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

An Expressive and Implementable Formal Framework for Testing Real-Time Systems

Moez Krichen, Stavros Tripakis

Report n^o TR-2004-13

June 1, 2004

An Expressive and Implementable Formal Framework for Testing Real-Time Systems

Moez Krichen, Stavros Tripakis

June 1, 2004

Abstract

We propose a new framework for black-box conformance testing of real-time systems. The framework is based on the model of partially-observable, non-deterministic timed automata. We argue that partial observability and non-determinism are essential features for ease of modeling, expressiveness and implementability. We provide algorithms for on-the-fly or static generation of digital-clock tests. These tests measure time only with finite-precision, periodic clocks, another essential condition for implementability. We also propose a technique for location and edge coverage of the specification, by reducing the problem to covering a symbolic reachability graph. We report on a prototype tool and two case studies.

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

Reviewers:

Notes: 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”.

How to cite this report:

@techreport { ,
title = { An Expressive and Implementable Formal Framework for Testing Real-Time Systems},
authors = { Moez Krichen, Stavros Tripakis},
institution = { Verimag Technical Report },
number = {TR-2004-13},
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 [14, 26]) or finite labeled transition systems − LTSs (e.g., see [31, 11, 20, 4, 15]). 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 timeouts are modeled by special δ actions [30] 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.

A model which has become quite popular during the past decade for modeling and verifying real-time systems is the model of *timed automata − TA* [2]. A number of methods for testing real-time systems based on variants of the above model (or other similar models such as timed Petri nets) have been proposed (e.g., see [9, 16, 19, 23, 27, 29, 28, 12, 24, 22]). However, these methods present two major limitations.

First, only restricted subclasses of the TA model are considered. This is problematic, since it limits the class of specifications that can be expressed in the above frameworks. For example, [29, 22] consider TA 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 TA. For instance, [27] use *event-recording automata* [3] and [24] use a determinizable TA model with restricted clock resets. Most of the works also assume that specifications are *fully-observable*, meaning that it is assumed that all events can be observed by the tester. All these restrictions limit the applicability of the methods. Indeed, a specification must be able to leave freedom to potential implementations, especially on choosing different outputs or output times. Also, as we argue below, partial observability and non-determinism are essential for ease of modeling, expressiveness and implementability.

The second limitation concerns implementability of tests. Only *analog-clock* tests (in the sense of [21, 25]) 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 $(4.9, 5.1)$.

Moreover, delays occur in the transmission of events from the SUT to the tester and vice versa. For instance, a tester may issue an output command when its internal clock expires, say, at time 10. However, operating system and device-driver delays may result in the SUT receiving the input at time 10.5 or later. These delays must be accounted for, in order not to give incorrect PASS/FAIL verdicts. This point is also

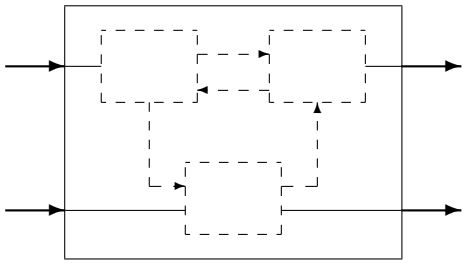

Figure 1: A compositional specification with internal (unobservable) actions.

made in [12], where other issues in interfacing tester and SUT are also brought up (e.g., observability problems). Some examples are given on how to control or observe SUT events during execution. However, no general theory of handling such issues is provided.

In this paper, we propose a new testing framework for real-time systems, which lifts the above limitations. Our framework is *expressive*: it can fully handle *partially-observable*, *non-deterministic* timed automata. It is also *implementable*: the tests we generate can be implemented using periodic clocks of finite precision.

We model specifications as timed automata with input, output or unobservable actions (without loss of generality, a single unobservable action is enough). The automata can also be non-deterministic, in the sense that a given action at a given time might lead the automaton to two different states. Such models arise often in practice. Specifications are usually built in a *compositional* way, from many components (see Figure 1). This greatly simplifies modeling.[1] In such cases, *internal* component interactions are typically unobservable to the external world, thus, also to the tester. *Abstractions* from low-level details are also used often, to simplify modeling and manage complexity. Such abstractions could, for instance, "hide" some variables, which typically results in non-determinism.

In general, timed-automata cannot be determinized [2] and unobservable actions cannot be removed [6]. It can be argued that, in practice, many models will be determinizable. However, checking this (and performing the determinization) is undecidable [33]. Thus, the user is left with two alternatives. Either attempt to fit the specification into a deterministic, fully-observable TA model, or use a framework like ours, which handles non-determinism and partial observability directly. Clearly, the first alternative is hardly feasible in practice, especially for large specifications consisting of many components, as it implies that the user has to perform determinization of such a model "manually".

