Specification and Test of Real-Time Systems Bibliographical Notes

Brian Nielsen

In the following we review some of the work that have appeared on methods and tools for automated testing, and relate this to our work. The discussion is structured into an untimed part appearing in Section 1, and a timed part appearing in Section 2. We outline the novelties of our approach in Section 3.

We shall introduce an alternative method of testing based on checking sequences for finite state machines. This has recently been applied also to real-time systems. Other topics include the choice of implementation relation, the approach to test generation, be it online or offline, and the strategy used to select tests.

1 Untimed Testing

This section discusses related work that does not deal explicitly with real-time, but which is nevertheless important. We first take a deeper look at both the theoretical and practical issues that arise from testing of concurrent systems. Then two extreme approaches to automatic testing, offline and online test generation, are identified. The description of test generation using checking sequences follows. Finally, we discuss test selection. Our test selection method is rooted in well known sequential testing techniques.

1.1 Test Observations

In the presented untimed testing theory we have assumed that the outcome of executing a test could be found by reading the verdict label of the state in which the execution of the tester and implementation *deadlocks*. However, this assumption is not without practical complications, as we shall discuss in the following. The comments also apply to our real-time preorder.

The first problem is how to conclude that the execution of the tester and implementation has entered a deadlocked configuration, i.e., that no further synchronizations are ever possible. It is of course impossible to wait an infinite amount of time to determine whether a deadlock has occurred, or whether the implementation is just too slow to respond. In practise, extremely long response

times are unacceptable, and consequently expiration of a carefully set timer can be used to declare deadlock. Another consequence of our assumption is that deadlocks and internal divergence (live lock) in the implementation are indistinguishable. However, such information about divergence could be of great value when diagnosing why a test failed.

The second problem is related to the certainty with which observations can be made in face of non-determinism. The essential problem is that satisfaction of a must property is quantified over all computations of the implementation, whereas a single test execution only reveals one.

When the execution of a must test deadlocks in a fail state, it can safely be concluded that the system is erroneous. However, from the observation of a successful computation, it cannot be concluded that the implementation always passes that test. Similarly, when a may test is successful, the implementation certainly had the desired trace, but when it is inconclusive, neither presence nor absence of the trace can be concluded.

Even if the same test were executed multiple times, it is unlikely that the implementation has followed all possible paths or interleavings. Hence, there is no means by which a black box tester can guarantee satisfaction of a must test. It is frequently assumed in practice that a finite number of re-executions of the same test will pass through all computations of the implementation (called the *complete-testing* assumption in [49]). An alternative to the complete testing assumption is to equip the tester with capabilities for monitoring which internal computations have been taken, or possibly even controlling which are taken, c.f., Brinch-Hansen [20] and Taylor et al. [50].

Deadlocks are not the only possible observations; indeed an abundance have been proposed. For an overview and classification we refer to Glabbeek [52]. Different assumptions induce different implementation relations that differ in how discriminating they are. For example, we could assume that the tester, in addition to observing deadlocks, also could recover from the deadlock and *continue* testing by enabling an alternative set of actions. Adopting observation of such communication failures leads to an implementation relation called the *failure trace* preorder. It turns out that such observations become central when time is taken into consideration, see Section 2.1.

Another fascinating example, albeit quite exotic, is observations corresponding to making copies[1] of the implementation in its current state. Each copy can then be subjected to different experiments. This technique enables testing of the very discriminating the 2/3-bisimulation (ready-simulation) [30] preorder.

Nearly all of these theoretically based preorders are based on symmetric and synchronous communication between tester and implementation. However, in practice communicating systems often distinguish between inputs and outputs: *Inputs* signify data given to the system, and *outputs* data being produced by

[1] Provided that we have yet to master the sophisticated technology required to realize Star Trek Replicators [28] we may need to settle with snapshots or core dump approximations.

the system. Inputs cannot be refused, but may of course be ignored, i.e, system entities are assumed to be input enabled. Tretmans proposes in [51] a preorder **ioconf** that relates processes when the outputs of the implementation after a trace are included in the outputs of the specification after the same trace. Tretmans also defines a stronger version **ioco** that additionally requires that the implementation only refuses to deliver outputs when the specification also refuses to deliver outputs.

1.2 Approaches to Test Generation

We distinguish between two extreme types of test generators: Online and offline generators. An *offline* generator does all work offline before test execution begins. It interprets the specification, constructs the success graph, traverses it to construct test cases, and performs test selection. The entire test suite is thus constructed a priori, and is typically stored in a set of files from where it can be recalled when a new product is to be tested.

An *online* generator constructs a test case as it is being executed. During test execution the test driver simulates the specification. The test driver constructs a set of actions that the implementation is expected to be able to synchronize on in its current state. This set of actions is typically chosen randomly from the success sets from the specification's current state. If synchronization were successful, the synchronization action is fed to the simulator that computes the states that the specification can reach after performing the action. A new success set is then computed and the procedure is continued. If the synchronization attempt was unsuccessful the test driver stops execution and reports the appropriate verdict. The test driver can be restarted to perform a new test for as long as time and other resources permit. Thus, only the parts of the state space and success graph actually executed needs to be constructed. Online testers are thus special cases of *environment simulators* where the interactions are derived from a well defined testing theory.

An example of an offline generator is the TGV (Test Generation with Verification technology) toolkit, developed at the University of Rennes, France by Jéron et al. [26, 5, 19]. Input to TGV consists of an SDL or LOTOS specification and a collection of test purposes. A *test purpose* expresses a particular property that the implementation must satisfy. The tool then constructs one *test graph* for each test purpose such that failure to pass the test implies failure to satisfy the test purpose. A test purpose is modeled as an automaton with a subset of states labeled with 'pass' or 'reject' verdicts. The test graph is a controllable subset of the graph consisting of traces leading to pass verdicts. A test case is controllable if it never has a choice of outputs, or a choice of producing an output or accepting an input. TGV is based on an asynchronous testing theory that distinguishes between inputs and outputs.

