# Black-box Conformance Testing for Real-Time Systems*

Moez Krichen and Stavros Tripakis

VERIMAG
Centre Equation, 2, avenue de Vignate, 38610 Gières, France. www-verimag.imag.fr.

**Abstract.** We propose a new framework for black-box conformance testing of real-time systems, where specifications are modeled as non-deterministic and partially-observable timed automata. We argue that such a model is essential for ease of modeling and expressiveness of specifications. The conformance relation is a timed extension of the input-output conformance relation of [29]. We argue that it is better suited for testing than previously considered relations such as bisimulation, must/may preorder or trace inclusion. We propose algorithms to generate two types of tests for this setting: analog-clock tests which measure dense time precisely and digital-clock tests which measure time with a periodic clock. The latter are essential for implementability, since only finite-precision clocks are available in practice. We report on a prototype tool and a small case study.

## 1 Introduction

Testing is a fundamental step in any development process. It consists in applying a set of experiments to a prototype system, 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 a system to a given specification. The system under test is "black-box" in the sense that we do not have a model of it, thus, can only rely on its observable input/output behavior.

Formal testing frameworks have been proposed (e.g., see [10]), where specifications are described in models with precise semantics and mathematical relations between such models define conformance. Then, under the assumption that the system under test (or *implementation*) can be modeled in the given framework, a set of tests can be automatically derived from the specification to test conformance of the (unknown) model. A number of issues arise, regarding the appropriateness of the models and conformance relation, the correctness of the testing process, its adequacy, its efficiency, and so on. Tools for test generation have been developed for various languages and models, both untimed (e.g., see [17, 3, 13]) and timed (e.g., see [8, 15, 12, 20, 25, 26, 19]).

* Work partially supported by European IST projects "Next TTA" under project No IST-2001-32111 and "RISE" under project No IST-2001-38117, and by CNRS STIC project "CORTOS".

In this paper, we propose a new testing framework for real-time systems, based on *timed automata* [1]. Existing works based on similar models (e.g., [14, 16, 25, 28, 11, 23, 19]) present two major limitations.

First, only restricted subclasses of timed automata are considered. This is problematic, since it limits the class of specifications that can be expressed. For example, [28, 19] consider timed automata where outputs are *isolated* and *urgent*. The first condition states that, at any given state, the automaton can only output a single action. Therefore, a specification such as "*when input a is received, output either b or c*" cannot be expressed in this model. Worse, the second condition states that, at any given state, if an output is possible, then time cannot elapse. This essentially means that outputs must be emitted at precise points in time. Therefore, a specification such as "*when input a is received, output b must be emitted within at most* 10 *time units*" cannot be expressed. Most other works consider deterministic or determinizable subclasses of timed automata. For instance, [25] use *event-recording automata* [2] and [23] use a determinizable timed automata model with restricted clock resets. It is also typically assumed that specifications are *fully-observable*, meaning that all events can be observed by the tester.

The second limitation concerns implementability of tests. Only *analog-clock* tests are considered in the works above. These are tests which can observe the time of inputs precisely and can also react by emitting outputs in precise points in time. For example, a test like "*emit output a at time 1; if at time 5 input b is received, announce PASS and stop, otherwise, announce FAIL*" is an analog-clock test. Analog-clock tests are problematic, since they are difficult, if not impossible, to implement with finite-precision clocks. The tester which implements the test of the example above must be able to emit *a precisely* at time 1 and check whether $b$ occurred *precisely* at time 5. However, the tester will typically sample its inputs periodically, say, every 0.1 time units, thus, it cannot distinguish between $b$ arriving anywhere in the interval $(4.9, 5.1)$.

In this paper, we lift the above limitations. Our main contributions are the following.

First, we develop a framework which can fully handle *non-deterministic* and *partially observable* specifications. Such specifications arise often in practice: when the model is built compositionally, component interactions are typically non-observable to the external world; abstraction from low-level details often results in non-determinism. In general, timed-automata cannot be determinized [1] and non-observable actions cannot be removed [5]. It can be argued that in practice many models will be determinizable. However, checking this (and performing the determinization) is undecidable [32]. Thus, it is important to offer a modeling framework which is general enough to relief the user from the burden of performing determinization "manually".

Second, we propose a conformance relation, called *timed input-output conformance* or tioco, inspired from the "untimed" conformance relation ioco of [29]. According to ioco, $A$ conforms to $B$ if for each observable behavior specified in $B$, the possible outputs of $A$ after this behavior is a subset of the possible outputs

of $B$. tioco is simply defined by including time delays in the set of observable outputs. This permits to capture the fact that an implementation producing an output too early or too late (or never, whereas it should) is non-conforming. A number of different conformance relations have been considered in previous works. [28] use bisimulation (which in that case reduces to trace equivalence, because of determinism). Bisimulation is also used in [12]. [25] use a must/may preorder. A must/may testing criterion is also considered in [20]. [19] use trace inclusion. [23] use an adaptation of ioco which, under the hypotheses of the model, is shown to be equivalent to trace inclusion. We argue that tioco is more appropriate for conformance testing than the above conformance relations, because it leaves more design freedom to potential implementations (see Section 3).

Finally, we consider both analog-clock and *digital-clock* (or *periodic-sampling*) tests. Analog-clock tests can measure precisely the delay between two events, whereas digital-clock tests can only count how many "ticks" of a periodic clock have occurred between the two events. Digital-clock tests are clearly more realistic to implement. Analog-clock tests can still be useful, however. For instance, when the implementation is discrete-time but its time step is not known a-priori.

The issue of determinization arises during test generation, since most algorithms rely on an implicit determinization of the specification. This presents problems for analog-clock test generation, due to the fact that timed automata are not determinizable in general, as mentioned above. To deal with the problem, we follow the idea of [31]: the automaton is "determinized" *on-the-fly*, during test generation and execution. The algorithm uses standard symbolic reachability techniques for timed automata. With a simple modification of the specification model, similar techniques can be used to generate digital-clock tests. The latter can be generated either on-the-fly or off-line, in which case they are represented as finite trees. We discuss a simple heuristic to reduce the size of these trees by eliminating chains of ticks. We also briefly discuss coverage, proposing a heuristic to generate a test suite which covers the edges of the specification automaton.

