Verimag

Unité Mixte de Recherche 5104 CNRS - INPG - UJF
Centre Equation
2, avenue de VIGNATE
F-38610 GIERES
tel : +33 456 52 03 40
fax : +33 456 52 03 50
http://www-verimag.imag.fr

Real-time Testing with Timed Automata Testers and Coverage Criteria

Moez Krichen and Stavros Tripakis

Report n^o TR-2004-15

June 15, 2004

Reports are downloadable at the following address
http://www-verimag.imag.fr

UNIVERSITE JOSEPH FOURIER

CNRS

INPG

Real-time Testing with Timed Automata Testers and Coverage Criteria

Moez Krichen and Stavros Tripakis

June 15, 2004

Abstract

In previous work, we have proposed a framework for black-box conformance testing of real-time systems based on timed automata specifications and two types of tests: analog-clock or digital-clock. Our algorithm to generate analog-clock tests is based on an on-the-fly determinization of the specification automaton during the execution of the test, which in turn relies on reachability computations. The latter can sometimes be costly, thus problematic, since the tester must quickly react to the actions of the system under test. In this paper, we provide techniques which allow analog-clock testers to be represented as deterministic timed automata, thus minimizing the reaction time to a simple state jump. We also provide a method for (statically) generating a suite of digital-clock tests which covers the specification with respect to a number of criteria: location, edge or state coverage. This avoids having to generate too many tests, as can be evidenced on a small example.

Keywords: real-time systems, timed automata, conformance testing, black-box, partial observability, coverage

Reviewers:

Notes: Work partially supported by CNRS STIC project "CORTOS".

How to cite this report:

@techreport { ,
title = { Real-time Testing with Timed Automata Testers and Coverage Criteria},
authors = { Moez Krichen and Stavros Tripakis },
institution = { Verimag Technical Report },
number = {TR-2004-15},
year = { },
note = { }
}

1 Introduction

Testing is a fundamental step in any development process. It consists in applying a set of experiments to a system (*system under test – SUT*), with multiple aims, from checking correct functionality to measuring performance. In this paper, we are interested in so-called *black-box conformance testing*, where the aim is to check conformance of the SUT to a given specification. The SUT is a "black box" in the sense that we do not have a model of it, thus, can only rely on its observable input/output behavior.

Our work targets *real-time* systems, that is, systems which operate in an environment with strict timing constraints. Examples of such systems are many: embedded systems (e.g., automotive, avionic and robotic controllers, mobile phones), communication protocols, multimedia systems, and so on. When testing real-time systems, one must pay attention to two important facts. First, it is not sufficient to check whether the SUT produces the correct outputs; it must also be checked that the timing of the outputs is correct. Second, the timing of the inputs determines which outputs will be produced as well as the timing of these outputs.

Classical testing frameworks are based on Mealy machines (e.g., see [13, 23]) or finite labeled transition systems – LTSs (e.g., see [29, 11, 18, 3, 14]). These frameworks are not well-suited for real-time systems. In Mealy machines, inputs and outputs are synchronous, which is a reasonable assumption when modeling synchronous hardware, but not when outputs are produced with variable delays, governed by complex timing constraints. In testing methods based on LTSs, time is typically abstracted away and time-outs are modeled by special δ actions [28] which can be interpreted as "no output will be observed". This is problematic, because timeouts need to be instantiated with concrete values upon testing (e.g., "if nothing happens for 10 seconds, output FAIL"). However, there is no systematic way to derive the timeout values (indeed, durations are not expressed in the specification). Thus, one must rely on empirical, ad-hoc methods.

In previous work [22] we have proposed a testing framework for real-time systems based on specifications modeled as *timed automata – TA* [1] with inputs, outputs and unobservable actions. We have presented techniques for generating two types of tests: *analog-clock* tests which measure real-time precisely and *digital-clock* tests which can only count the ticks of a digital clock.

Our technique for generating analog-clock tests is based on an *on-the-fly determinization* of the specification automaton during the execution of the test. This technique, introduced in [30] for purposes of fault detection, is essential in order to avoid two problems. First, the fact that timed automata cannot always be determinized [1] and it is undecidable to check determinizability [31]. Second, the problem that analog-clock tests cannot be represented (statically) as finite trees. This is because the response delays of the SUT are unknown and in dense-time, which requires a tree of infinite branching.

On-the-fly testing avoids both problems above, by generating the test strategy during the execution of the test (when the response delays become known). The on-the-fly determinization algorithm is essentially a reachability computation on the specification automaton. This can be problematic: timed automata reachability can be costly; but the tester must quickly respond to the inputs it receives from the SUT since the two interact in real-time.

Digital-clock tests, on the other hand, *can* be represented statically as finite trees. A special input, tick, models the reaction of the tester to the ticks of its own clock. This reaction is simply "jumping" to a successor node in the tree, thus, fast. However, a new problem arises, namely, how many tests to generate and which ones. A simple approach is to generate all possible tests up to a given depth, defined by the user. However, this quickly leads to explosion, even for small examples, as shown in [22].

In this paper, we show how to improve the testing framework by providing solutions to the two problems discussed above. First, we provide techniques which allow analog-clock tests to be represented as deterministic timed automata. This results in minimizing the reaction time of the tester to a simple state jump. Since timed automata determinization is undecidable [31], we take a pragmatic approach. We suppose that the tester disposes of a single clock and that this clock is reset every time the tester receives an observable input.[1] Then, we provide techniques to compute the locations, edges, guards and deadlines of the tester automaton. Naturally, having only one clock implies that the tester will not be *complete* in general, i.e., it might accept behaviors of the SUT which should be rejected. However, we guarantee that the tester is *sound* (i.e., when it announces "FAIL", the SUT is indeed non-conforming). We can also show

[1] The technique can be extended to more than one clocks, assuming we fix the points where each clock is reset.

that the tester "does its best" given the information that it has, that is, the tester is in a sense the optimal one-clock tester (which resets its clock at every transition).

The second contribution of this paper is a method to statically generate a suite of digital-clock tests which covers the specification with respect to a number of criteria, namely, *location, edge or state coverage*. The benefits can be significant. For the example considered in [22], suites of less than ten tests suffice to achieve coverage, whereas exhaustive suites contain several thousands of tests even for small depths.

The rest of this paper is organized as follows. Section 2 reviews our testing framework, namely, the specification model, the conformance relation and the types of tests we consider. Section 3 presents our technique for generating timed automata testers. Section 4 presents our coverage technique. Section 5 discusses our tool and two case studies. Section 6 presents the conclusions and future work plans.

Related work

Most existing real-time testing frameworks [15, 12, 21, 24, 27, 25, 20] consider only restricted subclasses of the TA model. For instance, TA with *isolated* and *urgent* outputs in [27, 20] or event-recording automata in [24]. Our framework can fully handle non-deterministic and partially observable specifications. This is essential for ease of modeling, expressiveness and implementability. Specifications are often built in a *compositional* way, from many components. This greatly simplifies modeling.[2] In such cases, internal component interactions are typically unobservable to the external world, thus, also to the tester. Abstractions can also result in non-determinism and partial observability. The latter is also essential for implementability, since it may be hard, in practice, to export all events to the tester. Other differences of our work with other frameworks include the conformance relation used and the types of tests generated. For an extensive comparison the reader is referred to [22].

To our knowledge, there is no work on generating testers represented as timed automata. The closest in spirit is the work on timed controller synthesis reported in [9]. There, it is shown that the problem of synthesizing a timed automaton controller is decidable iff the *resources* of the controller are fixed, where the resources are the number of clocks, their granularity and the maximal constant that can be used in the guards of the controller. The decidability result relies on a region-graph construction and also uses the notion of symbolic alphabet, which essentially encodes all possible reset/guard combinations for the given resources. Our approach fixes the number of clocks, maximal constant and points where the clocks are reset. Our generation algorithm does not rely on the region graph but on symbolic reachability.