The test graph is constructed in several steps. First the specification is τ reduced and determinized. It is then composed with the test purpose via a synchronous

product construction. The result is the behavior that is relevant for the test purpose. The 'reject' verdict is used to limit the size of the product graph by not constructing behavior that has been deemed irrelevant by the test engineer. The graph is then traversed in two phases to resolve controllability conflicts. It should be noted that TGV performs τ reduction and determinization *on-the-fly*, i.e., only the used state space is constructed and processed.

The test purpose plays an essential role in this approach since it determines which tests to be are generated. TGV thus uses test purpose based test selection. Because the test purposes are written by test engineerers, this is a manual strategy.

The techniques of TGV has been integrated into the commercial testing tool OBJECTGEODE from Verilog [27].

TORX by de Vries and Tretmans [17] is a tool for online testing. It accepts specifications written in the Promela protocol specification language [23]. A TORX configuration consists of three logical components: A specification interpreter, a test driver, and the implementation under test. The current interpreter is a modified version of the SPIN model checker [23]. The interpreter keeps track of the states reachable after the trace executed so far, and computes the actions possible in the next step. The test driver is the "middle man" which supplies the inputs produced by the interpreter to the implementation, and which invokes the interpreter with the resulting output.

In our view a good online tester systematically constructs and applies all possible tests, and avoids re-executing a test that has already been passed. Thereby it may obtain better coverage. From the description in [17] TORX does not appear to address the issue of obtaining or measuring coverage: It does not save information about the test cases already executed, and thus risks re-executing the same test. It is also unclear whether the tool chooses between the actions produced by the interpreter randomly or in order.

The VVT-RT (Verification, Validation, and Test for Reactive Real-Time Systems) testing tool by Peleska et al. [38, 36] is also based on an online approach, but addresses the issues of systematic test generation and re-execution of test cases. A CSP specification is compiled to a deterministic graph labeled with refusal information similar to our notion of a success graph by the CSP refinement checking tool FDR [43, 44]. This graph is then interpreted at run time to generate test cases and evaluate their outcome.

In addition, a *test monitor* is used to determine the achieved coverage. It is concerned with two types of coverage. The first type is what parts of the refusal graph have been covered, i.e., which requirements remain to be tested. The second type detects what internal paths or components have been activated during a test. Because the implementations may be non-deterministic, this may vary from one execution to another. It may be important for the test engineerer to know precisely which components or paths have been activated, for example which of a set of redundant components was active in a fault tolerant system.

For this reason, the testing monitor can also be equipped with probes into the internals of the implementation under test that enables the monitor to track which internal paths have been executed. The necessary instrumentation of the implementation must usually be done manually.

A final remark is that VVT-RT proposes *hardware in-the-loop* testing. This means that a separate test computer is used instead of the implementations operational environment, and that the test computer is connected to the external physical interface of the implementation. This tool also has the capability of generating real-time tests, see Section 2.3.

Our approach is based on an offline approach where we systematically analyze the specification and cover this with tests. However, handling timing uncertainty requires symbolic test cases or using an online approach. We shall therefore consider how to interpret event recording automata dynamically in future work.

1.3 Checking Experiments

A substantial amount of research was carried out in the period from the 1950's to the early 1970's on theories and algorithms for testing of sequential hardware circuits. This research resulted in efficient test generation algorithms that even guarantees full fault coverage under a specific set of assumptions. We refer to Lee and Yannakakis [31] for a recent survey of these results. The techniques are now being resurrected, generalized, and tried out in the context of conformance testing of communication protocols and reactive real-time systems. We shall therefore outline the key points of these techniques.

The theories were originally developed for Mealy-machines which are finite state machines where transitions are labeled with an input/output pair $s \xrightarrow{i/o} s'$ such that the machine upon receiving input action i produces output action o. Two *states are equivalent* iff for every input sequence both states produce the same output sequence. Two *machines are equivalent* iff for every state in one automata there exists an equivalent state in the other, and vice versa. This implies trace equivalence. A *checking sequence* is a sequence of input actions that is able to distinguish inequivalent implementation machines from a known specification machine under the following assumptions [31]:

1. The specification and implementation are both *deterministic* Mealy machines.
2. The machines has the same set of input and output actions.
3. The specification is minimized (it has no equivalent states).
4. The specification is strongly connected (for every pair of states there is an input sequence that transfers the machine from one state to the other).

5. The specification is completely specified (for every state there is an outgoing transition for every input action).

6. The specification has n states, and the implementation has at most m states, $m \geq n$.

7. The particular W-testing method discussed below also requires a reliable reset operation.

The basic idea in a checking experiment is to ensure that every transition of the specification is correctly implemented by the implementation. A checking experiment follows the following generic algorithm:

1. For every specification transition $s \xrightarrow{i/o} s'$, apply an input sequence that transfers (a correct) implementation to state s.

2. Apply input i, and verify that the output equals o.

3. Verify that the destination state is (equivalent to) state s'.

There is a host of techniques for verifying that the implementation is in the expected state (algorithm step 3). One, the so called W-method proposed by Chow in [11], uses a *characterizing set* for state verification. A characterizing set W for the specification machine is a set of input sequences that can distinguish the behaviors of all states, that is, for every pair of states s, s' $(s \neq s')$ there is an input sequence in W such that the output sequence produced by s differs from the output sequence produced by s'. This method thus requires that the same state is checked using all input sequences in W. This implies re-application of the transfer sequence in step 1, hence the need for a reliable reset action.