Our second contribution is that we propose a technique to generate *digital-clock* (or *periodic-sampling*) tests. Whereas analog-clock tests can measure precisely the delay between two input events, digital-clock tests can only count how many "ticks" of a periodic clock have occurred between the events. Regarding outputs, analog-clock tests can emit outputs at precise points in time, whereas in digital-clock test only guarantees that the output will be emitted at some point between two clock ticks. Other test-interfacing issues can also be taken into account, such as clock *skew* (when the clock of the tester is not perfectly periodic) or delays in transmitting inputs and outputs from the SUT to the tester and back.

Note that generating digital-clock tests does *not* mean that we discretize time. Indeed, the specification still has a continuous-time semantics. It is the tester which samples these with a digital clock.

Our test-generation method relies on a *symbolic reachability* algorithm, inspired from similar algorithms used in timed automata model-checking tools such as Kronos [17]. The technique has been first introduced in [32], where it is used for fault detection (thus, *passive* testing, where the tester only observes the SUT without providing inputs to it). Symbolic reachability allows us to represent the infinite state-space of a TA model as a finite graph. Test generation can be performed either in a *static* way by extracting a test suite from this graph, or in an *on-the-fly* way, by generating the test during test execution. Static test generation can suffer from the so-called *state-explosion* problem, because the graph can be very large. On the other hand, on-the-fly testing requires a time-efficient reachability algorithm, since the tester must be able

[1] 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.

to react to the SUT in real-time. To alleviate this problem, we propose a technique which permits to reduce the number of symbolic operations required for each time observation, thus, accelerating the reaction time of the tester.

Generating one test is not enough, and generating complete (or exhaustive up to some depth) test suites is infeasible in practice. Thus, we propose a new technique to generate test suites with respect to a *coverage* criterion. Two standard coverage criteria are considered in this paper, namely, *location* and *edge* coverage, where locations and edges refer to the specification automaton. The technique relies on the fact that in the symbolic reachability graph, every node (resp. edge) can be associated to a set of locations (resp. edges) of the specification automaton. Thus, covering locations (resp. edges) of the specification reduces to covering nodes in the symbolic reachability graph.

We have implemented our framework in a prototype tool, called TTG, on top of the IF environment [8]. We have experimented with our tool on a few toy examples, as well as on a real case study, namely, the execution software of the K9 Rover by NASA [10].

The rest of this paper is organized as follows. Section 2 reviews the model of timed automata. Section 3 introduces the testing framework and shows how various specifications can be modeled and how conformance is formally defined. Section 4 shows how digital-clock tests can be generated. Section 5 presents the coverage technique. Section 6 discusses our tool and two case studies. Section 7 presents the conclusions and future work plans.

2 Timed Automata with Inputs, Outputs and Unobservable Actions

To model the specification, we use timed automata [2] with *deadlines* to capture urgency [7], and input, output and unobservable actions, to capture inputs, outputs and internal actions of the SUT.

As the TA model is well-known, we only give a brief overview here. In particular, we do not consider a rich discrete-variable state-space and we also omit discussion of how to compose TA. In practice, these features are essential for ease and clarity of modeling (they are indeed part of our tool, see Section 6). As mentioned in the introduction, specifications are usually modeled compositionally, using a *network of communicating TA*. There are different ways of communication, using shared variables, synchronizing actions (rendez-vous), FIFO queues, and so on. Moreover, each automaton can have other local variables than clocks, including booleans, integers, lists, etc. As long as the discrete state-space of the automaton remains finite, these features do not add to the expressiveness of the model.

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 $\mathsf{time}(a\,1\,b\,0.5) = 1.5$.

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 $s = (q, v)$, where $q \in Q$ and $v : X \rightarrow \mathsf{R}$ is a clock *valuation*. $\vec{0}$ 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 transitions of the form $(q, v) \xrightarrow{a} (q', v')$, where $a \in \mathsf{Act}$ and there is an edge (q, q', ψ, 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. 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'', 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 $v \models \psi$. 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 naturally extends to timed sequences. For example, $s \xrightarrow{a1b} s'$ if there exist

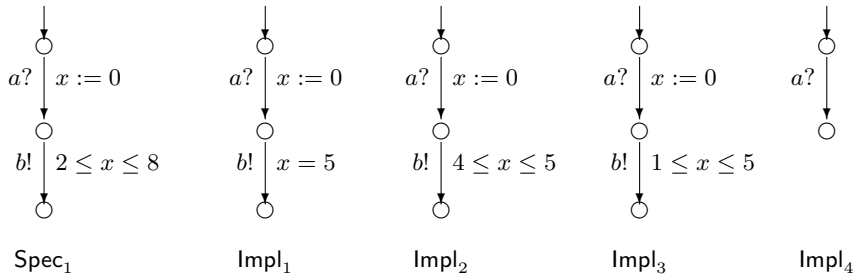

Figure 2: Examples of specifications and implementations.

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* $\mathsf{Act}_{\mathsf{in}}$ 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 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: $\forall s \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Act}_{\mathsf{in}} \,.\, s \xrightarrow{a}$. 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

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

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

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

3 Specifications 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 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, as we show below.

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.