Regarding coverage, [20] provides techniques for generating tests covering edges, locations or definition-use pairs for clock variables of the specification. These techniques rely on the assumption that outputs are urgent and isolated. Thanks to this assumption, every input sequence results in a unique output sequence. This means that tests are *sequences* rather than trees. Thus, finding a test can be reduced to a standard reachability problem for timed automata.

2 The Testing Framework

We briefly present our testing framework. See [22] for more details and examples.

2.1 Timed Automata with Inputs, Outputs and Unobservable Actions

To model the specification, we use timed automata [1] with *deadlines* to capture urgency [26, 7], and input, output and unobservable actions, to capture inputs, outputs and internal actions of the SUT.

Let R be the set of non-negative reals. Given a finite set of *actions* Act, the set $(\mathsf{Act} \cup \mathsf{R})^*$ of all finite *real-time sequences* over Act will be denoted $\mathsf{RT}(\mathsf{Act})$. $\epsilon \in \mathsf{RT}(\mathsf{Act})$ is the empty sequence. Given $\mathsf{Act}' \subseteq \mathsf{Act}$ and $\rho \in \mathsf{RT}(\mathsf{Act})$, $P_{\mathsf{Act}'}(\rho)$ denotes the *projection* of ρ to Act', obtained by "erasing" from ρ all actions not in Act'. For example, if $\mathsf{Act} = \{a, b\}$, $\mathsf{Act}' = \{a\}$ and $\rho = a\,1\,b\,2\,a\,3$, then $P_{\mathsf{Act}'}(\rho) = a\,3\,a\,3$. The time spent in a sequence ρ, denoted $\mathsf{time}(\rho)$ is the sum of all delays in ρ, for example, $\mathsf{time}(\epsilon) = 0$ and $\mathsf{time}(a\,1\,b\,0.5) = 1.5$.

[2] Notice that a compositional specification does not require that the SUT be implemented following the same structure. Composition is merely a way of modeling the specification.

A *timed automaton over* Act is a tuple $(Q, q_0, X, \mathsf{Act}, \mathsf{E})$ where Q is a finite set of *locations*; $q_0 \in Q$ is the initial location; X is a finite set of *clocks*; E is a finite set of *edges*. Each edge is a tuple (q, q', ψ, r, d, a), where $q, q' \in Q$ are the source and destination locations; ψ is the *guard*, a conjunction of constraints of the form $x\#c$, where $x \in X$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$; $r \subseteq X$ is the set of clocks to be *reset*; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*; and $a \in \mathsf{Act}$ is the action. We will not allow eager edges with guards of the form $x > c$.

A TA A defines an infinite labeled transition system (LTS). Its states are pairs $s = (q, v) \in Q \times \mathsf{R}^X$, where $q \in Q$ is a location and $v : X \to \mathsf{R}$ is a clock *valuation*. Given state $s = (q, v)$ and clock x, we write $x(s)$ to denote the value of x at s, i.e., $v(x)$. $\vec{0}$ is the valuation assigning 0 to every clock of A. S_A is the set of all states and $s_0^A = (q_0, \vec{0})$ is the initial state. There are two types of transitions, discrete and timed. Discrete transitions are of the form $s = (q, v) \xrightarrow{a} s' = (q', v')$, where $a \in \mathsf{Act}$ and there is an edge $e = (q, q', \psi, r, d, a)$, such that v satisfies ψ and v' is obtained by resetting to zero all clocks in r and leaving the others unchanged. We say that e is enabled at s and write $s \models e$ (or $s \models \psi$). Timed transitions are of the form $(q, v) \xrightarrow{t} (q, v + t)$, where $t \in \mathsf{R}, t > 0$ and there is no edge (q, q'', ψ, r, d, a), such that: either $d = \mathsf{delayable}$ and there exist $0 \leq t_1 < t_2 \leq t$ such that $v + t_1 \models \psi$ and $v + t_2 \not\models \psi$; or $d = \mathsf{eager}$ and there exists $0 \leq t_1 < t$ such that $v + t_1 \models \psi$. We use notation such as $s \xrightarrow{a}$, $s \not\xrightarrow{a}$, ..., to denote that there exists s' such that $s \xrightarrow{a} s'$, there is no such s', and so on. This notation naturally extends to timed sequences. For example, $s \xrightarrow{a1b} s'$ if there exist s_1, s_2 such that $s \xrightarrow{a} s_1 \xrightarrow{1} s_2 \xrightarrow{b} s'$. A state $s \in S_A$ is *reachable* if there exists $\rho \in \mathsf{RT}(\mathsf{Act})$ such that $s_0^A \xrightarrow{\rho} s$. The set of reachable states of A is denoted $\mathsf{Reach}(A)$.

In the rest of the paper, we assume given a set of actions Act, partitioned in two disjoint sets: a set of *input actions* $\mathsf{Act_{in}}$ and a set of *output actions* $\mathsf{Act_{out}}$. We also assume there is an *unobservable action* $\tau \notin \mathsf{Act}$. Let $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$.

A *timed automaton with inputs and outputs* (TAIO) is a timed automaton over Act_τ. A TAIO is called *observable* if none of its edges is labeled by τ. A TAIO A is called *input-complete* if it can accept any input at any state: $\forall s \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Act_{in}} \,.\, s \xrightarrow{a}$. It is called *deterministic* if $\forall s, s', s'' \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Act}_\tau \,.\, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Rightarrow s' = s''$. It is called *non-blocking* if

$$\forall s \in \mathsf{Reach}(A) \,.\, \forall t \in \mathsf{R} \,.\, \exists \rho \in \mathsf{RT}(\mathsf{Act_{out}} \cup \{\tau\}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho} . \quad (1)$$

The non-blocking property states that at any state, A can let time pass forever, even if it does not receive any input. This is a sanity property which ensures that a TAIO does not "force" its environment to provide an input by blocking time.

The set of *observable timed traces* of A is defined to be

$$\mathsf{Traces}(A) = \{P_{\mathsf{Act}}(\rho) \mid \rho \in \mathsf{RT}(\mathsf{Act}_\tau) \wedge s_0^A \xrightarrow{\rho}\}. \quad (2)$$

Finally, given a set of states $S \subseteq S_A$ and $a \in \mathsf{Act}$, we define the following operator:

$$\mathsf{succ}(S, a) = \{s' \in S_A \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, s \xrightarrow{a \cdot \rho} s'\} \quad (3)$$

$\mathsf{succ}(S, a)$ contains all states that can be reached from some state in S by performing a followed by an unobservable sequence ρ.

2.2 Specifications, Implementations and Conformance

We assume that the specification of the system to be tested is given as a non-blocking TAIO A_S. We assume that the SUT, also called *implementation*, can be modeled as a non-blocking, input-complete TAIO A_I. Notice that we do not assume that A_I is known, simply that it exists. The assumption of A_S and A_I being non-blocking is natural, since in reality time cannot be blocked. The assumption of A_I being input-complete is also reasonable, since a system usually accepts all inputs at any time, possibly ignoring them or issuing an error message when the input is not valid. Notice that we do not assume, as is often done, that the specification A_S is input-complete. This is because A_S needs to be able to model assumptions on the environment, i.e., restrictions on the inputs. For instance, a guard $x \leq 2$ on an edge labeled with input a is interpreted as "*if* a is received while $x \leq 2$ *then* it must be guaranteed that ...".