Let P be a set of input sequences that visits every transition of the specification machine (i.e., that constitutes a transition cover). P is the set of actions needed in steps 1–2 in the above algorithm. If $n = m$ the sequences $P \circ W$ constitutes an exhaustive test suite ($X \circ Y =_{\text{def}} \{x \cdot y \mid x \in X, y \in Y\}$). These sequences can be joined via a reset-action to form a checking sequence. When $m > n$ it must be checked that the extra $m - n$ states has an equivalent state in the specification. Because the extra states could be attached anywhere to the minimal implementation machine, additional input sequences of exponential length are required to ensure that all extra states are visited by the transition cover, i.e., the sequences $P \circ Act_I^{m-n} \circ W$ constitute an exhaustive test suite (Act_I^{m-n} is the set of all input sequences of length less than or equal to $m - n$).

The total length of a checking sequence constructed using the W-method when $m = n$ is consequently at most n^3k, where k is the number of input actions, and can be constructed in polynomial time. When $m > n$ the length grows to n^2mk^{m-n+1}, and thus becomes exponential in the number of extra states [11].

Test generation tools for FSMs using state characterization techniques exist. An example is TAG (Test Automatic Generation) developed by Tan et al. at

the University of Montreál [48]. The tool uses harmonized state identification sets instead of Chow's characterization set. This produces fewer test cases.

The methodology was developed for Mealy machines, but have in [49] been reformulated for LTSs, and are thus applicable in our testing setup in the case of deterministic systems. Some work exists on generalizing the theory to nondeterministic systems [32, 49], but it does not appear as well developed as the deterministic theory.

The employed implementation relation is trace equivalence. Ours is based on Hennessy tests which, contrary to traces, also have the capability of detecting deadlocks. We believe that this is essential for testing concurrent and distributed systems. Checking experiments appear ideal for relatively small deterministic systems, but where full fault coverage is critical.

1.4 Domain Based Selection

Most introductory books on software engineering or testing, e.g., Pressman [41] and Beizer [4], describe a technique for functional black box testing involving partitioning of the input values into equivalence classes (called domains in [4]) in which the implementation is expected to behave similarly. It is usually recommended to test each equivalence class once in its interior and a number of times on its borders or extreme values.

The rationale for this approach can be explained by considering the faults that can occur. A *computation fault* is wrong processing of all inputs in a domain, thus potentially causing wrong outputs. Hence interior selection. A *domain fault* is an incorrectly implemented domain, and thus inputs are classified wrongly. This is expected to occur most frequently on the border or at extreme values of the domain. Hence extreme value selection.

There is no formal definition of what precisely constitutes an equivalence class or what "same behavior" is. When the source code is available, inputs that follow the same path can be considered equivalent [54, 15]. Another common approach is to collect all predicates occurring in the formal or informal specification describing the pre- and post-conditions of its operations, rewrite these to a disjunctive normal form in order to obtain nice domains, and find their dependencies. Each disjunct is then treated as an sub-domain which is tested separately [4].

These ideas have been applied to formal specifications with the aim of ensuring coverage and automatizing testing, especially test selection and test outcome evaluation. Hierons [22], and Hörcher and Peleska [24] aim at automatizing testing against Z specifications.

Raymond et al. propose in [42] a technique for testing deterministic discrete time reactive systems against specifications given in the synchronous data flow language Lustre. Their work does not deal with testing of real-time constraints,

but whether the implementation computes permissible output *values*, given an history of input *values*. Their work shares with ours the use of efficient data structures commonly used in model checkers for computing relevant inputs. They use binary decision diagrams to solve boolean equations, and use convex polyhedra to represent solutions to numerical constraints. Their test inputs are then randomly chosen from these solution sets. They neither propose an explicit notion of coverage nor measure the resulting coverage.

In our approach we apply these selection principles to clock valuations, thus regarding clocks as parameters, although oddly behaving ones. We use the actions possible in a partition and its deadlocks properties as "outputs" to verify that the implementation responds correctly. We also propose extreme value selection to check that the time constraints are implemented correctly. Guards could be implemented erroneously by initializing timers with wrong values, or timers could be reused unsafely. A premature timeout could be caught by an "upper" or late extreme value because an action that should have been enabled (resp. disabled) have been disabled (resp. enabled). A missed deadline could be detected by a "lower" or early extreme because an action that should have been enabled (resp. disabled) is still disabled (resp. enabled). The notions of computation faults and domain faults therefore also make perfect sense in the time domain.

2 Timed Testing

The introduction of real-time influences all aspects of automated testing. We first discuss potential revisions of the theoretical foundation and the implementation relation in Section 2.1, and thereafter turn to, in our view, mostly theoretical testing methods based on checking sequences. Obtaining a manageable set of tests is a key issue in real-time testing. Some promising approaches are discussed in Section 2.3. We make some remarks in Section 2.4 about other potentially interesting algorithms for the analysis of real-time systems.

2.1 Observations and Timed Preorders

Our timed testing preorder was derived by including time in the traces of the untimed may and must properties. It is important to note that the satisfaction of the resulting implementation relation $\sqsubseteq_{\mathbf{tte}}$ does not imply that no arbitrary test automaton can distinguish the implementation from the specification. An ideal preorder would satisfy the relations stated in Definition 1.

Definition 1 *Test Preorder*:
Let $\mathcal{L}_{tta}$ be the class of test automata, i.e., timed automata whose locations has been labeled with verdicts **pass** or **fail**. Let $\mathcal{S}, \mathcal{I}$ be timed automata.

1. $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{tta}.\ \mathcal{S}$ **must** $\mathcal{T}$ *implies* $\mathcal{I}$ **must** $\mathcal{T}$
2. $\mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$ *iff* $\forall \mathcal{T} \in \mathcal{L}_{tta}.\ \mathcal{S}$ **may** $\mathcal{T}$ *implies* $\mathcal{I}$ **may** $\mathcal{T}$
3. $\mathcal{S} \sqsubseteq_{\mathbf{te}} \mathcal{I}$ *iff* $\mathcal{S} \sqsubseteq_{\mathbf{must}} \mathcal{I} \wedge \mathcal{S} \sqsubseteq_{\mathbf{may}} \mathcal{I}$

□

That is, $\mathcal{L}_{\mathbf{tmust}}$ does not fully characterize such a preorder. One reason is that the observations one would naturally make in a timed model change because the progression of time can be used to observe refusal of actions.