We have implemented our test-generation algorithms in a prototype tool, called TTG. The tool is built on top of the IF environment [7] and uses the modeling language of the latter. This language allows to specify systems of many processes communicating through message passing or shared variables and also includes features such as hierarchy, priorities, dynamic creation and complex data types. We have applied TTG to a small case study, presented in Section 6. We have also applied TTG to test behaviors of the K9 Martian Rover executive of NASA [9]. The results of TTG on this case study are reported in [4].

The rest of this paper is organized as follows. Section 2 reviews timed automata and timed automata with inputs and outputs. Section 3 introduces the testing framework. Section 4 defines analog and digital-clock tests. Section 5 presents the test generation methods for the two types of tests. Section 6 discusses a prototype implementation and illustrates the method on a small case study. Section 7 presents the conclusions and future work plans.

## 2 Timed Automata

Let $\mathsf{R}$ be the set of non-negative reals. Given a finite set of *actions* $\mathsf{Act}$, the set $(\mathsf{Act} \cup \mathsf{R})^*$ of all finite-length *real-time sequences* over $\mathsf{Act}$ 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 $\rho$ to $\mathsf{Act}'$, obtained by "erasing" from $\rho$ all actions not in $\mathsf{Act}'$. 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 $\rho$, denoted $\mathsf{time}(\rho)$ is the sum of all delays in $\rho$, for example, $\mathsf{time}(\epsilon) = 0$ and $\mathsf{time}(a\,1\,b\,0.5) = 1.5$.

We use timed automata [1] with *deadlines* to model urgency [27, 6]. A *timed automaton over* $\mathsf{Act}$ is a tuple $A = (Q, q_0, X, \mathsf{Act}, \mathsf{E})$ where: $Q$ is a finite set of *locations*; $q_0 \in Q$ is the initial location; $X$ is a finite set of *clocks*; $\mathsf{E}$ is a finite set of *edges*. Each edge is a tuple $(q, q', \psi, r, d, a)$, where $q, q' \in Q$ are the source and destination locations; $\psi$ 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 a set of clocks to *reset* to zero; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*; and $a \in \mathsf{Act}$ is the action. We will not allow $\mathsf{eager}$ edges with guards of the form $x > c$.

A timed automaton $A$ defines an infinite labeled transition system (LTS). Its states are pairs $s = (q, v)$, where $q \in Q$ and $v : X \to \mathsf{R}$ is a clock *valuation*. $\mathbf{0}$ is the valuation assigning 0 to every clock of $A$. $S_A$ is the set of all states and $s_0^A = (q_0, \mathbf{0})$ is the initial state. There are two types of transitions. Discrete transitions of the form $(q, v) \xrightarrow{a} (q', v')$, where $a \in \mathsf{Act}$ and there is an edge $(q, q', \psi, r, d, a)$, such that $v$ satisfies $\psi$ and $v'$ is obtained by resetting to zero all clocks in $r$ and leaving the others unchanged. Timed transitions of the form $(q, v) \xrightarrow{t} (q, v + t)$, where $t \in \mathsf{R}, t > 0$ and there is no edge $(q, q'', \psi, r, d, a)$, such that: either $d = \mathsf{delayable}$, $v \models \psi$ and $v + t \not\models \psi$; or $d = \mathsf{eager}$ and $v \models \psi$. 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 extends to timed sequences, in the usual way. 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 $\mathsf{Reach}(A)$.

*Timed Automata with Inputs and Outputs:* In the rest of the paper, we assume given a set of actions $\mathsf{Act}$, partitioned in two disjoint sets: a set of *input actions* $\mathsf{Act}_{\mathsf{in}}$ and a set of *output actions* $\mathsf{Act}_{\mathsf{out}}$. 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 $\mathsf{Act}_\tau$. A TAIO is called *observable* if none of its edges is labeled by $\tau$. A TAIO $A$ is called *input-complete* if it can accept any input at any state: $\forall s \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Act}_{\mathsf{in}} \,.\, s \xrightarrow{a}$. 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

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

This condition guarantees that $A$ will not block time in any environment.

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

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

## 3 Specifications, Implementations and Conformance

We now describe our testing framework. We assume that the specification of the system to be tested is given as a non-blocking TAIO $A_S$. We assume that the implementation (i.e., the system to be tested) 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. Input-completeness is required so that the implementation can accept inputs from the tester at any state (possibly ignoring them or moving to an error state, in case of illegal inputs).

In order to define the conformance relation, we define a number of operators. Given a TAIO $A$ and $\sigma \in \mathsf{RT}(\mathsf{Act})$, $A$ $\mathsf{after}$ $\sigma$ is the set of all states of $A$ that can be reached by some timed sequence $\rho$ whose projection to observable actions is $\sigma$. Formally:

$$A \text{ after } \sigma = \{s \in S_A \mid \exists \rho \in \mathsf{RT}(\mathsf{Act}_\tau) \,.\, s_0^A \xrightarrow{\rho} s \wedge P_{\mathsf{Act}}(\rho) = \sigma\}. \tag{3}$$

Given state $s \in S_A$, $\mathsf{elapse}(s)$ is the set of all delays which can elapse from $s$ without $A$ making any observable action. Formally:

$$\mathsf{elapse}(s) = \{t > 0 \mid \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}\}. \tag{4}$$

Given state $s \in S_A$, $\mathsf{out}(s)$ is the set of all observable "events" (outputs or the passage of time) that can occur when the system is at state $s$. The definition naturally extends to a set of states $S$. Formally:

$$\mathsf{out}(s) = \{a \in \mathsf{Act}_{\mathsf{out}} \mid s \xrightarrow{a}\} \cup \mathsf{elapse}(s), \qquad \mathsf{out}(S) = \bigcup_{s \in S} \mathsf{out}(s). \tag{5}$$