In order to formally define the conformance relation, we introduce a number of operators. In the definitions that follow, A is a TAIO, $\sigma \in \mathsf{RT}(\mathsf{Act})$, s is a state of A and S is a set of states of A.

$$\sigma(A) = \{s \in S_A \mid \exists \rho \in \mathsf{RT}(\mathsf{Act}_\tau) \,.\, s_0^A \xrightarrow{\rho} s \,\wedge\, P_{\mathsf{Act}}(\rho) = \sigma\}$$
$$\mathsf{elapse}(s) = \{t > 0 \mid \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}\}$$
$$\mathsf{out}(s) = \{a \in \mathsf{Act_{out}} \mid s \xrightarrow{a}\} \cup \mathsf{elapse}(s)$$
$$\mathsf{out}(S) = \bigcup_{s \in S} \mathsf{out}(s).$$

$\sigma(A)$ is the set of all states of A that can be reached by some timed sequence ρ whose projection to observable actions is σ. $\mathsf{elapse}(s)$ is the set of all delays which can elapse from s without A making any observable action. $\mathsf{out}(s)$ is the set of all observable "events" (outputs or delays) that can occur when the system is at state s.

The *timed input-output conformance relation*, denoted tioco, requires that after any observable sequence specified in A_S, every possible observable output of A_I (including delays) is also a possible output of A_S. Notice that this requirement only refers to outputs, thus, the fact that A_I accepts generally "more inputs" than A_S does not pose problems of non-conformance: it simply means that the implementation is required to be conforming only with respect to the input assumptions given in the specification. tioco is inspired from its "untimed" counterpart, ioco [28]. The key idea is that delays are considered to be observable events, along with output actions. Formally, A_I conforms to A_S, denoted A_I tioco A_S, if

$$\forall \sigma \in \mathsf{Traces}(A_S) \,.\, \mathsf{out}(\sigma(A_I)) \subseteq \mathsf{out}(\sigma(A_S)). \tag{4}$$

Due to the fact that implementations are assumed to be input-complete, it can be easily shown that tioco is a transitive relation, that is, if A tioco B and B tioco C then A tioco C. It can be also shown that checking tioco is undecidable. This is not a problem for black-box testing: since A_I is unknown, we cannot check conformance directly, anyway.

tioco permits to express most useful types of requirements for real-time systems, such as the requirements that an output must be generated neither too late nor too early. It can also capture "observable deadlocks", that is, situations where no output is generated for a "long" time.[3] Finally, it can capture assumptions on the environment. For examples illustrating these features of tioco, see [22].

2.3 Analog-clock and Digital-clock Tests

A test (or *test case*) is an experiment performed on the implementation by an agent (the *tester*). There are different types of tests, depending on the capabilities of the tester to observe and react to events. Here, we consider two types of tests (the terminology is borrowed from [19]). *Analog-clock* tests can measure precisely the real-time delay between two observed actions. *Digital-clock* tests can only count how many "ticks" of a finite-granularity clock have occurred between two actions. For simplicity, we assume that the tester and the implementation are started precisely at the same time. In practice, this can be achieved by having the tester issuing the start command to the implementation.

It should be noted that we consider *adaptive* tests (following the terminology of [23]), where the action the tester takes depends on the observation history. Adaptive tests can be seen as *trees* representing the strategy of the tester in a game against the implementation. Due to restrictions in the specification model, which essentially remove non-determinism from the implementation strategy, some existing methods [27, 20] generate non-adaptive test *sequences*.

An analog-clock test can be defined as a total function $T : \mathsf{RT}(\mathsf{Act}) \to \mathsf{Act_{in}} \cup \{\bot, \mathsf{pass}, \mathsf{fail}\}$ specifying the action the tester must take given its current observation (if $a \in \mathsf{Act_{in}}$ then the tester emits a; if $\bot$ then the tester waits; if $\{\mathsf{pass}, \mathsf{fail}\}$ then the tester produces a verdict and stops). For the purpose of this paper, which is to represent analog-clock testers as timed automata, it makes more sense to define an analog-clock test directly as a TAIO T. T has as inputs (resp. outputs) the outputs (resp. inputs) of the specification A_S. T is observable (i.e., has no τ actions), deterministic and non-blocking. Locations of T are marked

[3] The requirement "*output b must be emitted* **sometime** *after input a is received*" cannot be expressed by tioco. However, this requirement is hardly testable: if we do not have an upper bound on the time that it takes to emit b, how can we check conformance within a finite amount of time?

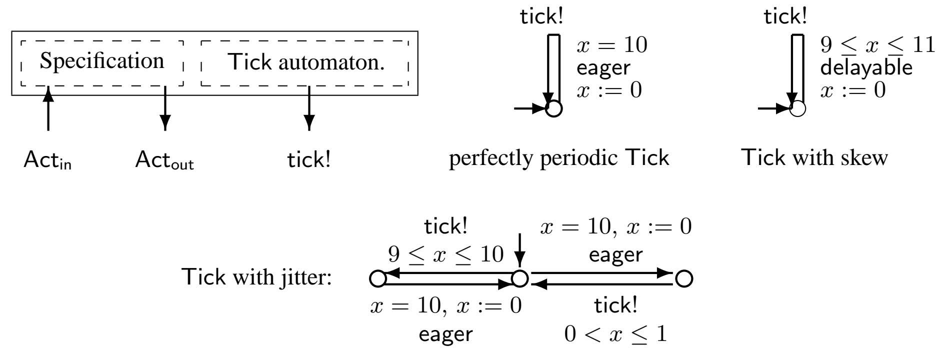

Figure 1: Extending the specification with a tester clock model and possible such models.

are either "input" or "output". In an input location, the tester waits for an input from the SUT (i.e., some $a \in \mathsf{Act_{out}}$). In an output location, the tester emits an output (i.e., some $b \in \mathsf{Act_{in}}$). Input locations of T are input-complete with respect to $\mathsf{Act_{out}}$, that is, for each input location q, for any $a \in \mathsf{Act_{out}}$ and any state $s = (q, v)$, $s \xrightarrow{a}$. T must also satisfy the *urgent* and *isolated* output condition of [27] with respect to $\mathsf{Act_{in}}$. This means that if $s \xrightarrow{a}$ for some state s and $a \in \mathsf{Act_{in}}$ then (a) time cannot elapse at s and (b) there is no other $b \in \mathsf{Act_{in}}$ such that $s \xrightarrow{b}$. Thus, there is no ambiguity as to which output must be emitted and when. The states of T will be partitioned into *accepting* and *rejecting*, corresponding to "PASS" and "FAIL" verdicts, respectively.

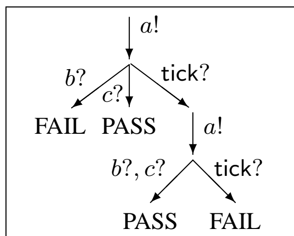

Figure 2: A digital-clock test represented as a finite tree.

A digital-clock test can also be defined as a function $D : (\mathsf{Act} \cup \{\mathsf{tick}\})^* \rightarrow \mathsf{Act_{in}} \cup \{\bot, \mathsf{pass}, \mathsf{fail}\}$, where tick is a new output action, not in Act_τ, modeling the tick of the tester's clock. In fact, the clock of the tester can be modeled directly by extending the specification automaton with a Tick automaton, as shown in Figure 1 (we use notation ! for outputs and ? for inputs). The Tick automaton models the digital clock of the tester. Different Tick automata can be used, depending on whether we want to model a clock which is assumed to be perfectly periodic, or a clock with skew, and so on. At the end, we obtain an extended specification model, denoted A_S^{tick}, and the objective becomes to generate an *untimed* test for A_S^{tick}. The test is untimed because it only has access to discrete observable actions, namely, all actions in the set $\mathsf{Act} \cup \{\mathsf{tick}\}$. The test has no direct access to time, it merely observes tick actions. Such a test can be represented as a finite tree, like the one shown in Figure 2. Nodes in this tree are marked either "input" or "output". In an input node, the tester waits for an input or the next tick of its clock. In an output node, the tester emits an output. Leaves are labeled PASS or FAIL.

