13 Real-Time and Hybrid Systems Testing

Kirsten Berkenkötter[1] and Raimund Kirner[2]

[1] Department of Computer Science
University of Bremen
kirsten@informatik.uni-bremen.de

[2] Real-Time Systems Group
Vienna University of Technology
raimund@vmars.tuwien.ac.at

13.1 Introduction

Real-Time and Hybrid Systems A system whose functionality is not only dependent on the logical results of computation but also on the time in which this computation takes place is called **real-time system**. We speak of **hard real-time** if timing constraints always have to be met exactly. In contrast, **soft real-time** allows lateness under specified conditions.

Similarly, **hybrid systems** also consider time to determine if computation works correctly. They are called hybrid as both time-discrete and time-continuous observables exist as well as time-discrete and time-continuous behavior. Variables may have dense values that change with respect to time while events occur discretely. Assignments to variables are also made at discrete points in time. Therefore the behavior of a hybrid system consists of time-continuous parts where variable evaluations change with respect to time and of time-discrete parts where events occur and assignments to variables are performed.

Both kinds of systems are used, e.g. in avionics, in automotive control, and in chemical processes control systems. They are often embedded systems with a probably safety-critical background. This leads to high demands on both modeling and testing for providing high quality.

Testing As stated above, both real-time and hybrid systems are potentially hazardous systems. Obviously, temporal correctness is an important issue of real-time systems. As a result, testing real-time systems is more complex than testing untimed systems as time becomes an additional dimension of the input data space. In case of hybrid systems, complexity increases again as the value domain is continuous instead of discrete.

For model-based testing, the main goal is handling this complexity. On the one hand, this means building models that allow to abstract from details to reduce complexity in a way that test cases can be generated. On the other hand, test cases must be selected in a meaningful way to achieve a manageable number out of them.

Outline In Section 13.2, we discuss test automation in general and with respect to the special needs of real-time and hybrid systems. We then focus on model-based test case generation in Section 13.3. Different modeling techniques like timed process algebras and timed automata are discussed for real-time and hybrid systems as well as their application in test case generation. After that, evolutionary testing is introduced as a method for improving generated test suites with respect to testing timing constraints in Section 13.4. We conclude with a discussion of the presented techniques in Section 13.5.

13.2 Test Automation

Testing is one of the most time consuming parts of the software development process. Hence, a high degree of **automation** is needed. The general structure for a test automation system is the same for untimed, timed, and hybrid systems. In contrast, the internals of the different parts of a test automation system differ, as the specific characteristics of timed and hybrid systems must be considered.

13.2.1 Overview

First, we have to introduce some testing terminology. **Testing** itself means the execution of the system to be tested which we call **system under test** (**SUT**). The SUT is fed input data while the output data is monitored for checking its correctness.

A **test case** is a set of test inputs, execution conditions, and expected results. This does not mean that the test case necessarily gives explicit input data, it may also specify rules for generating test data. The set of test cases for a SUT is called **test suite**. The **test procedure** gives detailed information about the set-up and execution of test cases and the evaluation of the test results. If only the interfaces of the SUT are accessible, a test is called **black-box test**. In contrast, if also internal states can be observed and influenced, a test is called **white-box test**.

As stated in Peleska et al. [PAD$^+$98], a test automation system consists of several logical building blocks as depicted in Figure 13.1:

- Model The **test model** represents the required functionality of the SUT. More precisely, it abstracts the functionality to a sensible size to allow the generation of a test suite with manageable size. Different representations can be chosen, e.g. automata or process algebra.
- Test Generation The **test generator** uses the model for deriving test cases. In addition, more specific **test case specifications** that describe a test case or a set of test cases can be used for this purpose. The test generator is responsible for achieving a manageable number of test cases out of the infinitely many possible ones. Therefore it plays an important role in the test automation system.
- Test Monitoring The execution of test cases must be monitored to keep track of inputs and outputs. This is done by the **test monitor**. On the one hand, this is needed for documentation purposes. On the other hand, results

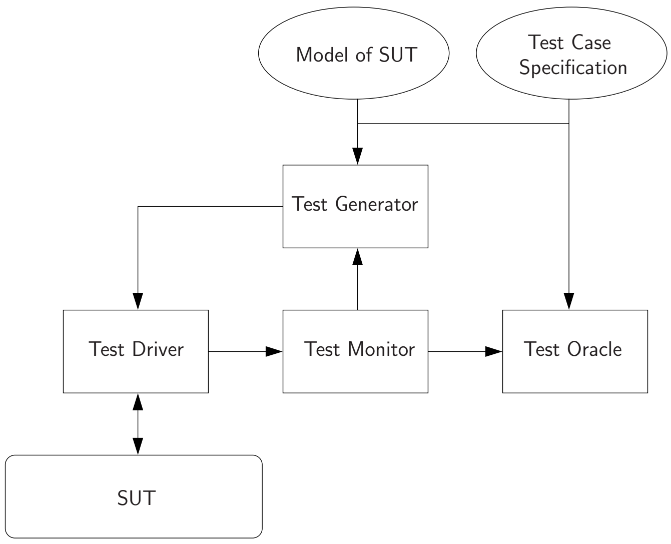

Fig. 13.1. Test automation system

may be needed during execution. To give an example, nondeterminism can occur. Then the output of the SUT is needed to decide which input can be sent next. It is also desirable to know if a test case fails during execution as the test can be aborted then. This holds especially for long time tests.

- Test Evaluation Either after or during test execution, the test result has to be assessed. This is the task of the **test oracle**. According to the monitored test data, SUT model, and test specifications, it calculates the **test verdict**. **Passed**, **failed**, and **inconclusive** are frequently used as verdicts.
- Test Driver The parts of the test automation system converge in the **test driver**. It executes the generated test cases, i. e. it posts inputs to the SUT and receives corresponding outputs. This includes providing interfaces from and to the SUT. Therefore, the test driver works hand in hand with the test monitor.

13.2.2 From Untimed to Timed to Hybrid Models

Due to the time-dependent behavior of a real-time system, new problems arise when testing it as time is a relevant factor. For hybrid systems, also dense-valued variables must be considered. These problems must be analyzed to find suitable abstractions for the SUT model as done in Peleska et al. [PAD+98].

With respect to testing, the sequential components of an untimed system are obviously best understood. Correct behavior is surveyed by looking at the initial and final states of such a component. Concurrent components are more difficult to test as not only the data processed must be correct but also the order in which it is processed. Furthermore, the interactions between the processes lead to many internal states. The amount of test cases and test data to be evaluated can be reduced by describing not only the system under test but also the environment. In this way, input and output are specified for a certain environment, so the number of possible values is reduced to a more manageable size. Also redundancy checks are performed that delete redundant test sequences.

Adding time to the requirements of a system means adding complexity to testing. In addition to the correct sequencing of inputs and outputs, the time at that they occur is crucial. If only **discrete time** is considered, time can be abstracted as a counter that is regularly incremented. The fuzziness of measuring time must be considered in the test automation system. This is the case for real-time systems with both discrete time and data domain.

If it becomes hard to define time as multiples of discrete time intervals, **dense time** has to be taken into account. This is closer to reality as time is naturally dense. Timing constraints are then given with respect to the real numbers. Data is still considered **discrete**. Nevertheless, a model that uses dense time increases complexity even more, so this must be taken into account, e.g. for test case generation. This is the case for real-time systems with discrete data domain and dense time domain.

The last step to be taken is regarding also **dense-valued**, i. e. analog data as done for hybrid systems. This is an abstraction for both analog sensors and actuators and also sensors and actuators that are discrete but have very high sampling rates. In the model the evaluation of dense-valued variables is specified by piecewise continuous functions over time that may be differential. Therefore the time domain and the value domain of hybrid systems are both dense.

13.2.3 Real-Time and Hybrid Systems

The different components of the test automation system that has been described in Section 13.2.1 have to be modified as the time and value domain of real-time and hybrid systems must be considered. As described in Section 13.2.2, there are real-time systems with discrete time domain, real-time systems with dense time domain, and hybrid systems with both dense value and dense time domain. These different abstractions of the SUT have to be mirrored in the corresponding test automation system.

- Model Obviously, adequate modeling techniques have to be chosen for testing real-time and hybrid systems. For real-time systems, there are several types of timed automata and timed process algebras that consider either discrete time or dense time in modeling. In case of hybrid systems, dense values for variables must also be taken into account. Here, hybrid automata and hybrid process algebra can be used for describing the SUT. It is important to notice that appropriate abstractions from the SUT must be found to obtain a manageable model.
- Test Generation In comparison with untimed systems, test generation is much more difficult for timed and hybrid systems as the search space of the model increases with discrete time, dense time, and (in case of hybrid

systems) dense values. Therefore, the algorithms needed for generating test cases have to be chosen carefully to gain a meaningful and manageable test suite.
- Test Monitoring Again, the important factor is time. For monitoring the execution of test cases, not only inputs and outputs to and from the SUT are relevant, also the time at that these inputs and outputs occur must be logged adequately.
- Test Evaluation Test evaluation also depends on time. Not only the correct order of inputs and outputs is needed for deciding if a test has failed or passed, the correct timing is also a crucial factor. An unavoidable fuzziness for measuring time must be considered. The same holds for measuring dense-valued data with respect to hybrid systems.
- Test Driver For real-time systems, the test driver must be capable of giving inputs at the correct time, i. e. it must be fast enough for the SUT. For hybrid systems, the same holds for dense data values, e. g. differential equations have to be processed fast enough to model the valuation of data values over time. If analog, i. e. dense-valued, data is expected as input from the SUT, this must be generated. The most important function of the test driver for real-time and hybrid systems is therefore bridging the gap between the abstract model and the corresponding implementation, i. e. the SUT. On the one hand, it has to concretize the input from the model to the SUT. On the other hand, it has to abstract the output from the SUT for the test monitor and test oracle.