The *timed input-output conformance relation*, denoted $\mathsf{tioco}$, is defined as

$$A_I \text{ tioco } A_S \equiv \forall \sigma \in \mathsf{Traces}(A_S) \,.\, \mathsf{out}(A_I \text{ after } \sigma) \subseteq \mathsf{out}(A_S \text{ after } \sigma). \tag{6}$$

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

*Examples:* Before we proceed to define tests, we give some examples that illustrate the meaning of our testing framework. In the examples, input actions are denoted $a?$, $b?$, etc, and output actions are denoted $a!$, $b!$, etc. Unless otherwise mentioned, deadlines of output edges are $\mathsf{delayable}$ and deadlines of input edges

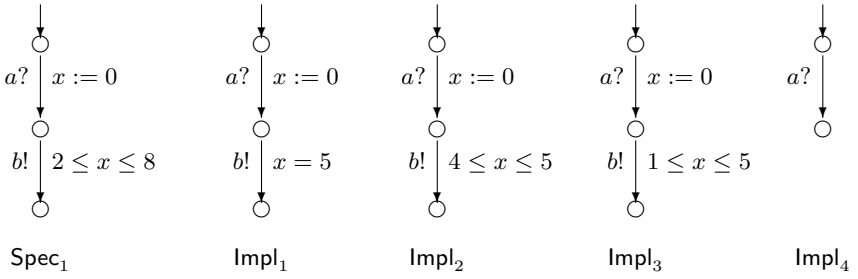



**Fig. 1.** Examples of specifications and implementations.

are lazy. In order not to overload the figures, we do not always draw input-complete automata. We assume that implementations ignore the missing inputs (this can be modeled by adding self-loop edges covering these inputs).

Consider the specification $\mathsf{Spec}_1$ shown in Figure 1. $\mathsf{Spec}_1$ could be expressed in English as follows: "after the first $a$ received, the system must output $b$ no earlier than 2 and no later than 8 time units". Implementations $\mathsf{Impl}_1$ and $\mathsf{Impl}_2$ conform to $\mathsf{Spec}_1$. $\mathsf{Impl}_1$ produces $b$ exactly 5 time units after reception of $a$. $\mathsf{Impl}_2$ produces $b$ within 4 to 5 time units. $\mathsf{Impl}_3$ and $\mathsf{Impl}_4$ do not conform to $\mathsf{Spec}_1$. $\mathsf{Impl}_3$ may produce a $b$ after 1 time unit, which is too early. $\mathsf{Impl}_4$ fails to produce a $b$ at all. Formally, $\mathsf{out}(\mathsf{Impl}_3 \text{ after } a\,1) = (0,4] \cup \{b\}$ and $\mathsf{out}(\mathsf{Impl}_4 \text{ after } a\,1) = (0,\infty)$, whereas $\mathsf{out}(\mathsf{Spec}_1 \text{ after } a\,1) = (0,7]$.

Now consider specification $\mathsf{Spec}_2$ shown in Figure 2. This specification could be written down as: "if the first input is $a$ then the system should output $b$ within 10 time units; if the first input is $c$ then the system should either output $d$ within 5 time units or, failing to do that, output $e$ within 7 time units". The second branch of $\mathsf{Spec}_2$ is a typical specification of a timeout. If the "normal" result $d$ does not appear for some time, the system itself should recognize the error and output an error message not much later. None of the four implementations of Figure 1 conform to $\mathsf{Spec}_2$, as they do not react to input $c$ (they ignore it). On the other hand, $\mathsf{Impl}_5$ and $\mathsf{Impl}_6$ of Figure 2 are conforming. It is worth noticing that $\mathsf{Impl}_6$ may output a $b$ some time after receiving input $f$. The fact that input $f$ does not appear in $\mathsf{Spec}_2$ does not affect the conformance of $\mathsf{Impl}_6$. (In fact, $\mathsf{Impl}_5$ and $\mathsf{Impl}_6$ conform not only to $\mathsf{Spec}_2$ but also to $\mathsf{Spec}_1$.) This example illustrates another property of tioco, namely, that an implementation is free to accept inputs not mentioned in the specification and behave as it wishes afterwards. This property is essential for capturing assumptions on the inputs (i.e., on the environment) in the specification. This is why we do not require specifications to be input-complete.

*Comparison:* [28] define conformance as timed bisimulation (TB), which in their case reduces to timed trace equivalence (TTE), since determinism is assumed. [25] define conformance using a must/may preorder (MMP). None of $\mathsf{Impl}_1, \mathsf{Impl}_2$

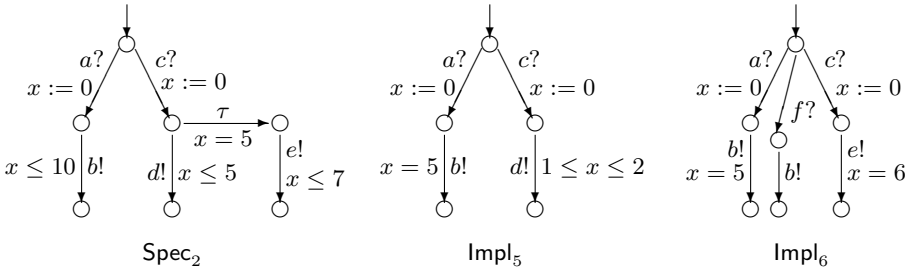



**Fig. 2.** More examples of specifications and implementations.

conform to $\mathsf{Spec}_1$ w.r.t. TB, TTE or MMP. We believe that this is too strict.[1] [23, 19] define conformance as timed trace inclusion (TTI). TTI is generally stricter than tioco: tioco allows an implementation to accept inputs not accepted by the specification, whereas TTI does not. When the specification is input-complete, tioco and TTI are equivalent. A deterministic (and fully observable) specification can be made input-complete without changing its conformance semantics by adding edges covering the missing inputs and leading to a "don't care" location where all inputs and outputs are accepted. This transformation is not always possible for non-deterministic specifications. Moreover, if the transformation is performed, care must be taken to instruct the test generation algorithm not to explore the "don't care" location, so that it does not generate useless tests. We opt for tioco, which avoids these complications in a simple way. For an extensive discussion of various untimed conformance relations, see [30].