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

$\sigma(A)$ 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

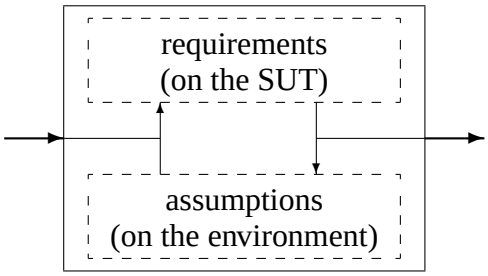

Figure 3: Specification including assumptions on the environment.

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. tioco is inspired from its "untimed" counterpart, ioco [30]. 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

$$\forall \sigma \in \mathsf{Traces}(A_S) \;.\; \mathsf{out}(\sigma(A_I)) \subseteq \mathsf{out}(\sigma(A_S)). \tag{3}$$

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.[2] Finally, it can capture assumptions on the environment. In the remaining of this section, we illustrate these features of tioco with simple examples.

In the examples, input actions are denoted $a?$, $b?$, ..., and output actions are denoted $a!$, $b!$, ... Unless otherwise mentioned, deadlines of output edges are delayable and deadlines of input edges 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 Spec_1 shown in Figure 2. 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". Thus, this specification requires that the output b is not emitted neither too early nor too late. Implementations Impl_1 and Impl_2 conform to Spec_1. Impl_1 produces b exactly 5 time units after reception of a. Impl_2 produces b sometime in the interval $[4, 5]$. Implementations Impl_3 and Impl_4 do not conform to Spec_1. Impl_3 may produce a b after 1 time unit, which is too early. Impl_4 fails to produce a b at all. Formally, letting $\sigma = a\,1$, we have $\mathsf{out}(\sigma(\mathsf{Impl}_3)) = (0, 4] \cup \{b\}$ and $\mathsf{out}(\sigma(\mathsf{Impl}_4)) = (0, \infty)$, whereas $\mathsf{out}(\sigma(\mathsf{Spec}_1)) = (0, 7]$.

Impl_4 offers an example of the "observable deadlock" situation mentioned above. "Doing nothing" is not an option for the SUT, since doing nothing is equivalent to letting time pass. But time is observable to the tester, which timeouts when the deadline for producing an output is violated. This example also illustrates how a real-time framework such as the one based on tioco handles deadlocks and timeouts in a seamless way, without the need of adding modeling artifacts.

Often, the SUT is supposed to operate correctly only in a particular environment, not in any environment. This brings up the issue of how to incorporate *assumptions* on the environment when building a

[2] 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?

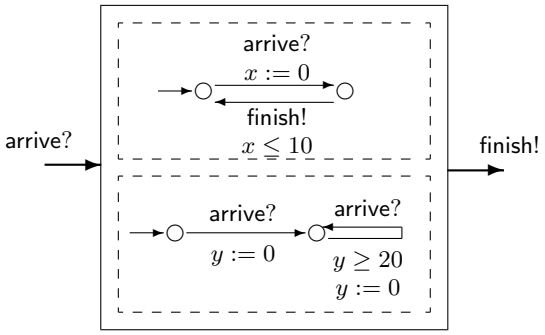

Figure 4: A task with minimal inter-arrival time 20 and deadline 10.

model of specification. Figure 3 shows how this can be done. The specification can be modeled compositionally, in two parts: one part modeling the environment (assumptions) and another part the nominal behavior of the SUT in this environment (requirements). In this case, the interactions between the two components are not unobservable, but are exported as inputs and outputs of the global specification.

Let us give a simple example of such a situation. Consider the following specification, ensuring schedulability of an aperiodic task in a typical real-time operating system: "assuming the minimal inter-arrival time of task A is 20 time units, the task must be executed within 10 time units". This specification can be modeled as shown in Figure 4. Notice that environment assumptions generally make the specification non-input-complete. In the above example, the second arrive input cannot be accepted until at least 20 time units have elapsed since the first arrive.

Modeling input/output variables

The TA model we have presented uses the notion of input/output *actions* (or events). In practice, many systems communicate with the external world using input/output *variables*. This situation can also be modeled without difficulty in our framework.

There are basically two possibilities to specify real-time requirements related to variables. One is to refer to variable *updates* and the other to refer to value *durations*. The first can be modeled in our framework using an action for each update. The second can be modeled using a "begin" action for the point in time where a variable changes its value to the value that is of interest and an "end" action for the moment where the variable changes to a different value.

For example, assume x is an input variable and y an output variable. Consider the requirement "y will be updated at most 10 time units after x is updated". Notice that x is updated by the environment (or the tester) while y is updated by the SUT. Thus, update_x can be introduced as an input action and update_y as an output action. The specification can be modeled as a TA similar to the one for Spec_1 of Figure 2, with $a?$ replaced by $\mathsf{update}_x?$ and $b!$ replaced by $\mathsf{update}_y!$.