The notion of *refusal testing* in the untimed setting was first explored by Phillips in [40]. Contrary to our Hennessy based tests which deadlock when the implementation refuses to engage in one of the offered actions, refusal testing assumes that the tester can observe such refusals and continue with an alternative set of actions. In [40] this is explained as a button pressing experiment where the implementation is equiped with a button for each action as well as a green light. In a basic experiment a set of actions are continuously pressed until one is accepted, or until the green light goes off. The green light is constructed to be on while the implementation has internal processing to do, and to be off when the implementation has reached a stable state without being able to synchronize with the offered actions.

In a timed setting this observation of refusals becomes more natural since one can offer a set of actions and wait some amount of time. If the timer goes off a time bounded refusal has been observed.

This idea of refusals seems to underly the testing theory developed by Hennessy and Regan in [21] for the discrete Timed Process Language (TPL). They define the notions of may and must testing, and give an alternative characterizations based on barbs, and based on passing tests in an associated test language (F-tests). Further, they also give a proof system for TPL. This work is thus a timed dual to the classical untimed work of De Nicola and Hennessy [34].

To apply the notion of refusals to dense timed automata we conjecture the need for test automata structured like the one illustrated in Figure 2. The automaton continuously offers a set of actions for c time units, and uses an internal action to time out.

The *timed failures* model of timed CSP [46] could also serve as basis for a continuous time testing theory. A timed failure is a timed trace and a set of refusal tokens describing which actions are continuously refused in which time intervals along that trace. The semantics of timed CSP is defined using timed failures.

An alternative testing theory is developed by Cleaveland and Zwarico in [16]. In

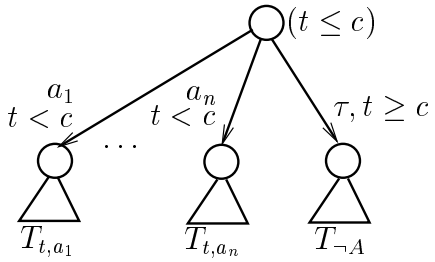

Figure 2. *Test automaton for densely timed automata? t is a clock used by the test automata, c is a real-valued constant, T_{t,a_i} is the sub test after executing a_i at time t, and $T_{\neg A}$ is the sub test after refusing $A = \{a_1 \ldots a_n\}$ for c time units.*

this theory an internal computation step (τ action) is defined to take one time unit. Systems are related by a *faster-than* relation. This work thus suggests a link between real-time conformance testing and performance testing.

We conclude that the theoretical ground for real-time testing is not completely covered. We are looking forward to a well-developed and practical theory for timed automata.

We finally remark that Mok's Real-Time Logic (RTL) [25] like ERA expresses time constraints on the occurrences of events. Central to RTL is the occurrence relation $R(e, i, t)$, which states that the ith occurrence of event e happens at time t. Time constraints are expressed by a first order predicate logic on time- and occurrence variables. The event recording automata model is similar in the sense of stating timing constraints on event occurrences. However, in the basic definition, event recording automata only permit reference to the last occurrence of an event, and permits a limited set of guards only.

2.2 Checking Experiments for Real-time Systems

It would be natural to assume that exhaustive testing of densely timed systems would be impossible because of the infinite state spaces. However, it was shown by Springintveld et al. in [47] that a *finite* set of finite length tests suffices. Like the untimed case, exhaustiveness is only ensured under a set of assumptions about the implementation. Before stating the exact result and its assumptions, some definitions are necessary.

A *region* is a symbolic representation of a set of clock valuations, or formally, an equivalence class on clock valuations induced by the equivalence relation defined in Definition 3. The region concept was proposed by Alur and Dill in [2, 1] as a vehicle for studying decision procedures for timed automata, and has also been applied to model checking.

Definition 3 *Region Equivalence [2]*:
Let X be a set of clocks, and let $\bar{u}, \bar{u}'$ be clock valuations. Two clock valuations are region equivalent, written $\bar{u} \doteq_\rho \bar{u}'$, *iff* $\forall x, y \in X$

1. $\lfloor \bar{u}(x) \rfloor = \lfloor \bar{u}'(x) \rfloor$ or $\bar{u}(x) > c_x$ and $\bar{u}'(x) > c_x$
2. if $\bar{u}(x) \leq c_x$ and $\bar{u}(y) \leq c_y$ then
   $(frac(\bar{u}(x)) \leq frac(\bar{u}(y))$ *iff* $frac(\bar{u}'(x)) \leq frac(\bar{u}'(y)))$
3. if $\bar{u}(x) \leq c_x$ then $(frac(\bar{u}(x)) = 0$ *iff* $frac(\bar{u}'(x)) = 0)$

□

A clock value is divided into two parts, the integral part $\lfloor \bar{u}(x) \rfloor$, and the fractional part $frac(\bar{u}(x))$. The integral part is the largest integer not larger than $\bar{u}(x)$. Two clock valuations are equivalent if the clocks agree on their integral parts, and if all fractional parts are 0 or if they have the same ordering on the fractional parts. A clock can be assigned the designated value ∞ when it exceeds the fixed constant c_x. Beyond c_x the precise value of x is irrelevant with respect to the evaluation of the guards in the automaton. c_x equals the largest constant used in guards on x when the only assignments to x are $x := 0$.