## 4 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 [18]). *Analog-clock* tests can measure precisely the delay between two observed actions and can emit an input[2] at any point in time. *Digital-clock* (or *periodic-sampling*) tests can only count how many "ticks" of a periodic clock have occurred between two actions and emit an input immediately after observing an action or tick. For simplicity, we assume that the tester and the implementation are started precisely at the same time.

[1] It should be noted, however, that the issue does not arise in [28] because outputs are assumed to be urgent, thus, $\mathsf{Spec}_1$ cannot be expressed.

[2] We always use terms "input" and "output" to mean input/output of the implementation. Thus, we write "the test emits an input" rather than "emits an output". We follow the same convention when drawing test automata. For example, the edge labeled $a?$ in the TAIO of Figure 3 corresponds to the tester emitting $a$, upon execution of the test.

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 [24]), 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 [28, 19] generate non-adaptive test *sequences*.

### 4.1 Analog-clock tests

An analog-clock test for a specification $A_S$ over $\mathsf{Act}_\tau$ is a total function

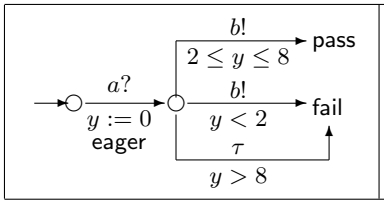
$$T : \mathsf{RT}(\mathsf{Act}) \to \mathsf{Act}_{\mathsf{in}} \cup \{\bot, \mathsf{pass}, \mathsf{fail}\}. \tag{7}$$

$T(\rho)$ specifies the action the tester must take once it observes $\rho$. If $T(\rho) = a \in \mathsf{Act}_{\mathsf{in}}$ then the tester emits input $a$. If $T(\rho) = \bot$ then the tester waits (lets time elapse). If $T(\rho) \in \{\mathsf{pass}, \mathsf{fail}\}$ then the tester produces a verdict (and stops). To represent a valid test, $T$ must satisfy a number of conditions:

$$\exists t \in \mathsf{R} \,.\, \forall \rho \in \mathsf{RT}(\mathsf{Act}) \,.\, \mathsf{time}(\rho) > t \Rightarrow T(\rho) \in \{\mathsf{pass}, \mathsf{fail}\} \tag{8}$$

$$\forall \rho \in \mathsf{RT}(\mathsf{Act}) \,.\, T(\rho) \in \{\mathsf{pass}, \mathsf{fail}\} \Rightarrow \forall \rho' \in \mathsf{RT}(\mathsf{Act}) \,.\, T(\rho \cdot \rho') = T(\rho) \tag{9}$$

Condition (8) states that the test reaches a verdict in bounded time $t$ (called the *completion time* of the test). Condition (9) is a "suffix-closure" property ensuring that the test does not recall a verdict. We also need to ensure that the test does not block time, for instance, by emitting an infinite number of inputs in a bounded amount of time. This can be done by specifying certain conditions on the LTS defined by $T$. The states of this LTS are sequences $\rho \in \mathsf{RT}(\mathsf{Act})$. The initial state is $\epsilon$. For every $a \in \mathsf{Act}_{\mathsf{out}}$ there is a transition $\rho \xrightarrow{a} \rho \cdot a$. There is also a transition $\rho \xrightarrow{t} \rho \cdot t$ for every $t \in \mathsf{R}$, provided $\forall t' \leq t . T(\rho) = \bot$. If $T(\rho) = b \in \mathsf{Act}_{\mathsf{in}}$ then there is a transition $\rho \xrightarrow{b} \rho \cdot b$. As a convention, all states $\rho$ such that $T(\rho) = \mathsf{pass}$ are "collapsed" into a single sink state pass, and similarly with fail. We require that states of this LTS are non-blocking as in Condition (1), unless pass or fail is reached.




$$\begin{aligned}
T(\epsilon) &= a? \\
T(a?) &= \bot \\
T(a?\,t) &= \bot, \quad \forall t \leq 8 \\
T(a?\,t) &= \mathsf{fail}, \quad \forall t > 8 \\
T(a?\,t\,b!) &= \mathsf{fail}, \quad \forall t < 2 \\
T(a?\,t\,b!) &= \mathsf{pass}, \forall t \in [2, 8]
\end{aligned}$$

**Fig. 3.** Analog-clock test represented as a TAIO or a function.

Analog-clock tests can sometimes be represented as TAIO.[3] For example, the test defined in the right part of Figure 3 can be equivalently represented by the TAIO shown in the left part. Function $T$ is partially defined in the figure. The remaining cases are covered by the suffix-closure property of pass/fail − Condition (9). For instance, $T(a?\,9\,b!) = \mathsf{fail}$, because $T(a?\,9) = \mathsf{fail}$.

Execution of the test $T$ on the implementation $A_I$ can be defined as the *parallel composition* of the LTSs defined by $T$ and $A_I$, with the usual *synchronization* rules for transitions carrying the same label. We will denote the product LTS by $A_I \| T$. The execution of the test reaches a pass/fail verdict after bounded time. However, since the implementation can be non-deterministic or non-observable, the verdict need not be the same in all experiments (i.e., runs of the product). To declare that the implementation passes the test, we require that *all* possible experiments lead to a pass verdict. This implies that in order to gain confidence in pass verdicts, the same test must be executed multiple times, unless the implementation is known to be deterministic.

Formally, we say that $A_I$ *passes* the test, denoted $A_I$ passes $T$, if state fail is not reachable in the product $A_I \| T$. We say that an implementation passes (resp. fails) a set of tests (or *test suite*) $\mathcal{T}$ if it passes all tests (resp. fails at least one test) in $\mathcal{T}$. We say that $\mathcal{T}$ is *sound with respect to* $A_S$ if $\forall A_I \;.\; A_I$ tioco $A_S \Rightarrow A_I$ passes $\mathcal{T}$. We say that $\mathcal{T}$ is *complete with respect to* $A_S$ if $\forall A_I \;.\; A_I$ passes $\mathcal{T} \Rightarrow A_I$ tioco $A_S$.

