constitute the condition which must hold to enable the transition. An action can either be a statement manipulating the data space or the generation of new events. The action sequence and the subsequently active target state constitute the effect of the transition.

In this paper we use a substantial subset of state machines to study automated test case generation and evaluation based on state machines. We assume the reader to have some basic knowledge of transition systems. In the following we briefly describe state machines by means of an example. Afterwards we discuss semantic issues which make automated test case generation a challenge. A complete and detailed description as well as a precise definition of the semantics (including the integration of complex data) can be found in [18].

2.1 Example

To demonstrate the state machine notation we use a state machine model specifying the behavior of a simple sound device in a car. Figure 1 shows this model. The requirements for such sound device could be as follows:

It should be possible to turn the Car Audio System on and off. When turned on, it should play one of three different audio sources, namely radio, tape or compact disc, respecting the presence of a tape or a compact disc. It should be possible to change between available sources. Furthermore, it should be possible to switch between four radio stations, to spool a tape backward or forward, or to select the previous or the next track of a compact disc.

Abstracting from any physical devices we introduce the following events to model the required behavior: `power`, `src` (to switch between the different sources), `next`, `back` and `play`. Additionally we introduce events signaling the insertion and the ejection of a tape or a compact disc as well as events to signal system reactions. Furthermore, we use data variables to store detailed information about the current state. For example, we use an integer variable `trackCount` to store the number of titles of an inserted compact disc. Figure 1 shows a state machine model of the sound device including the underlying state space.

At the highest level of abstraction the model consists of an orthogonal state comprising three regions. The two regions `CD Player` and `Tape Player` model the information if a tape or a compact disc is inserted into the system or not. The more complex region `Audio Player` models the control of the system. The region is refined by two states: `Off` and `On`. Initially the system is assumed to be switched off, expressed by the small arrow leaving a bullet and ending at the `Off` state. When the event *power* is processed the system is switched on and starts to play the radio (again expressed by a small arrow). The composite state `On` is refined into states modeling the three signal sources. The transitions between these states describe the changes between the sources as reaction

inria-00268864, version 2 - 23 Apr 2008

to an event *src*. For example, when the system is in `Tuner Mode` and a tape and a compact disc are inserted into the system (i. e. both in-predicates are true) and the event *src* is processed, the system can either switch to the tape mode or switch to the compact disc mode because both transitions are enabled and can fire. All three substates of `Audio Player` are further refined to describe the particular behavior in reaction to the events `next`, `back` and `play` in each state.

2.2 State Machine Semantics

The semantics of state machines is adapted from the STATEMATE semantics [17, 16] to fit into the object-oriented paradigm. As described above a state machine can be refined by simple composite and orthogonal states. Simple composite states contain exactly one region and orthogonal states contain at least two regions. In every region only one substate can be active at a time. The state which is entered by default when a region is entered is marked by an arrow emanating from a filled circle. The hierarchical ordering of states forms a tree structure with a region as the root node, simple states at the leave nodes and in between (alternating) composite states and regions.

Due to orthogonal regions a state machine can be in several states at a time. We call the set of all active states a *configuration*. For the same reason it is possible that more than one transition can fire at a time; one in every active orthogonal region. We call the set of all jointly firing transitions at a time *firing transition set* (FTS in short). Due to the hierarchical structure of state machines it can happen that two transitions are enabled for firing on different hierarchy levels of a state. Taking both would lead to a configuration which is not well-formed. A similar situation arises if a transition leaves an orthogonal region. In this case the transition cannot fire together with an enabled transition in another orthogonal region. In both cases the transitions are said to be in conflict with each other. Such situations are identified if two transitions leave identical states in the state hierarchy. The UML describes a two step process to resolve such conflicts. In the first step a priority scheme is used. Transitions emanating from a state deeper in the state hierarchy has priority over the other transition. Thus the more refined transition is taken.[1] Nevertheless, not all conflicts can be resolved using this priority scheme. In the second step only transitions are selected which are not in conflict to each other respecting maximal progress of the system. A so-called *transition selection algorithm* selects all maximal sets $T_{\|} \subseteq T$ of enabled transitions fulfilling the following requirements:

$$\forall t : T_{\|} \bullet enabled\,(t, c, e, d) \tag{1}$$

$$\forall t_1, t_2 : T_{\|} \mid t_1 \neq t_2 \bullet t_1 \parallel t_2 \tag{2}$$

$$\nexists t' : T \setminus T_{\|} \mid enabled\,(t', c, e, d) \bullet \forall t : T_{\|} \bullet\ t \parallel t' \vee t' \prec t \tag{3}$$

First, all transition in the firing transition set must be enabled regarding the current configuration, the trigger event and the current data assignments. Second, all transitions in the set are mutually conflict free (expressed by the $\parallel$ operator). Third, there is no enabled transition outside the set which is conflict free with the transitions in the set or with higher priority than a transition inside the set. Thus, transitions with the highest priority are taken and maximal sets are chosen. Result of the transition selection algorithm is a set of firing transition sets ($\mathbb{P}\mathbb{P}T$).

[1] This differs from classical Statecharts. But it reflects the object-oriented inheritance behavior.

Each set represents a valid firing transition set. It is important to mention that for execution one such set is arbitrarily chosen, and that the order in which the transitions are fired is arbitrarily chosen, too. In consequence, all set choices and transition permutations form the set of all possible semantic steps of the state machine at a time. This is important if we want to compute the possible correct behavior for an input sequence to evaluate the test execution. In opposite to the classical Statecharts, the event processing takes place in a so-called *run-to-completion* step. This asynchronous event processing demands the processing of the previous event to be completely finished before the next event can be processed. Therefore it is necessary to buffer received events in an event store. Consequently, the occurrence of an event and its processing are asynchronous, i. e. take place at different times. It follows immediately that a possible (observable) reaction of the system also takes place asynchronously.

