Online Testing of Real-time Systems Using UPPAAL

Kim G. Larsen, Marius Mikucionis, and Brian Nielsen

Department of Computer Science, Aalborg University
Fredrik Bajers Vej 7B, 9220 Aalborg Øst, Denmark
{kgl,marius,bnielsen}@cs.auc.dk

Abstract. We present T-UPPAAL — a new tool for online black-box testing of real-time embedded systems from non-deterministic timed automata specifications. We describe a sound and complete randomized online testing algorithm and how to implement it using symbolic state representation and manipulation techniques. We propose the notion of relativized timed input/output conformance as the formal implementation relation. A novelty of this relation and our testing algorithm is that they explicitly take environment assumptions into account, generate, execute and verify the result online using the UPPAAL on-the-fly model-checking tool engine. A medium size case study shows promising results in terms of error detection capability and computation performance.

1 Introduction

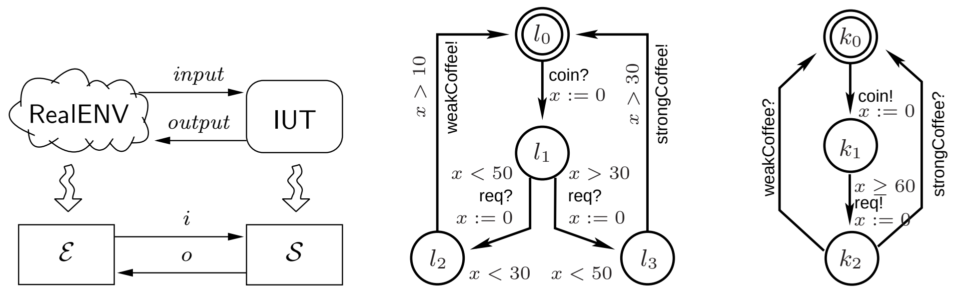
The goal of testing is to gain confidence in a physical computer based system by means of executing it. More than one third of typical project resources is spent on testing embedded and real-time systems, but still it remains ad-hoc, based on heuristics, and error-prone. Therefore systematic, theoretically well-founded and effective automated real-time testing techniques is of great practical value.

1.1 Model Based Testing

Testing conceptually consists of three activities: test case *generation*, test case *execution* and *verdict assignment*. Using model based testing, a behavioral model can be interpreted as a specification that defines the required and allowed observable (real-time) behavior of the implementation. It can therefore be used for automatic generation of sound and (theoretically) complete test suites.

An embedded system interacts closely with its environment which typically consists of the controlled physical equipment (the plant) accessible via sensors and actuators, other computer based systems or digital devices accessible via communication networks using dedicated protocols, and human users. A major task of the embedded system development is to ensure that it works correctly in its real operating environment. Due to lack of development resources it is not feasible to validate the system for all possible environments. Also it is not necessary if the system environments are known to a large extend. However, the requirements to the system and the assumptions made about the environment should be clear and explicit.

We denote the system being developed IUT, and its real operating environment RealENV. These communicate by exchanging $input$ and $output$ signals (seen from the perspective of IUT). Using a model-based development approach, the environment assumptions and system



(a) Abstraction of an embedded system. (b) Example Specification $\mathcal{S}_c$. (c) Example environment $\mathcal{E}_c$.

Fig. 1. Embedded system and example models.

requirements are captured through abstract behavioral models denoted $\mathcal{E}$ and $\mathcal{S}$ respectively, communicating on abstract signals $i \in A_{in}$ and $o \in A_{out}$ corresponding (via a suitable abstraction) to the real $input$ and $output$. This setup is depicted in Figure 1(a).

Modeling the environment explicitly and separately and taking this into account during test generation has several advantages: 1) the test generation tool can synthesize only relevant and realistic scenarios for the given type of environment, which in turn reduces the number of required tests and improves the quality of the test suite; 2) the test engineer can guide the test generator to more specific situations of interest; 3) a separate environment model makes it easy to test the system under different assumptions and use patterns.

The goal of relativized conformance testing is to check whether the behavior of the IUT is correct (conforming) to its specification $\mathcal{S}$ when operating under assumptions $\mathcal{E}$ about the environment. We propose relativized timed input/output conformance relation between model and IUT which coincides with timed trace inclusion taking the environment behavior into account.

1.2 Online Testing

Test cases can be generated from the model offline where the complete test scenarios and verdicts are computed apriori and before execution. Another approach is *online (on-the-fly) testing* that combines test generation and execution: only a single test primitive is generated from the model at a time which is then immediately executed on the IUT. Then the produced output by the IUT as well as its time of occurrence are checked against the specification, a new test primitive is produced and so forth until it is decided to end the test, or an error is detected. An observed test run is a timed trace consisting of an alternating sequence of (input or output) actions and time delays.

There are several advantages of online testing: 1) testing may potentially continue for a long time (hours or even days), and therefore long, intricate, and stressful test cases may be executed; 2) the state-space-explosion problem experienced by many offline test generation tools is reduced because only a limited part of the state-space need to be stored at any point in time; 3) online test generators often allow more expressive specification languages, especially wrt. allowed non-determinism in real-time models.

1.3 Related Work

Model based test generation for real-time specifications has been investigated by others (see e.g., [23, 18, 5, 11, 9, 7, 25, 22, 15, 10, 17]), but remain relatively immature.

A solid and widespread implementation relation used in model based conformance testing of untimed systems is the input/output conformance relation by Tretmans [27]. Informally, input/output conformance requires for all traces in the specification that the implementation never produces an output not allowed by the specification, and that it never refuses to produce an output (stays quiescent) when the specification requires one.

As also noted in [15, 17] a timed input/output conformance relation can be obtained (assuming input enabledness) as timed trace inclusion between the implementation and its specification. Our work further extends this to a *relativized* conformance relation taking environment assumptions explicitly into account. In [27] the specification is permitted to be non-input enabled (thus making the conformance relation non-transitive in general) in order to capture environmental constraints. However, this requires explicit rewriting of the specification when different environments are to be used. Following the seminal work [16] our approach is based on an separate model of the environment. In particular, once conformance has been established with respect to a particular environment we can automatically conclude conformance under more restricted environments. Also, when the IUT is to be used in different environments, it suffices to test it under the most liberal environment assumptions. Furthermore, relativized conformance is transitive.