Soundness is a minimal correctness requirement. Is is rather weak, since many tests can be sound and useless (by always announcing pass). Completeness, on the other hand, is usually impossible to achieve with a finite test suite (see Section 5.3). We are thus motivated to define another notion. We say that a test $T$ is *strict with respect to* $A_S$ if $\forall A_I \;.\; A_I$ passes $T \Rightarrow A_I \| T$ tioco $A_S$. What the above definition says is that a strict test must not announce pass when the implementation has behaved in a non-conforming manner *during the execution of the test*. In the untimed setting, a similar notion of *lax* tests is proposed in [22]. The test shown in Figure 3 is sound and strict w.r.t. $\mathsf{Spec}_1$ of Figure 1. Changing the fail state of the test into pass would yield a test which is still sound, but no longer strict.

### 4.2 Digital-clock tests

Consider a specification $A_S$ over $\mathsf{Act}_\tau$ and let tick be a new output action, not in $\mathsf{Act}_\tau$. A digital-clock test (or *periodic sampling* test) for $A_S$ is a total function

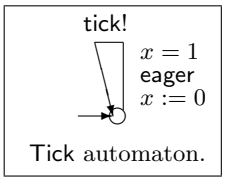
$$D : (\mathsf{Act} \cup \{\mathsf{tick}\})^* \to \mathsf{Act}_{\mathsf{in}} \cup \{\bot, \mathsf{pass}, \mathsf{fail}\}. \tag{10}$$

The digital-clock test can observe all input and output actions, plus the action tick which is assumed to be the output of the tester's digital clock. We assume

[3] But not always: the test which moves to pass once it observes a sequence of $a$'s such that the time distance between two $a$'s is 1 cannot be captured by a timed automaton with a bounded number of clocks. This is related to the fact that timed automata are not determinizable whereas a test is by definition deterministic.

that the initial phase of the clock is 0 and its period is 1. We further assume that the clock is never reset, and that ticks have priority over other observable actions (i.e., if tick and $a$ occur at the same time, tick will be always observed before $a$). With these assumptions, if action $a$ is observed after the $i$-th and before the $(i+1)$-st tick, then the tester knows that $a$ occurred at some time in the interval $[n, n+1)$.

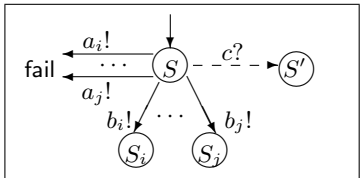
Validity conditions similar to those for analog-clock apply to digital-clock tests as well. Due to lack of space, we omit the formal definitions. A digital-clock test $D$ defines a LTS with states in $(\mathsf{Act} \cup \{\mathsf{tick}\})^*$ and labels in $\mathsf{Act} \cup \{\mathsf{tick}\} \cup \mathsf{R}$. Given state $\pi$, if $D(\pi) \notin \mathsf{Act_{in}}$ then $\pi$ has a self-loop transition labeled with $t$, for all $t \in \mathsf{R}$. The reason such transitions are missing from states such that $D(\pi) = a \in \mathsf{Act_{in}}$ is that we assume that the digital-clock test emits $a$ immediately after the last event in $\pi$ is observed.



Tick automaton.

Execution of a digital-clock test is defined by forming the parallel product of three LTSs, namely, the ones of the test $D$, the implementation $A_I$, and the Tick automaton shown to the left. Tick implicitly synchronizes with $A_I$ through time. Tick explicitly synchronizes with $D$ on transitions labeled tick. The parallel product is built so that tick transitions have priority over other observable transitions. Thus, if $s$ is a state of the product and $s \xrightarrow{\mathsf{tick}}$, then $s$ has no other outgoing transition. The definition of passes for digital-clock tests is similar to the one for analog-clock tests, with $A_I \| T$ being replaced by $A_I \| \mathsf{Tick} \| D$. The definitions of soundness, completeness and strictness also carry over in the natural way.

## 5 Test generation



Generic test-generation scheme.

We adapt the untimed test generation algorithm of [29]. Roughly speaking, the algorithm builds a test in the form of a tree. A node in the tree is a set of states $S$ of the specification and represents the "knowledge" of the tester at the current test state. The algorithm extends the test by adding successors to a leaf node, as illustrated in the figure to the left. For all *illegal* outputs $a_i$ (outputs which cannot occur from any state in $S$) the test leads to fail. For each legal output $b_i$, the test proceeds to node $S_i$, which is the set of states the specification can be in after emitting $b_i$ (and possibly performing unobservable actions). If there exists an input $c$ which can be accepted by the specification at some state in $S$, then the test may decide to emit this input (dashed arrow from $S$ to $S'$). At any node, the algorithm may decide to stop the test and label this node as pass.

Two features of the above algorithm are worth noting. First, the algorithm is only partially specified. Indeed, a number of decisions need to be made at each

node: (1) whether to stop the test or continue, (2) whether to wait or emit an input if possible, (3) which input, in case there are many possible inputs. Some of these choices can be made according to user-defined parameters, such as the desired depth of the test. They can also be made randomly or systematically using some book-keeping, in order to generate a test suite, rather than a single test. We discuss this option in more detail in Section 5.3.

The second feature of the algorithm is that it implicitly *determinizes* the specification automaton. Indeed, building $S_i, S_j$ and so on corresponds to a classical *subset construction*. The latter can be performed either off-line, that is, before the test generation, or on-line, that is, during the test generation or even during the test execution. Test generation during test execution has been termed *on-the-fly* and is supported by the tool Torx [3].

### 5.1 Generating analog-clock tests

Analog-clock tests cannot be represented as a finite tree, because there is an a-priori infinite set of possible observable delays at a given node. To remedy this, we use the idea of [31]. We represent an analog-clock test as an *algorithm*. The latter essentially performs subset construction on the specification automaton, during the execution of the test. Thus, our analog-clock testing method can be classified as on-the-fly.