The semantic model of state machines builds on the semantic steps a state machine can execute during its lifetime. Such a step moves the state machine from one semantic state to another semantic state while receiving events from and emitting events to the environment. A semantic state (called a *status*) comprises of three components: a configuration (a set of active states), an event queue, and the variable assignments. We depict the components of a status in double square brackets $[\![c, q, d]\!]$ and a semantic step as follows:

$$[\![c, q, d]\!] \xrightarrow{in, out} [\![c', q', d']\!] \tag{4}$$

Please note that the chosen set of firing transitions and the execution order of these transitions can be identified (if necessary) from this representation. Assuming a state machine to be input enabled (cf. the next section) a *semantic step* can be described as follows. We have to distinguish two situations. First, the situation when the event store does not contain any events:

$$\frac{q = <> \qquad q' = \oplus(q, E_{in})}{[\![c, q, d]\!] \xrightarrow{E_{in}, \langle\rangle} [\![c, q', d]\!]} \tag{5}$$

During the step, only the events received from the environment (E_{in}) are added to the event store ($\oplus(q, E_{in})$). The active configuration and the data assignments are left unchanged. Second, the situation when the event store contains events for processing:

$$\frac{\begin{array}{c} q \in \mathsf{ran} \oplus \\ (q'', e) = \ominus(q) \\ c' = (c \setminus \bigcup_{\forall t : T_{\|}} exits\,(t)) \cup \bigcup_{\forall t : T_{\|}} enters\,(t) \\ A_{seq} \in perm(\{t : T_{\|} \bullet effect\,(label\,(t)(e))\}) \\ (d', E_{gen}) = performAll(\frown / A_{seq})(d) \\ (E_{int} = E_{gen} \upharpoonright E_{SM}) \wedge (E_{out} = E_{gen} \upharpoonright E_{env}) \\ q' = (q'' \oplus E_{int}) \oplus E_{in} \end{array}}{[\![c, q, d]\!] \xrightarrow{E_{in}, E_{out}} [\![c', q', d']\!]} \tag{6}$$

During the step, the trigger event will be selected from the event store ($\ominus(q)$). The next configuration c' results from leaving all states the transitions exit, and entering all states the transitions enter. Next, an execution order for the firing transition set is chosen (*perm*), and the effect of this transition sequence is calculated

(*performAll*). The effect includes the new data assignments (d') and the sequence of newly generated events (E_{gen}). Finally, this event sequence is processed. The generated internal events (E_{int}) and the events received from the environment (E_{in}) are added to the event store. The remaining external events (E_{out}) are sent to the environment. Now we can describe the execution of a state machine based on this definitions as a concatenation of semantic steps. We call such a sequence of semantic steps a *computation*:

$$[\![c_1, q_1, d_1]\!] \xrightarrow{in_1, out_1} [\![c_2, q_2, d_2]\!] \xrightarrow{in_2, out_2} \dots \xrightarrow{in_{n-1}, out_{n-1}} [\![c_n, q_n, d_n]\!]$$

All formal definitions of the used state machine semantics can be found in [18].

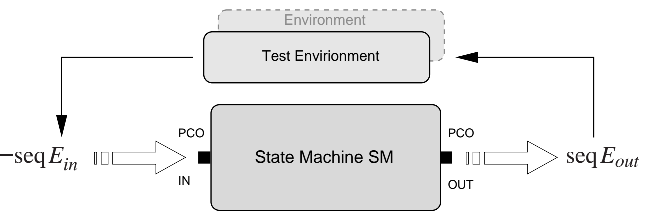

Figure 2: Abstract Test Architecture for Embedded Systems.

3. TEST CASE GENERATION

We use the formal execution model from the previous section as the bases for the definition of our automated test approach. Only such mathematical precise models with a clear interpretation offers the basis for automated processes. To define such a process we need to fix some last open points.

In the UML (and in our semantics, too) not all semantic details are fixed. Such points are called *semantic variation points*. Semantic variation points have been introduced to avoid unnecessary restrictions on semantic details. Instead, there should be some space for different realizations.[2] A user of the semantics has to instantiate these variation points before working with the semantics. For our test approach the most interesting semantics variation points are: the nature of the event store, events not enabling any transition, the selection policy of possible firing transition sets, and the execution order of the transitions in a chosen set.

For our test approach we have to instantiate the first two semantic variation points. We do not instantiate the latter two but leave them uninstantiated. Thus the test approach works correctly for different implementations of a state machine specification. Precisely, we neither want to restrict how to choose a possible set of firing transitions (if there is more than one) nor do we want to restrict the order these transitions will be executed. This is different for the event store. In order to be able to calculate the possible correct behavior allowed by the state machine specification, we need to know the nature of the event store, or with other words, we have to decide for a specific nature. In most practical contexts a FIFO queue is used to store events for further processing. Hence we assume an unbounded reliable FIFO queue as event store. Second, we assume that events that do not enable a transition when they are processed are just deleted and the next event from the event store will be processed. This implies that the state machines do not block. Technically they are called *input enabled*.

In summary, the result of the discussion about semantic variation points is twofold: first, an event queue and events to be omitted are introduced into the semantic model of state machines. Second, we need to respect different firing transition set selections and execution strategies in a test approach.

3.1 Conformance Relation for State Machines

Before we describe how to generate test cases based on our semantics definitions we describe the general test setting. As mentioned in the introduction, an embedded system comprises of hardware and software components. Thus we have to treat the system under test (sut) as a black box. We only require the sut to have so-called *points of control and observations* (pco). Thus it is possible to control the sut from the outside, i. e. to send inputs, and to observe the outputs of the sut. Figure 2 shows the abstract test architecture. As a consequence of this architecture only the inputs to the sut and the outputs of the sut are visible in the environment and thus for the tester. This particularly implies that the event queue is not visible from the outside. Thus we need to restrict the test process to the observable parts of a system under test and must respect internal details, which influence the possible behavior.