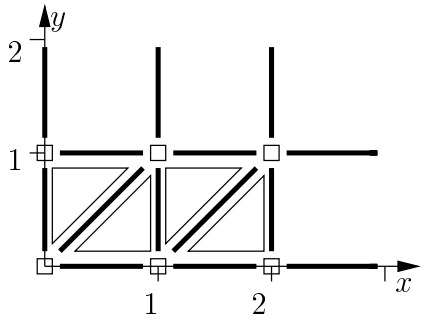

Figure 4. *The regions (boldfaced line segments, corner points, and interior triangles) induced by two clocks x, y and maximum domains $c_x = 2$ and $c_y = 1$. There are 28 regions in this example.*

Figure 4 visualizes the region concept. The key observation is that no guard can distinguish between two clock valuations in the same region. A timed automaton can therefore be analyzed by picking a single representative clock valuation from each region. A *region state* is a pair consisting of a location vector and a region. The reachable state space of a timed automaton can be computed from the initial region state, and by recursively computing its successor regions.

The number of regions in a timed automata with $|X|$ clocks is bounded by $|X|! \cdot 2^{|X|} \cdot \Pi_{x \in X}(2c_x + 2)$. It can thus be noted that the number of regions depends exponentially on both the number of clocks and the clock constants. The number of region states is bounded by multiplying the number of regions

with the number of locations.

Define the *grid automata* $\mathcal{G}(\mathcal{A}, \delta)$ as the sub automata of timed automaton $\mathcal{A}$ that only contains clock valuations that are multiples of δ, where $0 < \delta < 1$. Let S be the states in $\mathcal{A}$, and X the set of clocks. Thus, a state $\langle \bar{l}, \bar{u} \rangle \in S$ is also a state in the grid automaton iff $\forall x \in X.\ \exists k \in \mathbb{N}.\ \bar{u}(x) = k\delta$. $\mathcal{G}(\mathcal{A}, \delta)$ represents a discrete version of $\mathcal{A}$ with discretization step δ.

The algorithm developed in [47] generates test cases for a flavor of timed automata called Bounded Time Domain Input/Output Automata (TIOA). The TIOA model distinguish between input and output actions; inputs are controlled by the environment and outputs by the automaton itself. A TIOA is input enabled which means that it is able to receive inputs at every time instant. The time domain of a clock is a bounded interval of real numbers united with the infinite element ∞. Intuitively, the value of a clock is defined to be ∞ when it exceeds the upper bound of the interval. Beyond this bound the exact value of the clock is irrelevant—it suffices to know that it is large.

[47] proves that bisimilarity of two TIOA can be decided by checking bisimilarity of their (finite state) grid automata, provided that the step size δ is chosen sufficiently small, i.e., $\mathcal{A}_1 \simeq \mathcal{A}_2 \quad \textit{iff} \quad \mathcal{G}(\mathcal{A}_1, \delta) \simeq \mathcal{G}(\mathcal{A}_2, \delta)$.

The basic idea is now to use Chow's algorithm to derive a checking sequence from such a sufficiently fine grained specification grid automata. Note that in the deterministic case trace equivalence coincides with bisimilarity. A sufficiently small step size is 2^{-n} where n is chosen to be greater than or equal to the number of *regions* in the product automata of the specification and implementation TIOA.

Because the number of regions in realistic specifications is very large, it should be clear that the step size becomes infinitesimal, and consequently that the algorithm, while theoretically exhaustive, is highly impractical.

The assumptions of the algorithm can now be stated as:

1. The specification is a known controllable deterministic TIOA. The implementation can be modeled by some controllable deterministic TIOA.
2. The number of regions in the implementation does not exceed n', and the step size is chosen sufficiently small as defined above.
3. The number of states in the grid automaton for the implementation does not exceed m.

A more recent result also using checking sequences of grid automata is presented by En-Nouaary et al. in [18]. The algorithm also uses a deterministic TIOA-model, but here the step size is chosen much larger than in [47]. It has been shown that when the step size is chosen to be $1/(|X| + 2)$ [29], all reachable regions has a representative state in the grid automaton. The checking sequence derived from the grid automata can thus be viewed as a checking sequence for the

region graph of the specification. A fault model based on wrongly implemented regions is also presented.

The resulting test suite is exhaustive wrt. trace equivalence if *uniformity* can be assumed about the implementation. Their uniformity assumption states that if the implementation behaves correctly on some points in a clock region, it also behaves correctly for the remaining points. Although not explicit from the paper, it also seems necessary to assume that the implementation uses the same number of clocks as the specification, and has a no more than m states in its grid automaton.

The authors of [18] give an example of an on-off switch specification. The timed automata has two locations, two edges, one clock, and uses a maximum clock constant of 1. For this example, their algorithm generates 30 test cases. Thus, although the step size is more reasonable than [47], we still believe that it will be too small for most practical applications.

A final effort using checking sequences is reported by Cardell-Oliver and Glover in [10]. Their specification language, termed timed transition systems, is different from timed automata in that no explicit clocks exist. Instead actions are guarded by an upper and lower bound. One of the enabled actions must be taken from a state before any of them disables. Their testing methodology assumes a discrete time interpretation of deterministic, finite state, deadlock and live lock free specifications and implementations. As usual an upper bound on the number of states in the implementation must be assumed. Their approach is implemented in a tool which is applied to a series of small cases. Their result indicates that the approach is feasible, at least for small systems, but problems arise if the implementation has more states than the specification.

Recently Cardell-Oliver [8, 9] has outlined how to generate tests in the form of timed traces from continuously timed automata. Testing is based on generating a checking sequence from a digitized approximation of the original automaton. However, it is unclear from the presentation what properties this approximation has. The step size is chosen much larger than that required to visit every region, and possibly only such that every edge can be visited. Further, it is unclear what kind of communication interface is assumed to exist between the tester and the implementation: She seem to assume that the tester can observe the values of the clocks and state variables in the implementation.

Our symbolic method maintains exact information of the state space of the specification, and only assumes communication with the implementation via synchronizing on actions.