3 Generating Timed Automata Testers

The problem of generating an analog-clock test represented as a TAIO can be anything from trivial to undecidable, depending on its precise definition. If we require a test which is only sound, then the problem is trivial, because a test always announcing PASS is sound. On the other hand, if we require a test which is also complete, then we face two problems: (a) such a test may not exist because the specification is non-deterministic whereas the test has to be deterministic, but TA are not determinizable; (b) checking and producing a deterministic test when it exists can be shown to be an undecidable problem [31].

To avoid these difficulties, we take a pragmatic approach. We suppose that the tester has only one clock which is reset every time the tester observes an action, that is, at any edge of the tester TAIO. We then provide techniques to compute the locations and edges of the tester automaton and the guards and deadlines of the edges.

It should be noted that the above technique can be easily extended to generate testers with more than one clock, provided the *skeleton* of the tester is given. The skeleton is a deterministic finite automaton the transitions of which are labeled with resets of the clocks of the tester. This information is necessary since, for a given number of clocks (even for one clock) there exist many possible testers which differ in their logic of resetting clocks. A special case is an *event-clock* tester which has one clock for each observable action, reset when this action occurs, as in *event-clock automata* [2].

3.1 "One-clock Determinization" of Timed Automata

For pedagogical reasons, we first explain our technique for plain timed automata, which can be seen as TAIO with an empty set of input actions. For such an automaton A, the technique amounts to determinizing A "as best as possible", given that we can only use one clock. Formally, the deterministic counterpart of A, denoted A_{mon}, will accept a superset of $\mathsf{Traces}(A)$. Notice that A may contain unobservable actions and non-determinism. Viewing A as the specification, A_{mon} is a *monitor* for A.

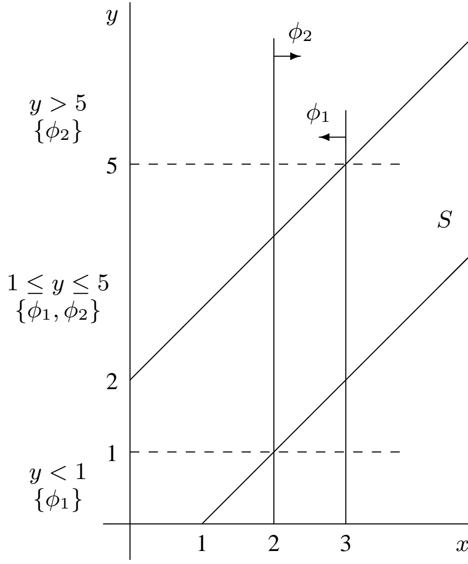

Figure 3: Illustration of the $\sim_S^a$ equivalence.

A_{mon} is a TAIO which has as inputs the outputs of A. A_{mon} is observable, deterministic and input-complete. All its locations are input locations. A_{mon} uses a single clock, y, which is reset to zero every time an action is observed. A_{mon} tries to estimate the state of A based on its current observation (including the value of its own clock y). A_{mon} has no urgency constraints: all its deadlines are lazy, thus, A_{mon} is non-blocking. A_{mon} needs no urgency because it acts as an "acceptor" rather than a "generator" of traces. On the other hand, the states of A_{mon} (including locations and values of the clock y) are divided into accepting and rejecting.

Let $A = (Q, q_0, X, \mathsf{Act}, \mathsf{E})$ and suppose y is a new clock, not in X. Let S_A^y be the set of states of A extended with the clock y, that is, $S_A^y = Q \times \mathsf{R}^{X \cup \{y\}}$. For an action $a \in \mathsf{Act}$, let $\mathsf{E}_a \subseteq \mathsf{E}$ be the set of edges of A which are labeled with a. For a given set of extended states $S \subseteq S_A^y$ and a value $u \in \mathsf{R}$ of clock y, we define the set of edges:

$$\mathsf{E}_a(S, u) = \{e \in \mathsf{E}_a \mid \exists s \in S \,.\, y(s) = u \wedge s \models e\}. \quad (5)$$

$\mathsf{E}_a(S, u)$ contains all edges labeled a which are satisfied by a state in S where y equals u. Finally, we define the following equivalence on values $u_1, u_2 \in \mathsf{R}$ of the clock y:

$$u_1 \sim_S^a u_2 \quad \text{iff} \quad \mathsf{E}_a(S, u_1) = \mathsf{E}_a(S, u_2). \quad (6)$$

The intuition is as follows. Two values of y are equivalent if they give the same information on the enabledness of an edge labeled with a, assuming S holds. S captures the current "knowledge" of the monitor. In particular, it captures the relation between values of y and possible states where A can be in.

Let us illustrate the meaning of $\sim_S^a$ with the example shown in Figure 3. We assume that

$$S = (q, -2 \leq x - y \leq 1)$$

and that q has two outgoing edges e_1 and e_2 labeled a, with guards $\phi_1 \equiv x \leq 3$ and $\phi_2 \equiv x \geq 2$, respectively. Then, $\sim_S^a$ induces three equivalence classes, namely, $y < 1$, $1 \leq y \leq 5$ and $y > 5$. Indeed, given the assumption $-2 \leq x - y \leq 1$, $y < 1$ implies $x < 2$. Thus, when $y < 1$ we know that ϕ_2 does not hold, therefore, e_2 is not enabled. Similarly, when $y > 5$ we know that e_1 is not enabled. When $1 \leq y \leq 5$, both e_1 and e_2 may be enabled. It is important to note that *not all* states in S for which $1 \leq y \leq 5$ satisfy ϕ_1, and similarly for ϕ_2. However, given our information on y, we cannot be sure. Thus, we need to include both e_1 and e_2 in the set of possible enabled edges, given the constraint $1 \leq y \leq 5$.

We now explain the construction of the monitor automaton A_{mon}. A location of A_{mon} is associated with a set of extended states of A, $S \subseteq S_A^y$. For each action a, for each equivalence class ψ in the (coarsest)

partition induced by $\sim_S^a$, A_{mon} has an edge $e = (S, S', \psi, \{y\}, \mathsf{lazy}, a)$, where the destination location S' is computed as follows:

$$S' = \mathsf{succ}(S \cap \psi, a) \tag{7}$$

where $S \cap \psi$ denotes the set of all states $s \in S$ such that $y(s) \models \psi$. Notice that S' can be empty, even when S is non-empty. This is because ψ may be unsatisfied in S. Also note that S' is the "best" possible estimate, in the sense that S' is the smallest set possible, given the knowledge the monitor has when a arrives. This knowledge is captured by $S \cap \psi$. Indeed, the monitor knows that A cannot be in a state outside S. It also knows that clock y satisfies ψ, which further restricts the possible states A can be in.

Let A^y be the automaton A extended with clock y and recall that $s_0^{A^y}$ denotes the initial state of A^y. Then, the initial location of A_{mon} is defined to be $S_0 = \{s \in S_{A^y} \mid \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, s_0^{A^y} \xrightarrow{\rho} s\}$. S_0 captures the initial knowledge of the monitor. The latter knows that initially y and all clocks of A equal zero. However, S_0 must also include all states that A can move to by performing unobservable sequences.