All in all, we can identify two main issues for testing real-time and hybrid systems:

- Building manageable and meaningful models from the SUT.
- Finding manageable and meaningful test suites.

Test oracle, test monitor, and test driver are more a challenge in implementation. They have in common that they depend on the model and the test case generation to work efficiently. Therefore, the focus in the following chapters is on modeling and test case generation.

13.3 Model-Based Test Case Generation

The aim of model-based testing is to derive test cases from an application model that is an abstraction of the real behavior of the SUT. As described in Section 13.2.2, time can be abstracted either as discrete or dense time for real-time systems. Values are considered discrete. For hybrid systems, the time domain is dense just as the value domain. There are approaches for modeling each of these systems as well as test case generation algorithms based on these models. These are described in the following.

13.3.1 Real-Time Systems – Discrete Time and Discrete Values

First, we focus on real-time systems that consider both time and value domain as discrete. Different modeling techniques can be used here, e. g. process algebras as the **Algebra of Communicating Shared Resources** (**ACSR**) [LBGG94, Che02] or **Timed Communicating Sequential Processes** (**TCSP**) [DS95, Hoa85] or different variants of **Timed Automata** [AD94].

In this section, we discuss ACSR as a process algebra and **Timed Transition Systems** (**TTS**) as an example for automata as test case generation algorithms for these exist. Timed CSP has also been used for testing purposes, but here research has been done with regard to test execution and test evaluation and not test case generation [PAD$^+$98, Pel02, Mey01]. Test case specifications are written in Timed CSP and then executed instead of deriving a test suite from a Timed CSP model of the SUT.

Describing Real-Time Constraints Before describing the formalisms used in testing frameworks to model real-time applications, we introduce a more intuitive graphical language to describe real-time constraints called **constraint graph** language. We use it to present a simple example of a real-time application that is used throughout this chapter.

A constraint graph (CG) is a directed graph $G(V, E)$ with a distinguished starting node $\varepsilon \in V$. The nodes of a CG represent I/O events where input events are given by there name while output events are marked with a horizontal line above their name, e.g. $\overline{event}$. The edges E of CG are illustrated as $f \xrightarrow[F]{T} g$, where f denotes the source I/O event and g denotes the target I/O event. T is the time constraint that guards the edge. It can be an interval $[t_1, t_2)$ or a fixed delay t. F denotes the possibly empty set of forbidden I/O events that may not occur before receiving the target event g to allow the edge to be taken. Multiple edges starting from a single node have the semantics of alternative executions, except multiple edges that are marked with a common diamond at their origin. These represent concurrent execution. The constraints of concurrent edges must be valid simultaneously.

Edges with a given timing constraint where the target event is an input event are called **behavioral constraints**. These limit the rate at which inputs are applied to a system. In contrast, edges with a given timing constraint where the target event is an output event are called **performance constraints** that dictate the rate at which outputs are produced by a system.

Real-Time Monitor Example For a better understanding of the described modeling techniques, we will use in this section a simple real-time monitor example called *RTMonitor*. The task of *RTMonitor* is to observe whether events occur within a certain time interval after triggering the monitor application. Such a tool could be, for example, used to monitor whether a robot arm moves correctly by sampling its temporal position at two control points. The initial state of *RTMonitor* is denoted as ε while the start of *RTMonitor* is triggered by event *Start*. The incidents of two control points are reported by signals *Pos*1 respectively *Pos*2.

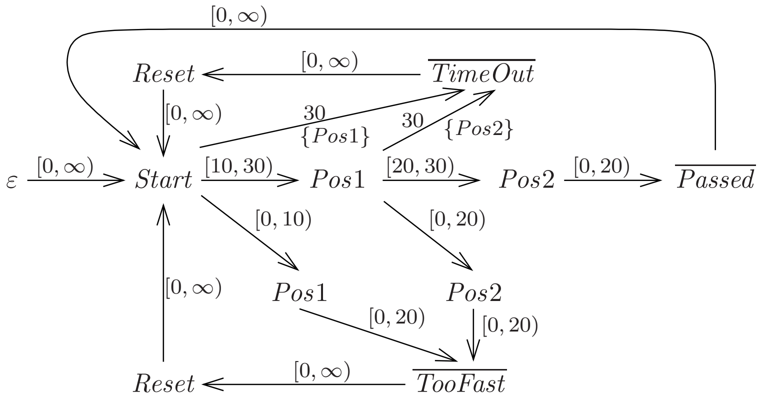

Fig. 13.2. Constraint graph of the real-time monitor example

The constraint graph of *RTMonitor* that shows the relative timing constraints is given in Figure 13.2. The start of monitoring can be triggered at an arbitrary time instant. Once it is triggered:

- The first event *Pos*1 has to be reported within time interval $[10, 30)$.
- The second event *Pos*2 has to be reported within time interval $[20, 30)$ after the first event.
- If one of these two events occurs too early, event $\overline{TooFast}$ has to be triggered.
- In case that one of these two events has not been observed after passing the time interval, event $\overline{TimeOut}$ has to be stimulated.
- After output event $\overline{TooFast}$, respectively $\overline{TimeOut}$, has been used to indicate an incorrect timing, the input event *Reset* can be used to restart the monitoring.
- If both events have occurred with a correct timing, event $\overline{Passed}$ will be triggered within time interval $[0, 20)$ after event *Pos*2.
- The monitoring application is restarted by waiting again for trigger event *Start*.

There is no timeout mechanism for the time constraint $[0, 20)$ of the event $\overline{Passed}$. $\overline{Passed}$ is an output event and therefore its time constraint $[0, 20)$ is the allowed delay introduced between processing the previous event and generating the output event $\overline{Passed}$. However, for testing the real-time behavior of the SUT, the validity of such a performance constraint also has to be tested on the SUT.

To exemplify the semantics of a constraint graph's edge with forbidden I/O events we describe the meaning of the edge "$Start \xrightarrow[\{Pos1\}]{30} \overline{TimeOut}$". It implies that the last received input event was *Start*. Then, this edge is taken if for the time interval of $[0, 30)$ the application receives no input event $Pos1$. In this case, the output event $\overline{TimeOut}$ will be emitted.

ACSR-based Test Case Generation ACSR is based on the Calculus of Communicating Systems (CCS) [Mil89], a process algebra to specify untimed, concurrent, and communicating systems. ACSR adds several operators to describe timed behavior and handle the communication and resource consumption of concurrent real-time processes. These operators support mechanisms for modeling bounded execution time, timeouts, delays, and exceptions.

Modeling: The ACSR computation model considers a real-time system as a collection of communicating processes competing for shared resources. Every computation step is either an **event** or a resource consuming **action**:

- EVENT (e_i, p_i): An event e_i having a priority level p_i is denoted as (e_i, p_i). It serves as a synchronization or communication mechanism between processes. The execution of events does not consume any time in contrast to the execution of actions. The example event (e_i, p_i) describes an input event in contrast to an output event that is drawn with a top bar above its name: $(\bar{e_i}, p_i)$.
- ACTION $\{(r_i, p_i)\}^t$: An action is a set of consumptions of resources r_i at corresponding priority level p_i $(1 \leq i \leq n)$ that needs t time units to execute. A resource consumption is denoted by a pair $(r_i, p_i)^t$.

A process P can be one of the following expressions:

NIL	– process that executes no action (deadlock).
$A^t : P_1$	– executes action A for t time units and proceeds with process P_1. The action $\emptyset^t$ represents idling for t time units.
$e.P_1$	– executes event e and proceeds with process P_1.
$P_1 + P_2$	– nondeterministic selection among the processes P_1 and P_2.
$P_1 \parallel P_2$	– concurrent execution of processes P_1 and P_2.
$P_1 \triangle_t^a (P_2, P_3, P_4)$	– temporal scope construct that binds the execution of event a by process P_1 with a time bound t. If P_1 terminates successfully within time t by executing the event a, the "success-handler" P_2 is executed. If P_1 fails to terminate within t, process P_3 is executed as a "time-out exception handler". Lastly, the execution of P_1 may be interrupted by the execution of a timed action or an instantaneous event of process P_4.
$[P_1]_I$	– process P_1 that only uses resources in set I.
$P_1 \backslash F$	– process P_1 where externally observable events with labels in F are disallowed while P_1 is executing.
$P_1[R_e, R_a]$	– relabels the externally observable events of P_1 according to the relabeling function R_e and the resources of P_1 according to the relabeling function R_a.

$recX.P_1$	– process P_1 that is recursive, i.e. it may have an infinite execution. Every free occurrence of X within P_1 represents a recursive call of the expression $recX.P_1$.
X	– recursive call of the surrounding recursive process $recX.P_1$.