To generate test cases for a black box sut from a state machine specification, we need to extract the observable parts of the computations we defined for the semantic model of state machines. These are the events received from the environment and the generated events sent to the environment. Corresponding to the computation defined above we yield an *observable computation* by extracting and concatenating these events:

$$in_1 \frown out_1 \frown \dots \frown in_{n-1} \frown out_{n-1} \qquad (7)$$

An observable computation is a sequence of two types of events. Precisely, the sequences of received events and the sequences of generated events are concatenated.[3] The set of all observable computations form our observable execution model of state machines.

A prerequisite to evaluate automatically whether a sut conforms to a specification is a formal definition of conformance. To define conformance, we use the notion of implementation relations. De Nicola and Hennessy studied various possible characterizations of conformance [9, 8]. Brinksma and Tretmans studied various implementation relations for synchronous transition systems [4, 22]. In general, relevant implementation relations are based on the same idea of an external observer. In this idea an implementation I conforms to its specification S, if and only if all observations *obs* any external observer o can make on the implementation can be related to the observations this observer can make on the specification:

$$I \leq_o S \Leftrightarrow \forall o : \mathcal{O} \bullet obs(I, o) \sqsubseteq obs(S, o) \qquad (8)$$

To get an applicable relation you need to define the type of observers ($\mathcal{O}$), which observations these observers can make (*obs*), and how to relate these observations ($\sqsubseteq$). In our test approach we use sequences of inputs to the system under test as observers. The observations these observers can make are the resulting outputs,

[2]Note that many problems with the UML semantics arise from that point. On the one hand some of these points are not obvious in the semantics and on the other hand decisions taken by the users of the semantics are often not propagated to the outside.

[3]We assume the event store to be a queue so that received events will be stored one after another in sequence. Furthermore, transitions and actions on transitions are executed in sequence, whereat generated events are stored in a sequence.

inria-00268864, version 2 - 23 Apr 2008

i. e. the generated events, of the system under test. The relation we use to compare observations of the system under test with the observations of the specification is set inclusion ($\subseteq$). Thus we can argue that a system under test conforms to its specification if and only if the output sequences for all possible input sequences are included in the set of all output sequences of the specification for the same input sequence:

$$I \leq_{out} S \Leftrightarrow \forall \sigma : \mathrm{seq}\, E_S \bullet out(I,\sigma) \subseteq out(S,\sigma) \qquad (9)$$

Following the idea of Tretmans [22] we restrict the set of possible inputs to that of the specification. The set of outputs we calculate from the set of observable computations of a specification:

$$out(S,\sigma) == \{\delta : otraces(S) \mid \sigma = \delta \restriction E_{in} \bullet \delta \restriction E_{out}\} \qquad (10)$$

Precisely, the set of all observations $out(S,\sigma)$ for S with input sequence σ results from all observable computations of S ($otraces(S)$) for which σ denotes the input sequence ($\sigma = \delta \restriction E_S$) and $\delta \restriction E_{env}$ denotes the resulting output sequence.

Now we have a precise meaning of conformance and a guideline how to compute test cases. Based on the specification we need to calculate the traces of the state machine for all possible inputs and extract the possible correct observations. For testing the sut we need to stimulate the system under test with the particular inputs, observe the outputs and compare them to the pre-calculated possible correct observations. That means to check for their existence. Obviously a problem arrises when thinking about practical testing: the set of inputs is infinitely large[4] or pretty huge.

3.2 Selecting Inputs for Test Case Generation

When testing in practice only we are interested in relevant and interesting test cases to advantage the quality assurance process, and to use time and computation power at an optimum. Therefore, we generate a test case for a prior selected input sequence. This two-step process clearly separates the input selection problem from the test case generation problem. Thus it is possible to use different selection strategies with the same generation process and it allows to adapt the input selection process to different test aims or to different project stages.

In the TEAGER Tool Suite we implemented several input selection strategies. The strategies range from using given fixed input sequences to using specific models describing the environment. The former allows so called special value testing and is used for very specific test aims like the coverage of a certain path or state. The latter allows to model varied behavior of an environment. We use probabilities for inputs to model different environments. The most general one is an environment in which all inputs can happen at any time with the same probability (*uniform distribution*). In a more specific environment different probabilities are assigned to the inputs (*a prior distribution*). Thus the occurrence of specific inputs can be influenced. We also use a variant of this strategy where we adapt the probabilities once an input is chosen (*dependant distribution*). For every input a weight is assigned and decremented if the input is selected. If all weights are equal to zero the initial assignments will be used. With this strategy we ensure that eventually every event is chosen. The most expressive way to describe the environment is to model it with probabilistic state machines. Using state machines allows to model dependencies among inputs in

[4] If we think of embedded systems as non-terminating systems.

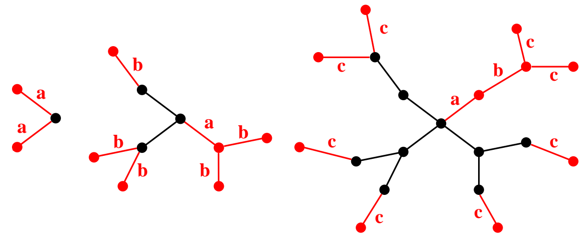

Figure 3: Stepwise State Space Exploration for `[a,b,c]`.

a sequence. It also allows to completely reassign input probabilities depending on the assumed state of the system under test. For example, the probability of dialing a number before lifting the receiver of a telephone is certainly different from the probability of dialing after lifting the receiver. In summary, we use different complex strategies to describe assumed environments to select relevant and interesting inputs.