Model based *offline testing* is often based on a coverage criterion of the model like in [12, 10], on a test purpose as e.g. [14, 15], or a fault-model as [11, 9]. When specifications allow non-determinism, the generated test cases cannot simply be a sequence, but take the form of *behavior trees* adaptive to implementation controlled actions, e.g different outputs or timing. Therefore, most offline test generation algorithms explicitly determinize the specification [8, 14, 22]. However, for expressive formalisms like timed automata this approach is infeasible because in general they cannot be determinized [2] and their unobservable actions cannot always (and when they can it may be very costly) be removed [29]. Much work on timed test generation from timed automata therefore restricts the amount and type of allowed non-determinism. Some works [25, 9, 10] completely disallow non-determinism, whereas others [15, 22] restrict the use of clocks, guards or clock resets. However, in many cases it is important to allow non-determinism, because 1) specifications are often given as a parallel composition of model-components, 2) it allows the implementor some freedom, and 3) the tester is usually concerned with abstract requirements rather than concrete details of the IUT. Note that in particular for real-time systems it may be crucial to allow specification of timing uncertainty, e.g., that an output is expected in some interval of time (e.g., between 1 and 5 time units from now), but not exactly when. Timed automata model this by a non-deterministic choice of letting time pass or outputting an event.

In contrast, online testing is automatically adaptive and only implicitly determinizes the specification, and only partially up to the concrete trace observed so far. The (untimed) online testing algorithm proposed by Tretmans et. al. in [31, 4] continually computes the set of states that the specification can possibly occupy after the observations made so far. Based on this the tester can at any time decide to either perform one of the inputs enabled in the specification, or wait for output from the implementation, and then check whether the output (or its absence) is allowed in the state-set. Online testing from Promela [31] and LOTOS specifications for untimed systems have been implemented in the TORX [30] tool, and practical application to real case studies show promising results [28, 30, 4]. However, TORX provides no support for real-time.

Our work generalize the TORX approach to timed systems and to the handling of explicit environment assumptions. We allow a quite generous (non-deterministic) timed automata language. In addition, we compute the state-set symbolically to track the (potentially dense) timed state space.

Online testing from unrestricted non-deterministic timed automata using symbolic state-set computation [24] was first published by Krichen and Tripakis [17]. We implement a similar approach by extending the UPPAAL model-checker resulting in an integrated and mature testing and verification tool. Our work (originating from [6, 21, 19]; an abstract appeared in [20]) is different from [17] in that 1) the exact timed automata language variant is different and includes separable environment models, 2) we propose a relativized version of timed input/output conformance, 3) our algorithm (presented in much greater detail) generate tests relevant only for the specified environment, and 4) is shown to be sound and complete under certain assumptions, and finally 5) we provide experimental evidence of the feasibility of the technique.

1.4 Contributions

In this paper we describe a tool for online testing of real-time systems. Our main contributions are the notion of *relativized timed input/output conformance* and an implementation based on UPPAAL of a *symbolic algorithm* that performs online testing based on a (possibly densely timed and potentially non-deterministic) timed automata model of the IUT and its assumed environment. We prove under a certain testing hypothesis that our algorithm is sound and (in a precise probabilistic sense) complete. Furthermore, we apply T-UPPAAL to a medium sized case that demonstrates good error detection potential and very encouraging performance.

2 Test Specification

This section formally presents our semantic framework, and introduces TIOTS, timed automata, and our relativized input/output conformance relation.

2.1 Timed I/O Transition Systems

We assume a given set of actions A partitioned into two disjoint sets of output actions A_{out} and input actions A_{in}. In addition we assume that there is a distinguished unobservable action $\tau \notin A$. We denote by A_τ the set $A \cup \{\tau\}$.

Definition 1. *A timed I/O transition system (TIOTS) $\mathcal{S}$ is a tuple $(S, s_o, A_{in}, A_{out}, \rightarrow)$, where S is a set of states, $s_0 \in S$, and $\rightarrow \subseteq S \times (A_\tau \cup \mathbb{R}_{\geq 0}) \times S$ is a transition relation satisfying the usual constraints of* time determinism *(if $s \xrightarrow{d} s'$ and $s \xrightarrow{d} s''$ then $s' = s''$) and* time additivity *(if $s \xrightarrow{d_1} s'$ and $s' \xrightarrow{d_2} s''$ then $s \xrightarrow{d_1+d_2} s''$), $d \in \mathbb{R}_{\geq 0}$, where $\mathbb{R}_{\geq 0}$ denotes non-negative real numbers.*

Notation for TIOTS. Let $a, a_{1...n} \in A$, $\alpha \in A_\tau \cup \mathbb{R}_{\geq 0}$, and $d, d_{1...n} \in \mathbb{R}_{\geq 0}$. We write $s \xrightarrow{\alpha}$ iff $s \xrightarrow{\alpha} s'$ for some s'. We use $\Rightarrow$ to denote the τ-abstracted transition relation such that $s \overset{a}{\Rightarrow} s'$ iff $s \xrightarrow{\tau}^* \xrightarrow{a} \xrightarrow{\tau}^* s'$, and $s \overset{d}{\Rightarrow} s'$ iff $s \xrightarrow{\tau}^* \xrightarrow{d_1} \xrightarrow{\tau}^* \xrightarrow{d_2} \xrightarrow{\tau}^* \cdots \xrightarrow{\tau}^* \xrightarrow{d_n} \xrightarrow{\tau}^* s'$ where $d = d_1 + d_2 + \cdots d_n$. We extend $\Rightarrow$ to sequences of actions and delays in the usual manner.

We assume that the TIOTS $\mathcal{S}$ is strongly *input enabled* and *non-blocking*. $\mathcal{S}$ is strongly input enabled iff $s \xrightarrow{i}$ for all states s and for all input actions i. $\mathcal{S}$ is non-blocking iff for any state s and any $t \in \mathbb{R}_{\geq 0}$ there is a timed output trace $\sigma = d_1 o_1 \ldots o_n d_{n+1}$ such that $s \stackrel{\sigma}{\Rightarrow}$ and $\sum_i d_i \geq t$. Thus $\mathcal{S}$ will not block time in any input enabled environment.

To model potential implementations it is usefull to define the properties of *isolated outputs* and *determinism*. We say that $\mathcal{S}$ has isolated outputs if whenever $s \xrightarrow{o}$ for some output action o, then $s \stackrel{\tau}{\not\rightarrow}$ and $s \stackrel{d}{\not\rightarrow}$ for all $d > 0$ and whenever $s \xrightarrow{o'}$ then $o' = o$. Finally, $\mathcal{S}$ is deterministic if for all delays or actions α and all states s, whenever $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ then $s' = s''$.