RTMonitor Example: We will now model the RTMonitor example according to the constraint graph given in Figure 13.2. An ACSR model of *RTMonitor* is shown in Figure 13.3. It does not use any actions as resources are not considered in this example. The model excessively uses the concept of temporal scopes to model the allowed time intervals.

$$\begin{array}{ll}
RTMonitor & = recX.((Start, 1).P_{MA_1}) \\
P_{MA_1} & = \emptyset^{\infty} \triangle_{10}^{(Pos_1,1)}(P_{early}, P_{MA_2}, NIL) \\
P_{MA_2} & = \emptyset^{\infty} \triangle_{20}^{(Pos_1,1)}(P_{MB_1}, P_{miss}, NIL) \\
P_{MB_1} & = \emptyset^{\infty} \triangle_{20}^{(Pos_2,1)}(P_{early}, P_{MB_2}, NIL) \\
P_{MB_2} & = \emptyset^{\infty} \triangle_{10}^{(Pos_2,1)}(P_{ok}, P_{miss}, NIL) \\
P_{ok} & = (\overline{Passed}, 1).X \\
P_{miss} & = (\overline{TimeOut}, 1).(Reset, 1).X \\
P_{early} & = (\overline{TooFast}, 1).(Reset, 1).X
\end{array}$$

Fig. 13.3. ACSR model of the real-time monitor example

First, process *RTMonitor* is defined as a recursive process where the recursion is performed in subprocess P_{ok}. *RTMonitor* is waiting an unlimited time period for the occurrence of event $(Start, 1)$ and continues then with process P_{MA_1}. Process P_{MA_1} together with process P_{MA_2} checks whether event $(Pos_1, 1)$ occurs within time interval $[10, 30)$ and continues then with process P_{MB_1}. If the event comes too early respectively too late, the corresponding output events $(\overline{TooFast}, 1)$ or $(\overline{TimeOut}, 1)$ are generated by process P_{early} respectively P_{miss}. After receiving an input event $(Reset, 1)$ process *RTMonitor* is recursively called. Analogous to P_{MA_1} andP_{MA_2}, processes P_{MB_1} and P_{MB_2} check whether event $(Pos_2, 1)$ occurs within time interval $[20, 30)$. After that, control is taken over by process P_{ok} that emits the output event $(\overline{Passed}, 1)$ and then recursively calls *RTMonitor*.

The use of a process algebra like ACSR allows to abstract from the real application behavior by modeling only the dynamic aspects of interaction. In case of ACSR these aspects of interaction include events as well as resource consuming actions. Mechanisms like temporal scope and time consuming actions can express the temporal behavior of the application. ACSR does not support the modeling of numerical calculations or direct communication of parameters. Therefore, the use of ACSR is adequate in cases where the behavior of event communication and resource consuming actions of concurrent processes are the only interesting aspects.

Test Case Generation: As stated above, ACSR describes the interaction of concurrent processes. Testing such a system of concurrent processes is done by expressing a test as a separate process that we call T. The application of a test T to a process P is denoted as $T \triangleright P$ as done by Clarke and Lee [CL95, CL97a, CL97b]. The test operator $\triangleright$ is introduced for testing purposes and is not part of the ACSR specification itself. It implies an auxiliary sink process that absorbs unsynchronized output events between the tester process and the process under test. For testing the ACSR model of the SUT a test T written in ACSR can be directly applied to the model. But for model-based testing the test T has to be translated into another language so that it can be applied to the SUT, i.e. it must be executable.

A test T indicates by signaling whether a test was a success or failure. The notion of success or failure of a test is modeled by the special event labels $\overline{success}$ and $\overline{failure}$. Since the generality of ACSR's syntax obscures some common testing operations, the following notational conventions have been introduced by Clarke and Lee [CL97a]:

$\top$	– process that signals successful termination of a test: $\top \equiv (\overline{success}, 1).recX.(\emptyset : X)$.
$\bot$	– process that signals the failing of a test: $\bot \equiv (\overline{failure}, 1).recX.(\emptyset : X)$.
$\delta.T$	– *unbounded wait* for the occurrence of an action or event of test process T: $\delta.T \equiv (recX.(\emptyset : X))\triangle_{\infty}(NIL, NIL, T)$.
$T_1 \triangle_t T_2$	– simplified *timeout notation*: $T_1 \triangle_t T_2 \equiv T_1 \triangle_t (NIL, T_2, NIL)$.
$T_1; T_2$	– sequential composition of tests $T1$ and T_2: $T_1; T_2 \equiv (T_1[\{e_{sccs}/success\}, \emptyset] \mid\mid \delta.(e_{sccs}, 1).T_2)\backslash\{e_{sccs}\}$ where event e_{sccs} is not used in T_1 or T_2.
$(e,p)!T$	– applies event (e,p) as input to the SUT and proceeds with T: $(e,p)!T \equiv (\overline{e}, p).T$.
$(e,p)?T$	– the specific output event (e,p) from the SUT must be received; if it is not received, the *required response* is a failed test: $(e,p)?T \equiv (e,p).T + (\tau, 1).\bot$ where $p>1$.

The constraint graph given in Figure 13.2 can now be used to derive a test suite to verify whether the temporal behavior of the application conforms to the ACSR model given in Figure 13.3. The test suite is then transformed by a trivial rewriting step into a test language suitable for testing the SUT. We have to notice that this test case generation method does not deal with infinite application behavior.

Two kinds of constraints in the SUT are tested by this approach: **behavioral constraints** and **performance constraints**:

- PERFORMANCE CONSTRAINT This kind of constraint describes a delay interval that ends when a required output is produced.
- BEHAVIORAL CONSTRAINT This kind of constraint describes a delay interval that ends when a required input is applied.

A *performance constraint* is tested by a simple test that verifies that the correct response is received during the required interval. Figure 13.4 shows the three situations of an erroneous implementation of a sample constraint S. Implementation I_1 shows the case that the required output $\overline{E_1}$ is not produced within the interval $[t_1, t_2)$. In contrast, output $\overline{E_1}$ occurs to early in implementation I_2. I_3 demonstrates the situation in which an output is produced within the required interval $[t_1, t_2)$ but the event associated with the output is incorrect. Since the quantity being tested is the delay introduced by the SUT, there are no input parameters for the tester to vary. Therefore, it also does not make a difference whether the time domain boundaries are open or closed ones.

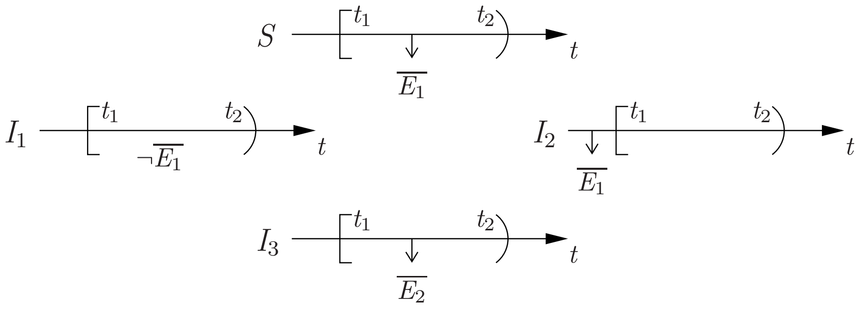

Fig. 13.4. Erroneous implementations of performance constraints

For *behavioral constraints* each of the two time domain boundaries is verified by up to two test points. As shown in Figure 13.5, the number and position of the test points for each domain boundary depends on its type and value. In contrast to the performance constraints, for behavioral constraints it makes a difference whether the time domain boundaries are open or closed ones, because test points have to be used close to these boundaries. One test point is always placed directly on the domain boundary. For closed domain boundaries the second test point is placed at a distance of ϵ outside the boundary. For open domain boundaries the second test point is placed at a distance ϵ inside the boundary. For the special case where the upper boundary is ∞, it is not possible to place a test point at the domain boundary. It is approximated by placing a test point at time T_{max} after the start of the interval, where T_{max} is the longest constraint interval in the system specification. For both open and closed interval boundaries, the up to two tests per boundary are sufficient to verify that the required change in system behavior has occurred within ϵ time units with respect to the required point in time.

A test suite can be derived by generating test cases so that both coverage criteria are fulfilled. The following three steps are used for test case generation:

(1) Deriving test process templates from the constraint graph. These templates will supply inputs at some time within the required interval, observe the outputs of the SUT to verify that they are generated within the correct time interval, and terminate with $\top$ if the test is successful or $\bot$ if the test fails.

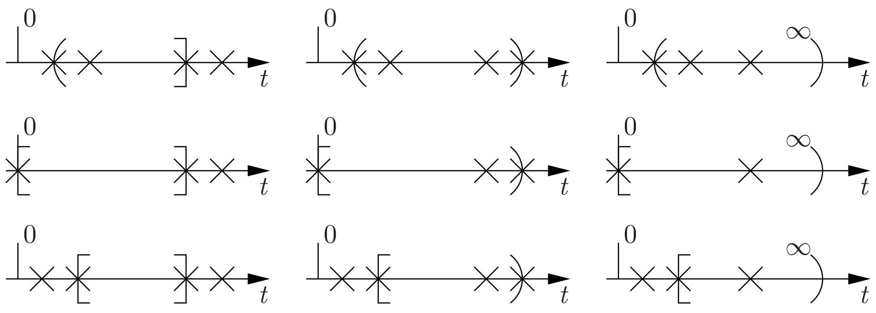

Fig. 13.5. Test points to test interval boundaries of behavioral constraints

(2) Derive input delay values that must be covered by the test in order to satisfy the intended coverage requirements.
(3) The output of the two previous steps is used to determine all test case candidates. As describing each delay requirement separately would lead to a high degree of redundancy within the test processes, a further optimization pass is necessary to reduce the number of test cases. Since this optimization problem is NP-hard, the usage of heuristics is necessary.

Since a process may also contain recursive elements (i.e. loops in the constraint graph), a full depth-first traversal of the constraint graph in step 1) and step 2) is not feasible. Therefore, the traversal must be bounded to a maximum depth.

Summary: The test framework described by Clarke and Lee has already been applied to real applications such as a relatively simple communication protocol [CL97b]. However, there is still room for further research in improving this method. A useful extension would be the development of coverage criteria that address interactions between different timing constraints (such as race conditions). Furthermore, coverage metrics that exploit the ACSR's focus on resource requirements and priorities in interactions would improve the generality of the test case generation framework.

An extension of the framework of Clarke and Lee to handle also infinite executions of processes is necessary for testing typical reactive systems that are nonterminating. The critical aspect of this extension is the design of the required test coverage criteria. For applications that can be expressed by a finite constraint graph without loops, it is sufficient to use coverage criteria that guarantee local coverage across the constraint graph. In case of infinite executions, it is required to define the coverage criteria such that the overall amount of test cases that are required for testing is limited. One possible method is giving a fixed upper bound for the length of test sequences. This strategy can be directly combined with the coverage criteria described by Clarke and Lee.