3.3 Test Case Generation Algorithm

With the decision to consider a finite set of finite sequences of inputs we can calculate all possible correct observations for these inputs. We use this information to be able to automatically evaluate the test execution. Considering complex data during the test case generation process is not scope of the present paper and we skip the corresponding details here. In the current implementation data are chosen randomly while generating test cases. The problem of test cases with data and which specific data to choose is part of ongoing research. To calculate the possible correct observations we stepwise explore the state machine's state space for the given input. The challenge here is to correctly consider all semantic subtleties. We do this in a two step algorithm:

First, we initialize the state machine with its initial status, i. e. with its initial configuration and an empty queue. Then we insert the first input event to the event queue. Now we apply a semantic step to this configuration: first, we calculate all possible firing transitions sets. For every transition set and every possible execution order of the transitions inside these sets we calculate the resulting status. It is important to note that we calculate a fix-point for this set. That means, that no new status can be reached from any calculated status. Thus we yield a set of all reachable status including all intermediate status for the first event. To store the intermediate status is important for handling possible interleavings of input and internally generated events. Second, we insert the next event to every reachable status in the previously calculated set. By doing so we respect possible interleavings of events in the event queue. Then we again calculate all reachable status for this input and proceed in the same way for the other inputs. We calculate the graph of all execution paths which includes the reachable status. Figure 3 and 4 show such graphs. Only this stepwise calculation of all reachable status ensures that all possible execution paths for the given input are calculated. This includes all non-determinism in the specification (modeled and arising from the semantic model of state machines) and effects from processing events asynchronously.

Example. Let us assume an internal event `i`. Processing this event from the queue `[a]` will produce a new internal event `j`. Event `i` will be generated in response of input event `a`. For the next step we want to process the input sequence `a b`. The problem is, that while testing we cannot observe the queue of the system under test. So we do not know how event `b` will interleave with the internal

inria-00268864, version 2 - 23 Apr 2008

inria-00268864, version 2 - 23 Apr 2008

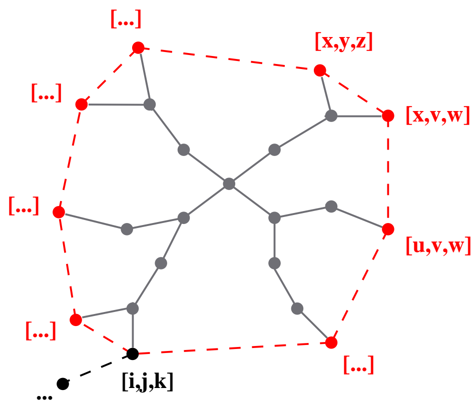

Figure 4: Hull with calculated sequences of observations.

events. So we first insert event `a` into the queue and calculate that there are three reachable status with different queues: `[i]`, `[j]`, `[]`. The first queue results from just processing `a`. The second results from processing `a` and `i`. The third results from processing `a`, `i` and `j`. By inserting input `b` into all the queues we prepare for respecting all possible interleavings. The resulting queues are `[i,b]`, `[j,b]`, `[b]` and during the next step `[b,j]` which properly respects one possible interleaving. Due to inserting the next event to all reached status, event `b` will also be inserted to the queue `[a]`. This results in the queue `[a,b]` reflecting the situation that the environment triggered both events before the system under test processed the first one. Figure 3 shows the principle of the stepwise state space exploration for the input sequence `[a,b,c]`.

After processing all events from the input sequence we can identify among the set of all reached status those status which are finally reached by processing the complete input sequence. These nodes on the hull of the execution graph are so-called *quiescent*. That means that their event queue is empty and thus they cannot proceed without a new input from the environment. Figure 4 shows an execution graph with the status on the hull. We now extract from these status the observations, which would be emitted when executing a particular path. These observations are the events which the state machine sends to the environment. All observation sequences comprise the possible correct observations we can make when triggering the system under test with the input sequence. Our idea is now to treat all observations as an alphabet for a language. The calculated observation sequences form accepted words of these language causing the test execution to pass. All other sequences cause the test execution to fail. We now just need to build an acceptor for the calculated observation sequence and use them as our test oracle to automatically evaluate the test execution.

Before we can do that we need to solve one problem which can arise when calculating the observation sequences. We argued that we calculate a fix-point for the set of reachable status. Due to the fact that the state machine can generate (internal) events and produce internal infinite loops the calculation of the fix-point does not terminate in any case.[5] Figure 4 shows such a situation in the lower left corner. To solve the problem we limit the number of steps to an upper bound. Technically, every reached status has got a counter

[5]Here we subsume the problem that the time to calculate the fix-point is unacceptable high.

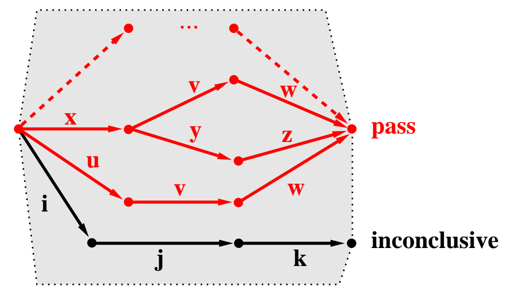

Figure 5: Acceptance graph with an *inconclusive* test verdict.

for the number of steps necessary to reach this status. If a counter reaches a specified upper bound we mark this status and abort further processing of this status. As a consequence we calculate two types of observation sequences. One which could be calculated within the given bound, and one which could not. The latter type could be interpreted as follows: all observations made so far are correct, but not all observations could be calculated. Hence, after processing all observations, we have no further information to compare the output of the system under test. We can neither say that further observations are correct nor can we say that they are not correct. We only can stop testing the system under test with this input sequence and give an *inconclusive* test verdict. This verdict says that all observations so far are correct but that we stopped processing the current execution path further. It would also be possible to decide for a *pass* or a *fail* verdict. But introducing a third verdict allows a finer distinction of differently caused test executions results.

As a consequence we have to distinguish the two sets of possible observation sequences. The acceptance graph we build out of these sets comprises two accepting nodes. One for all observation sequences which could completely be generated and one for all observation sequences which were bounded. The acceptor itself is a deterministic finite automaton accepting both sets of observation sequences. A test case execution finishing in one of these nodes results in a pass or an inconclusive verdict. All observations not covered by the acceptance graph result in a fail verdict. Figure 5 shows an acceptance graph for the observations of Figure 4.