An observable *timed trace* $\sigma \in (A \cup \mathbb{R}_{\geq 0})^*$ is of the form $\sigma = d_1 a_1 d_2 \ldots a_k d_{k+1}$. We define the observable timed traces $\mathsf{TTr}(s)$ of a state s as:

$$\mathsf{TTr}(s) = \{\sigma \in (A \cup \mathbb{R}_{\geq 0})^* \mid s \stackrel{\sigma}{\Rightarrow}\} \tag{1}$$

For a state s (and subset $S' \subseteq S$) and a timed trace σ, s After σ is the set of states that can be reached after σ:

$$s \;\mathsf{After}\; \sigma = \{\, s' \mid s \stackrel{\sigma}{\Rightarrow} s' \,\}, \quad S' \;\mathsf{After}\; \sigma = \bigcup_{s \in S'} s \;\mathsf{After}\; \sigma \tag{2}$$

The set $\mathsf{Out}(s)$ of observable outputs or delays that can occur in $s \in S' \subseteq S$ is defined as:

$$\mathsf{Out}(s) = \{\, a \in A_{out} \cup \mathbb{R}_{\geq 0} \mid s \stackrel{a}{\Rightarrow} \}, \quad \mathsf{Out}(S') = \bigcup_{s \in S'} \mathsf{Out}(s), \tag{3}$$

Timed Automata [2] is an expressive and popular formalism for modelling real-time systems. Let X be a set of $\mathbb{R}_{\geq 0}$-valued variables called *clocks*. Let $\mathcal{G}(X)$ denote the set of *guards* on clocks being conjunctions of simple constraints of the form $x \bowtie c$, and let $\mathcal{U}(X)$ denote the set of *updates* of clocks corresponding to sequences of statements of the form $x := c$, where $x \in X$, $c \in \mathbb{N}$, and $\bowtie \in \{\leq, <, =, >, \geq\}$. A *timed automaton* over (A, X) is a tuple (L, ℓ_0, I, E), where L is a set of locations, $\ell_0 \in L$ is an initial location, $I : L \rightarrow \mathcal{G}(X)$ assigns invariants to locations, and E is a set of edges such that $E \subseteq L \times \mathcal{G}(X) \times A_\tau \times \mathcal{U}(X) \times L$. We write $\ell \xrightarrow{g,\alpha,u} \ell'$ iff $(\ell, g, \alpha, u, \ell') \in E$.

The semantics of a timed automaton is defined in terms of a TIOTS over states of the form $s = (\ell, \bar{v})$, where ℓ is a location and $\bar{v} \in \mathbb{R}_{\geq 0}^X$ is a clock valuation satisfying the invariant of ℓ. Intuitively, there are two kinds of transitions: delay transitions and discrete transitions. In delay transitions, $(\ell, \bar{v}) \xrightarrow{d} (\ell, \bar{v} + d)$, the values of all clocks of the automaton are incremented by the amount of the delay, d. Discrete transitions $(\ell, \bar{v}) \xrightarrow{\alpha} (\ell', \bar{v}')$ correspond to execution of edges $(\ell, g, \alpha, u, \ell')$ for which the guard g is satisfied by $\bar{v}$. The clock valuation $\bar{v}'$ of the target state is obtained by modifying $\bar{v}$ according to updates u and satisfies the invariants on ℓ'.

Figure 1(b) shows a timed automaton specifying the requirements to a coffee machine. It has a facility that allows the user, after paying, to indicate his eagerness to get coffee by pushing a request button on the machine forcing it to output coffee. However, allowing insufficient brewing time results in a weak coffee. Waiting less then 30 time units definitely results in weak coffee, and waiting more than 50 definitely in strong coffee. Between 30 and 50 time units the choice is non-deterministic, meaning that the IUT/implementor may decide what to produce. After the request, it takes the machine an additional (non-deterministic) 10 to 30 (30 to 50) time units to produce weak coffee (strong coffee). The timed automaton in Figure 1(c) models a potential (nice) user of the machine that pays before requesting coffee and wants strong coffee thus requesting only after 60 time units.

TIOTS Composition. Let $\mathcal{S} = (S, s_0, A_{in}, A_{out}, \rightarrow)$ be an input enabled, non-blocking TIOTS. An *environment* $\mathcal{E}$ for $\mathcal{S}$ is itself an input enabled, non-blocking, TIOTS $\mathcal{E} = (E, e_o, A_{out}, A_{in}, \rightarrow)$. Here E is the set of environment states and the set of input (output) actions of $\mathcal{E}$ is identical to the output (input) actions of $\mathcal{S}$. The parallel composition of $\mathcal{S}$ and $\mathcal{E}$ forms a *closed system* $\mathcal{S} \parallel \mathcal{E}$ whose observable behavior is defined by the TIOTS $(S \times E, (s_0, e_0), A_{in}, A_{out}, \rightarrow)$ where $\rightarrow$ is defined as

$$\frac{s \xrightarrow{a} s' \quad e \xrightarrow{a} e'}{(s,e) \xrightarrow{a} (s',e')} \qquad \frac{s \xrightarrow{\tau} s'}{(s,e) \xrightarrow{\tau} (s',e)} \qquad \frac{e \xrightarrow{\tau} e'}{(s,e) \xrightarrow{\tau} (s,e')} \qquad \frac{s \xrightarrow{d} s' \quad e \xrightarrow{d} e'}{(s,e) \xrightarrow{d} (s',e')} \tag{4}$$

The timed automata $\mathcal{S}_c$ and $\mathcal{E}_c$ respectively shown in Figure 1(b) and 1(c) can be composed in parallel on actions $A_{in} = \{\mathsf{req}, \mathsf{coin}\}$ and $A_{out} = \{\mathsf{weakCoffee}, \mathsf{strongCoffee}\}$ forming a closed network[1].

2.2 Relativized Timed Conformance

In this section we define our notion of conformance between TIOTSs. Our notion derives from the input/output conformance relation (ioco) of Tretmans and de Vries [27, 31] by taking time and environment constraints into account. Under assumptions of input enabledness our relativized timed conformance relation coincides with relativized timed trace inclusion. Like ioco, this relation ensures that the implementation only has behavior allowed by the specification. In particular, 1) it is not allowed to produce an output at a time (too late or too early) when one is not allowed by the specification, 2) it is not allowed to omit producing an output when one is required by the specification by delaying more than allowed. Thus, timed trace inclusion offers the notion of time-bounded quiescence that—in contrast to ioco's conceptual eternal quiescence—can be observed in a real-time system.