2.3 Real-Time Testing

The application of black-box domain testing to real-time systems is also proposed by Clarke and Lee in [13, 12, 14]. Although their primary goal of using testing as a means of approximating verification to reduce the state explosion

problem is different from ours, their generated tests could potentially be applied to physical systems as well. Their tests are not applied to a physical system, but to an Algebra of Communicating Shared Resources (ACSR) model thereof.

Time requirements are specified as directed acyclic graphs called *constraint graphs*. Nodes in a constraint graph correspond to actions, and edges express a time constraint between the source and target action. An edge is labeled with two pieces of information; an interval describing the permissible delays between the two actions, and a set of actions that may not occur during this interval. Tests can be automatically generated from such constraint graphs.

The authors define the *domain* of an action to be the permissible delays preceding the action. They further define different coverage criteria for these domains, such as observation of all actions, and/or observation of all extreme values in the domains. These criteria are then organized in a subsumption hierarchy.

Their domains are "nice" linear intervals that are directly available in the constraint graph. Also, since their constraint graphs must be acyclic this only permit specification of finite behaviors. Our specifications are given as event recording automata without these restrictions. Our stable edge set partitioning were obtained, not only by looking at single actions, but sets thereof, i.e., we do not assume independence of these. Moreover, since we operate with constraints over many clocks, our partitions are no longer just intervals, but of a dimension corresponding to the number of clocks. We further subdivided these into convex polyhedra, and applied symbolic reachability analysis to find the reachable parts thereof. Thus, we are faced with a more difficult analysis problem, and the constraint graph can to some extend be viewed as the outcome of this analysis.

Braberman et al. [6] describe an approach where a structured analysis/structured design real-time model is represented as a timed Petri net. Analysis methods for timed Petri nets based on constraint solving can be used to generate a symbolic *timed reachability tree* up to a predefined time bound. From this, specific timed test sequences can be chosen. This work shares with ours the generation of tests from a symbolic representation of the state space. The paper also proposes other selection criteria, mostly based on the type and order of the events in the trace. However, they seem to be concerned with generating traces only, and not on deadlock properties as we are. The paper describes no specific data structures or algorithms for constraint solving, and states no results regarding their efficiency. Their approach does not appear to be implemented.

The VVT-RT (now known as RT-Tester) tool developed in cooperation between Bremen University and Verified Systems GmbH. [38, 36, 35], briefly discussed in Section 1.2, also facilitate real-time testing. The specification language is untimed CSP extended with a set of special actions $\texttt{set}_i$ and $\texttt{elapse}_i$ for setting and waiting for timers provided by the runtime system. CSP specification processes can synchronize on these actions, and use them to signal error if an action is not received when required, or for delaying inputs, etc. There are two types of timers. Fixed timers time out after a specified amount of time.

Random timers time out at a random instant in a specified time interval.

It should be noted that the specification accepted by RT-Tester is not a model of the desired target system behavior, but rather is a *test specification* consisting of CSP expressions representing the *joined* behavior of a collection of use cases, each describing a communication scenario, that the test engineerer wish to have tested. Thus much of the burden of generating and selecting test cases lies with the test engineerer. In the reviewed literature there is no description of how a test specification can be derived systematically from a model of the environment and desired target system behavior.

The employed coverage criterion aims at executing every edge in the refusal graph of the test specification at least once, including timer events. There is no coverage criterion for the time domain except that all time outs will be covered. Their approach has detected implementation faults in industrial applications [37, 45, 7, 39], but whether a more systematic and detailed treatment of time could reveal further faults is an open issue.

The approaches reviewed so far are based on so called behavioral specification languages. Another school is logic specifications. Mandrioli et al. [33] proposes a technique for tool *assisted* generation of tests from TRIO discrete time temporal logic specifications. The user assists in selecting the tests to be generated by guiding the decomposition of the specification into subformulae. The tool then generates a history (execution trace) satisfying the chosen subformula. With appropriate input/output labeling this trace can be used as a test case. The authors propose to measure coverage in terms of the number of axioms and predicates that have been tested.

2.4 Algorithms

Yannakakis and Lee [55] describe an alternative algorithm for computing a minimized reachable symbolic transition system from a deterministic timed automaton. The symbolic states resulting from their algorithm are *stable* in the sense that all its members have the same symbolic a successor states for all actions a, including the immediate time successor action. Our symbolic states do not have this property which implies that we must strengthen the symbolic states through a back propagation step prior to trace generation. Their algorithm is of potential interest to us because avoiding back propagation will be a big advantage if our techniques are to be applied in an online testing approach where the test case is generated while being executed. It will enable us to systematically visit all equivalence states without use of back propagation. It is unclear whether we will benefit from the acclaimed efficiency of the algorithm because the initial partitioning that must be provided (although the same as ours where the same edges must be enabled) is required to be convex.

Clock Difference Diagrams [3] is a binary decision diagram inspired data structure that permits representation of non-convex unions of convex sets. This data

structure will allow a much more compact representation of the passed list than presently done. It will possibly also enable reachability analysis to be made based on non-convex partitions, rather than on their present convex subsets.

3 Novelties of Our Approach

Our work focuses on fully automatic generation of tests for timed automata using an offline approach. Compared to the related work outlined in this chapter, our work distinguishes itself by treating time thoroughly and systematically, yet in a way we claim have practical relevance.

We have defined a partitioning of the timed automata specification which is much coarser than the previous approaches based on regions. It is our view that the region based techniques in most cases are too fine grained, and neither scale well, nor provide good guidance in the test selection process.

We have given algorithms that systematically explore the partition graph and cover this with tests. To our knowledge, the employed zone and DBM based algorithms and data structures for symbolic execution and reachability analysis of the specification have not previously been applied to testing.

