Conformance Testing for Real-Time Systems

Moez Krichen* Stavros Tripakis†

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. The framework allows the user to define, through appropriate modeling, assumptions on the environment of the system under test (SUT) as well as on the interface between the tester and the SUT. We consider two types of tests: analog-clock tests and digital-clock tests. Our algorithm to generate analog-clock tests is based on an on-the-fly determinization of the specification automaton during the execution of the test, which in turn relies on reachability computations. The latter can sometimes be costly, thus problematic, since the tester must quickly react to the actions of the system under test. Therefore, we provide techniques which allow analog-clock testers to be represented as deterministic timed automata, thus minimizing the reaction time to a simple state jump. We provide algorithms for static or on-the-fly generation of digital-clock tests. These tests measure time only with finite-precision, digital clocks, another essential condition for implementability. We also propose a technique for location, edge and state coverage of the specification, by reducing the problem to covering a symbolic reachability graph. This avoids having to generate too many tests. We report on a prototype tool TTG and two case studies: a lighting device and the Bounded Retransmission Protocol. Experimental results obtained by applying TTG on the Bounded Retransmission Protocol show that only a few tests suffice to cover thousands of reachable symbolic states in the specification.

Key words: conformance, testing, real-time, timed automata, specification, analog, digital, coverage, tool, case studies.

*Verimag Laboratory, Centre Equation, 2, avenue de Vignate, 38610 Gières, France. Web: http://www-verimag.imag.fr. Email: krichen@imag.fr

†Verimag Laboratory and Cadence Berkeley Labs, 1995 University avenue, Suite 460, Berkeley, CA 94704, USA. Web: http://www.cadence.com/company/cadence_labs. Email: tripakis@cadence.com

Contents

List of Figures

List of Algorithms

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*). There exist many types of testing with multiple aims, from checking correct functionality to measuring performance. In this work, we are interested in so-called *conformance testing* where the aim is to check conformance of the SUT to a given specification. The SUT is often a "black box" in the sense that we do not have knowledge about its internals (e.g., its state is not known) , thus, can only rely on its observable input/output behavior.

We focus our attention on *real-time systems*. These are systems that operate in an environment with strict timing constraints. Examples of such systems are many: embedded control systems from the automotive, aerospace or other domains, mobile phones, multimedia systems, and so on. When testing a real-time system, 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. Moreover, the timing of these outputs depends on the timing of the inputs. In turn, the timing of applicable future inputs is determined by the outputs.

Classical testing frameworks are based on Mealy machines (e.g., see [19, 39]) or finite labeled transition systems (e.g., see [46, 16, 26, 3, 20]). These formalisms are not well-suited for modeling real-time systems. In Mealy machines, inputs and outputs are synchronous, which is a reasonable assumption when modeling synchronous hardware, but not when the delays among inputs and outputs are governed by complex timing constraints. In testing methods based on LTSs, time is typically abstracted away and timeouts are modeled by special δ actions [45] which can be interpreted as "no output will be observed" (this is the property of *quiescence*). 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 real-time systems is the model of *timed automata − TA* [1]. 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 [14, 21, 25, 30, 41, 44, 42, 18, 33, 29, 38, 17]). However, these methods present one or both of the following two limitations.

First, only restricted subclasses of timed automata are considered, thus limiting the expressiveness of specifications. For example, [44, 29] 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, [41] use *event-recording automata* [2] and [33] 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 is that only *analog-clock* tests are considered in the works above. These are tests which can observe the delays between inputs and outputs precisely. For example, a test like "*emit a; if b is received at most 5 time units later, 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 measure the delay t between a and b and announce PASS if and only if $t \leq 5$. In practice, the tester will typically sample time periodically, say, every 0.1 time units. Thus, it cannot distinguish between t being anywhere in the interval $(4.9, 5.1)$. In this case, in order to be sound, the tester should announce PASS for every t in this interval, thus accepting also some non-conforming specifications (those where $5 < t < 5.1$). A modeling framework which allows to formally specify such implementation considerations and derive sound tests from them is not provided in the above works.

In this work, we propose a 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 digital clocks of finite precision. Our framework has been presented in previous publications [34, 35, 36]. This paper unifies and extends the results presented in these publications.

We next summarize the main characteristics of our framework.

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. This greatly simplifies modeling. Figure 1 illustrates this fact: it shows a specification (solid-line box) communicating with the external world through some observable interface (solid arrows). The specification is built internally using three components (dotted boxes). These components communicate using signals that are unobservable to the external world (dotted arrows). *Abstractions* from low-level details are also used often, to simplify modeling and manage complexity. Such abstractions could, for instance, "hide" some variables and behavior, which typically results in non-determinism.

In general, timed-automata cannot be determinized [1] 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 [48]. 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".