Definition 2. *Given an environment $e \in E$ the e-relativized timed input/output conformance relation rtioco_e between system states $s, t \in S$ is defined as:*

$$s \,\mathsf{rtioco}_e\, t \quad \text{iff} \quad \forall \sigma \in \mathsf{TTr}(e).\, \mathsf{Out}\big((s,e) \text{ After } \sigma\big) \subseteq \mathsf{Out}\big((t,e) \text{ After } \sigma\big)$$

Whenever $s \,\mathsf{rtioco}_e\, t$ we will say that s is a correct implementation (or refinement) of the specification t under the environmental constraints expressed by e. Under the assumption of input-enabledness of both $\mathcal{S}$ and $\mathcal{E}$ we may characterize relativized conformance in terms of trace-inclusion as follows:

Lemma 1. *Let $\mathcal{S}$ and $\mathcal{E}$ be input-enabled with states $s, t \in S$ and $e \in E$ respectively. Then*

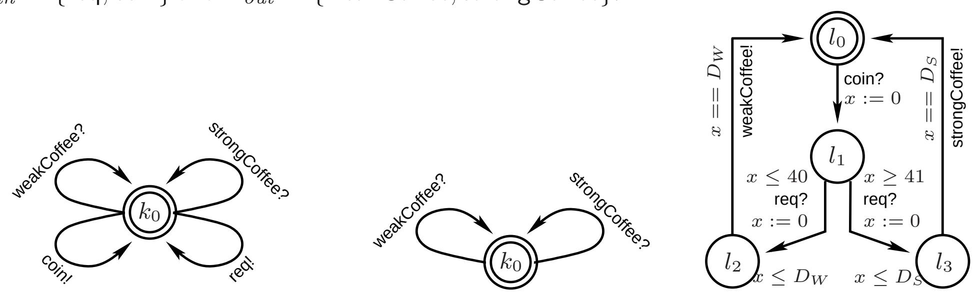
$$s \,\mathsf{rtioco}_e\, t \quad \text{iff} \quad \mathsf{TTr}(s) \cap \mathsf{TTr}(e) \subseteq \mathsf{TTr}(t) \cap \mathsf{TTr}(e)$$

Thus if $s \,\mathsf{rtioco}_e\, t$ does not hold then there exists a trace σ of e such that $s \overset{\sigma}{\Rightarrow}$ but $t \overset{\sigma}{\not\Rightarrow}$. Given the notion of relativized conformance it is natural to consider the preorder on environments based on their discriminating power, i.e. for two environments e and f:

$$e \sqsubseteq f \quad \text{iff} \quad \mathsf{rtioco}_f \subseteq \mathsf{rtioco}_e \tag{5}$$

[1] To avoid cluttering the figures we have not made them explicitly input enabled; for the unspecified inputs there is an undrawn self looping edge that merely consumes the input without changing the location.

(to be read f is more discriminating than e). It follows from the definition of rtioco that $e \sqsubseteq f$ iff $\mathsf{TTr}(e) \subseteq \mathsf{TTr}(f)$. In particular there is a most (least) discriminating input enabled and non-blocking environment U (O) given by $\mathsf{TTr}(U) = (A \cup \mathbb{R}_{\geq 0})^*$ $\big(\mathsf{TTr}(O) = (A_{out} \cup \mathbb{R}_{\geq 0})^*\big)$. The corresponding conformance relation rtioco_U (rtioco_O) specializes to simple timed trace inclusion (timed output trace inclusion) between system states. In Figure 2(a) and Figure 2(b) the most-discriminating and the least-discriminating environments are given when $A_{in} = \{\mathsf{req}, \mathsf{coin}\}$ and $A_{out} = \{\mathsf{weakCoffee}, \mathsf{strongCoffee}\}$.



(a) most-$\sqsubseteq$ environment $\mathcal{E}_U$. (b) least-$\sqsubseteq$ environment $\mathcal{E}_O$. (c) IUT: $\mathcal{I}(D_S, D_W)$.

Fig. 2. Implementation of coffee machine

2.3 Examples

The specification machine $\mathcal{S}_c$ and environment $\mathcal{E}_c$ were described in Section 2.1. The (deterministic) implementation $\mathcal{I}(D_S, D_W)$ in Figure 2(c) produces weak coffee (strong coffee) after less than 40 time units (more than 41 time units) and an additional brewing time of D_S (resp. D_W) time units. Observe that any trace of the implementation $\mathcal{I}(40, 20)$ (in any environment) can be matched by the specification; hence $\mathcal{I}(40, 20)$ $\mathsf{rtioco}_{\mathcal{E}_U}$ S. Thus also $\mathcal{I}(40, 20)$ $\mathsf{rtioco}_{\mathcal{E}_c}$ $\mathcal{S}_c$. In contrast $\mathcal{I}(70, 5)$ $\not\mathsf{rtioco}_{\mathcal{E}_U}$ $\mathcal{S}_c$ for two reasons: 1) it has the timed trace $\mathsf{coin} \cdot 30 \cdot \mathsf{req} \cdot 5 \cdot \mathsf{weakCoffee}$ that $\mathcal{S}_c$ does not, i.e., it may produce weak coffee too soon (no time to insert a cup); 2) it has the trace $\mathsf{coin} \cdot 50 \cdot \mathsf{req} \cdot 70$ not in $\mathcal{S}_c$ meaning that it produces strong coffee too slowly. Assume now that the strong coffee error is fixed, and that the machine $\mathcal{I}(40, 5)$ is used in the restricted environment of nice users $\mathcal{E}_c$. Here, despite the remaining weak coffee error in $\mathcal{E}_U$, $\mathcal{I}(40, 5)$ $\mathsf{rtioco}_{\mathcal{E}_c}$ $\mathcal{S}_c$ because $\mathcal{E}_c$ never requests weak coffee.

3 Test Generation and Execution

We present the main algorithm, its soundness and completeness proof, and how to implement it.

3.1 The Main Algorithm

The input to Algorithm 1 is two TIOTSs $\mathcal{S} \parallel \mathcal{E}$ respectively modelling the IUT and environment. It maintains the current reachable state set $\mathcal{Z} \subseteq S \times E$ that the test specification can possibly occupy after the timed trace observed so far. Knowing this state estimate allows it to choose appropriate test primitives and to validate IUT outputs.

Algorithm 1 Test generation and execution: $TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T)$. $\mathcal{Z} := \{(s_0, e_0)\}$.

```
while Z ≠ ∅ ∧ ♯iterations ≤ T do switch(action, delay, restart) randomly:
    action:                                              // offer an input
        if EnvOutput(Z) ≠ ∅
            randomly choose a ∈ EnvOutput(Z)
            send a to IUT
            Z := Z After a
    delay:                                               // wait for an output
        randomly choose δ ∈ Delays(Z)
        sleep for δ time units and wake up on output o
        if o occurs at δ' ≤ δ then
            Z := Z After δ'
            if o ∉ ImpOutput(Z) then return fail
            else Z := Z After o
        else                                             // no output within δ delay
            Z := Z After δ
    restart:                                             //reset and restart
        Z := {(s0, e0)}
        reset IUT
if Z = ∅ then return fail
else return pass
```

The tester can perform three basic actions: either send an input (enabled environment output) to the IUT, wait for an output for some time, or reset the IUT and restart. If the tester observes an output or a time delay it checks whether this is legal according to the state set. The state set is updated whenever an input is offered, or an output or delay is observed. Illegal occurrence or absence of an output is detected if the state set becomes empty which is the result if the observed trace is not in the specification. The functions used in Algorithm 1 are defined as: $\mathsf{EnvOutput}(\mathcal{Z}) = \{a \in A_{in} \mid \exists (s,e) \in \mathcal{Z}. e \xrightarrow{a}\}$, $\mathsf{ImpOutput}(\mathcal{Z}) = \{a \in A_{out} \mid \exists (s,e) \in \mathcal{Z}. s \xrightarrow{a}\}$, and $\mathsf{Delays}(\mathcal{Z}) = \{d \mid \exists (s,e) \in \mathcal{Z}. e \xRightarrow{d}\}$. Note that EnvOutput is empty if the environment has no outputs to offer. Similarly, Delays cannot pick at random from the entire domain of real-numbers if the environment must produce an input to the IUT model before a certain moment in time. We use the efficient reachability algorithm implementation [3] to compute the operator After. It operates on bounded symbolic states, checks for inclusions and thus always terminates even if the model contains τ action loops.