This way of modeling supposes that updates are immediately perceived (by the SUT or by the tester) when they occur. This is not always true. For instance, a sampling controller typically reads its inputs only periodically (but may write the outputs as soon as they are ready). In this case, it could be that the specification only requires that the output be produced at most 10 time units after the input is sampled by the controller, not after it is updated by the environment. This situation can also be modeled in our framework by explicitly adding automata modeling the sampling (either at the SUT side, or at the tester side, or both). In fact, we will add such an automaton, called the Tick automaton (see Figure 6), to the specification, in order to generate digital-clock tests, as shown in the section that follows. The Tick automaton models sampling at the tester side. A similar automaton can be used to model sampling at the SUT side, with the difference that the tick event would in this case be an input event.

```
Input := nil;
State := init_state();
Alarm := set_alarm( State );
loop
  await( Alarm or Input );
  State := update( State, Alarm, Input );
  Verdict := check( State );
  if ( Verdict = FAIL or Verdict = PASS ) then
    announce( Verdict );
    stop;
  end if;
  Output := decide_output( State );
  if ( Output <> nil ) then
    emit( Output );
    State := update( State, Output );
  end if;
  Alarm := set_alarm( State );
end loop
```

Figure 5: Pseudo-code of a generic timed tester.

Other conformance relations

We end this section with a few comments on other conformance relations, such as *timed bisimulation* (e.g., considered in [29, 13]), *timed trace equivalence* (e.g., in [24, 22]), or *a must/may preorder* [27]. We believe that these relations are not appropriate for real-time testing, because they are too strict. For instance, none of $\mathsf{Impl}_1, \mathsf{Impl}_2$ conform to Spec_1 with respect to any of the above relations. This is because Spec_1 does not specify what should happen if a second input a is received before the output b is emitted. Thus, it is safe for Impl_1 and Impl_2 to ignore all but the first input a (recall that we implicitly assume input-completeness of Impl_1 and Impl_2). This results, however, in traces such as $a\,1\,a\,1\,b$, which are in the implementation but not in the specification, hence, non-conformance with respect to the above relations. For an extensive discussion of various untimed conformance relations, see [31].

4 Testing

A test is an experiment performed on the SUT by an agent, the *tester*. A generic algorithm for a real-time tester is shown in Figure 5. The tester maintains a current state and has access to an alarm (more than one alarms are also possible). The tester is triggered by two types of events: inputs received from the SUT or expirations of the alarm. After initializing its state and setting the initial expiration date of the alarm, the tester enters the following loop:

– the tester awaits for a trigger (an input or the expiration of the alarm);

– when the trigger occurs, the tester updates its state accordingly;

– then it checks whether the test must stop with a PASS or FAIL verdict;

– if the test should continue, the tester decides whether to emit an output to the SUT and if so, it emits the output and updates its state;

– finally, the tester resets the alarm and re-enters the loop.

Notice that a periodic-sampling tester is a special case of the above generic tester. It resets its alarm always to the same value (the period) and reads the input when triggered by the alarm (and upon updating its state).

In building a concrete tester out of this generic scheme, we must answer a number of questions: *what is the initial state*? *how is the state updated*? *how is the verdict computed*? *when to emit an output and which output if many are possible*? *how to set the alarm*? We provide provide answers to these questions

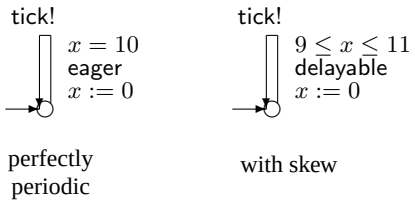

Figure 6: Possible Tick automata.

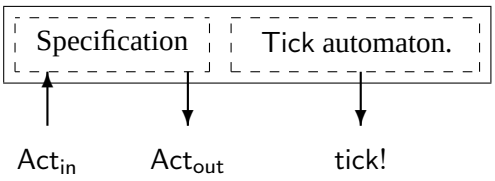

Figure 7: Specification composed with the Tick automaton.

in the sequel.

The first step in building the tester is to compose the specification automaton A_S with a Tick automaton. The latter models the digital clock of the tester. Two possible Tick automata are shown in Figure 6. The one on the left models a perfectly periodic clock while the one on the right models a clock with a skew. Notice that the time units in the Tick automaton must be in accordance with the time units in A_S (if necessary, scaling must be applied to bring all constants to integer values). The composition of A_S with Tick yields a new TAIO, denoted A_S^{Tick}. The latter is depicted in Figure 7. It has as inputs the inputs of A_S and as outputs the outputs of A_S plus the new output tick. Indeed, since the tester cannot measure time precisely, but only read the digital clock, we can consider that the tester reacts to time simply by observing tick (possibly counting the number of tick events it observes).

A state of the tester will be a set S of states of A_S^{Tick}. S represents the *uncertainty* of the tester on which state the specification could be according to the history of observations. Note that S does not contain states of the SUT (since its model is unknown) but states of A_S^{Tick}. The initial state of the tester will be the set containing the singleton $s_0^{A_S^{\mathsf{Tick}}}$, i.e., the initial state of A_S^{Tick}. The tester updates S with each input $a \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$ (notice that inputs of the tester are outputs of the specification). If S becomes empty, this implies that a is not allowed by the specification, given the present history of observations. Thus, the SUT is non-conforming and the tester issues a FAIL verdict.