More precisely, the test will maintain a set of states $S$ of the specification TAIO, $A_S$. $S$ will be updated every time an action is observed or some time delay elapses. Since the time delay is not known a-priori, it must be an input to the update function. We define the following operators:

$$\mathsf{dsucc}(S,a) = \{s' \mid \exists s \in S \,.\, s \xrightarrow{a} s'\} \tag{11}$$

$$\mathsf{tsucc}(S,t) = \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho} s'\} \tag{12}$$

where $a \in \mathsf{Act}$ and $t \in \mathsf{R}$. $\mathsf{dsucc}(S,a)$ contains all states which can be reached by some state in $S$ performing action $a$. $\mathsf{tsucc}(S,t)$ contains all states which can be reached by some state in $S$ via a sequence $\rho$ which contains no observable actions and takes exactly $t$ time units. The two operators can be implemented using standard data structures for symbolic representation of the state space and simple modifications of reachability algorithms for timed automata [31].

The test operates as follows. It starts at state $S_0 = \mathsf{tsucc}(\{s_0^{A_S}\}, 0)$. Given current state $S$, if output $a$ is received $t$ time units after entering $S$, then $S$ is updated to $\mathsf{dsucc}(\mathsf{tsucc}(S,t),a)$. If no event is received until, say, 10 time units later, then the test can update its state to $\mathsf{tsucc}(S,10)$. If ever the set $S$ becomes empty, the test announces $\mathsf{fail}$. At any point, for an input $b$, if $\mathsf{dsucc}(S,b) \neq \emptyset$, the test may decide to emit $b$ and update its state accordingly. At any point, the test may decide to stop, announcing $\mathsf{pass}$.

It can be shown that the test defined above is both sound and strict.

### 5.2 Generating digital-clock (periodic-sampling) tests

Since its set of observable events is finite ($\mathsf{Act} \cup \{\mathsf{tick}\}$), a digital-clock test can be represented as a finite tree. In this case, we can decide whether to generate tests on-the-fly or off-line. This is a matter of a space/time trade-off. The on-the-fly method does not require space to store the generated tests. On the other hand, a test computed on-the-fly has a longer reaction time than a test which has been computed off-line.

Independently of which option we choose, we proceed as follows. We first form the product $A'_S = A_S \| \mathsf{Tick}$. We then define the following operator on $A'_S$:

$$\mathsf{usucc}(S) = \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, s \xrightarrow{\rho} s'\}. \tag{13}$$

$\mathsf{usucc}(S)$ contains all states which can be reached by some state in $S$ via a sequence $\rho$ which contains no observable actions. Notice that, by construction of $A'_S$, the duration of $\rho$ is bounded: since $\mathsf{tick}$ is observable and has to occur after at most 1 time unit, $\mathsf{time}(\rho) \leq 1$.

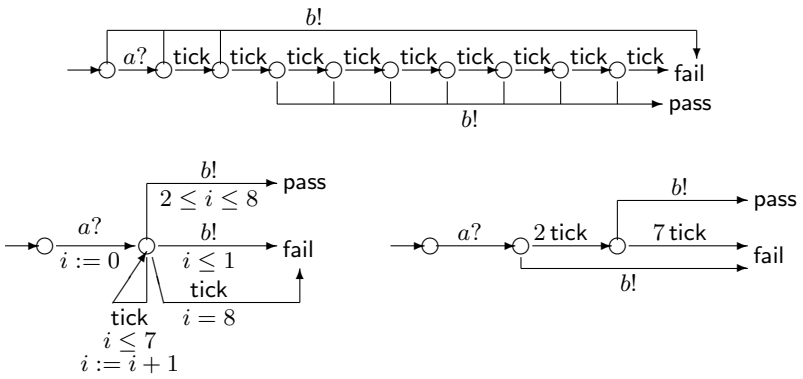



**Fig. 4.** A digital-clock test (top) and two alternative representations (bottom).

Finally, we apply the generic test-generation scheme presented above. The root of the test tree is defined to be $S_0 = \{s_0^{A_S}\}$. Successors of a node $S$ are computed as follows. For each $a \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$, there is an edge $S \xrightarrow{a} S'$ with $S' = \mathsf{dsucc}(\mathsf{usucc}(S), a)$, provided $S' \neq \emptyset$, otherwise there is an edge $S \xrightarrow{a} \mathsf{fail}$. If there exists $b \in \mathsf{Act}_{\mathsf{in}}$ such that $S'' = \mathsf{dsucc}(\mathsf{tsucc}(S, 0), b) \neq \emptyset$, then the test generation algorithm may decide to emit $b$ at $S$, adding an edge $S \xrightarrow{b} S''$. Notice the asymmetry in the ways $S'$ and $S''$ are computed. The reason is that the tester is assumed to emit an input $b$ immediately upon entering $S$. Thus, $S''$ should only contain the immediate successors of $S$ by $b$.

The tests generated in this way are guaranteed to be sound. However, they are not strict in general. This is expected, since the tester cannot distinguish

between outputs being produced *exactly* at time 1 or, say, at time 1.5. A sound (but not strict) digital-clock test for $\mathsf{Spec}_1$ of Figure 1 is shown in the top of Figure 4.

*Reducing the size of digital-clock tests:* Digital-clock tests can sometimes grow large because they contain a number of "chains" of ticks. On the other hand, standard test description languages such as TTCN [21] permit the use of variables and richer data structures. We would like to use such features to make the representation of digital-clock tests more "compact". For example, the test shown in the top of Figure 4 can be equivalently represented as the automaton with counter $i$, shown in the bottom-left of the figure.

Reducing the size of test representations is a non-trivial problem in general, related to compression and algorithmic complexity theory. In our context, we only use a heuristic which attempts to eliminate tick chains as much as possible. To this purpose, we generalize the labels of the digital-clock test to labels of the form $k\,\mathsf{tick}$, where $k$ is a positive integer constant. A transition labeled with $k\,\mathsf{tick}$ is taken when the $k$-th tick is received, counting from the time the source node is entered. Naturally, tick is equivalent to $1\,tick$. Now, consider two nodes $S$ and $S'$ such that: (1) $S \stackrel{\mathsf{tick}}{\rightarrow} S'$, (2) for all $a \in \mathsf{Act}$, the successors of $S$ and $S'$ are identical, (3) $S' \stackrel{k\,\mathsf{tick}}{\rightarrow} S''$. In this case, we remove node $S'$ (and corresponding edges) and add the edge $S \stackrel{(k+1)\,\mathsf{tick}}{\rightarrow} S''$. We repeat the process until no more nodes can be removed. The result of applying this heuristic to the test in the top of Figure 4 is shown in the bottom-right of the figure.