Algorithm 1 shows the control structure of the test case generation algorithm. The loop will be executed as often as inputs should be sent to the system under test in the test case. The inner while-loop controls the fix-point calculation of reachable status. While there are newly generated status the simulation step is successively repeated to calculate all reachable status. If there are no newly generated status the algorithm proceeds with the next input event. The results of the loop are a set of all completely calculated observation sequences and a set of all incompletely calculated observation sequences. Out of these sets an acceptance graph will be calculated. Algorithm 2 shows the calculation of the successive status for the calculated status in the previous step. First, the state machine is initialized with the configuration from the status and the next trigger event is selected from the corresponding event queue. Then, all possible firing transitions sets and all possible transition execution orders are executed to estimate the resulting status and the generated events. This includes: saving reached configuration, adding internal events to the input queue, and saving generated events which should be sent to the environment. The latter events are the possible correct observations which we use to build the acceptance graphs.

inria-00268864, version 2 - 23 Apr 2008

```
input  : state machine: sm
output: an acceptance graph
sm.configuration ← initial configuration
result ← initial simulation node
inconclusives ← ∅
while |trigger| < input length do
    trigger ← generate a new trigger
    store ← ∅
    forall node ∈ result do
        node.queue ⊕ trigger
        store ∪ {node}
    steps ← 0
    while result ≠ ∅ ∧ steps < limit do
        temp ← simulationStep(result)
        steps ← steps + 1, result ← ∅
        forall node ∈ temp do
            if steps = limit then
                inconclusives ∪ {node }
            else
                store ∪ {node}
                result ∪ {node}
    result ← store;
generateAcceptanceGraph(result,inconclusives)
```

Algorithm 1: Test Case Generation: Control Structure.

```
input  : set of simulation nodes: input
output: set of new generated simulation nodes: result
result ← ∅
forall node ∈ input do
    if node.queue ≠ <> then
        sm.configuration ← node.configuration
        event ← node.dequeue
        forall T_∥ : sm.getFTS(event) do
            permutations ← permute(T_∥)
            forall firing_transitions: permutations do
                effects ← []
                forall t: firing_transitions do
                    effects ← fire(t)
                temp ← node
                temp.configuration ← sm.configuration
                forall effect: effects do
                    forall ev: effect do
                        if ev ∉ E_SM then
                            temp.observation ⊕ ev
                        else
                            temp.queue ⊕ ev;
                result ∪ {temp }
                sm.configuration ← node.configuration
return result
```

Algorithm 2: Test Case Generation: Simulation Step.

Both: the successive status and the generated events will be stored in a new simulation node. The set of all new simulation nodes will be returned as the result of the *simulationStep*.

A test case comprises of the input sequence to stimulate the system under test and an acceptance graph to automatically evaluate the execution of this test case. The length of a test case and the number of test cases can be influenced by the selection policy of input sequences as explained above. The generated test suite is *sound*. That means that no correct systems under test will be rejected due to a test case. Instead, the test verdict fail will only be assigned if the observation of the system under test cannot be explained by the possible correct observations of the specification (see the conformance relation for state machines). This is true because we calculate all possible execution paths to generate the sets of possible correct observations. With unlimited computation power and time the presented algorithm is able to compute a *complete* test suite, which is capable to exactly differentiate between correct and incorrect implementations.

The presented algorithm has exponential complexity. The exponential complexity arises from the branch factor introduced by the different sets of firing transitions, the different possible execution orders of transitions, and the necessity to consider possible interleavings in the event queue. Thus the effort to calculate a test case grows with the length of the input sequence and indirectly by the number of internally generated events ($f(\tilde{x})$). The branch factor is bounded by the finite number of transitions and the finite number of events (c). Thus we can approximate the effort A to generate a test case for a given input sequence of length x as follows:

$$A(x) \sim e^{c \cdot (x + f(\tilde{x}))} \tag{11}$$

The exponential effort is visualized in Figure 6 by the doubly dotted curve.

3.4 Combining Test Sequences

When testing non-terminating embedded systems it is also interesting to execute longer input sequences. To reduce non-determinism in the specification is not possible without any further knowledge about the system under test. Thus we concentrate on the asynchronous event processing. The lion's share of the calculation effort results from respecting all interleavings of the input sequence with internal generated events. Now we can argue that it is not necessary to consider all of these interleavings. For example, in practice it is the case that the system under test immediately starts to process the first received input. It usually does not wait until "ten" events are received from the environment. With the distance of these events the probability falls that an internally generated event (as a consequence of processing, for example, the first input) interleaves with the tenth generated event.

Based on this idea we developed various strategies to reduce the calculation effort. To demonstrate the core idea we implemented a strategy where we introduce so called *observation points*. Observation points are points in time where we give the system under test enough time to calculate its reaction. Compared to our semantic model of state machines the system under test reaches a status in which the event queue must be empty. So no more reaction can be produced for the given input. This is the same situation as explained above for the general algorithm to calculate the possible correct observations – all the status on the hull are quiescent. Continuing after such an observation point now means: to enqueue the next input to all (non-inconclusive) status on the hull of the previously calculated execution graph (note that for these status the

inria-00268864, version 2 - 23 Apr 2008

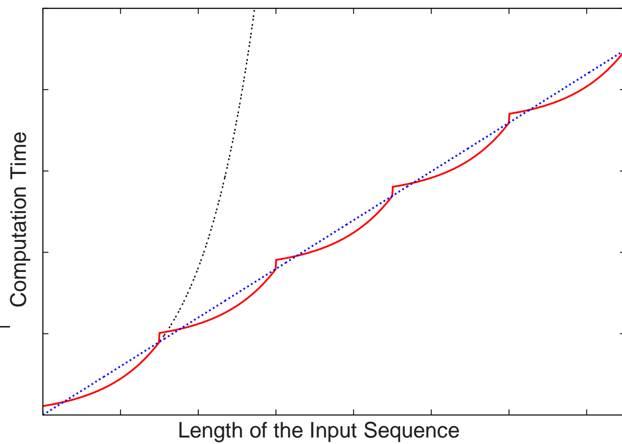

Figure 6: Linearization of the exponential Complexity.

event queue is empty). We also reset the collected possible observations. We can do so because at an observation point we assume the system under test to have completely calculated its reactions. These reactions will be checked by the last proceeding acceptance graph.[6] Now we can proceed to calculate the possible correct observation sequences for the complete next input sequence.