The updates are based on the operator defined below, where $a \in \mathsf{Act} \cup \{\mathsf{tick}\}$:

$$\mathsf{succ}(S, a) = \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \,.\, s \xrightarrow{\rho} \xrightarrow{a} s'\}. \tag{4}$$

$\mathsf{succ}(S, a)$ contains all states which can be reached by some state in S via a sequence of unobservable actions ending with the observable action a. Let $\mathsf{succ}_k(\cdot)$ be a shorthand notation for the application of $\mathsf{succ}(\cdot, \mathsf{tick})$ k times. For example, $\mathsf{succ}_2(S) = \mathsf{succ}(\mathsf{succ}(S, \mathsf{tick}), \mathsf{tick})$.

The updates will be performed as follows. Suppose the current state of the tester is S. Suppose the alarm reads k and it is set to expire at $m > k$. If the alarm expires with no input being received meanwhile, this means that $m - k$ tick events occurred, thus, the next state of the tester is computed as $S' = \mathsf{succ}_k(S)$. If an input $a \in \mathsf{Act}_{\mathsf{out}}$ is received before the alarm expires, when the alarm reads n, $m > n > k$, then $n - k$ ticks occurred followed by a, thus, the next state of the tester is computed as $S' = \mathsf{succ}(\mathsf{succ}_n(S), a)$. As said above, if ever $S' = \emptyset$, the tester announces FAIL and stops.

The tester also uses succ to check whether to emit an output and update its state in such a case. For each $a \in \mathsf{Act}_{\mathsf{in}}$, the tester computes a temporary value $S_a = \mathsf{succ}(S, a)$. If $S_a = \emptyset$, then a is not valid. This means that issuing a would violate the specification assumptions on the inputs, thus, it is useless to issue a at this point. On the other hand, if S_a is non-empty, then a can be emitted and the state updated to S_a.

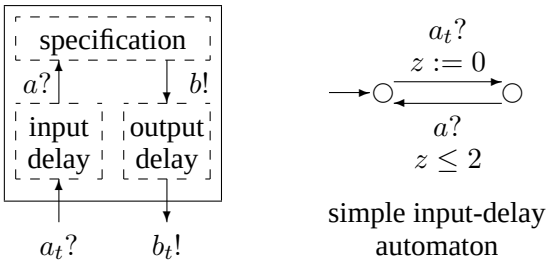

Figure 8: Specification composed with I/O delay automata.

Modeling interfacing delays

As mentioned in the introduction, it is often the case that there are interfacing delays between the tester and the SUT. Unless such delays are taken into account, the tester may issue wrong verdicts. Luckily, the TA model allows such delays to be captured directly in the specification. This can be done in the same way as in modeling environment assumptions or tester sampling directly in the specification. The specification is composed with input/output delay automata, as shown in Figure 8. A simple input delay automaton is shown to the right of the figure. Input action a is the original action whereas a_t is the output command of the tester. This automaton models the assumption that the tester output may experience a delay of at most 2 time units until it is perceived by the SUT. Notice that this automaton does not allow a new input to be produced while the previous one is still in "transit". For this, a more complicated automaton is necessary, which buffers input events. We omit it due to lack of space.

Symbolic tester implementation

Sets of states of a timed automaton can be represented *symbolically* using simple polyhedra to encode the continuous state-space. For instance, the polyhedron $1 \leq x \leq 2 \wedge x = y$ represents the fact that clock x reads some value within $[1, 2]$ and clock y is equal to x. Such polyhedra can be implemented using various data structures, such as DBMs (*difference bound matrices*) [18]. Computing updates using the succ operator can be also done symbolically, using a *bounded-time reachability analysis* for timed automata, as shown in [32]. Reachability algorithms are standard technology in timed automata verification tools such as Kronos [17], thus, we do not discuss them further here.

On-the-fly testing versus static test generation

The testing method presented above can be applied in an *on-the-fly* or in a *static* manner. On-the-fly testing means that the tester computes its state and makes decisions such as whether to emit an output or wait during the execution of the test. In static test generation, these choices are resolved off-line and are encoded in a test represented in the form of a finite tree, like the one shown in Figure 9. Nodes in this tree are either "input" or "output". In an input node, the tester waits for an input from the SUT or for the next tick of its digital clock. In an output node, the tester emits an output to the SUT. Leaves are labeled PASS or FAIL. In the context of "untimed" systems, on-the-fly testing is supported by the tool Torx [4] and static test generation by the tool TGV [20].

On-the-fly testing is advantageous in what concerns memory, since the tester need only keep the current state, whereas the number of nodes in the static test tree can explode. On the other hand, static test execution is faster, since the tester need only move to a child node in the tree, whereas a reachability analysis needs to be performed for each application of succ. This can sometimes be costly, especially when succ_k is computed, for large k.