In order to capture conformance of a SUT to a specification we propose a formal relation, called *timed input-output conformance* or tioco. The latter is inspired from the "untimed" conformance relation ioco of [45]. The main difference is that ioco uses the notion of quiescence, according to which absence of outputs is observable. In tioco we do not use quiescence because we want timeouts to be explicitly specified as said above. For instance, we do not allow specifications stating "a must *eventually* occur" but only "a must occur with x time units". Apart from this important difference, tioco is similar in spirit to ioco: intuitively 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. In

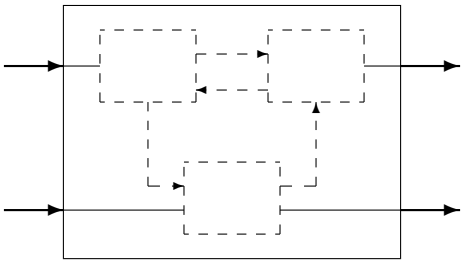


Figure 1: A compositional specification with internal (unobservable) actions.

tioco time delays are included 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.

We consider both analog-clock and *digital-clock* tests. The former measure time precisely, whereas the latter can only count how many "ticks" of a digital clock have occurred between two events. In our framework, the digital clock is itself modeled as a timed automaton. In that way, the user has full control on the assumptions about the execution platform where the tester will execute. We provide examples that show how to model different types of digital clocks, from simple strictly-periodic clocks to more complex clocks with jitter or skew. Note that generating digital-clock tests does not mean that we assume a discrete-time setting. Indeed, the specification is still continuous-time. The SUT operates in dense time as well. Only the tester (which is a digital system) is sampling this time using a digital clock.

We propose algorithms to generate tests both *on-line* (or *on-the-fly*) and *off-line*. On-line test generation means that the test is generated essentially during execution. Thus, a large number of computations and choices must be performed and resolved on-the-fly: compute the next state of the tester, decide whether to wait and for how long, etc. In off-line test generation these choices are resolved at generation time. The test is usually represented as a finite tree with PASS/FAIL annotations on the leaves. All that the tester has to do then during test execution is follow this tree. On-line versus off-line test generation is essentially a time versus space trade-off: on-line generation saves space at the expense of requiring more time during the execution of the test (reaction time). This in turn limits the *reactivity* of the tester to the actions of the SUT. On the other hand, off-line test generation requires space to store the generated tests.

Classic test generation algorithms use a pre-processing step consisting in determinizing the specification [26]. For reasons explained above, this is impossible in the case of timed automata. To solve this problem, we employ different techniques. In the case of on-line generation of analog-clock tests, we use an *on-the-fly subset construction* technique. This consists in determinizing the specification on-the-fly, based on the sequence of observed time delays and discrete actions. This technique has been first introduced in [47] where it was used for monitoring and fault-detection purposes.

The case of off-line generation of analog-clock tests is tricky. The question is how to represent analog-clock tests in a "static" way? An immediate thought is to use timed automata. However, the tester automata must be deterministic, otherwise determinization must be performed during test execution, and we are back to the case of on-the-fly generation. But generating a deterministic test from a non-deterministic specification raises the undecidability issues discussed above. We thus take a pragmatic approach. We suppose

that the user fixes the number of clocks that the tester automaton has and also the points where these clocks are to be reset (we call the latter a *reset skeleton*). The user must also specify the maximum constants the clocks are to be compared to in the guards of the tester. We provide an algorithm which, given this information, generates a test represented as a deterministic timed automaton.

Generation of digital-clock tests (both on-line and off-line) is based on the following idea: if we consider the "tick" of the digital clock as any other discrete action, then digital-clock tests become ordinary "untimed" automata. Such automata can be generated from special "untiming" constructions of the product of the two input timed automata: the specification and the digital clock model.

All our test-generation methods rely on *symbolic reachability* algorithms similar to those used in timed automata model-checking tools such as Kronos [22]. Test generation suffers from the state-explosion problem less than model-checking. On the other hand, the number of tests that can be generated from a given specification is infinite! Even up to a given depth, the number of possible tests is exponentially large. To limit explosion in the number of tests, we consider *coverage criteria* such as state or transition coverage, inspired from software testing [40]. These criteria can drastically reduce the number of generated tests: the tests required to cover a specification are often a very small subset of the set of all possible tests up to a given depth. We provide algorithms to generate digital-clock tests with respect to *location, state or edge* coverage. The algorithms are based on the fact that in the symbolic reachability graph of a timed automaton every node or edge can be associated to a set of locations, states or edges of the specification automaton. Thus, covering locations, states or edges of the specification reduces to covering nodes or edges in the symbolic reachability graph.