It remains to define the accepting and rejecting states of A_{mon}. Given $S \subseteq S_A^y$, let $S_{/y}$ be the projection of S on clock y, that is, $S_{/y} = \{u \in \mathsf{R} \mid \exists s \in S \,.\, y(s) = u\}$. Then, all states $(S, S_{/y})$ of A_{mon} are accepting, provided $S \neq \emptyset$. The rest of the states are rejecting. The above algorithm is essentially a *subset construction* for A, with the addition that clock y is used to infer knowledge about states that A can possibly be in. The construction relies on repeating two basic steps: (a) computing the partition induced by equivalences $\sim_S^a$, and (b) computing successor locations S' using reachability. We show how step (a) can be implemented below. As for step (b), standard symbolic reachability techniques, coupled with so-called *extrapolation abstractions* can be used to ensure that the number of possible locations of A_{mon} remains finite [16, 4, 8].

In such abstractions, the maximal constants compared with each clock play an essential role. These constants are known in the case of the clocks of A but must be specified by the user for the monitor clock y. Indeed, increasing the maximal constant for y amounts to increasing the observational power of the monitor. In fact, there are cases where there is no optimal monitor: the greater the maximal constant allowed for y is, the more precise A_{mon} will be, in the sense of how "close" the language of A_{mon} is to the language of A. An example is shown in Figure 4. The TAIO shown in the figure can produce a single output a at any time k, where $k \in \{1, 2, ...\}$. It can be seen that for any such k, a monitor able to compare y to constants up to k is "less accurate" than a monitor able to compare y to constants up to $k+1$. Indeed, the former cannot distinguish between $a!$ happening precisely at time k or at time strictly greater than k, while the latter can.

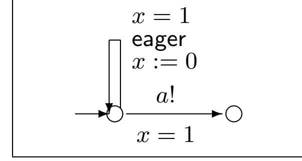

Figure 4: A TAIO which can produce $a!$ at times 1, 2, 3, ...

A simple algorithm for computing the coarsest partition induced by $\sim_S^a$ is the following. Given a constraint ψ on clock y, let $\mathsf{E}_a^{S,\psi} = \{e \in \mathsf{E}_a \mid S \cap (\psi \wedge \mathsf{guard}(e)) \neq \emptyset\}$, where $\mathsf{guard}(e)$ is the guard of edge e. $\mathsf{E}_a^{S,\psi}$ contains all edges labeled a whose guards may be satisfied by a state in S where y lies in the interval ψ. In other words, $\mathsf{E}_a^{S,\psi}$ is the union of $\mathsf{E}_a(S, u)$ over all values u satisfying ψ. Now, let K be the greatest constant appearing in a constraint defining S or a guard of an edge in E_a. For each ψ in the set of intervals $\{[0,0], (0,1), [1,1], (1,2), ..., [K,K], (K,\infty)\}$, compute $\mathsf{E}_a^{S,\psi}$. For this, the condition $S \cap (\psi \wedge \mathsf{guard}(e)) \neq \emptyset$ needs to be checked. This can be done symbolically, using standard techniques and data structures such as DBMs [6, 17]. Once $\mathsf{E}_a^{S,\psi}$ is computed for all intervals ψ, the coarsest partition is obtained by "merging" (i.e., taking the union of) intervals having the same set $\mathsf{E}_a^{S,\psi}$. For the example of Figure 3, $\mathsf{E}_a^{S,y<1} = \{e_1\}$, $\mathsf{E}_a^{S,1\leq y\leq 5} = \{e_1, e_2\}$ and $\mathsf{E}_a^{S,y>5} = \{e_2\}$. Notice that the correctness of the above algorithm relies on the fact that all values in an interval $(i, i+1)$ are equivalent, and the same is true for the interval (K, ∞). This is because constraints only have integer constants.

Let us give an example illustrating the construction of A_{mon}. Consider the non-deterministic timed automaton shown in Figure 5. All its edges are lazy, except the one from location 2 to location 4, which is $\mathsf{delayable}$. Its one-clock monitor automaton is shown in Figure 6. Not all locations and edges of the monitor are shown, in order not to overload the figure. In particular, the empty location and all edges leading to it are not shown. For instance, there is an edge labeled a with guard $y > 5$ from the initial location to the empty location, since a is not accepted if it arrives after 5 time units from start. All states of the monitor are accepting, except from the empty location and the states of location $S = (2, x = y \leq 2)$

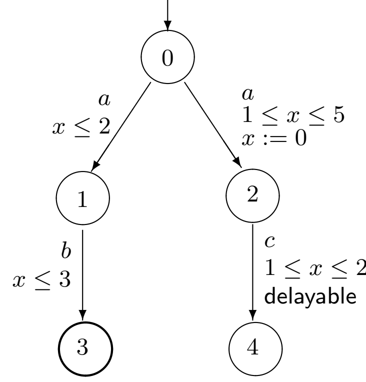

Figure 5: A non-deterministic timed automaton.

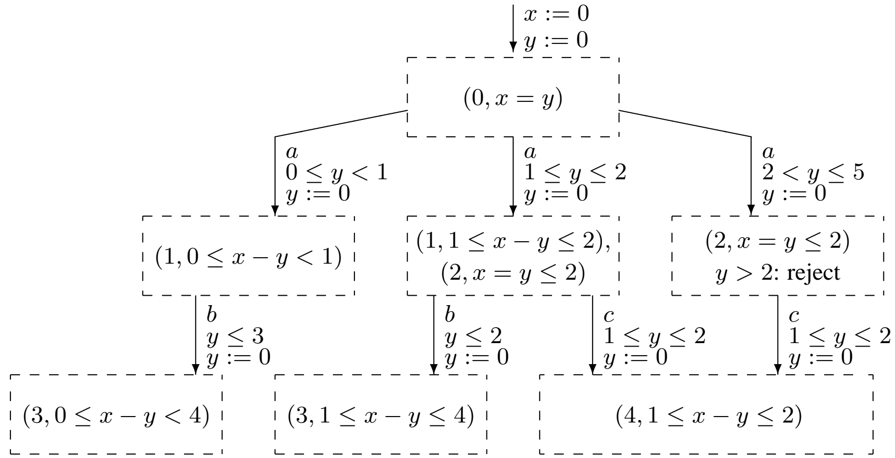

Figure 6: The one-clock deterministic monitor of the automaton of Figure 5.

where $y > 2$ (notice that $S_{/y}$ is the constraint $y \le 2$). This is because c must be received at most 2 time units after a, in order to be accepted. Note that there are no such rejecting states at location $S' = (1, 1 \le x - y \le 2) \cup (2, x = y \le 2)$. This is because the monitor does not know whether the original automaton is at location 1 or 2, and there is no urgency at location 1. Indeed, $S'_{/y}$ is the constraint $true$.

3.2 From Monitors to Testers

We now consider the general case of TAIO with both input and output actions. In this case, the monitor becomes a tester, since it must supply inputs to the SUT. Formally, the tester is an analog-clock test TAIO, denoted A_{test}, as defined in Section 2.3.