TTS-based Test Case Generation Test case generation with discrete time and value domain has also been done based on timed automata. Originally, timed automata work with a dense time domain [AD94]. In contrast, we present here results from Cardell-Oliver that use discrete time [CO00]. In this approach, it is argued that events cannot be observed at arbitrarily close times even if this can be specified in the dense time domain. Therefore digital clocks that model discrete time are used.

In the original algorithm, the basic assumption is that the implementation does not possess more states than its model. In this case, it can be proven that the generated test suite detects non-conformance between model and SUT and is furthermore complete. We do not believe that this is always guaranteed as the model is generally derived from the requirements of the SUT and not from the SUT itself. We refer to Chapter 8 for theoretical background. However, we believe that the presented ideas for generating a manageable test suite are useful as their focus is decreasing the amount of test cases.

Modeling: The basis for this approach are **Timed Transition Systems** (**TTS**) that mainly consist of three components:

- States Each TTS owns a finite set of states.
- Initial state One of these states is the initial state, where execution starts.
- Labeled transition relations Source and target states are connected by labeled transition relations. The label is discussed with respect to the used timed automata definition.

The timed transition system itself is described by a network of communicating timed automata as used in the tool UPPAAL [LPY97]. These automata further consist of:

- Variables Each automaton owns a finite set of data and clock variables. All variables are bounded. Clocks have to be reset if the specified bound is reached.
- Guards Guards are predicates that are conjunctions of constraints on clock and data variables. They are used for labeling transitions. If the guard evaluates to true, the transition is enabled and can be taken.
- Events Each automaton owns a finite set of events. Different automata communicate via these events over synchronization channels. For this purpose, each event is classified either as input or as output. If a is the name of a synchronization channel, $a!$ is the corresponding output event and $a?$ the corresponding input event. Like guards, events are used to label transitions. Two transitions are involved and must therefore be enabled: one that emits event $a!$ and one that receives event $a?$.
- Assignments With assignments both data and clock variables can be reset. They are also used to label transitions.
- Clock Invariants Each state of the automaton can own an invariant that specifies when the state has to be left due to a given timing constraint.

With respect to transitions, labels are composed of guards, events, and assignments in this order. The initial state is marked with an inner circle in the state symbol as shown in Figure 13.6. The network of communicating automata

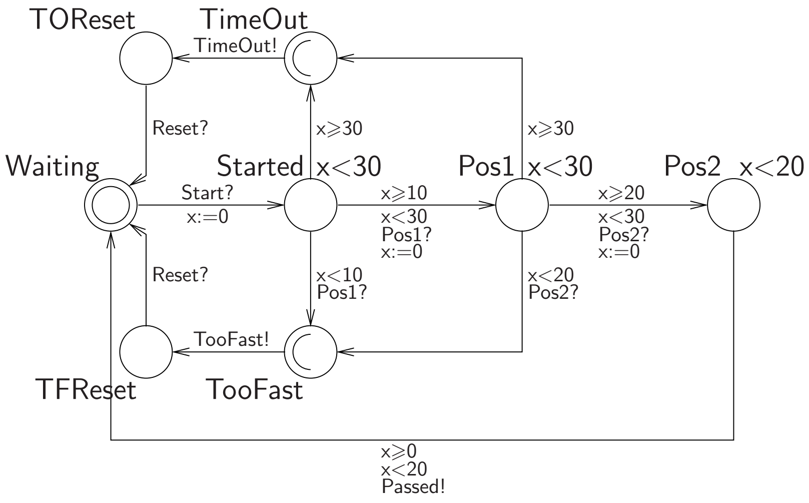

Fig. 13.6. UPPAAL timed automaton model of the real-time monitor example

is merged to a product automaton that is given as a TTS enriched with the clock invariants that are not included in the general TTS notation.

RTMonitor Example: Again, we will look on the RTMonitor example. It is realized by an UPPAAL timed automaton in Figure 13.6. In the initial state, we wait until event *Start* is received. After that, event *Pos*1 should occur in time interval $[10, 30)$. If this has been received correctly, we wait again if event *Pos*2 occurs in time interval $[20, 30)$. In this case, event *Passed* is generated in time interval $[0, 20)$.

If events *Pos*1 or *Pos*2 occur too early, event *TooFast* is generated and control is switched to state *TFReset*. Vice versa, event *TimeOut* is generated and control is switched to state *TOReset*, if these events do not occur in their specified time intervals. After *TooFast*, respectively *TimeOut*, has been generated, we wait for event *Reset* to restart the RTMonitor.

The states *Started* and *Pos*1 own both an invariant $x < 30$, so after 30 time units these states must be left. The same holds for state *Pos*2 where the invariant is $x < 20$. States *TooFast* and *TimeOut* are marked committed which is shown with the symbol C inside the state. A committed state must be left immediately after entering it, i.e. events *TimeOut* and *TooFast* are sent immediately.

In this example, we have seen that real-time systems can be modeled adequately with timed automata. By using states and transitions, the control flow of the modeled application can be easily captured. The timed automata variant of UPPAAL also allows modeling inputs and outputs explicitly. This further simplifies understanding of the automaton. However, large systems become intractable by using this modeling technique. Hierarchical structuring of models can help here.

Test Case Generation: The central idea of the approach of Cardell-Oliver is using **test views** for transforming a given TTS to a **Testable TTS** (**TTTS**). As described above, the model used for test case generation is given by a TTS that is derived from a network of UPPAAL automata. In combination with this model, different **test views** can be used that each describe a specific test purpose. The TTS is transformed to a smaller TTS by a view. This is the TTTS that is used for test case generation.

Each test view is designed to fulfill a given test purpose. This is described by different parameters:

- Interface With respect to the TTS, we have to identify events that are produced from the test driver and are therefore input events for the SUT and events that are produced by the SUT for the test driver and are therefore output events. That is, we identify the interface between the test driver and the SUT.
- Discrete Clock The digital clock used in the test automation system must be specified. The clock grain must be chosen according to the needed precision to distinguish between observed and stimulated events and the possibilities of the used hardware.
- Hiding The set of events can be divided into observable and hidden events. Therefore, only the events of interest with respect to the test purpose are observable in the TTTS. This can reduce the search space of the system as also less states are visible if traces to and from them are not observable any more.

With the help of the test view, the test designer can control the size of the test suite. The TTTS can be detailed if important test cases are generated and less detailed if the test purpose is not crucial. The size of the search space is determined by the discrete clock as its granularity influences the search space and by the hidden and observable events that lead to less observable transitions. In general, the search space is decreased as states cannot be distinguished anymore after eliding invisible transitions. Only under specific circumstances it is increased.

This happens if a state has n incoming transitions that are all hidden and m outcoming transitions that are all visible. Before hiding, the number of visible edges in the TTS is $n + m$, after hiding, the number of visible edges in the transformed TTTS is $n * m$. The test designer can react to this by using a different test view where these events are not hidden. A similar problem is a cycle of transitions where all events are hidden. This is not allowed for creating a test view as the SUT may cycle forever in this loop without a possibility of observation when test cases are executed, i.e. unbounded nondeterminism occurs. Here, at least one event must be made visible in the test view.

Another problem in this context is that the resulting TTTS may have redundant states. These could be distinguished in the TTS before the transformation by different distinguishing traces of inputs, outputs, and delays. After the transformation, these distinguishing traces can be equivalent due to hidden events. In this case the TTTS can be minimized before test case generation. This is not necessary but helpful as test case generation can be performed more efficiently if the size of the TTTS is further reduced.

The test case generation algorithm itself is based on the W method [Cho78] discussed in detail in Chapter 4. It works in the following way:

(1) For each state all acyclic traces that lead to that state are generated. It is possible that one or more of these traces are tester controlled, i.e. the inputs of the test driver to the SUT produce deterministic outputs of the SUT. If such a trace exists, this can be used as a test case in the following. If nondeterminism is possible, the correct test case is selected out of all generated ones during testing with respect to the output of the SUT.
(2) After that, we have to check that the reached state is really the state we expected. As the underlying TTS has persistent variables, it may be possible to identify states based on variable values. If this is not possible, distinguishing sequences as introduced by Chow [Cho78] can be used. Again, tester controlled test cases are preferred as this reduces the test suite. Else all possible test cases must be present to be chosen during test execution.
(3) Furthermore, not just every state but also every transition should be visited. Therefore one test case for every transition is generated.
(4) At least, the test suite is created based on the test cases produced in step 1 to 3. We check for redundant test cases as a short test trace may be included in a longer one. In this case, the short traces can be elided from the test suite. It is expected that the nondeterministic test cases are all executed at least once if testing the implementation is performed long enough. In practice, this may not happen but cannot be prevented. The test suite can be further reduced if it is possible to limit the possible set of input values to the SUT by making assumptions about the possible ones. Often, a system is expected to run in a specific environment so some input values can never occur.

Summary: The main idea of this test case generation algorithm is obviously the usage of test views. These specify the interface between the SUT and the test driver, the clock granularity, and the amount of observable events based on the test purpose. By using UPPAAL automata that differentiate between sending and receiving events, interfaces can be easily specified. The clock grain can be chosen with respect to the used timing constraints in the model and the used hardware. The art of creating test views is the subdivision of the event set into hidden and observable events. The amount of test cases generated by the algorithm depends mainly on these. Therefore the test designer has to consider carefully which events should be observed in a test view and which not.

As we cannot guarantee that the set of states is equivalent in the implementation and the model, the completeness results for the generated test suite is not relevant. However, test views are a means to reduce the set of test cases and are therefore useful. Moreover, the usage of persistent variables helps reducing the amount of test cases as states can be often distinguished based on variable values. If we do not consider time in test views, these can also be used to reduce test suites of untimed systems.