A technique to tackle this problem is to use a *parametric* Tick automaton, as shown in Figure 10. In such an automaton, the period is defined by one or more parameters, e.g., P in the exactly periodic automaton or $[L, U]$ in the automaton with skew. These parameters are instantiated upon execution, when their value is known. For example, if we use the exactly periodic automaton and we know that 1023 ticks

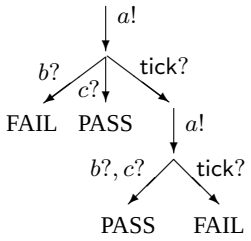

Figure 9: A test represented statically as a tree.

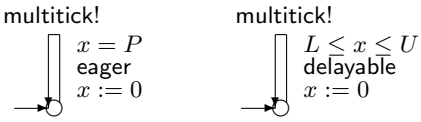

Figure 10: Parametric Tick automata.

occurred, we can set $P = 1023$. In the skewed clock automaton, we would set $L = 1023 \cdot (1 - \epsilon)$ and $U = 1023 \cdot (1 + \epsilon)$, where ϵ is the skew. In this case, the multitick event no longer represents a single tick, but 1023 ticks. The advantage is that we now need apply the symbolic successor operator $\mathsf{succ}(S, \mathsf{multitick})$ only once, instead of 1023 times. Note that this technique relies on the fact that our prototype tool (discussed in Section 6) allows parametric automata to be created and destroyed dynamically, which is a feature few such tools possess.

Soundness and strictness

The tests generated by the method presented above are *sound*, in the sense that if the tester announces FAIL and the SUT is indeed non-conforming to the specification. Obviously, this is a minimal correctness requirement from a testing framework. In fact, our tests satisfy a stronger property than soundness (which can be trivially satisfied by a test always announcing PASS), called *strictness* [25]. Informally, a test is strict if, every time it outputs PASS, then indeed the behavior of the SUT was a conforming behavior according to the specification. In other words, the subset of the SUT tested by this particular test is guaranteed to be conforming.

5 Coverage

A test suite is *complete* if it is sufficient to guarantee conformance (i.e., passing all tests in the suite implies that the SUT is conforming). It is generally impossible to generate a finite complete test suite.[3] Even when this is possible in principle (e.g., by assuming an upper bound on the number of states of the implementation, it is usually infeasible in practice, because the number of tests required is prohibitively large [29]).

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. Different coverage criteria have been proposed for software, such as statement coverage (every statement of the program must be "explored" by at least one test), branch coverage, and so on (see, for instance, the survey [35]). In the TA case the state space is infinite, thus, existing methods attempt to cover either finite abstractions of the state space or structural elements of the specification. For instance, [29, 19] cover the *region graph* abstraction [2] and [27] cover the *time-abstracting partition graph* [34]. [22] propose techniques for edge, location, or definition-use pair coverage and can also generate time-optimal tests. [9] consider various coverage criteria in the context of timed Petri nets.

[3] 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.

Here, we propose a new technique for edge or location coverage. Notice that we cannot use the technique of [22]. This technique relies on the assumption that outputs in the specification 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. Finding a test can then be reduced to finding a run from a source to a target state, which can be done using standard reachability algorithms for timed automata.

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 $\{s_0^{A_S^{\mathsf{Tick}}}\}$. 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. Using arguments similar to the finiteness of the *region graph* [2], it can be shown that the observable graph is finite. Its size, however, can be exponential in the number of clocks.[4]

Every node of OG corresponds to a set of states S of A_S^{Tick}. Each state $s \in S$ includes a location of A_S. Thus, every node of OG can be associated with a set of locations of A_S, which can be computed while generating OG. On the other hand, every static test tree is essentially a sub-graph of OG. We say that such a test *covers* the set of locations associated with all nodes of OG appearing in the test. We say that a set of tests (or *test suite*) achieves *location coverage* if every reachable location of A_S is covered by some test in the suite.[5] Clearly, since OG is finite and the set of locations of A_S is also finite, a finite number of tests suffices to achieve location coverage.

Similarly, 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 $s \xrightarrow{\rho} \xrightarrow{a} s'$ for an un observable 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 is covered by some test in the suite. As with location coverage, a finite number of tests suffices to achieve edge coverage.

The above definitions imply a straightforward algorithm to generate a test suite which achieves location or edge coverage. The first step is to build the observation graph of A_S^{Tick}. Then, tests are extracted statically from OG, until coverage is achieved. Tests are extracted as follows.

While there are 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. 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.

An edge-covering test suite can be extracted in a similar way. The only difference is that instead of finding a path reaching a target node of OG, the algorithm must find a path reaching a target edge. Notice that these algorithms are very similar to an AND/OR search in a finite graph (AND for tester inputs and OR for tester outputs).

The worst-case complexity of the above algorithm is polynomial (quadratic) 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 locations (or edges) in A_S.

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 test suites.