The algorithm for constructing A_{test} is a generalization of the algorithm for building A_{mon}. As with A_{mon}, each location of A_{test} is a set $S \subseteq S_A^y$. The choice of marking a location as input or output is made by the algorithm non-deterministically. For locations marked as input, their outgoing edges are computed as shown in the previous section, using the equivalence $\sim_S^a$, where, in this case, $a \in \mathsf{Act}_{\mathsf{out}}$. In order to mark a location S as output, there must exist $a \in \mathsf{Act}_{\mathsf{in}}$ and $u \in \mathsf{R}$ such that

$$\forall s \in S \,.\, y(s) = u \Rightarrow s \xrightarrow{a} .$$

This condition guarantees that when $y = u$ then a is a relevant input, that is, it satisfies the environment assumptions given in the specification. If S is indeed marked as output, then the edge $(S, S', y = u, \mathsf{eager}, a)$

is added to A_{test}, where S' is computed as shown in the previous section. Notice that the deadline of the edge is eager: this is because we want the output to be emitted at a precise point in time, otherwise the tester is not time-deterministic. To find a u satisfying the condition above, we first compute the constraint $P_{S,a}(y)$ on clock y:

$$P_{S,a}(y) \equiv \forall q, \forall x \,.\, (q,x,y) \in S \Rightarrow \exists e \in \mathsf{E}_a \,.\, (q,x) \models \mathsf{guard}(e)$$

where q and x are variables denoting the location and clocks of A, respectively (we slightly abuse notation and write $\forall x$ instead of $\forall v \in \mathsf{R}^X$). $P_{S,a}(y)$ can be computed symbolically by using quantifier elimination to eliminate variables q and x. $P_{S,a}(y)$ is a linear constraint on y. Thus, we can check satisfiability in a constructive way and find the value u we seek. If we cannot find an integer value u, then we pick a rational value and multiply at the end of the construction all constants in the automaton with a sufficiently large constant to make them integer.

The states of A_{test} are defined to be either accepting or rejecting, as with A_{mon}. Rejecting states correspond to the tester emitting a "FAIL" verdict. On the other hand, there is no specific point in time where the tester emits a "PASS". Indeed, the execution of the test can go on as long as the tester remains in an accepting state. The user can stop the test when he/she is tired of waiting. Coverage criteria, similar to the ones discussed in Section 4 can also be considered, since each location of A_{test} essentially covers a set of states of A. The difference is that in this case the book-keeping of coverage must be performed on-the-fly, that is, during execution of the test. Also, since tester outputs are urgent, more than one tests will generally be necessary to achieve coverage. Studying such coverage methods is part of our ongoing work.

4 Generating Digital-clock Tests with respect to Coverage Criteria

In [22] we have given an algorithm to generate a digital-clock test using symbolic techniques similar to the ones presented in the previous section. The algorithm takes as input an extended specification model A_S^{tick} and generates a test tree like the one shown in Figure 2. Nodes of the tree correspond to sets of states of A_S^{tick}. Nodes are marked input or output non-deterministically, as when generating timed automata testers. In order for a node S to be marked output, there must exist an action $a \in \mathsf{Act}_{\mathsf{in}}$ such that $S' = \mathsf{succ}(S,a) \neq \emptyset$. In this case, the algorithm chooses such an action and generates an edge $S \xrightarrow{a} S'$. For each input node S and every action $b \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$, the algorithm generates an edge $S \xrightarrow{b} S''$ in the test tree, with $S'' = \mathsf{succ}(S,b)$. If $S'' = \emptyset$, then S'' is marked "FAIL". Otherwise, the algorithm continues to extend the test from S''.

The above algorithm is only partially specified. It must be completed by specifying a policy for marking nodes as input or output, for choosing which of the possible outputs to emit and for choosing when to stop the test. An easy way is to resolve these choices randomly. This may not be satisfactory when some completeness guarantees are required or when repetitions must be avoided as much as possible. Another possibility is to generate an *exhaustive* test suite up to a depth k specified by the user. This approach suffers from the *explosion* problem, since the number of tests is generally exponential in k.

To remedy the above problems, many approaches have been proposed for generating test suites with respect to a given *coverage criterion*. Different coverage criteria have been proposed for software, such as statement coverage, branch coverage, and so on [32]. In the TA case existing methods attempt to cover either finite abstractions of the state space (e.g., the region graph [27] or a time-abstracting quotient graph [24]) or structural elements of the specification such as edges or locations [20].

Here, we propose a new technique for covering states, locations or edges of the specification.[4] Our technique relies on the concept of *observable graph* of the composed automaton A_S^{tick}, denoted OG. This graph is generated as follows. The initial node of the graph is $S_0 = \{s \mid \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, s_0^{A_S^{\mathsf{tick}}} \xrightarrow{\rho} s\}$. For each generated node S and each $a \in \mathsf{Act} \cup \{\mathsf{tick}\}$, a successor node $S' = \mathsf{succ}(S,a)$ is generated and an edge $S \xrightarrow{a} S'$ is added to the graph. Extrapolation abstractions can be used here as well, to ensure that the graph remains finite.

[4] As mentioned in the introduction, we cannot use the technique of [20] because it relies on the assumption that outputs in the specification are urgent and isolated.

Every node of OG corresponds to a set of states S of A_S^{tick}. We say that the node *covers* S. On the other hand, every static test tree is essentially a sub-graph of OG. We say that such a test covers the union of all sets of states covered by its nodes. We say that a set of tests (or *test suite*) achieves *state coverage* if every reachable state of A_S is covered by some test in the suite.[5]

Similarly, a node S of OG covers a location q of A_S if S contains some state $s = (q, v)$. A test suite achieves *location coverage* if every reachable location of A_S is covered by some test in the suite. When A_S is built compositionally, we can distinguish between *global* and *local* location coverage. In global location coverage, we require that all reachable global locations be covered. A global location is a vector $(q_1, ..., q_n)$ where n is the number of components and q_i is the local location of component i. In local location coverage, we simply require that all reachable individual locations of components be covered. Clearly, a test suite achieving global location coverage also achieves local location coverage, but the converse is not generally true. Similarly, a test suite achieving state coverage also achieves both local and global location coverage, but the converse is not always true.

Every edge of OG can be associated to a set of edges of A_S. In particular, an edge $S \xrightarrow{a} S'$ will be associated to all edges which are visited during the reachability algorithm which computes S' from S. Formally, if $s \in S$, $s' \in S'$ and $s \xrightarrow{\rho \cdot a} s'$ for an unobservable sequence ρ, all edges in the path from s to s' are covered by the edge $S \xrightarrow{a} S'$. We say that a test suite achieves *edge coverage* if every reachable edge of A_S (i.e., an edge enabled at a reachable state of A_S) is covered by some test in the suite. A test suite achieving edge coverage also achieves local location coverage. However, it may not achieve global location (or state) coverage.

We now give an algorithm to generate a test suite achieving coverage with respect to a given criterion. The first step is to build the observation graph of A_S^{tick}. Then, tests are extracted statically from OG, until coverage is achieved. We first consider location coverage. Tests are extracted as follows.

While there are reachable locations not covered, the algorithm picks such a location, say q. Next, it picks a node v of OG associated with q (such a node exists since q is reachable) and finds a path in OG from the initial node to v. Then, it extends this path into a test tree. This can be done by completing the path with the missing edges, labeled with tester inputs. For instance, if there is an edge $v_1 \xrightarrow{a} v_2$ in the path, with $a \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$, then every outgoing edge of v_1 labeled with a tester input b, i.e., every edge $v_1 \xrightarrow{b} v'$, $b \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$, must be added.[6] The leaves of the tree are labeled PASS, except if a leaf is empty, in which case it is labeled FAIL. This new test is added to the set of tests already generated and the algorithm repeats choosing a new uncovered location, until all locations are covered. Notice that the algorithm is essentially an AND/OR search in a finite graph, AND nodes being input nodes and OR nodes being output nodes.

A state-covering suite can be extracted in a similar way. If some state s is not covered, we first find a node v of OG covering s. Then we extract a test including v as above. Notice that this test will cover not only s, but a set of states containing s. It will at least cover the region in which s belongs. This guarantees that the algorithm terminates with a finite test suite, even though the set of states is infinite. The algorithm is also similar for edge coverage, with the difference that instead of finding a path reaching a target node of OG, the algorithm finds a path reaching a target edge (the so-far uncovered edge).