The obvious drawback is that test views must be chosen carefully. The created test suites for each view may overlap and hence increase the overall testing time unnecessarily. Even worse, parts of the system may never be tested as no test view covers them. Therefore, the generated test suites must be compared before using them for testing.

13.3.2 Real-Time Systems – Dense Time and Discrete Values

For modeling real-time systems, also dense time can be used as this is the natural way time is passing. This approach is used in the original **Timed Automata** approach by Alur and Dill [AD94] and also in the more restricted **Event Recording Automata** (**ERA**) [AFH94]. As the different timed automata variants do not differ significantly we reuse the RTMonitor example from Section 13.3.1 and focus on test case generation with respect to the differences in using discrete and dense time.

ERA-based Test Case Generation In Section 13.3.1 we already presented one technique for generating test cases based on timed automata models. However, this approach is working with a discrete time domain. It is also possible to use a dense time domain for test case generation. This approach is driven by the natural flow of time that is not discrete but dense [Nie00]. Furthermore, processor clocks are discrete but their granularity is so fine that it can be regarded as dense.

The test case generation algorithm is based on Hennessy's testing theory for untimed systems, i.e. it is based on preorder relations. This is described in more detail in Chapter 5, Chapter 6, and Chapter 8. We are interested here in the ways the test cases in a test suite are chosen out of the possible ones.

Modeling: The timed automaton model ERA [AFH94] chosen by Nielsen is very similar to the one presented in Section 13.3.1 but more restricted [Nie00]. Due to these restrictions, ERA can be determinized. Briefly, there are states and transitions labeled with guards, actions, and assignments to clock variables. Actions are partitioned into hidden and observable ones that are either input or output actions used for synchronization while transitions are either urgent or non-urgent.

The characteristic feature of ERA is that clocks and actions are coupled as each action has an associated clock. This clock is reset every time the action is performed, other resets are not allowed. Therefore, a clock measures the time between two occurrences of its associated event. Similar to UPPAAL automata, input and output actions are always synchronized. The environment has control over clock resets as it performs the complementary actions to the ones of the model of the SUT, so clock valuations are determined. The environment is the test driver as it stimulates and records inputs and outputs to and from the SUT.

For test case generation, ERA are further restricted by permitting only observable and urgent actions and forbidding clock invariants in states. Urgent and non-urgent actions were not distinguished by Alur et. al. for ERA but are introduced by Nielsen for testing purposes. With only urgent actions, transitions must be taken immediately if they are enabled and their action synchronization can be performed. Therefore, it is determined when a transition must be taken. ERA are enhanced by allowing integer variables that can be shared between automata in a network just as clock variables.

RTMonitor Example: On the first sight, the RTMonitor model created with ERA shown in Figure 13.7 does not differ very much from the one created with UPPAAL automata in Section 13.3.1. We have to keep in mind that all clocks are associated to an event and are automatically reset. Therefore, there are no clock assignments shown. Clock names are given with respect to their associated action. To give an example, for event *Start* the corresponding clock is named *StartC*. The initial state has a further inner circle.

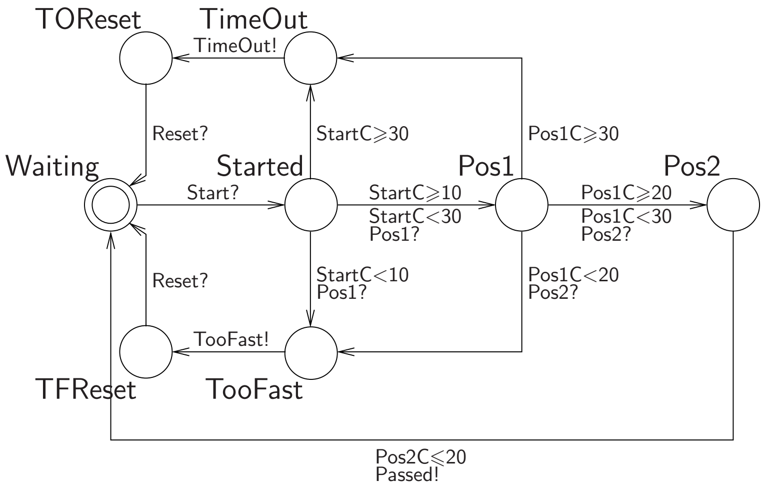

Fig. 13.7. ERA model of the real-time monitor example

The time intervals in which signals *Pos*1 and *Pos*2 should occur, respectively in which signals *TooFast*, *TimeOut*, and *Passed* must be sent, are obviously the same as in the example using discrete time in Section 13.3.1. Nevertheless, they are expressed differently as all clocks are related to events. The most important difference between the two examples using discrete respectively dense time is that in ERA events can be sent and received at any time and not only at fixed time points. To give an example, event *Start* can be received at time unit 1, 2, 3, ... in an UPPAAL automaton using discrete time. In contrast, *Start* can be received at time point 1.5, 1.578, or 2.3 in an ERA with dense time domain. Therefore this model is closer to reality.

Test Case Generation: In the test case generation algorithm presented by Nielsen [Nie00], two main principles are used: partitioning of the state space for decreasing it and using coverage criteria for selecting test cases out of the possible many ones. Determinized ERA automata serve as a basis.

The first step to be taken is partitioning the overall state space by grouping states into sets of equivalent states. These serve as a basis for testing. The motivation for this is that it is more interesting to test inequivalent states then testing equivalent states multiple times. It is also necessary as the underlying TTS of the ERA has infinitely many states due to the dense time domain.

The partitioning is done with respect to the **stable transition criterion**. Nielsen calls it stable edge criterion, but for homogeneity throughout this chapter we prefer the term transition instead of edge. Two sets of states are considered equivalent if they consist of the same states and enable the same set of transitions. A transition is enabled if its guard evaluates to true. A change in the enabled set of transitions may also induce a change in the enabled actions for synchronization. Therefore, different deadlock situations can be detected with respect to the different enabled transitions. Furthermore, guards may be dependent on clock valuations, so the set of enabled transitions changes with respect to time. This behavior requires corresponding test cases. Hence, using the set of enabled transitions is more convenient for testing than just considering transitions.

After partitioning the original ERA, respectively its underlying TTS, into sets of equivalent states, we can visualize the result as a **partition graph**. The sets of equivalent states are called **symbolic states**. Each partition should be tested by at least one test case in a test suite. Before generating it, the reachable parts of each partition are computed. A **symbolic reachability graph** for the partition graph is the result. This is created by starting at the initial state of the partition graph and traverse the graph with respect to symbolic states. Each trace is followed as long as new symbolic states are found.

Test cases are generated from the symbolic reachability graph in the following way:

(1) For each symbolic state, create a concrete trace leading to it with the initial state as a starting point. To do this, a **strengthened symbolic state** is created that consists of all states that will lead to the target state. This is necessary as not all states inside a partition will lead to the target partition. We do this by starting at the target transition and follow the trace back to the initial state so that all constraints in the transitions of the trace will evaluate to true. This proceeding is called **back propagation**.
(2) The strengthened traces created in step 1 are transformed to specific traces with concrete values for delays.

Delays can be chosen according to different strategies:

- PROMPTNESS The smallest possible delay is chosen. This is useful to stress the SUT with the shortest possible interval between inputs and outputs.
- PERSISTENCE A delay somewhere in the middle of the possible values is chosen. This is useful to check the persistence of the SUT.
- PATIENCE The largest possible delay is chosen. Here, the patience of the SUT is tested.

Nielsen claims that many bugs are found near extreme values of inputs and therefore choosing delays with respect to promptness and patience is preferable.

In principle, the test case generation algorithm depends on the reachability graph as the traces represented by this are concretized during test case generation. If the reachability graph or even the partition graph is too large further strategies have to be applied based on pragmatic reasons. These can be:

- Trace length limitation One possibility is limiting the trace length of a test case to a certain length. After that, processing of reachable states is aborted.
- Randomized state space exploration Another way of limiting the size of the partition or reachability graph is choosing the successor states of one state randomly out of all possible ones. This is done until a fixed number of states is reached or a specific time limit exceeds.
- Bit-State Hashing It is also possible to use a hash table with fixed length. Each state is stored there with respect to a key value that must be computed based on a given algorithm. Therefore, different states may be related to the same key. As only one bit is used for hashing, there can exist exactly one entry for each key value. If a state with an already used key is reached, it overwrites the hash entry. Exploration from the former state is stopped then.

A generated test suite can be reduced further by eliminating redundant test cases as one test case can be included in another, longer one. To perform this reduction, the test suite is transformed to a tree structure called **test tree**. This is also helpful for nondeterministic tests where new inputs must be chosen with respect to the nondeterministic output of the SUT. Obviously, this technique is also applicable to testing untimed systems.

Summary: The presented test case generation algorithm for models based on a dense time domain relies mainly on the partitioning of state sets into equivalent sets called partitions. Partitions are chosen with respect to state sets that enable the same transitions and consist of the same states. Therefore, the infinitely many states in the TTS underlying an ERA are grouped so that a symbolical finite partition graph is the result. This can be used for test case generation.

The generated test suite can still be very large as it depends on the size of the partition graph. If this is very large, heuristics must be used. Three possibilities are suggested, namely trace length limitation, randomized state space exploration, and bit-state hashing.

The delays used in timed traces also depend on heuristics. Stressing the SUT with the shortest possible delay values as well as testing its patience by choosing the largest delays are claimed to be most important for testing as extreme values are considered best for finding errors.

The chosen modeling language ERA restricts the original timed automata by using only event clocks and urgent transitions. However, this seems not to be a drawback as the resulting model is still expressive. The main problem with this approach is that the partition graph may be still too large to generate a test suite with practicable size without using further heuristics.