### 5.3 Coverage

It is generally impossible to generate a finite test suite which is complete, in particular when the specification has loops, which define an infinite set of possible behaviors. This is because implementations can have an arbitrary number of states, while a finite test suite can only explore a bounded number of states. But an implementation could be conforming up to a certain point and not conforming afterwards.

To remedy this fact, test generation methods usually make a compromise: instead of generating a complete test suite, generate a test suite which *covers* the specification.[4] Different coverage criteria have been proposed for untimed systems, such as state coverage (every state of the specification must be "explored" by at least one test), transition coverage (every transition must be explored), and so on. A survey of coverage criteria and their relationships, in the context of software testing, can be found in [33]. In the case of timed automata the state space is infinite, thus, existing methods attempt to cover: either finite abstractions of the state space, e.g., the region graph in [28, 16], a time-abstracting partition graph in [25]; or the structural elements of the specification, e.g., [19] propose

[4] Some methods [28, 12] generate a suite which is complete w.r.t. a given upper bound on the number of states of the implementation.

techniques for edge, location, or definition-use pair coverage and [8] consider various criteria in the context of timed Petri nets.

In the spirit of [19], we propose a heuristic for generating a digital-clock test suite covering the edges of the specification automaton. Notice that we cannot use the technique of [19], which is based on formulating coverage as a reachability problem. Indeed, this technique relies on the assumption that outputs in the specification are urgent and isolated, which results in tests being *sequences*, rather than trees.

Our method aims at covering edges labeled with an input action. Then, edges labeled with outputs will also be covered, since a test must be able to accept any output at any state. Let $\mathcal{T}$ be a test suite and $S \xrightarrow{a} S'$ be an edge in some test of $\mathcal{T}$, with $a \in \mathsf{Act_{in}}$. If $e$ is an edge of $A_S$ labeled with $a$ and enabled at some state in $S$, then we say that $e$ is *covered* by $\mathcal{T}$. We say that $\mathcal{T}$ covers $A_S$ if all input edges of $A_S$ are covered by $\mathcal{T}$. Then, the test generation algorithm can stop once it has generated a test suite covering $A_S$.

## 6 Tool and case study

We have built a prototype test-generation tool, called TTG, on top of the IF environment [7]. 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. The IF tool-suite includes a simulator, a model checker and a connection to the untimed test generator TGV [17]. TTG is implemented independently from TGV. It is written in C++ and uses the basic libraries of IF for parsing and symbolic reachability of timed automata with deadlines.

TTG takes as main input the specification automaton, written in IF language, and can generate two types of tests: (1) analog-clock tests under the assumption that the implementation is discrete-time and has a time step of 1; (2) digital-clock tests with respect to a given Tick automaton. By modifying the Tick automaton, the user can implement different sampling rates, model jitter in the sampling period, and so on. TTG can be executed in an interactive mode, where the user guides the test generation by resolving decision points. TTG can also be asked to generate a single test randomly or the exhaustive test suite, up to a user-defined depth. The depth of a test is the longest path from the initial state to a pass or fail state. The tests are output in IF language.

We have applied TTG to a small case study, which is a modification of the light switch example presented in [19]. The (modified) specification is shown in Figure 5. 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.

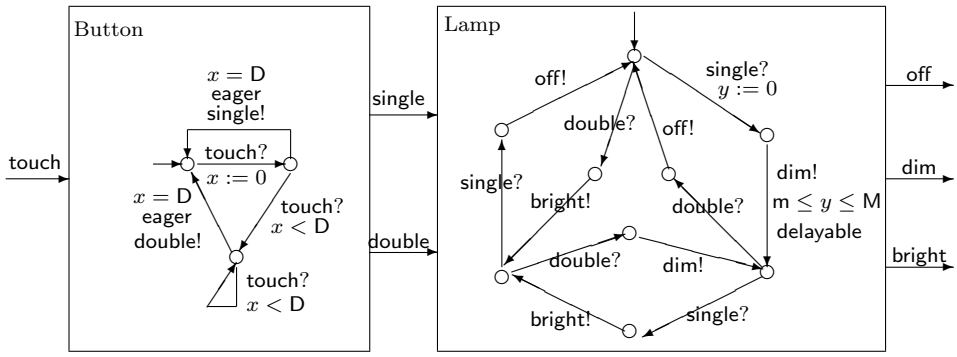



**Fig. 5.** A lighting device.

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 non-observable 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 deadlines in 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 have used TTG to generate the exhaustive digital-clock test suite for the above specification, with parameter set $\mathsf{D} = 1, \mathsf{m} = 1, \mathsf{M} = 2$, for various depth levels. We have obtained 68, 180, 591 and 2243 tests, for depth levels 5, 6, 7 and 8, respectively. Notice that these are the sets of all possible tests up to the specified depth: no test selection is performed. Moreover, the current implementation is sub-optimal because it generates tests announcing pass before the maximum depth is reached. Implementation of test selection criteria is underway. One of the tests generated by TTG is shown in Figure 6. The drawing has been produced automatically using the `if2eps` tool by Marius Bozga.

## 7 Summary and future work

We have proposed a testing framework for real-time systems based on non-deterministic and partially-observable timed-automata specifications. To our knowledge, this is the first framework that can fully handle such specifications. We introduced a timed version of the input-output conformance relation of [29] and proposed techniques to generate analog-clock and digital-clock tests for this relation. We reported on a prototype tool and a simple case-study.

Regarding future work, our priority is to study test selection methods, in order to reduce the number of generated tests. To this aim, we are currently implementing the edge-coverage heuristic discussed in Section 5.3. We are also working on reducing the size of generated tests, implementing the reduction heuristic for digital-clock tests discussed in Section 5.2. Another direction that we pursue is to identify classes of specifications for which analog-clock tests can be represented as timed automata. One such class is deterministic and observable specifications. The advantage of a timed automata representation is that it avoids on-the-fly reachability computation, thus reducing the reaction time of the test.