A further novelty is that we permit both *non-deterministic* timed specifications and implementations. Most other related work on timed testing limits attention to only deterministic systems, whereas non-determinism is permitted in most untimed approaches. Our work thus levels out this discrepancy. To handle non-determinism, we made a slight generalization to Hennessy's testing theory and adopted a specification language, event recording automata, that enabled us to perform the necessary analysis. Interestingly, the ease of analyzing event recording automata does not seem to have been exploited elsewhere.

4 Summary

We have identified two main approaches to test generation. In the preorder based approach, an implementation relation defines the correcness of implementations. A tool interprets the specification with respect to this preorder and generates test cases. Checking sequence based test generation checks that the states of the specification are equivalent to those of the implementation, i.e., that the implementation has no output- or transfer-faults. Both techniques have also been applied in the timed setting.

Test case generators can be online or offline. Online generators execute test cases as they are being generated. Offline generators output completed test suites before test execution. Further, test selection can be manual or fully automatic.

Our approach is preorder based, generates tests offline, and selects tests fully

automatically. Our work focuses on testing real-time constraints. The novelties include automatic test selection from a coarse grained state partitioning, handling non-deterministic timed specifications, and the application of symbolic verification techniques to test generation.

References

[1] Rajeev Alur, Costas Courcoubetis, and David Dill. Model–Checking for Real–Time Systems. In *Fifth IEEE Symposium on Logic in Computer Science*, pages 414–425, 1990.

[2] Rajeev Alur and David L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 25 April 1994.

[3] Gerd Behrmann, Kim G. Larsen, Justin Pearson, Carsten Weise, and Wang Yi. Efficient Timed Reachability Analysis Using Clock Difference Diagrams. In *Computer Aided Verification (CAV'99)*, volume LNCS 1633, pages 22–24. Springer Verlag, July 1999. Trento, Italy.

[4] Boris Beizer. *Software Testing Techniques*. International Thompson Computer Press, 1990. 2nd edition, ISBN 1850328803.

[5] Marius Bozga, Jean-Claude Fernandez, Lucian Ghirvu, Claude Jard, Thierry Jéron, Alain Kerbrat, Pierre Morel, and Laurent Mounier. Verification and Test Generation for the SSCOP Protocol. *Science of Computer Programming*, 36(1):27–52, 2000.

[6] V. Braberman, M. Felder, and M. Marré. Testing Timing Behaviors of Real Time Software. In *Quality Week 1997. San Francisco, USA.*, pages 143–155, April-May 1997 1997.

[7] Bettina Buth, Michel Kouvaras, Jan Peleska, and Hui Shi. Deadlock Analysis for a Fault-Tolerant System. In Michael Johnson, editor, *Algebraic Methodology and Software Technology. AMAST'97, Sidney, Australia*, pages 60–75, December 1997. Springer LNCS 1349.

[8] Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. In *Nordic Workshop on Programming Theory*, Oct 6-8 1999.

[9] Rachel Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, (12):350–371, 2000.

[10] Rachel Cardell-Oliver and Tim Glover. A Practical and Complete Algorithm for Testing Real-Time Systems. In *5th international Symposium on Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'98)*, pages 251–261, September 14–18 1998. Also in LNCS 1486.

[11] Tsun S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, may 1978.

[12] Duncan Clarke. *Testing Real-Time Constraints.* PhD thesis, A dissertation in Computer and Information Science. University of Pennsylvania. Department of Computer and Information Science, December 1996.

[13] Duncan Clarke and Insup Lee. Testing Real-Time Constraints in a Process Algebraic Setting. In *17th International Conference on Software Engineering*, 1995.

[14] Duncan Clarke and Insup Lee. Automatic Test Generation for the Analysis of a Real-Time System: Case Study. In *3rd IEEE Real-Time Technology and Applications Symposium*, 1997.

[15] Lori A. Clarke, Johnette Hassel, and Debra J. Richardson. A Close Look at Domain Testing. *IEEE Transactions of Software Engineering*, 8(4):380–390, 1982.

[16] Rance Cleaveland and Amy E. Zwarico. A Theory of Testing for Real-Time. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 110–119, Amsterdam, The Netherlands, 15–18 July 1991. IEEE Computer Society Press.

[17] René G. de Vries and Jan Tretmans. On the fly Conformance Testing using Spin. In *4th International Spin Workshop*, 1998. In Conjunction with IFIP FORTE/PSTV98.

[18] Abdeslam En-Nouaary, Rachida Dssouli, and Ferhat Khendek. Timed Test Cases Generation Based on State Characterization Technique. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, pages 220–229, December 2–4 1998.

[19] Jean-Claude Fernandez, Claude Jard, Thierry Jéron, and César Viho. An Experiment in Automatic Generation of Test Suites for Protocols with Verification Technology. *Science of Computer Programming*, 29:123–146, 1997.

[20] Per Brinch Hansen. Reproducible Testing of Monitors. *Software—Practice and Experience*, 8:712–729, 1978.

[21] Matthew Hennessy and Tim Regan. A Process Algebra for Timed Systems. *Journal of Information and Computing*, 117:221–239, 1994.

[22] Robert M. Hierons. Testing from a Z Specification. *Software Testing, Verification and Reliability*, 7:19–33, 1997.

[23] Gerard J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, New Jersey, U.S.A, 1991. ISBN 0-13-539925-4.

[24] Hans-Martin Hörcher and Jan Peleska. Using Formal Specifications to Support Software Testing. *Software Quality Journal*, 7:309–327, 1995.

[25] Farnam Jahanian, Aloysius K. Mok, and Douglas A. Stuart. Formal Specification of Real-Time Systems. Technical Report UTCS-TR-88-25, University of Texas, 1988.

[26] Thierry Jéron and Pierre Morel. Test Generation Derived from Model-Checking. In *International Conference on Computer Aided Verification (CAV'99)*, July 7–10 1999. Italy.