13.3.3 Hybrid Systems – Dense Time and Dense Values

The last step is taking also dense values into account as done in **hybrid** systems. These have been studied in many ways during the last ten years. There are different attempts for modeling them and applying concepts known from the formal methods community to them, like model checking or theorem proving, e.g. by Henzinger [Hen96], Alur et al. [ACH$^+$95], Zhou et al. [CJR96], Kapur et al. [KHMP94], Ábrahám-Mumm et al. [ÁMHS01], Lynch et al. [LSV01], or Larsen et al. [LSW97]. Hence, models of hybrid systems are well understood today. In contrast, there are only few attempts for using hybrid models in model-based testing.

We focus on Hybrid Automata [Hen96] and their extension to CHARON [ADE$^+$01, AGLS01, ADE$^+$03] and HybridUML [BBHP03] here. There exists also approaches for hybrid process algebras, e.g. Hybrid CSP as a further extension of Timed CSP [CJR96, Amt00]. Here, research with respect to testing deals with test case specifications and their execution as e.g. in Peleska et al. [PAD$^+$98]. Test case generation based on a model of the SUT is not covered in detail.

Thermostat Example To give an idea of hybrid systems, we will use the thermostat example taken from Alur et al. [ACH$^+$95] throughout this section. The thermostat continuously measures the room temperature x. It turns a heater on and off due to the current temperature. The initial temperature is named θ, K and h are constants describing the power of the heater and the room. The following requirements hold for the SUT and must be considered in the SUT model:

- If the heater is off, x is decreasing according to $x(t) = \theta * e^{-Kt}$.
- If the heater is on, x is increasing in the following way: $x(t) = \theta * e^{-Kt} + h(1 - e^{-Kt})$.
- The heater is turned *on* if the temperature falls below m.
- The heater is turned *off* if the temperature rises above M.

Hybrid Automata-Based Test Case Generation With considering dense values in addition to dense time, the state space further explodes. Nevertheless, also hybrid automata can be used for test case generation if appropriate abstractions are found. We can build up on the techniques for discrete- and dense-timed automata presented in Section 13.3.1 and Section 13.3.2.

Modeling: For modeling hybrid systems, **Hybrid Automata** as developed by Henzinger [Hen96] can be used. In general, the time-discrete part of the system is described by transitions while the time-continuous part is modeled inside states. Automata are enriched with flow conditions for describing the evolution of dense-valued variables over time. Clocks are modeled in the same way with a constant rate of change of *1*.

In general, a hybrid automaton H consists of five components:

- Variables A finite set of real-valued variables.
- Control graph The system is described by a finite directed graph with vertices called control nodes and edges called control switches. As long as control resides inside one node, time is passing and the values of the dense variables evolve according to time. This is a continuous change called **flow**. When an edge is taken, control switches to another mode, i. e. a discrete change is performed called **jump condition**. Discrete changes do not consume time.
- Initial, invariant, and flow conditions Three different predicates can be attached to control nodes. First, the **initial condition** specifies the initial values of variables inside a node. Second, an **invariant** can be assigned that describes under which conditions the node has control. If the invariant is violated, control must be switched to another node. At last, a node may have a **flow condition** that describes the evaluation of analog variables over time.
- Jump conditions A control switch can be labeled by a predicate called **jump condition**. The edge is enabled if the condition evaluates to true. Only then, the control switch can be taken.
- Events Furthermore, an edge can also have an assigned **event**. If the jump condition of an edge holds and the associated event occurs, the edge is taken and control is switched to the target node. Different hybrid automata $H1$ and $H2$ can interact via events, i. e. they synchronize over event a if a is both an event of $H1$ and of $H2$.

CHARON is a further development of hybrid automata as described by Alur et al. in [ADE$^+$01], [AGLS01], and [ADE$^+$03] with two main improvements. First, not only behavior but also structure of a system can be modeled. This is done in an agent whose behavior is described in a mode. Second, both structure and behavior may be built hierarchically. This allows better structuring of models as large systems become unmanageable with flat structures. A further enhancement is **HybridUML**, a profile of the **Unified Modeling Language** 2.0 (**UML**) with formal semantics [BBHP03]. In addition to the possibilities of CHARON, HybridUML gives better support for datatypes and allows communication not only via shared variables but also via signals. The specification of datatypes, structure, and behavior is handled in different UML diagrams, so no confusion between the different aspects of modeling occurs.

Thermostat Example: We assume that the thermostat consists of a controller and a heater. In Figure 13.8 we can see a hybrid automaton describing the behavior of the controller. This is switching the heater on and off via events named *on* and *off*. The evaluation of the temperature is described by flow conditions. If the temperature falls below m or rises above M, jump conditions will trigger the switch from state *On* to state *Off*. θ is set to 20 here, while $K = 0.1$, $h = 5$, $m = 20$ and $M = 22$. The heater can be modeled as a separate hybrid automaton which has two states *On* and *Off* that have control if the heater is on, respectively off. The switch from one state to another one is triggered by events *on* and *off* that are sent by the controller's hybrid automaton, i. e. the two automata synchronize over these events.

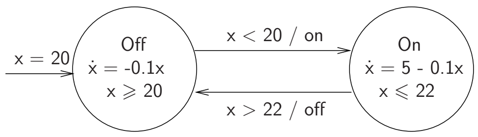

Fig. 13.8. Hybrid automaton for the thermostat controller

For comparison, we look at the same controller as a HybridUML model. The structure of the thermostat is modeled as an agent in Figure 13.9. The thermostat consists of a heater and a controller that communicate via signals *on* and *off*. Temperature x is measured by the controller and is also visible in the thermostat itself, e.g. to monitor the temperature from the environment. Therefore x is a shared variable. Sending of a signal, respectively write access to a shared variable, is shown as a black-filled box, while receiving a signal, respectively read access to a shared variable, is shown as a white-filled box. These boxes are connected to visualize communication structures.

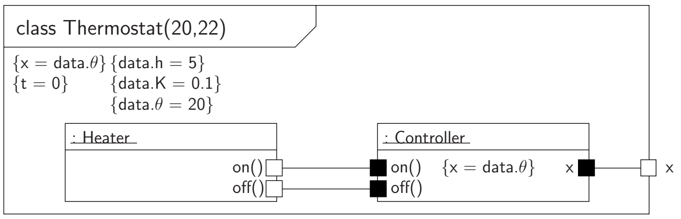

Fig. 13.9. Composite structure diagram for the thermostat agent

In the upper left corner, parameters m and M are set to 20, respectively 22. Both values are given as parameters for better reusability of the thermostat model. Furthermore, variables and constants of the thermostat must be initialized. Constants K, h, and θ are all included in structure *data* and are set to 5, 0.1, and 20. Shared variable x that measures the temperature is set to the initial value θ in both the thermostat and the controller. At last, t is a global clock that must be set to 0 in the beginning.

The behavior of the controller is visualized in Figure 13.10, similar to the hybrid automaton modeled above, i.e. states and transitions coincide in both variants of the thermostat example. Here, flows and invariants are marked explicitly with keywords *flow*, respectively *inv*. Jump conditions of transitions are given in brackets while events are given after a slash.

As we have seen, hybrid automata offer the possibility to model time-discrete and time-continuous behavior. As they are based on well-known automata, they

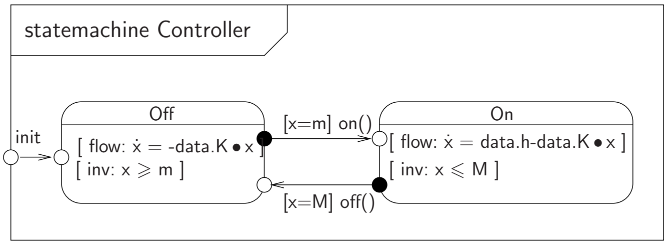

Fig. 13.10. Statechart diagram for the thermostat controller mode

are easy to understand. Hybrid Automata themselves have been introduced for theoretical purposes. In contrast, CHARON and HybridUML have been designed for practical purposes and are able to model both structure and behavior of a system.

Test Case Generation: New problems appear for test case generation based on hybrid systems. In addition to the problems that arise when dense time is considered as described in Section 13.3.2, new problems occur due to the dense value domain. On the one hand, the SUT expects dense values in form of curves as input, e.g. the velocity of a car. The SUT must receive these during test execution from the test driver. Such curves must be selected from the infinitely many possible ones. This is a problem that has not been tackled until now. On the other hand, dense values are outputs from the SUT that must be evaluated. We have to consider a certain fuzziness with respect to time and values as the test itself is performed with a discrete-working computer. We can image this as a tolerance tube like in Hahn et al.[HPPS03b, HPPS03a] where some tolerance is added for both expected time and values. The output from the SUT must lie inside this tube as depicted in Figure 13.11.

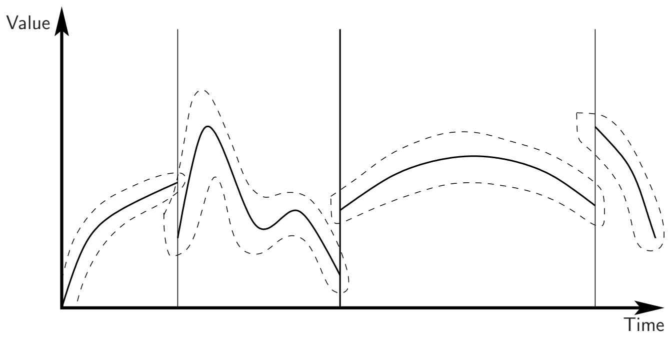

Fig. 13.11. Tolerance tubes for time and values [HPPS03a]

Until now, there is one attempt for test case generation based on hybrid systems. In this approach, two models instead of one are used as done by Hahn et al. [HPPS03b, HPPS03a]. The first model describes the hybrid system, i.e.

the discrete and the continuous part of the SUT. The second model is a purely discrete model that describes abstract control flow in the SUT. This reduces the problem of test case generation for hybrid systems to the one for real-time systems with dense time. This is possible as the hybrid part of the model is always hidden inside states while the discrete part is modeled by transitions. These transitions, respectively their triggers, are needed for generating test cases.