The reduction in the computation effort results from the fact that we do not consider possible interleavings resulting from events in the first input sequence with events in the second input sequence. Figure 6 visualizes this effect. We now repeatedly calculate only the first part of the exponential curve. The overall calculation effort follows from adding the efforts needed to calculate the observations for the individual input sequences. The average effort has a linear gradient depicted by the dotted line. Compared to the effort for processing one input sequence with the length of the sum of all sub-sequences this is an enormous reduction in the calculation effort. Now the effort for combined test sequences still grows exponentially with the length n of the particular input sequences but linear with the number x/n of combined sequences and consequently with the length x of the overall input sequence:

$$A_{comb}(n,x) \sim \frac{e^{c\cdot(n+f(\tilde{n}))}}{n} \cdot x \qquad (12)$$

The consequence of this reduction is that now the possible correct behavior is over-approximated. We do not calculate all possible observation sequences for the composed input sequence. Instead, we approximate them in the way that we treat more behavior to be correct, i. e. more observation sequences to be correct. Therefore, this kind of a test case is weaker because it is not able to detect all errors a test case with only one input sequence would detect. But the test case is still sound. We do not reject correct systems under test with this strategy. The over-approximation follows from the fact that observation sequences from different acceptance graphs can be combined in any possible order. This would not be possible for a complete input sequence. Depending on the used testing strategy we can now parameterize how test cases should be generated and combined. On the one hand by the effort we need to process the total count of inputs, and on the other hand by the reduction capability when splitting the input sequence into smaller parts. Figure 7 shows the general structure of a combined test case. When reaching a pass node in an acceptance graph for an input sequence we can continue to trigger the system under test with the next input sequence and check the newly generated output of the system under test at the next observation point.

[6] An improvement of this strategy would be to collect possible correct observations for more than one observation point.

First experiments with this static strategy showed that if we can introduce such observation points for the system under test this strategy works quite well. But further research and experiments are needed to investigate more elaborate (dynamic) strategies. We especially think of using knowledge about the system under test like specific properties of the used buffer to store events, introduce probabilistic strategies to handle possible interleavings or to observe the memory consumption of the system.

3.5 Evaluating the Test Process

When a test suite is generated with the algorithm above and a system under test is tested with this test suite we would like to know how extensively we tested the system under test. The number of test cases and the length of the input sequences in the test cases only conditionally allow to draw conclusions related to that question. Still today the question is hard to answer. The mostly used approach is to measure the coverage of different (structural) elements of the system under test or the specification. For general code this is common practice. The used criteria are usually based on control flow or data flow information in the code or on functional description in the specification. With our test approach we address embedded systems composed of hardware and software components. You can apply well known techniques to measure coverage in the software components, but our impression is that this is not sufficient for such systems. To measure coverage in the hardware components is usually not possible. The only way to regard the whole system is to use the specification. Thus we need to develop meaningful criteria for state machines. Our current work introduces different criteria based on structural elements of state machines, like states and transitions, and on semantic elements, like configurations and sets of firing transitions. First results show that especially semantic criteria are able to evaluate the behavior in a meaningful manner. For the future we particularly address such semantic criteria based on the semantic model of state machines and their relation to selected input sequences. A still open and interesting question is whether it is possible to use those criteria to control the test case generation process, viz. to measure coverage while generating test cases and to select the next inputs according to this coverage.

4. TOOL SUPPORT

To evaluate and to show the practicability of our approach we implemented the TEAGER Tool Suite. Figure 8 shows the general architecture of the TEAGER Tool Suite. TEAGER consists of an environment to automatically generate and execute test cases, and additionally of an environment to execute state machine specifications. The latter we use to analyze the execution behavior and the testability of a state machine, and to measure coverage on a state machine specification to evaluate generated test suites.

The *Test Case Generation and Driver* component contains the *Test Case Generator* and the *Test Driver*. The Test Case Generator we use to automatically generate test cases out of a state machine specification. For selected inputs a loaded state machine specification will be executed step by step to compute the possible correct observation sequences. Based on them an acceptance graph as the test oracle is generated. Input sequences and acceptance graphs will be stored for each test case in separate files for later execu-

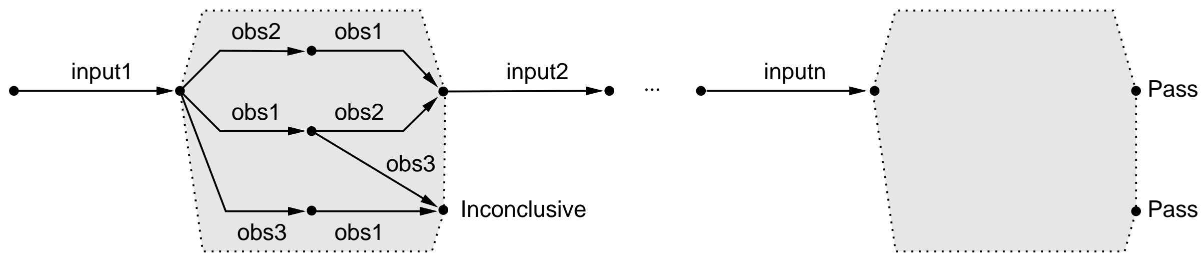

Figure 7: General Structure of a combined Test Case.

tion. The Test Driver in turn loads saved test cases and executes them. The execution includes both: stimulating the system under test and comparing the observation to the computed possible correct behavior in the acceptance graphs. The communication with the system under test takes place over a socket connection using pre-implemented adaptors. This concept offers a flexible way to connect the system under test. It also offers the possibility to use our *State Machine Executor* as a system under test stub. Thus we can analyze the execution behavior of state machine specification or measure the coverage of a used specification.