3.2 Soundness and Completeness

Algorithm 1 constitutes a randomized algorithm for providing stimuli to (in terms of input and delays) and observing resulting reactions from (in terms of output) a given IUT. Assuming the behavior of the IUT is a non-blocking, input enabled, deterministic TIOTS with isolated outputs the reaction to any given timed input trace $\sigma = d_1 i_1 \dots d_k i_k d_{i+1}$ is completely deterministic. More precisely, given the stimuli σ there is a unique $\rho \in \mathsf{TTr}(\mathsf{IUT})$ such that $\rho \uparrow A_{in} = \sigma$, where $\rho \uparrow A_{in}$ is the natural projection of the timed trace ρ to the set of input actions.

Under a certain (theoretically necessary) testing hypothesis about the behaviour of IUT and given that the TIOTSs $\mathcal{S}$ and $\mathcal{E}$ satisfy certain assumptions, the randomization used in Algorithm 1 may be chosen such that the algorithm is both complete and sound in the sense that it (eventually with probability one) gives the verdict “fail” in all cases of non-conformance and the verdict “pass” in cases of conformance. The hypothesis and assumptions are based on the results on digitization techniques in [26][2] which allow the dense-time trace inclusion problem between two sets of timed traces to be reduced to discrete time. In particular it suffices to choose unit delays in Algorithm 1 (assuming that the models and IUT share the same magnitude of a time unit).

Theorem 1. *Assume that the behaviour of* IUT *may be modelled*[3] *as an input enabled, non-blocking, deterministic TIOTS with isolated outputs. Furthermore assume that* $\mathsf{TTr}(\mathsf{IUT})$ *and* $\mathsf{TTr}(\mathcal{E})$ *are* closed under digitization *and that* $\mathsf{TTr}(\mathcal{S})$ *is* closed under inverse digitization. *Then Algorithm 1 with only unit delays is sound and complete in the following senses:*

1. *Whenever* $TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail$ *then* $\mathsf{IUT}\ \not{\mathsf{rtioco}}_{\mathcal{E}}\ \mathcal{S}$.
2. *Whenever* $\mathsf{IUT}\ \not{\mathsf{rtioco}}_{\mathcal{E}}\ \mathcal{S}$ *then* $\mathsf{Prob}\big(TestGenExe(\mathcal{S}, \mathcal{E}, \mathsf{IUT}, T) = fail\big) \xrightarrow{T\to\infty} 1$ *where* T *is the maximum number of iterations of the while-loop before exiting.*

Proof. (Sketch) Soundness follows from an easy induction on $|\rho|$ that when starting each iteration of the while-loop the timed trace ρ observed since the last restart satisfies $\rho \in \mathsf{TTr}(\mathsf{IUT})$, $\rho \in \mathsf{TTr}(\mathcal{E})$ and $\rho \in \mathsf{TTr}(\mathcal{S})$ and that any chosen extension $\rho\alpha$ still lies in $\mathsf{TTr}(\mathsf{IUT}) \cap \mathsf{TTr}(\mathcal{E})$.

As for completeness assume that the IUT does not conform to $\mathcal{S}$ relative to $\mathcal{E}$. Then $\mathsf{TTr}(\mathsf{IUT}) \cap \mathsf{TTr}(\mathcal{E}) \not\subseteq \mathsf{TTr}(\mathcal{S})$. However due to the assumed properties of closure with respect to digitization respectively inverse digitization this failing timed trace inclusion is equivalent to the existence of a timed trace $\rho = d_1 a_1 d_2 a_2 \ldots d_k a_k d_{k+1}$ with all delays being integral such that $\rho \in \mathsf{TTr}(\mathsf{IUT}) \cap \mathsf{TTr}(\mathcal{E})$ but $\rho \notin \mathsf{TTr}(\mathcal{S})$. Now let $\sigma = \rho \uparrow A_{in}$; that is σ is the input-delay stimuli allowed by $\mathcal{E}$ which when given to IUT will result in the timed trace ρ. Now assume that the random choice of input action, unit delay and restart is made using a fixed discrete and finite probability distribution (with p being the smallest probability used) it is clear that:

$$\mathsf{Prob}(\sigma \text{ is generated between two given consecutive restarts }) \geq p^{K+D}$$

where K respectively D is the number of input actions respectively accumulated delay in σ. Now let $\epsilon = p^{K+D}$ it follows that

$$\mathsf{Prob}(\sigma \text{ is generated before k'th restart }) \geq 1 - (1 - \epsilon)^{k-1}$$

Obviously there will in general be several input stimuli that will reveal the lack of conformance. Hence the above probability just provides a lower bound for Algorithm 1 yielding the verdict “fail” before the k’th restart. Obviously, as $T \to \infty$ also the number of restarts diverges and hence we see that $\mathsf{Prob}(\sigma \text{ is generated}) = 1$. □

From [26, 13] it follows that the closure properties required in Theorem 1 are satisfied if the behaviour of IUT and $\mathcal{E}$ are TIOTSs induced by closed timed automata (i.e. where all guards and invariants are non-strict) and $\mathcal{S}$ is a TIOTS induced by an open timed automaton (i.e. with guards and invariants being strict). In practice these requirements are not restrictive, e.g. for strict guards one can always scale the clock constants to obtain arbitrary high precision.

---

[2] We refer the reader to [26] for the precise definition of digitization and inverse digitization.

[3] The assumption that the IUT can be modelled by a formal object in a given class is commonly refered to as the *test hypothesis*. Only its existence is assumed, not a known instance. In particular it may be extremely large, and structurally totally unrelated to the specification.