At this point, we have to ask ourselves, why we have not built a discrete model beforehand if this is used for the generation process. We must reconsider that the hybrid model is the most exact model for mirroring the required behavior of the SUT. The pure discrete model is too imprecise for testing the hybrid system. We therefore use the test suite created by the discrete model and feed both the SUT and the hybrid model the generated inputs.

We first consider an **open loop** system, i.e. we do not model the environment of the SUT and feedback from the environment to the SUT as in the closed loop system described in the next paragraph. Here, we have to compare the calculated output given by the hybrid model with the output from the SUT. The hybrid model is used to evaluate if a test has passed or failed as shown in Figure 13.12.

The situation is slightly different for **closed loop** systems. Here, the environment of the SUT is explicitly modeled due to the fact that most systems are required to work correctly in one specific environment and not in all possible ones. Therefore, values of inputs to the SUT can be restricted to possible inputs in the specified environment. As inputs often depend on the output of the SUT, a feedback construction is needed, so outputs of the SUT can be considered to calculate the next input, again with respect to the environment. The advantage of this approach is that complexity is reduced as the possible search space is restricted.

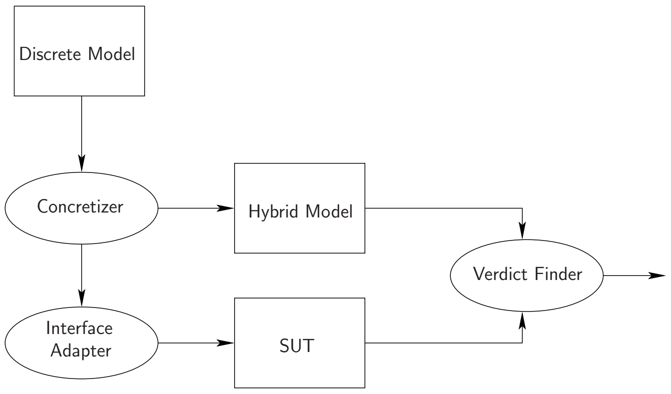

Fig. 13.12. Open loop test [HPPS03a]

Problems occur here with respect to the required feedback construction to the discrete model. The outputs we expect during testing are calculated in the hybrid model. Hence, this is used to give feedback to the discrete model as shown in Figure 13.13. New inputs to the SUT must be given with respect to this feedback, the modeled environment, and the modeled control flow of the SUT. We associate control flow with the hybrid model that defines it according to states and transitions in this model. If the hybrid model has reached a new state, i.e. control has switched from one state to another one, the discrete model must also perform a control switch to a new state, so inputs to the SUT are calculated with respect to the correct state of the system and the corresponding behavior of the environment that may be different in different states. Therefore, the discrete model, the hybrid model, and the environment model have to be synchronized to guarantee correctly generated test cases. This is done by using the output of the hybrid model to differentiate between its states. The output value space is split up into partitions that are related to states.

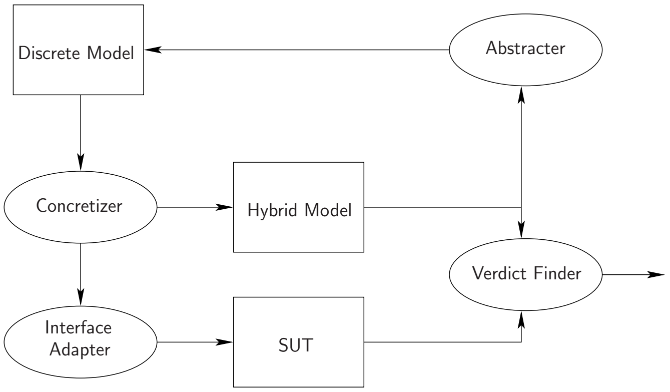

Fig. 13.13. Closed loop test [HPPS03a]

Summary: To summarize, by using one hybrid and one abstracted discrete model we can combine the advantages of the hybrid and discrete approaches. On the one hand, we can calculate results precisely, on the other hand, we are able to generate test traces efficiently. In case of open loop test systems, we just compare the results from the SUT and the hybrid model by the test oracle. In case of closed loop test systems, we have a feedback construction that requires more effort from test automation and test execution. The advantage of this proceeding is that the search space is further reduced. The problem is that the feedback construction is difficult to built as discrete model, hybrid model, and environment model must be synchronized. Partitioning the output space of the hybrid model to fulfill this task may be infeasible or not possible without ambiguities.

Obviously, the multiple model approach increases the modeling effort as we have to built two models instead of one. The key to the usability of this approach is designing a good discrete model. For one, it must mirror the hybrid model to guarantee working test traces, but in the same time, it has to be very abstract to take advantage of the discrete nature, i.e. generating a manageable set of test traces. If the discrete model is too detailed we would again get too much test cases. Another problem not tackled here is generating time-continuous input data as we have infinitely many possibilities for this.

13.4 Optimizing Test Suites by Evolutionary Testing

As we have seen in Section 13.3, there exist different techniques for model-based test case generation for real-time and hybrid systems. They have in common that they adapt algorithms known from testing theory to achieve a manageable test suite that consists of a finite set of test cases. Nevertheless, the number of test cases can still be very large. In that case, further reduction is required. Moreover, we do not know if the test cases derived are "good" test cases, i.e. they are able to find errors in the SUT or increase our confidence in the correct behavior of the SUT. We will present a possible way for improving this situation.

With respect to real-time systems, time is an important factor to guarantee correct behavior. Hence, we are also interested in the best-case execution times and especially in the worst-case execution times to see if deadlines are met. To test the timing constraints imposed by the SUT, the algorithms presented in Section 13.3 do not help since the values for delays generated by them are chosen according to a certain test strategy, e.g. promptness. They do not depend on the real behavior of the SUT.

Evolutionary algorithms can be adapted to optimize the size of the test suite or to test timing constraints imposed by the SUT.

13.4.1 Iterative Refinement Using Evolutionary Testing

Evolutionary testing is a testing technique where test data can be generated automatically by using search techniques. It is called **iterative refinement** because the test data to be optimized is iteratively refined due to a specified quality criterion.

Evolutionary testing has not been explicitly developed as a model-based method for test case generation, but it can be used in combination with this. As shown in Figure 13.14, the test model used in evolutionary testing is quite simple. It basically consists of a start state l_0 and an end state l_1 where the state l_0 can have an initial property p_0 assigned to it. Beside the initial property p_0, the only application specific part of the model is the property p_1 of the end state l_1. p_1 encodes certain aspects of the application that one is interested to be verified by testing. These aspects are typically maximum allowed boundaries such that the system is considered correct as long as it stays within these boundaries.

Reactive systems potentially run endlessly and therefore it is not possible to designate an end state of the system. To apply evolutionary testing, one idea is to identify an interesting intermediate state of the overall system model and use it as end state for the testing model. This allows testing the value of property p_1 when reaching state l_1.

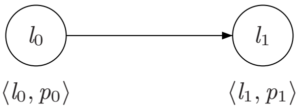

Fig. 13.14. Formal model for evolutionary testing of real-time systems

Though the application model used by evolutionary testing is quite simple, the challenge is to define an effective process of iterative test case generation together with a useful encoding of relevant system aspects by property p_1.

Evolutionary testing as described above assumes a model with a given boundary for the property of the end state p_1. The test goal is to increase the confidence whether this property is an invariant of the system respectively to show by counter-example whether it is invalid. If one is not interested to test a specific boundary of the property p_1, it is also possible to use evolutionary testing to get an idea of the feasible boundary of p_1.

Test Case Generation: As described above, the generation of test cases by evolutionary testing is a process that performs an iterative improvement of the test data. To achieve this, the test results of the previous test run are abstracted by using a fitness calculation. The calculated fitness values are used on the one side to decide whether the iterative test case generation can be stopped and on the other side to guide the calculation of new test data for the next test round. This is a typical optimization problem that can be solved, for example, by **evolutionary algorithms**. A characteristic of evolutionary algorithms is that there exists a whole population of solutions instead of only one current solution.

The choice of a certain search technique is often a question of the compromise between efficiency and robustness to generic problems. The difficulties of searching test data with maximum fitness are demonstrated in Figure 13.15. Figure 13.15(a) shows a relatively simple fitness distribution where even local search methods like *hill climbing* will find the solution easily. A more complex example is given in Figure 13.15(b) where the fitness distribution has more than one local maximum. Evolutionary algorithms have mechanisms to avoid getting stuck on a local maximum.

In the following, evolutionary algorithms are described as a technique for the iterative test case generation.

The iterative process of test-data generation based on evolutionary algorithms is shown in Figure 13.16. The algorithm starts with an initial set of test cases which is called **population**. Each member of the population is called **individual**. Each individual must be a valid parameter of the SUT. During **fitness evaluation** each individual is weighted according to a specified optimization criterion. The test can be stopped if the **exit test** has determined that the best fitness value has been reached or that the number of iterations has exceeded a certain value. In case of termination, the individual with the best fitness value is reported as final result. Otherwise, individuals from the current population are selected to create new individuals out of them. The new individuals will be

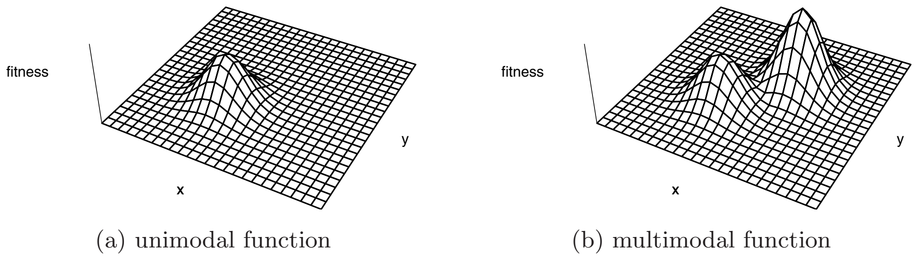

Fig. 13.15. Examples of fitness functions with different difficulty

the population for a new cycle of the evolutionary algorithm. This proceeding is explained in more detail by Goldberg [Gol89].