The complete test case generation process is parameterized to have maximal control over the structure of test cases and the effort needed to calculate them. First you can specify the number of test cases to be generated, the length of input (sub-)sequences and the number input sequences to be combined in a test case. Then you can specify the way input sequences are generated. As an example, we implemented different probabilistic strategies which describe possible environments (cf. Section 3.2). It is also possible to specify the sequence of events as a preamble to generate test cases related to specific parts of the specification. After configuring all parameters the test case generation works completely without any user interaction.

For test case execution you can control the frequence at which inputs are sent to the system under test. To avoid a fixed timing we use a Gaussian distributed trigger rate with a mean value and deviation to be specified by the tester. The tester also specifies the number a test case should be repeated, and the policy how several execution results should be combined. Executing a test case several times is especially necessary when dealing with non-deterministic systems. Every execution can cause the system to execute a different path for the same input. However, we need to check all resulting observations. How often test cases should be executed when dealing with different non-determinism cannot be fixed in advance. Thus the given number is a so-called test hypothesis. The term hypothesis expresses that we assume the number high enough to test the system under test adequately. The timeout value which can be specified is also an test hypothesis. This upper time bound specifies how long the test driver should wait for a desired observation. Usually, this time bound is higher than the reaction of the system. So the system under test has enough time to produce the reaction for an input. The timeout value directly influences the test case execution behavior since we use it to implement our observation points. Up to this bound all reactions can be observed and (as important as the previous fact) no other reactions can be observed. A value too short would cause unnecessary *false negatives*; a value too high would unnecessarily slow down the test execution.

For the combination of different execution results of the same test case different strategies are imaginable. Actually we use three different strategies: *MUST* requires every test execution to pass. *STRONG_MAY* requires at least one test execution to pass, whereat the test execution will be repeated (up to the number of test repetitions) until the first test case passes. *WEAK_MAY* requires that no test execution fails.

The State Machine Executor executes a state machine and thus allows an exploration of miscellaneous properties like the testability or the coverage of a specification which we use to evaluate the quality of a generated test suite. Here it is particularly important to avoid a fixed execution timing. For UML state machines the so-called *zero time assumption* does not hold. Instead, it is assumed that executing a transition consumes time. To respect this and especially to be able to investigate effects of the used asynchronous communication we also use a probability based scheme for the execution times of transitions. The tester can specify the mean value and the deviation for a Gaussian distribution which is used to select an execution time of every transition execution. Thus effects of different timings can be tested.

First experiments with the TEAGER showed, that we can meet the state explosion problems introduced through the semantics of state machines with our approximation strategy. The generation and execution process is parameterized. This allows the application of different testing strategies and to have maximal control over the complete process. For more information about the TEAGER Tool Suite, its individual components, and the used parameters, we refer the interested reader to our web site [19].

5. SUMMARY AND OUTLOOK

Testing benefits from the fact that the real system is brought to execution. Thus, the interaction of the real hardware and the real software can be evaluated. It aims in falsification, i. e. to show inconsistencies between the specification and developed system. Testing is applicable at different levels of abstraction and at different stages of the development. With our approach UML state machines can be used in the quality assurance to serve as a specification for the desired reactive behavior of the system. It is possible to select relevant and interesting inputs for a test case and to calculate the possible correct observations for given inputs. They allow to automatically evaluate test executions which is in general a difficult and time consuming task. Applied approximation makes the generation process practical, whereat it is possible to control this process depending on the time and computation power to invest.

Especially the modularization of the different task in automatically generating test cases makes the approach interesting for further research. All discussed strategies are implemented as modules of the tool suite. Thus, different strategies for selecting inputs, for com-

inria-00268864, version 2 - 23 Apr 2008

inria-00268864, version 2 - 23 Apr 2008

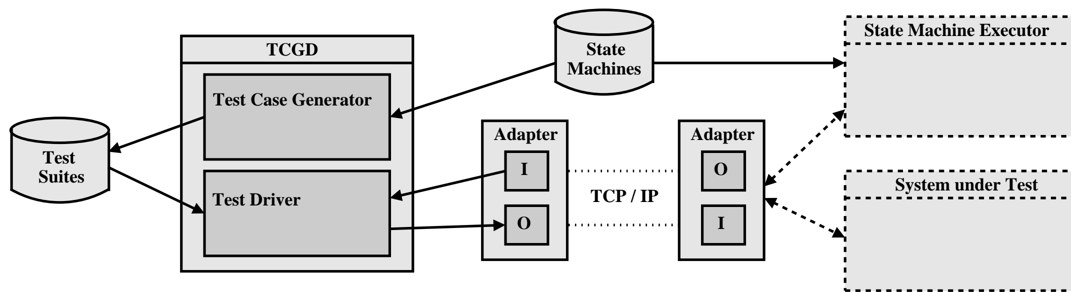

Figure 8: Architecture of the tool suite TEAGER.

bining test cases to reduce the calculation effort, or to select relevant data during test case generations can be studied independently from each other. For using this approach in practice it also enables to adapt it to different needs.

Compared to a lot other works we use a precisely defined semantics for UML state machines in our test approach. We do not restrict state machines to ease test case generation. Instead, we follow the semantics description of the UML standard [24] as much as possible. Only misleading or conflicting statements are clarified. We address all semantic details which arise from the different sources of non-determinism. In particular we also address the problem of asynchronous communication which is introduced to the *run-to-completion* semantics. You can question these points, but first, real embedded systems behave like this way and second, UML state machine models allow such behavior without any user action. More or later you have to deal with these points.