*Acknowledgment:* We would like to thank Marius Bozga for his help with IF.

## References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In *CAV'94*, volume 818 of *LNCS*. Springer, 1994.
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 $12^{th}$ *Int. Workshop on Testing of Communicating Systems*. Kluwer, 1999.
4. S. Bensalem, M. Bozga, M. Krichen, and S. Tripakis. Testing conformance of real-time applications by automatic generation of observers. In *Runtime Verification (RV'04)*. To appear in ENTCS.
5. B. Berard, A. Petit, V. Diekert, and P. Gastin. Characterization of the expressive power of silent transitions in timed automata. *Fundamenta Informaticae*, 36(2-3):145–182, 1998.
6. S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer, 1998.
7. 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.
8. V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. In *International Software Quality Week*, 1997.
9. G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, and W. Visser. Experimental evaluation of V&V tools on martian rover software. In *SEI Software Model Checking Workshop*, 2003.
10. E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *MOVEP 2000*, volume 2067 of *LNCS*. Springer, 2001.
11. R. Cardell-Oliver. Conformance test experiments for distributed real-time systems. In *ISSTA'02*. ACM Press, 2002.
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.
13. D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS'02*, volume 2280 of *LNCS*. Springer, 2002.
14. 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.

15. D. Clarke and I. Lee. Automatic test generation for the analysis of a real-time system: Case study. In *RTAS'97*, 1997.
16. A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS'98*. IEEE, 1998.
17. 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.
18. T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992.
19. 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.
20. 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.
21. ISO/IEC. Open Systems Interconnection Conformance Testing Methodology and Framework – Part 1: General Concept – Part 2 : Abstract Test Suite Specification – Part 3: The Tree and Tabular Combined Notation (TTCN). Technical Report 9646, International Organization for Standardization — Information Processing Systems — Open Systems Interconnection, Genève, 1992.
22. C. Jard, T. Jéron, and P. Morel. Verification of test suites. In *TESTCOM 2000*, 2000.
23. A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES'03*, Montreal, October 2003.
24. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.
25. B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS'01*. LNCS 2031, Springer, 2001.
26. J. Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *IDPT'02*, 2002.
27. 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.
28. J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254, 2001.
29. 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.
30. J. Tretmans. Testing techniques. Lecture notes, University of Twente, The Netherlands, 2002.
31. 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.
32. S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, LNCS. Springer, 2003.
33. H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997.

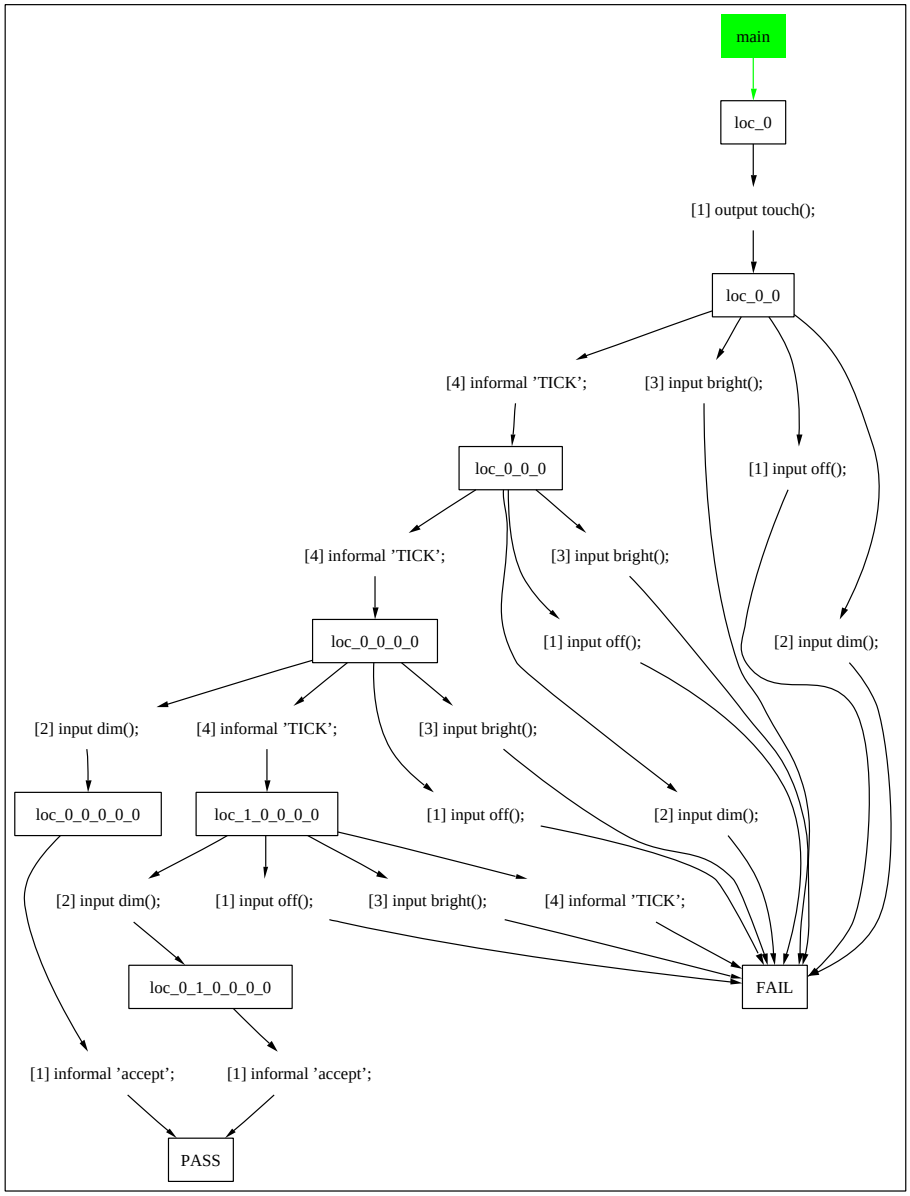



**Fig. 6.** A test generated automatically by TTG.