3.3 Symbolic State-set Computation

We now discuss the concrete realization of Algorithm 1. We use (well established) symbolic constraint solving techniques to represent sets of clock valuations compactly. A zone over a set of clocks X is a conjunction of clock inequations of the form $x_i - x_j \prec c_{i,j}$, $x_i \prec c_{iu}$, and $c_{il} \prec x_i$, where $\prec \in \{<, \leq\}$, $c_{i,j}, c_{il}, c_{iu}$ are integer constants including $\pm\infty$, and $x_i, x_j \in X$. A *symbolic state* is a pair $\langle \bar{\ell}, Z \rangle$ consisting of a vector $\bar{\ell}$ of locations for each parallel automaton and the zone Z. Z denotes a set of clock valuations, i.e., a symbolic state represents a set of concrete states: $\langle \bar{\ell}, Z \rangle = \{(\bar{\ell}, \bar{v}) \mid \bar{v} \in Z\}$. Henceforth $\mathcal{Z} = \{\langle \bar{\ell}_1, Z_1 \rangle \ldots \langle \bar{\ell}_n, Z_n \rangle\}$ denotes the set of concrete states represented by the union of the symbolic states of $\mathcal{Z}$.

We use the following operations on zones: conjunction $Z \wedge Z'$, future $Z^{\uparrow} = \{\bar{v} + \delta \mid \bar{v} \in Z, \delta \in \mathbb{R}_{\geq 0}\}$, clock x assignment to c value $Z_{x:=c} = \{\bar{v}[c/x] \mid \bar{v} \in Z\}$, Z_r the (successive) assignment of all clock assignments in r, containment check $Z \subseteq Z'$, and check for emptiness $Z = \emptyset$. The symbolic transition relation $\rightarrowtail$ between symbolic states denotes the possibility of taking a transition from a (concrete) state in the source symbolic state to one in the destination. It is computed as follows:

$$\langle \bar{\ell}, Z \rangle \overset{\gamma}{\rightarrowtail} \langle \bar{\ell}', (Z \wedge g)_r \wedge I(\bar{\ell}') \rangle \text{ if } \bar{\ell} \xrightarrow{g,\gamma,r} \bar{\ell}' \text{ where } \gamma \in A_\tau \tag{6}$$

The required symbolic algorithms are similar to those used for model checking [3, 1] except that only states up to a certain time limit needs to be computed. This is most easily accomplished by introducing an auxiliary clock t that is set to zero whenever an observable action occurs.

Algorithm 2 computes the function $\mathsf{Closure}_{\delta\tau}(\mathcal{Z}, d) = \bigcup_{0 \leq d' \leq d} \mathcal{Z} \text{ } \mathsf{After} \text{ } d'$ that collects the reachable symbolic state set within a delay of d. The predicate $\mathsf{Contains}(\mathcal{Z}, \langle \bar{\ell}, Z \rangle)$ tests whether a symbolic state $\langle \bar{\ell}, Z \rangle$ is covered by some symbolic state in $\mathcal{Z}$.

Algorithm 2 $\mathsf{Closure}_{\delta\tau}(\mathcal{Z}, d)$ $pass := \emptyset$, $wait := \mathcal{Z}$

```
while wait ≠ ∅ do
    wait := wait\{⟨ℓ̄, Z⟩}                          // pick a symbolic state
    Z := Z↑ ∧ (t ≤ d) ∧ I(ℓ̄)                              // limited delay
    pass := pass ∪ {⟨ℓ̄, Z⟩}
    for each symbolic transition ⟨ℓ̄, Z⟩ ↣τ ⟨ℓ̄′, Z′⟩
        if not Contains(pass, ⟨ℓ̄′, Z′⟩) then wait := wait ∪ {⟨ℓ̄′, Z′⟩}
return pass.
```

The function $\mathsf{Closure}_{\tau}(\mathcal{Z}) = \mathsf{Closure}_{\delta\tau}(\mathcal{Z}, 0)$ that collects the reachable symbolic state set after all possible internal transitions in zero delay can be computed similarly. Given these functions, the actual algorithms for computing $\mathcal{Z}$ After δ and $\mathcal{Z}$ After a become trivial:

$$\mathcal{Z} \text{ } \mathsf{After} \text{ } a = \mathsf{Closure}_{\tau}\Big(\{\langle \bar{\ell}', Z' \rangle \mid \langle \bar{\ell}, Z \rangle \in \mathsf{Closure}_{\tau}(\mathcal{Z}),\ \langle \bar{\ell}, Z \rangle \overset{a}{\rightarrowtail} \langle \bar{\ell}', Z' \rangle\}\Big) \tag{7}$$

$$\mathcal{Z} \text{ } \mathsf{After} \text{ } \delta = \Big\{\langle \bar{\ell}, Z' \rangle \;\Big|\; \langle \bar{l}, Z \rangle \in \mathsf{Closure}_{\delta\tau}(\mathcal{Z}, \delta),\ Z' = \big(Z \wedge (t == \delta)\big)_{t:=0}\Big\} \tag{8}$$

3.4 Choice of Delays

The environment model restricts the possible actions that can be chosen by the tester. It bounds the delays before an input must be given or output expected, and limits the possible input actions.

In particular it is important for the correctness of Algorithm 1 to choose delays not exceeding the time bound within which the environment is required to offer an input (environment invariant condition may force inputs). Thus $\mathsf{Delays}(\mathcal{Z})$ must not contain delays exceeding forced inputs.

To cheaply compute a safe delay given a symbolic state-set $\mathcal{Z}$ we propose the following technique: Pick a random symbolic state $\langle \bar{\ell}, Z \rangle \in \mathcal{Z}$, compute its timed future as $Z' = (Z \wedge t = 0)^{\uparrow} \wedge I(\bar{\ell})$, and pick randomly $\delta \in [0, \mathsf{max}_t(Z'))$, where $\mathsf{max}_t(Z)$ extracts the maximum value of the auxillary clock t in Z. Note that this procedure will not compute the exact longest possible delay because it does not follow internal transitions (i.e the conjuncted invariant I may force an internal transition rather than an observable input). When the chosen delay has been performed, the state-set will be updated for the next iteration of the algorithm. Computing the exact delays is possible but would involve computing the more expensive $\mathsf{Closure}_{\delta\tau}(\mathcal{Z}, \infty)$.

Furthermore, it is desirable to compute intervals of time where input transitions are enabled for two reasons: 1) to optimize the algorithm to avoid too many superflous attempts to offer inputs (condition $\mathsf{EnvOutput}(\mathcal{Z}) \neq \emptyset$ in Algorithm 1), and 2) to guide the algorithm to cover the structure (transitions and locations) of the specification [22]. This optimization can be performed using the presented techniques, but we omit the details due to space limitations.