An early work on testing on the basis of transition systems is the work of Chow [6] (W-Method). Bourhfir et al. [3] presented several approaches for conformance testing based on extended finite state machines. Ural [25] reviewed several methods for generating test sequences from finite state machine based specification. De Nicola and Hennessy [9] introduce a formal theory of testing on which (later on) Brinksma [4] and Tretmans [22] build approaches to derive test cases from a formal specification. In contrast to our work, the approaches of Brinksma and Tretmans assume the testing process to communicate synchronously with the system under test. Tretmans developed a tool called TorX which allows conformance testing of reactive systems [23]. The used internal representation is based on (input/output) labeled transition systems. TorX is mainly used to perform on-the-fly testing. Newer work also addresses problems of selecting inputs and data, batch test case generation or asynchronous behavior [10, 12]. A detailed overview of the fundamental literature for classical formal testing can be found in Brinksma's and Tretmans' annotated bibliography [5]. Grieskamp et al. at Microsoft Research use finite state machines as the underlying model for automated testing [14]. The approach is based on a AsmL [13] specification for which a FSM model is generated. In contrast to our work, the test approach uses synchronous method calls to test the system under test. The needed data to instantiate the methods parameter are automatically chosen. The state space exploration is controlled by so-called test properties and filters. The general principle to explore the state space is comparable to our approach. Belli et al. (see [2] and the work cited there) base their testing methodology on a variant of state machines. In contrast to our approach, they do test against a fault model that has to be set up explicitly. They do not execute the state machines directly, but represent them as event sequence graphs. Auguston et al. [1] use environment models for test case generation. In contrast to our approach, they do not use state machines, but attributed event grammars for modeling.

Beside 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 [20]. Simulink Verification and Validation generates test cases in Simulink and Stateflow, and measures test coverage for Statecharts [21]. Conformiq Software Ltd. offers a *Test Generator* which accepts "extended UML state charts" as the model of the system under test for dynamic testing [7]. The tool provides on-line and off-line test execution with test coverage measuring and report generation. Due to a lack of technical documentation, the relation to UML state machines is unclear. *AsmL 2* by Microsoft provides an executable specification language based on the theory of Abstract State Machines [11]. 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.

Our ongoing research deals with a comprehensive integration of our approach into an UML-based development. In particular we address questions: how to combine our approach with a component-based development approach and how to combine our technics with other successfully applied testing technics like scenario-based technics testing interfaces. Furthermore we integrate more and more syntactical elements into our formal semantics and analyze their influence on the automated test generation process. Perspectively we address two challenges, namely specifying and testing timed behavior and considering complex data, not only in random fashion, to generate "interesting" test cases. We also develop technics to control and evaluate our automated processes. Mesuaring coverage, especially on the specification, is one step into this direction. Currently we are analyzing criteria based on state machines and their semantic model. In conclusion we aim at reasonable automated processes to support and guide the developing embedded systems.

6. REFERENCES

[1] 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.

[2] F. Belli and A. Hollmann. Holistic Testing with Basic Statecharts. In W.-G. Bleek, H. Schwentner, and H. Züllighoven, editors, *Software Engineering 2007 –*

inria-00268864, version 2 - 23 Apr 2008

Beiträge zu den Workshops, Lecture Notes in Informatics 106, pages 91–100. Gesellschaft für Informatik, 2007.

[3] C. Bourhfir, R. Dssouli, and E. M. Aboulhamid. Automatic Test Generation for EFSM-based Systems, 1996.

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

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

[6] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Trans. Softw. Eng.*, 4(3):178–187, 1978.

[7] Conformiq Software Ltd. Conformiq Test Generator, 2008. `www.conformiq.com`.

[8] R. De Nicola. Extensional Equivalences for Transition Systems. *Acta Informatica*, 24(2):211–237, 1987.

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

[10] L. Feijs, N. Goga, M. S., and J. Tretmans. Test Selection, Trace Distance and Heuristics. In I. Schieferdecker, H. König, and A. Wolisz, editors, *Testing of Communicating Systems XIV*, pages 267–282. Kluwer Academic Publishers, 2002.

[11] Foundations of Software Engineering Group. Asml 2. Microsoft Research, 2008. `research.microsoft.com/fse/asml`.

[12] L. Frantzen, J. Tretmans, and T. A. C. Willemse. A Symbolic Framework for Model-Based Testing. In K. Havelund, M. Núñez, G. Rosu, and B. Wolff, editors, *Formal Approaches to Software Testing and Runtime Verification, First Combined International Workshops, FATES 2006 and RV 2006, Seattle, WA, USA, August 15-16, 2006, Revised Selected Papers*, volume 4262 of *Lecture Notes in Computer Science*, pages 40–54. Springer, 2006.

[13] W. Grieskamp, Y. Gurevich, W. Schulte, and M. Veanes. Generating Finite State Machines from Abstract State Machines. In *Proceedings of International Symposium on Software Testing and Analysis (ISSTA 2002)*, pages 112–22, Rome, Italy, 22–24 July 2002. IEEE.

[14] W. Grieskamp, L. Nachmanson, N. Tillmann, and M. Veanes. Test Case Generation from AsmL Specifications. In *10th International Workshop on Abstract State Machines (ASM'03)*, volume 2589 of *Lecture Notes in Computer Science*, page 413, Mar. 2003.

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

[16] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *Computer*, 30(7):31–42, 1997.

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

[18] D. Seifert. *Automatisiertes Testen asynchroner nichtdeterministischer Systeme mit Daten*. Shaker Verlag, 2007.

[19] D. Seifert. The TEAGER Tool Suite. Test Execution and Generation Framework for Reactive Systems. Technische Universität Berlin, 2008. `swt.cs.tu-berlin.de/~seifert/teager.html`.

[20] Telelogic. Rhapsody ATG und Rhapsody TestConductor, 2008. `www.telelogic.com`.

[21] The MathWorks Inc. Matlab/Simulink, 2008. `www.mathworks.com`.

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

[23] J. Tretmans and A. Belinfante. TorX. University of Twente, 2008. `fmt.cs.utwente.nl/tools/torx`.

[24] UML2. Unified Modeling Language: Infrastructure and Superstructure. Object Management Group, 2007. Version 2.1.1, formal/07-02-03, `www.uml.org/uml`.

[25] H. Ural. Formal Methods for Test Sequence Generation. *Comput. Commun.*, 15(5):311–325, 1992.