It can be shown that for every reachable state of A_S there exists a node S of OG covering this state, and similarly for locations and edges. Thus, covering all nodes in OG suffices to achieve coverage for each of the three criteria above. Since OG is finite, a finite number of tests suffices to achieve coverage, thus, the algorithm terminates. The worst-case complexity of the algorithm is polynomial in the size of OG. Indeed, finding a node (or edge) of OG associated with a location (or edge) of A_S is linear. Finding a path in OG and extending the path into a test tree is also linear. These steps are performed at most as many times as there are nodes in OG.

One drawback of the algorithm is that it does not always generate *minimal* test suites. A test suite is minimal in the sense that if any test is removed from the suite, then coverage is no longer achieved. In general the minimal suite is not unique. Moreover, adding a new test to the suite may result in making one or more previously generated tests redundant. We are currently studying methods of generating minimal

[5] Unreachable states of A_S can be ignored, since they play no role regarding conformance.

[6] In general, it is a good idea to continue extending the test tree in this way. This is because, using such a policy, a single test will cover as many locations as possible.

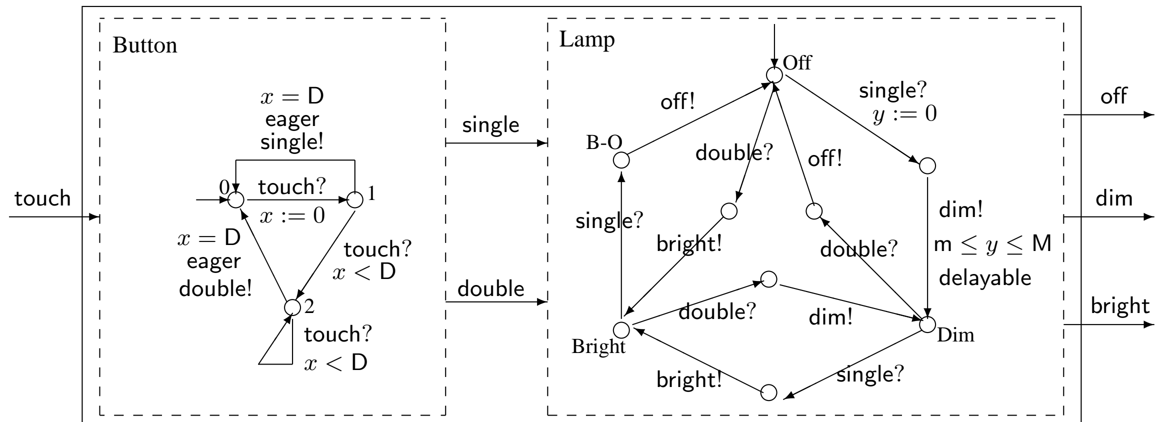

Figure 7: A lighting device specification.

test suites.

5 Prototype tool and experiments

We have built a prototype test-generation tool, called TTG, on top of the IF environment [10]. The IF modeling language allows to specify systems consisting of many processes communicating through message passing or shared variables and includes features such as hierarchy, priorities, dynamic creation and complex data types. Currently, TTG allows the user to generate digital tests interactively, randomly or exhaustively up to a given length. TTG can also generate analog on-the-fly testers or monitors for a given time granularity. Generation of timed automata testers and coverage criteria are being implemented.

We have used TTG to test the executive subsystem of the Mars rover controller K9, developed at NASA Ames. A detailed description of this case study can be found in [5]. Here, we illustrate the benefit of generating test suites with respect to coverage criteria on a small case study.

The case study is a modification of the light switch example presented in [20]. The (modified) specification is shown in Figure 7. It models a lighting device, consisting of two modules: the "Button" module which handles the user interface through a touch-sensitive pad and the "Lamp" module which lights the lamp to intensity levels "dim" or "bright", or turns the light off. The user interface logic is as follows: a "single" touch means "one level higher", whereas a "double" touch (two quick consecutive touches) means "one level lower". It is assumed that higher and lower is modulo three, thus, a single touch while the light is bright turns it off.

The device communicates with the external world through input touch and outputs off, dim, bright. Events single and double are used for internal communication between the two modules through *synchronous rendez-vous* and are unobservable to the external user. The Button module uses the timing parameter D which specifies the maximum delay between two consecutive touches if they are to be considered as a double touch. The Lamp module uses the timing parameters m and M which specify the minimum and maximum delay for the lamp to change intensity (e.g., to warm-up a halogen bulb).

In order not to overload the figure, we omit most guards, resets and deadlinesin the Lamp module. They are placed similarly to the ones shown in the figure (i.e., resets in inputs, guards and deadlines in outputs). We also omit the names of most locations of the Lamp module. There are three main locations named "Off, Dim, Bright" and a number of intermediate locations, for instance, "B-O" between "Bright" and "Off". The locations of the Button module are numbered $0, 1, 2$.

In [22] we have reported on using TTG to generate the exhaustive digital-clock test suite for the light switch specification, with parameter set $D = 1, m = 1, M = 2$ and for various depths. We have obtained 68, 180, 591 and 2243 tests, for depth levels 5, 6, 7 and 8, respectively. It can be seen that the number of tests grows exponentially with the depth and is very large, even for such a small example.

On the other hand, a very small test suite suffices to cover this specification with respect to any of the three criteria of Section 4. Consider, for instance, the two tests shown in Figure 8. In order not to overload the figure, each node of the tests is labeled only with the set of corresponding global locations; states are omitted. Also, for output nodes we only draw the outgoing edges which do not lead to FAIL. For example, node (2,Off) of the leftmost test has three outgoing edges labeled off?, dim?, bright? and leading to FAIL. Also, to save space, we draw the tree as a DAG (directed acyclic graph).

It can be seen from the figure that these two tests cover local locations. It is not difficult to check that the two tests cover edges as well. In fact, we can see from the figure that the two tests "walk trough" all observable edges of the specification. So it only remains to check that the unobservable edges are covered too. This is true since they are all visited between one of the pairs of successiveticks the two tests have (this is why nodes of the tests between successive ticks are labeled with pairs of global locations and not single global locations as for the other nodes).

The two tests do not achieve global location (and, consequently, neither state) coverage. For example, location (1,O-B) is not covered. However, 18 out of 30 global locations are covered. For covering the rest, it is possible either to generate more tests or to extend one of the two tests above. For instance, we can append the rightmost test at the end of the leftmost one.Also, in order to cover location (1,O-B), say, we can consider node (0,O-B) ofthe leftmost test as an output node instead of an input node (issuing the onlypossible output, touch!) and keep the remaining part of the test unchanged.Doing this, we can obtain a single test of depth 41 which achieves global location coverage. Alternatively, a suite of 8 tests of lengths smaller that those of Figure 8 suffices to achieve global location coverage. This suite can be generated by the algorithm of Section 4. Notice that the depth of the leftmost test of Figure 8 is 19. Generating an exhaustive test suite up to this depth would be infeasible due to explosion.

6 Conclusions and future work

The main contributions of this paper are two techniques for improving on-the-fly analog-clock testing and static digital-clock test generation. First, we have provided an algorithm to generate analog-clock testers which are represented as timed automata with one clock. This permits to minimize the reaction time of on-the-fly testing. Second, we have provided an algorithm to generate digital-clock test suites with respect to several coverage criteria, namely, state, location and edge coverage. This permits to significantly reduce the number of generated tests with respect to an approach of exhaustive test generation up to a certain depth, as evidenced on a small case study.