4 Experiments

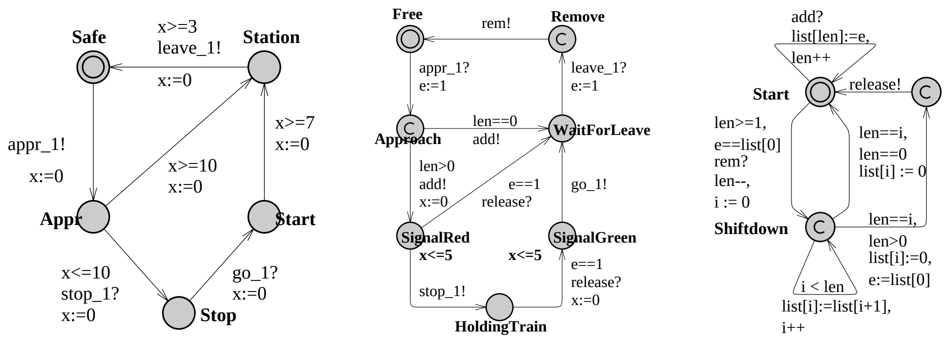
We implemented our algorithm by extending the mature UPPAAL model-checker tool to the testing tool T-UPPAAL. Besides a graphical timed automata editor, UPPAAL provides an efficient implementation of the needed basic symbolic operations. Unlike UPPAAL, T-UPPAAL does not store the reached state space, but only the current symbolic state set. We allow the full UPPAAL timed automata language, including non-deterministic (action and timing) specifications and discrete variables. The IUT is connected to T-UPPAAL via an adapter component translating abstract I/O actions into their real representation, and sends (receives) them to (from) the IUT.

This section presents the results of the first set of experiments using our implementation. The purpose is to give an indication of the feasibility of our technique in terms of applicability, error detection capability, and performance in terms of state-set size and computation time.

4.1 Test Specification

A rail-road intersection controller monitors trains on a set of rail-road tracks with a shared track segment, e.g. a train-station. Its main objective is to ensure that only one train occupies the shared segment at a time, and to grant access in arrival order. In this setup we assume 4 tracks, and for simplicity 1 train per track at a time. Trains on track i signal the controller when they approach and leave the station using signals appr_i and $\mathsf{leave}_\mathsf{i}$ respectively. When train i approaches an occupied station the controller is required to issue a stop_i within 5*mtu* (model time units), and is similarly required to issue go_i within 5*mtu* after the station becomes free.

The environment assumption model consists of 4 concurrent timed automata each modeling the assumed behavior of a train. The model for train 1 is shown in Figure 3(a); the remaining trains are identical except for the train-id. The model of the IUT requirements consists of 4 concurrent train control automata (Figure 3(b)) tracking the position of each potential train, and one queue automaton tracking their arrival order (Figure 3(c): list is an array of integers,



(a) Train 1 of 4. (b) Controller 1 of 4. (c) Queue

Fig. 3. Test specification for train controller: trains as environment, controller and queue as implementation.

and i is an index into the array). We use UPPAAL syntax to illustrate timed automata. Initial locations are marked using a double circle. Edges are by convention labeled by the triple: guard, action, and assignment in that order. The internal τ-action is indicated by an absent action label. Committed locations are indicated by a location with an encircled "C". A committed location must be left immediately as the next transition taken by the system. Finally, bold-faced clock conditions placed under locations are location invariants.

The complete test specification is a reasonably large and nontrivial first experiment: it consists of 9 concurrent timed automata, 8 clocks, and a sequential queue data structure.

4.2 Implementation Under Test

The IUT is implemented as an approximately 100 line C++ program following the basic structure of the specification. It uses POSIX Threads and POSIX locks and condition variables for multi-threading and synchronization. It consists of one thread per train, and queue data structure whose access is guarded by mutual exclusion and condition variables. In the experiment, the IUT runs in the same address space as the T-UPPAAL tool, and input and output actions are communicated to and from the driver/adapter via two single place bounded buffers.

In addition we have created a number of erroneous mutations based on the *assumed* correct implementation (**M0**):

M1: The stop_3 signal is issued 1*mtu* too late.
M2: The controller issues stop_1 instead of stop_3.
M3: The controller never issues stop_3
M4: The controller uses a bounded queue limited to 3 trains. Thus, the fourth train overwrites the third train in the queue.
M5: The controller uses LIFO queue instead of FIFO.
M6: The controller ignores appr_3 signals if a train arrives before 2*mtu* after entering the location Free.

4.3 Error Detection Capability

The experiments are run on a 8x900 MHZ Sun Sparc Fire v880R workstation with 32 GB memory running Sun Solaris 9 (SunOS 5.9). T-UPPAAL runs on one CPU whereas the IUT may

Table 4. Error detection and performance measures:

Mu-tant	Error detection capability						State-set size				Execution time, μs			
	Input actions			Duration, *mtu*			After(delay)		After(action)		After(delay)		After(action)	
	Min	Avg	Max	Min	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max
M1	2	4.8	16	0	68.8	318	2.3	18	2.7	28	1113	3128	141	787
M2	2	4.6	13	1	66.4	389	2.3	22	2.8	30	1118	3311	147	791
M3	2	4.7	14	0	66.4	398	2.2	22	2.7	30	1112	3392	141	834
M4	6	8.5	18	28	165.0	532	2.8	24	3.1	48	1113	3469	125	936
M5	4	5.6	12	14	89.8	364	2.8	24	3.3	48	1131	3222	146	919
M6	2	14.1	92	0	299.6	2077	2.7	27	2.9	36	1098	3531	110	861
M0	3565	3751.4	3966	10^5	10^5	10^5	2.7	31	2.9	46	1085	3591	101	950

run on one or more of the remaining. T-UPPAAL itself does not require these extreme amount of resources, and it runs well on a standard PC, but a multiprocessor allows T-UPPAAL and the IUT to run in parallel as they would normally do in a black-box system level test.

To allow for faster and more experiments and reduce potential problems with real-time clock synchronization between the engine and IUT, the experiments are run using a simulated clock progressing when both T-UPPAAL and the IUT needs to let time pass. Each mutant is tested 1100 times each with an upper time limit of 100000*mtu*. All runs of **M1-6** mutants failed and all runs of **M0** passed with timeout for testing. The minimum, maximum, and average running time and number of used input actions are summarized on the left side of Table 4.

The results show that all erroneous mutants are killed surprisingly quickly using less than 100 input actions and less than 2100*mtu*. In contrast the assumed correct implementation **M0** was not killed and was subjected to at least 3500 inputs stimuli and survived for more than 300 times longer than other mutants in average. In conclusion, the results indicate that online real-time testing may be a highly effective technique.

4.4 Performance

Based on the same setup from Section 4.3 we instrumented T-UPPAAL to record the number of symbolic states in the state-set, and the amount of CPU time used to compute the next state-set after a delay and an observable action. The right side of Table 4 summarizes the results. The state-set size is in average only 2-3 symbolic states per state-set, but it varies a lot, up to 48 states. In average, the state-set sizes reached after performing a delay appear larger than after an action. In average it costs only $1.1ms$ to compute the successor state-set after a delay, and less than $0.2ms$ after an action. Thus it seems feasible to generate tests from much larger specifications, obviously depending on the scale of time units.