[4] This is an inherent worst-case complexity barrier, even for the simplest analysis problems in timed automata (e.g., reachability). In practice, symbolic methods have permitted to treat industrial case studies of moderate size (tens of components).

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

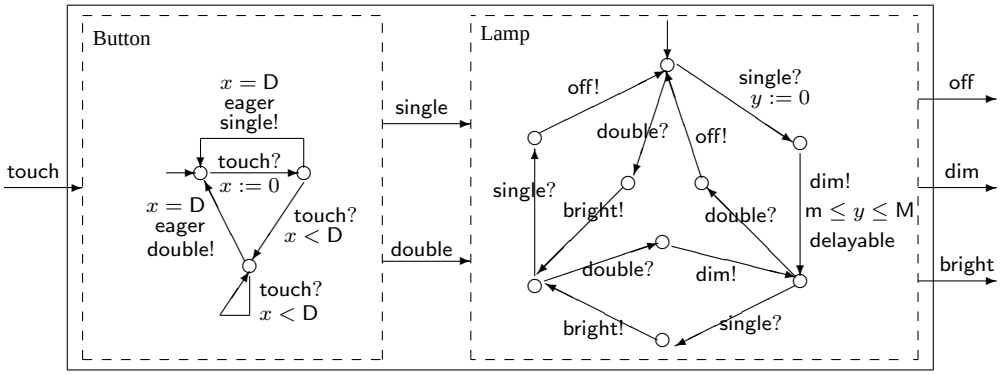

Figure 11: A lighting device.

6 Tool and case studies

We have built a prototype test-generation tool, called TTG, on top of the IF environment [8]. 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 [20]. 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 digital-clock tests with respect to a given (parametric) Tick automaton. By modifying the Tick automaton, the user can implement different sampling rates, model skew or jitter in the clock, and so on. TTG can be executed in *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. Implementation of test selection criteria is underway.

6.1 A toy example

We have applied TTG to a small case study, which is a modification of the light switch example presented in [22]. The (modified) specification is shown in Figure 11. 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 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. Notice that a single test

suffices to achieve location-coverage of this specification: it first applies three single touches and then three double touches.

One of the tests generated by TTG is shown in Figure 12. The drawing has been produced automatically using the `if2eps` tool by Marius Bozga.

6.2 The K9 Rover case study

We have also used TTG to test the executive subsystem of the Mars rover controller K9, developed at NASA Ames. For a more extensive description of the case study, the reader is referred to [10]. Our treatment is presented in detail in [5]. Here, we only present a brief overview.

The Rover executive is a multi-threaded program that consists of approximately 35000 lines of C++ code, of which 9600 lines of code are related to actual functionality. The executive is responsible of executing a given *plan*. The plan defines the steps to be performed in the mission, and also gives detailed information about their order, their timing, what to do in case of failures, and so on. Thus, the plan can be taken to be the specification. On the other hand, the SUT is the executive fed with this plan as input. Indeed, executing the plan on the executive must produce a behavior which meets the requirements specified in the plan.

We have automatically translated various plans into timed automata specifications using the technique of [1]. Notice that this technique is compositional, thus, the resulting TA contained unobservable internal events. From each TA specification, we generated a tester using TTG. Then we applied the tester to a number of "log-traces" generated by the executive on this plan, kindly provided to us by people at NASA. The results were very encouraging. In all cases, the tester found out the bugs that were already known to be contained in the traces. Generation and execution of each tester took a matter of seconds.

7 Summary and future work

We have proposed a testing framework for real-time systems based on partially-observable, non-deterministic timed-automata specifications and on digital-clock tests. To our knowledge, this is the first framework that can fully handle such specifications and such tests. We introduced a timed version of the input-output conformance relation of [30] and proposed techniques to generate location- or edge-covering test suites. We reported on a prototype tool and two case-studies.

We are currently implementing in TTG the coverage technique discussed in Section 5 and are also studying methods to generate minimal test suites. We are also examining heuristics to choose tester output times on-the-fly. This can permit to guarantee some coverage in on-the-fly testing, by performing multiple tests and using appropriate book-keeping, without having to generate the symbolic reachability graph.

References

[1] A. Akhavan, S. Bensalem, M. Bozga, and E. Orfanidou. Experiment on verification of a planetary rover controller, 2003. Submitted. 6.2

[2] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 1, 2, 2

[3] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In *CAV'94*, volume 818 of *LNCS*. Springer, 1994. 1

[4] A. Belinfante, J. Feenstra, R.G. de Vries, J. Tretmans, N. Goga, L. Feijs, S. Mauw, and L. Heerink. Formal test automation: A simple experiment. In 12^{th} *Int. Workshop on Testing of Communicating Systems*. Kluwer, 1999. 1

[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. 6.2

[6] 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. 1

[7] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer, 1998. 2

[8] 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. 1, 6

[9] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. In *International Software Quality Week*, 1997. 1, 2

[10] 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. 1, 6.2

[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. Conformance test experiments for distributed real-time systems. In *ISSTA'02*. ACM Press, 2002. 1