[27] Alain Kerbrat, Thierry Jéron, and Roland Groz. Automated Test Generation from SDL Specifications. In *Ninth SDL Forum*, 21-25 June 1999. Montral, Qubec, Canada.

[28] Lawrence M. Krauss. *The Physics of Star Trek*. HarperCollins Publishers, New Yourk, U.S.A, 1995. ISBN 0-06-097710-8.

[29] Kim G. Larsen and Wang Yi. Time Abstracted Bisimulation: Implicit Specifications and Decidability. In *Mathematical Foundations of Programming Semantics (MFPS 9)*, volume LNCS 802. Springer Verlag, April 1993. New Orleans, U.S.A.

[30] Kim Guldstrand Larsen and Arne Skou. Bisimulation Through Probabilistic Testing. *Information and Computation*, 94(1), 1991.

[31] David Lee and Mihalis Yannakakis. Principles and Methods of Testing Finite State Machines—A Survey. *Proceedings of the IEEE*, 84(8):1090–1123, august 1996.

[32] Gang Lou, Gregor v. Bochmann, and Alexandre Petrenko. Test Selection Based on Communicating Nondeterministic Finite-State Machines Using a Generalized Wp-Method. *IEEE Transactions on Software Engineering*, 20(2):140–162, february 1994.

[33] Dino Mandrioli, Sandro Morasca, and Angelo Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.

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

[35] Jan Peleska. Formal Methods and the Development of Dependable Systems. Habilitationsschrift 9612, Institut für Informatik und Praktische Mathematik, Christian-Albrechts Universität, Kiel, 1996.

[36] Jan Peleska, Peter Amthor, Sabine Dick, Oliver Meyer, Michael Siegel, and Cornelia Zahlten. Testing Reactive Real-Time Systems. In *Material for the School – 5th International School and Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'98*, 1998. Lyngby, Denmark.

[37] Jan Peleska and Bettina Buth. Formal Methods for the International Space Station ISS. In E.-R. Olderog and B. Steffen, editors, *Correct System Design*, pages 363–389, 1999. Springer LNCS 1710.

[38] Jan Peleska and Michael Siegel. Test Automation of Safety-Critical Reactive Systems. *South African Computer Journal*, 19:53–77, 1997.

[39] Jan Peleska and Cornelia Zahlten. Test Automation for Avionic Systems and Space Technology. In *GI Working Group on Test, Analysis and Verification of Software*, 1999. Munich, Extended Abstract.

[40] Iain Phillips. Refusal Testing. *Theoretical Computer Science*, 50:241–284, 1987.

[41] Roger S. Pressman. *Software Engineering—A practioner's Approach*. McGraw-Hill Series in Software Engineering and Technology. McGraw-HILL, Inc., New York, second edition, 1987. ISBN 0-07-100232-4.

[42] Pascal Raymond, Xavier Nicollin, Nicolas Halbwacs, and Daniel Weber. Automatic Testing of Reactive Systems. In *19th IEEE Real-Time Systems Symposium (RTSS'98)*, December 2–4 1998.

[43] Andrew W. Roscoe. *Model-Checking CSP*. Prentice-Hall, May 1994. ISBN 0-13-294844-3.

[44] Andrew W. Roscoe, Poul H.B. Gardiner, Michael H. Goldsmith, Jason R. Hulance, David M. Jackson, and J. Bryan Scattergood. Hierarchical Compression for Model-Checking CSP *or* How to Check 10^{20} Dining Philosophers for Deadlock. In Uffe H. Engberg, Kim G. Larsen, and Arne Skou, editors, *Proceedings of the Workshop on Tools and Algorithms for The Construction and Analysis of Systems, TACAS* (Aarhus, Denmark, 19–20 May, 1995), number NS-95-2 in Notes Series, pages 187–200, Department of Computer Science, University of Aarhus, May 1995. BRICS.

[45] Holger Schlingloff, Oliver Meyer, and Thomas Hülsing. Correctness Analysis of an Embedded Controller. In *Data Systems in Aerospace (DASIA99). ESA SP-447, Lisbon, Portugal*, pages 317–325, 1999.

[46] Steve Schneider. An Operational Semantics for Timed CSP. *Information and Computation*, 116:193–213, 1995.

[47] J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. TR CTIT 97-17, University of Twente, 1997. To appear in *Theoretical Computer Science*.

[48] Q.M. Tan, A. Petrenko, and G. v. Bochmann. A Test Generation Tool for Specifications in the Form of State Machines. Technical Report IRO 1016, Department d'IRO, Université de Montreál, february 1996.

[49] Q.M. Tan, A. Petrenko, and G. v. Bochmann. Checking Experiments with Labeled Transition Systems for Trace Equivalence. In *IFIP 10th International Workshop on Testing of Communication Systems (IWTCS'97*, 1997. Korea.

[50] Richard N. Taylor, David L. Levine, and Cheryl D. Kelly. Structural Testing of Concurrent Programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, March 1992.

[51] Jan Tretmans. Test Generation with Inputs, Outputs, and Quiescence. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, pages 127–146, 1996. LNCS 1055.

[52] R.J. van Glabbeek. The Linear Time — Branching Time Spectrum. In *International Conference on Concurrency Theory (CONCUR '90)*, pages 278–297, December 1990. Madras, India, Also in LNCS 458.

[53] Joachim Wegener and Matthias Groctmann. Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing. *Real-Time Systems*, (15):275–298, 1998.

[54] Lee J. White and Edward I. Cohen. A Domain Strategy for Computer Program Testing. *IEEE Transactions of Software Engineering*, 6(3):247–257, 1980.

[55] Mihalis Yannakakis and David Lee. An Efficient Algorithm for Minimizing Real-Time Transition Systems. *Formal Methods in System Design*, 11:113–136, 1997. Extended Abstract in Proceedings of Compute Aided Verification CAV'93, LNCS 697.