In conclusion, the performance of our technique looks very promising and appears to be fast enough for many real-time systems. Obviously, more experiments on varying size and complexity models are needed to find the firm limitations of the technique.

5 Conclusions and Future Work

We have presented the T-UPPAAL tool and approach to testing of embedded systems using real-time online testing from non-deterministic timed automata specifications. Based on an experiment with a non-trivial specification we conclude that our notion of relativized input/output conformance and our sound and complete randomized online testing algorithm appear correct and feasible. We further conclude that our algorithm is implementable, and T-UPPAAL tool implementation shows encouraging results both in terms of error detection capability and performance of the symbolic state-set computation algorithm. However, further work and real-life applications are needed to evaluate the algorithm and the tool in detail.

Besides practical application, we plan to improve the tool in several directions. For instance, to estimate model coverage of the trace and use it to guide the random choices made by the algorithm and investigate their impact on the error detection capability. Also we plan to include observation uncertainty into our algorithm (i.e., outputs and given stimuli classified in an interval of time rather than a time instance), to improve clock synchronization between T-UPPAAL and the implementation, and a value passing mechanism to make tool easier to adopt.

Acknowledgments. We would like to thank anonymous reviewer for a valuable insight to our relativized timed input/output conformance relation.

References

1. T. Henzinger and X. Nicollin and J. Sifakis and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2):193–244, June 1994.
2. R. Alur and D.L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, April 1994.
3. G. Behrmann, J. Bengtsson, A. David, K.G. Larsen, P. Pettersson, and W. Yi. Uppaal implementation secrets. In *Formal Techniques in Real-Time and Fault-Tolerant Systems: 7th International Symposium, FTRTFT 2002*, pages 3–22, September 2002.
4. A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In 12^{th} *Int. Workshop on Testing of Communicating Systems*, pages 179–196, 1999.
5. 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.
6. E. Brinksma, K.G. Larsen, B. Nielsen, and J. Tretmans. Systematic Testing of Realtime Embedded Software Systems (STRESS), March 2002. Research proposal submitted and accepted by the Dutch Research Council.
7. R. Cardell-Oliver. Conformance Testing of Real-Time Systems with Timed Automata. *Formal Aspects of Computing*, 12(5):350–371, 2000.
8. R. Cleaveland and M. Hennessy. Testing Equivalence as a Bisimulation Equivalence. *Formal Aspects of Computing*, 5:1–20, 1993.
9. A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. 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.
10. A. Hessel, K.G. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-Optimal Test Cases for Real-Time Systems. In *3rd International Workshop on Formal approaches to Testing of Software (FATES 2003)*, Montréal, Québec, Canada, October 2003.
11. T. Higashino, A. Nakata, K. Taniguchi, and A R. Cavalli. Generating test cases for a timed i/o automaton model. In *IFIP Int'l Work. Test. Communicat. Syst. (IWTCS)*, pages 197–214, 1999.
12. H.S. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. In *Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 327–341. Springer-Verlag, 2002.

13. J. Ouaknine and J. Worrell. Revisiting digitization, robustness, and decidability for timed automata. In *18th IEEE Symposium on Logic in Computer Science (LICS 2003) Ottawa, Canada*, pages 198–207. IEEE Computer Society, june 2003.
14. T. Jéron and P. Morel. Test generation derived from model-checking. In N. Halbwachs and D. Peled, editors, *CAV'99, Trento, Italy*, volume 1633 of *LNCS*, pages 108–122. Springer-Verlag, July 1999.
15. A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *3rd International Workshop on Formal Approaches to Testing of Software (FATES'03). LNCS 2931*, Montreal, Canada, 2003.
16. K.G. Larsen. A Context Dependent Equivalence Between Processes. *Theoretical Computer Science*, 49:185–215, 1987.
17. M. Krichen and S. Tripakis. Black-box Conformance Testing for Real-Time Systems. In *Model Checking Software: 11th International SPIN Workshop*, volume LNCS 2989. Springer, april 2004.
18. D. Mandrioli, S. Morasca, and A. Morzenti. Generating Test Cases for Real-Time Systems from Logic Specifications. *ACM Transactions on Computer Systems*, 13(4):365–398, 1995.
19. M. Mikucionis, K.G. Larsen, and B. Nielsen. Online on-the-fly testing of real-time systems. Technical Report RS-03-49, Basic Research In Computer Science (BRICS), December 2003.
20. M. Mikucionis, B. Nielsen, and K.G. Larsen. Real-time system testing on-the-fly. In *the 15th Nordic Workshop on Programming Theory*, number 34 in B, pages 36–38, Turku, Finland, October 29–31 2003. Åbo Akademi, Department of Computer Science, Finland. Abstracts.
21. M. Mikucionis and E. Sasnauskaite. On-the-fly testing using UPPAAL. Master's thesis, Department of Computer Science, Aalborg University, Denmark, June 2003.
22. B. Nielsen and A. Skou. Automated Test Generation from Timed Automata. In *TACAS 2001 - Tools and Algorithms for the Construction and Analysis of Systems*, pages 343–357, April 2001.
23. J. Peleska, P. Amthor, S. Dick, O. Meyer, M. Siegel, and C. 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.
24. S. Tripakis. Fault Diagnosis for Timed Automata. In *Formal Techniques in Real-Time and Fault Tolerant Systems (FTRTFT'02)*, volume LNCS 2469. Springer, 2002.
25. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing Timed Automata. *Theoretical Computer Science*, 254(1-2):225–257, March 2001.
26. T.A. Henzinger and Z. Manna and A. Pnueli. What good are digital clocks? In Werner Kuich, editor, *Automata, Languages and Programming, 19th International Colloquium, ICALP92, Vienna, Austria*, volume 623 of *LNCS*, pages 545–558. Springer, july 1992.
27. J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999.
28. J. Tretmans and A. Belinfante. Automatic testing with formal methods. In *EuroSTAR'99: 7^{th} European Int. Conference on Software Testing, Analysis & Review*, Barcelona, Spain, November 8–12, 1999. EuroStar Conferences, Galway, Ireland.
29. V. Diekert, P. Gastin, A. Petit. Removing epsilon-Transitions in Timed Automata. In *14th Annual Symposium on Theoretical Aspects of Computer Science, STACS 1997*, pages 583–594, Lübeck, Germany, February 1997. LNCS, Vol. 1200, Springer.
30. R. de Vries, J. Tretmans, A. Belinfante, J. Feenstra, L. Feijs, S. Mauw, N. Goga, L. Heerink, and A. de Heer. Côte de resyste in PROGRESS. In STW Technology Foundation, editor, PROGRESS *2000 – Workshop on Embedded Systems*, pages 141–148, Utrecht, The Netherlands, October 2000.
31. R.G. de Vries and J. Tretmans. On-the-fly conformance testing using SPIN. *Software Tools for Technology Transfer*, 2(4):382–393, March 2000.