[13] 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

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

[15] 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

[16] 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

[17] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool Kronos. In *Hybrid Systems III, Verification and Control*, volume 1066 of *LNCS*, pages 208–219. Springer-Verlag, 1996. 1, 4

[18] 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. 4

[19] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS'98*. IEEE, 1998. 1, 2

[20] 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, 6

[21] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992. 1

[22] 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, 1, 2, 6.1

[23] 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

[24] A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES'03*, Montreal, October 2003. 1, 1

[25] 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, 4

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

[27] B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS'01*. LNCS 2031, Springer, 2001. 1, 1, 2

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

[29] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254, 2001. 1, 1, 2

[30] 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. 1, 3, 7

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

[32] 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, 4

[33] S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, 2003. To appear in LNCS series by Springer. 1

[34] S. Tripakis and S. Yovine. Analysis of timed systems using time-abstracting bisimulations. *Formal Methods in System Design*, 18(1):25–68, January 2001. 2

[35] H. Zhu, P. Hall, and J. May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4), 1997. 2

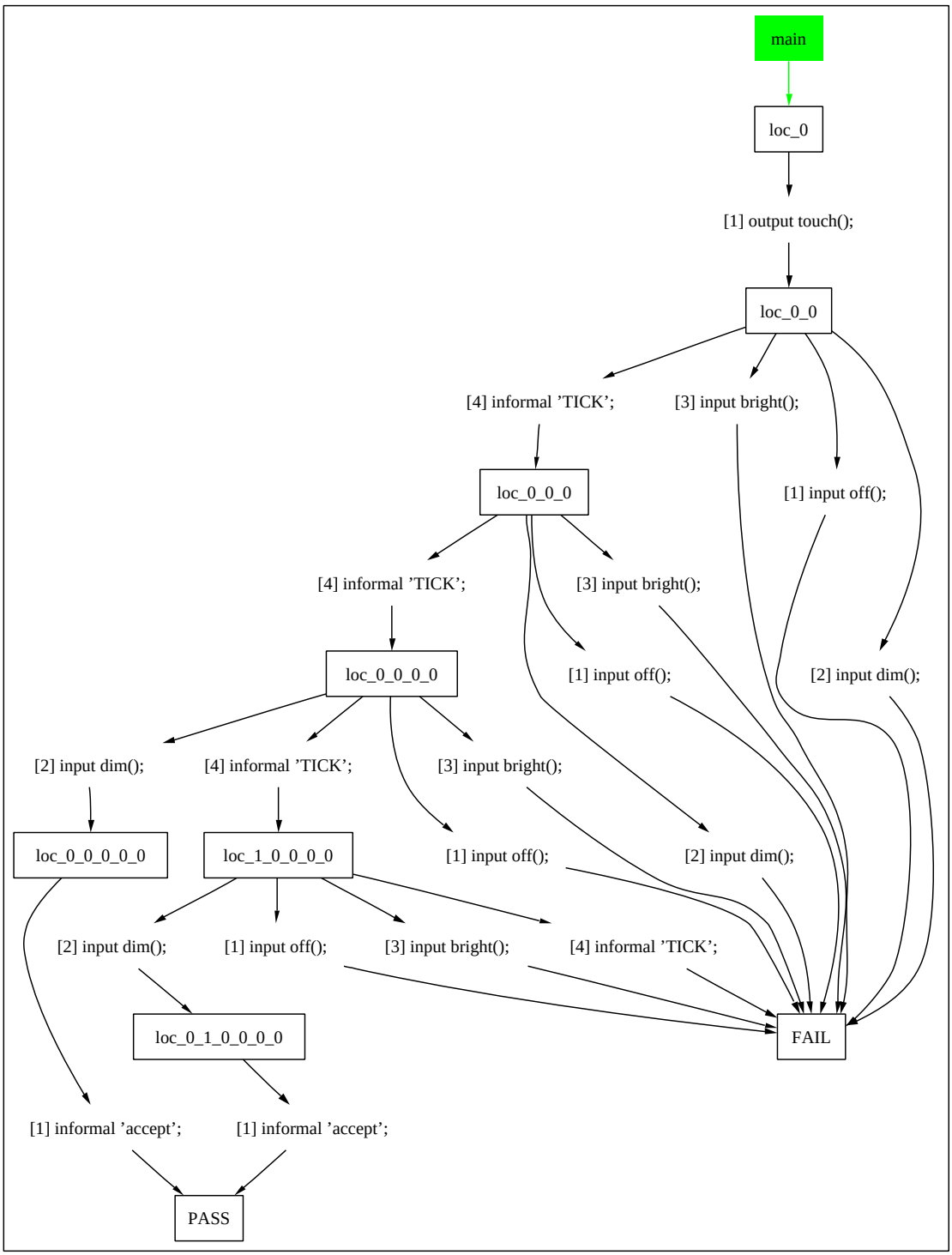

Figure 12: A test generated automatically by TTG.