We are currently implementing on TTG the test generation technique with respect to coverage criteria of Section 4 and studying methods to generate minimal test suites. We are also implementing the timed automata tester generation technique of Section 3 and examining notions of coverage in this context as well.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 1, 2.1

[2] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In *CAV'94*, volume 818 of *LNCS*. Springer, 1994. 3

[3] 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 12th *Int. Workshop on Testing of Communicating Systems*. Kluwer, 1999. 1

[4] J. Bengtsson and W. Yi. On clock difference constraints and termination in reachability analysis of timed automata. In *ICFEM'03*, volume 2885 of *LNCS*. Springer, 2003. 3.1

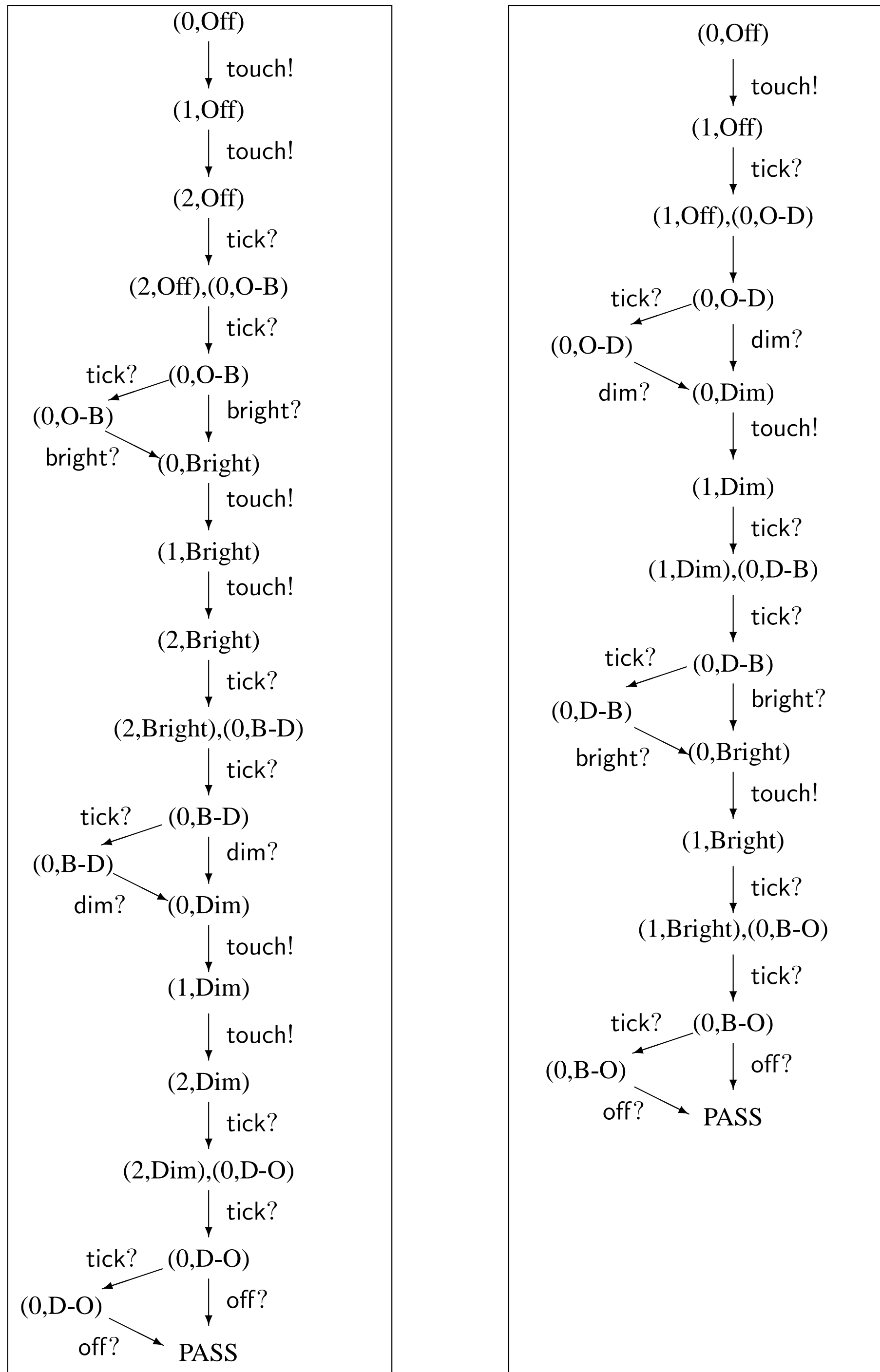

Figure 8: Two digital-clock tests covering most global locations of the specification of Figure 7.

[5] S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications by automatic generation of observers. In *4th International Workshop on Runtime Verification (RV'04)*, 2004. To appear in ENTCS series by Elsevier. 5

[6] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9:41–46, 1983. 3.1

[7] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer, 1998. 2.1

[8] P. Bouyer. Untameable timed automata! In *STACS'03*, volume 2607 of *LNCS*. Springer, 2003. 3.1

[9] P. Bouyer, D. D'Souza, P. Madhusudan, and A. Petit. Timed control with partial observability. In *CAV'03*, 2003. 2

[10] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In E.A. Emerson and A.P. Sistla, editors, *Proc. CAV'00*, volume 1855 of *LNCS*, pages 543–547. Springer Verlag, 2000. 5

[11] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *MOVEP 2000*, volume 2067 of *LNCS*. Springer, 2001. 1

[12] R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *FTRTFT'98*, volume 1486 of *LNCS*, 1998. 1

[13] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(1), 1978. 1

[14] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS'02*, volume 2280 of *LNCS*. Springer, 2002. 1

[15] D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *3rd Workshop on Object-Oriented Real-Time Dependable Systems (WORDS'97)*, 1997. 1

[16] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, volume 1384 of *LNCS*. Springer-Verlag, 1998. 3.1

[17] D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *Lecture Notes in Computer Science*, pages 197–212. Springer–Verlag, 1989. 3.1

[18] J.C. Fernandez, C. Jard, T. Jéron, and G. Viho. Using on-the-fly verification techniques for the generation of test suites. In *CAV'96*, LNCS 1102, 1996. 1

[19] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992. 2.3

[20] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal real-time test case generation using UPPAAL. In *FATES'03*, Montreal, October 2003. 1, 2, 2.3, 4, 4, 5

[21] T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed I/O automaton model. In *IFIP Int'l Work. Test. Communicat. Syst.* Kluwer, 1999. 1

[22] M. Krichen and S. Tripakis. Black-box conformance testing for real-time systems. In *11th International SPIN Workshop on Model Checking of Software (SPIN'04)*, volume 2989 of *LNCS*. Springer, 2004. 1, 1, 2, 2, 3, 4, 5

[23] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996. 1, 2.3

[24] B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS'01*. LNCS 2031, Springer, 2001. 1, 4

[25] J. Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *IDPT'02*, 2002. 1

[26] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, pages 347–359, Grenoble, France, February 1996. Lecture Notes in Computer Science 1046, Spinger-Verlag. 2.1

[27] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254, 2001. 1, 2.3, 2.3, 4

[28] J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10th Int. Conference on Concurrency Theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 46–65. Springer-Verlag, 1999. 1, 2.2

[29] J. Tretmans. Testing techniques. Lecture notes, University of Twente, The Netherlands, 2002. 1

[30] S. Tripakis. Fault diagnosis for timed automata. In *Formal Techniques in Real Time and Fault Tolerant Systems (FTRTFT'02)*, volume 2469 of *LNCS*. Springer, 2002. 1

[31] S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*. Springer, 2004. 1, 3

[32] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997. 4