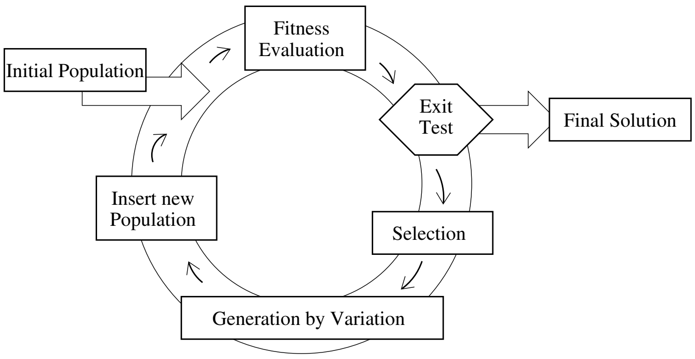

Fig. 13.16. Operational cycle of evolutionary algorithms

With respect to testing, the test case generator must be able to derive test cases. Furthermore, it has to be able to evaluate the fitness of each individual and to perform exit tests. This may require to collect and merge values of several observations into one fitness value.

Testing the Worst-Case Execution Time: To demonstrate the application of evolutionary testing, we describe its application to measurement-based timing analysis of real-time systems. Testing real-time systems means ensuring the correctness in the value and time domain. In contrast to other methods that test both properties in combination, the aim of the test method described in this section [PN98, Weg01, AHP99, GW98] is testing only the time domain. The value domain can be tested separately by other test methods. The property p_1 of the end state of a real-time system we are interested in is the tuple $\langle \text{BCET}, \text{WCET} \rangle$ where the **BCET** means the **best-case execution time** and **WCET** the **worst-case execution time**. In the following we will just describe test case generation for WCET as BCET can be handled similar. The application of evolutionary testing to measure the WCET conforms to the operational cycle of evolutionary algorithms given in Figure 13.16.

Testing the timing behavior of tasks individually requires that each task is free of synchronization points, i.e. it is a simple task [Kop97]. In the following we call the SUT just real-time program without making assumptions about its granularity compared to the whole real-time system.

Evolutionary testing for generating test cases to measure the WCET depends on **fitness evaluation** and the encoding of the real-time program's input variables. The fitness evaluation is realized as a black-box test that measures the execution time for concrete input test data. The execution time measurement is performed for each generated test case while the fitness value is calculated by comparing the relative execution time of each test result. The technical realization of the execution time measurement can be arbitrary.

As evolutionary testing is an iterative process, we also need a **stoppage criterion**. The simple case is that the fitness evaluation provides an execution time that is higher than the specified WCET bound. In this case the test has found a counter-example to the model and the test immediately stops. But as long as the execution times of all individuals of the population are smaller than the WCET bound, it is hard to decide whether the test can be stopped. Using an upper bound of the number of cycles in the iterative refinement does not provide a trustworthy result.

There are several approaches that demonstrate the applicability of evolutionary testing to analyze the WCET of so-called **transformative systems**, e.g., [PN98, Weg01, WBS02]. Transformative systems are typically subsystems that take input data and transform them into output data. In contrast to *reactive systems* that potentially run endlessly, transformative systems have to be triggered separately for each transformation. As a consequence, test data of transformative systems consist of a single test vector while test data of reactive systems consist of a sequence of test vectors.

The application of evolutionary testing based on evolutionary algorithms to reactive systems is not obvious as concrete techniques like *genetic algorithms* operate with individuals having a fixed length. A possible approach would be to test only test sequences of a fixed length, a method that is also used to limit the search space of test cases. However, in case of testing *performance constraints* (described in Section 13.3.1), the presented technique of WCET testing can be applied. This allows to reason at least whether a reactive system can perform its transitions within a certain time period. The verification of *behavioral constraints* (Section 13.3.1) of reactive systems would need another testing technique.

Summary: Evolutionary testing allows the iterative refinement of input data for testing. There exist several works on how to apply evolutionary testing for so-called *transformative systems*. The application of evolutionary testing to reactive systems has not been done and is not obvious. However, we have sketched in this section how evolutionary testing can be applied to test performance constraints of reactive systems. This can be done by applying the methods of WCET testing.

13.5 Summary

In this chapter, we have discussed test case generation for real-time and hybrid systems. As we have seen in the beginning, test automation for real-time and hybrid systems differs from that for untimed systems. The time domain has to be taken into account as well as the dense value domain for hybrid systems, e.g. for test evaluation. The model of the SUT must abstract from the real behavior of the SUT to allow sensible test case generation. In contrast, the test driver concretizes the generated inputs to feed the SUT. Vice versa, the concrete outputs of the SUT are abstracted again so that test evaluation can be done with respect to the abstract model.

Process Algebra vs. Automata Different modeling techniques can be used for this purpose. We have discussed ACSR as a process algebra and different variants of automata that already have been used for test case generation. A process algebra is capable of modeling a system consisting of processes that communicate with each other via events. Concurrency can be modeled explicitly as there are operators for interleaving and parallel execution with synchronization. Time is considered discrete. One problem of process algebras is that this modeling technique has no support in industry where graphical modeling is preferred. Nevertheless, testing based on process algebra has been proven useful and practicable.

In contrast, automata based modeling has more support as this is very popular, e.g. with respect to UML where state machines are used as automata variants. Timed automata introduce either discrete or dense time to be capable of modeling timing constraints. Graphical models are in principle easy to understand as control flow can be captured at one sight. However, large models become intractable in graphical representation. Hierarchy or different abstraction levels must be used to better this situation.

Discrete vs. Dense Time With respect to time, this is either modeled discrete or dense. In the first case, a timer is a counter that is incremented continuously. As the computer itself is working discrete, this seems appropriate. In contrast, time is naturally dense so modeling based on dense time is closer to reality, e.g. when analog sensors and actuators are used. For real-time systems, it must be chosen if the model represents natural time or computer time beforehand. For hybrid systems, time must be modeled dense as the continuous parts of the system rely on this.

Discrete vs. Dense Values Another aspect is the value domain that has to be tested. This is considered discrete for real-time systems. We have seen that test case generation builds up on methods for untimed systems and enhances the generated test cases with timing information. Discrete values for inputs and outputs to respectively from the SUT can be easily generated and evaluated. Time points and intervals for generating inputs and outputs are chosen with respect to the selected test case generation method. For hybrid systems, this situation differs as we have dense-valued variables. Until now, nobody has tackled the problem of generating curves as input data if time-continuous input is needed. With respect to output data, fuzziness is considered for test evaluation.

Test Case Generation Algorithms We presented two different test case generation algorithms for real-time systems that are based on discrete time and one algorithm that is based on dense time. Also one algorithm for creating test suites for hybrid systems was discussed. The most important function of these algorithms is the way in which the size of the test suite is reduced to a manageable size, i.e the way in which test cases are selected.

The first test case generation algorithm discussed is based on the process algebra ACSR where test cases are derived from the constraint graph of the ACSR model. Test data is selected to cover two type of timing constraints, namely performance constraints and behavioral constraints. To handle also applications with recursive elements, upper bounds on the length of tested execution scenarios have to be introduced.

Furthermore, test case generation for systems with discrete time can be done with timed automata models. Here, the central idea is using test views that restrict the model of the SUT to a specified test purpose. Test views are composed by parameters like clock granularity and the division of actions into observable and hidden actions. Therefore the task of the test engineer is creating test views that lead to manageable test suites. This must be done carefully as parts of the system may never be tested while other test views overlap. One important advantage of this approach is that the size of test suites is scalable.

With respect to dense time, the main idea is finding equivalent parts in the model called partitions. Due to the dense time domain, the model has infinitely many states with respect to the infinitely many possible time values. Partitions group these values so that a finite graph is the result that can be used for test case generation. As this may be too large for full exploration, heuristics must be used to limit the size of the state space.

For hybrid systems, the presented algorithm is based on the results of test case generation for real-time systems. In addition to the hybrid model, a second, discrete, model is created that abstracts from the hybrid system. This can be used in combination with the test case generation algorithms for real-time systems presented. The original hybrid model is needed to derive correct evaluations for variables as the discrete model being only an abstraction is not capable of doing this. The selection of dense input curves is not tackled.

Optimization of Test Suites Testing timing constraints is still a difficult topic as delays for stimulating the SUT must be chosen and the correctness of outputs of the SUT must be accessed. First attempts with using evolutionary theory based test case generation methods have shown that these can improve the test suite with respect to determining best- and worst-case-execution times. This can be helpful to prove if a SUT meets its timing requirements.

All other algorithms presented built up on results of testing untimed systems. In contrast, evolutionary testing has a different background and therefore provides a new point of view for future work in the field of real-time and hybrid systems testing.

Future Work Model-based test case generation for real-time and hybrid systems has been successfully applied in relatively small examples. More effort has to be put into this to prove the practicability of test case generation algorithms. There is also only few tool support for test case generation and execution for real-time systems and no tool support for hybrid systems. New techniques like evolutionary testing to test timing constraints must be further surveyed. Also cross-fertilization between the different approaches seems useful to improve the presented algorithms as each has its advantages and disadvantages.