We have implemented our real-time testing framework in a prototype tool called TTG. TTG is implemented on top of the IF environment [12]. We have experimented with TTG on a few case studies, including the execution software of the K9 Rover by NASA [15, 5] and the Bounded Retransmission Protocol [27, 36].

The rest of this document is organized as follows. In Section 2 we introduce timed automata with inputs and outputs. In Section 3 we define the conformance relation tioco and illustrate its usage with examples. Section 4 provides a description of analog-clock and digital-clock tests. In Section 5 we show how to automatically generate such tests, using on-line or off-line methods. Section 6 presents test generation based on coverage criteria. Section 7 discusses our tool and two case studies. In Section 8 we present our conclusions and future work plans.

2 Timed Automata with Inputs and Outputs

2.1 Real-Time Sequences

Let R be the set of non-negative reals and Q the set of non-negative rationals. Given a finite set of *actions* Act, the set $(\mathsf{Act} \cup \mathsf{R})^*$ of all finite-length *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 $\mathsf{time}(a\,1\,b\,0.5) = 1.5$.

In the rest of the document, we assume given a set of actions Act, partitioned in two

disjoint sets: a set of *input actions* $\mathsf{Act_{in}}$ and a set of *output actions* $\mathsf{Act_{out}}$. Actions in $\mathsf{Act_{in}} \cup \mathsf{Act_{out}}$ are called *observable* actions. We also assume there is an *unobservable* action $\tau \notin \mathsf{Act}$. Let $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$.

2.2 Timed Labeled Transition Systems

A *timed labeled transition system* (TLTS) over Act is a tuple $(S, s_0, \mathsf{Act}, T_d, T_t)$, where:

- S is a set of *states*;
- s_0 is the initial state;
- T_d is a set of *discrete transitions* of the form (s, a, s') where $s, s' \in S$ and $a \in \mathsf{Act}$;
- T_t is a set of *timed transitions* of the form (s, t, s') where $s, s' \in S$ and $t \in \mathsf{R}$.

Timed transitions must be deterministic, that is, $(s, t, s') \in T_t$ and $(s, t, s'') \in T_t$ implies $s' = s''$. T_t must also satisfy the following conditions:

- $(s, t, s') \in T_t$ and $(s', t', s'') \in T_t$ implies $(s, t + t', s'') \in T_t$;
- $(s, t, s') \in T_t$ implies that for all $t' < t$, there is some $(s, t', s'') \in T_t$.

A given TLTS $(S, s_0, \mathsf{Act}, T_d, T_t)$ is said to be *rational-delay* if for each timed-transition $(s, t, s') \in T_t$ the duration t is rational (i.e., $t \in \mathsf{Q}$).

We use standard notation concerning TLTS. For $s, s', s_i \in S$, $\mu, \mu_i \in \mathsf{Act}_\tau \cup \mathsf{R}$, $a, a_i \in \mathsf{Act} \cup \mathsf{R}$, $\rho \in \mathsf{RT}(\mathsf{Act}_\tau)$ and $\sigma \in \mathsf{RT}(\mathsf{Act})$, we have:

- General transitions:
  - $s \xrightarrow{\mu} s' \stackrel{Def}{=} (s, \mu, s') \in T_d \cup T_t$;
  - $s \xrightarrow{\mu} \stackrel{Def}{=} \exists s' : s \xrightarrow{\mu} s'$;
  - $s \not\xrightarrow{\mu} \stackrel{Def}{=} \nexists s' : s \xrightarrow{\mu} s'$;
  - $s \xrightarrow{\mu_1 \cdots \mu_n} s' \stackrel{Def}{=} \exists s_1, \cdots, s_n : s = s_1 \xrightarrow{\mu_1} s_2 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_n = s'$;
  - $s \xrightarrow{\rho} \stackrel{Def}{=} \exists s' : s \xrightarrow{\rho} s'$;
  - $s \not\xrightarrow{\rho} \stackrel{Def}{=} \nexists s' : s \xrightarrow{\rho} s'$.
- Observable transitions:
  - $s \stackrel{\epsilon}{\Rightarrow} s' \stackrel{Def}{=} s = s'$ or $s \xrightarrow{\tau \cdots \tau} s'$;
  - $s \stackrel{a}{\Rightarrow} s' \stackrel{Def}{=} \exists s_1, s_2 : s \stackrel{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \stackrel{\epsilon}{\Rightarrow} s'$;
  - $s \stackrel{a}{\Rightarrow} \stackrel{Def}{=} \exists s' : s \stackrel{a}{\Rightarrow} s'$;
  - $s \stackrel{a}{\not\Rightarrow} \stackrel{Def}{=} \nexists s' : s \stackrel{a}{\Rightarrow} s'$;
  - $s \stackrel{a_1 \cdots a_n}{\Longrightarrow} s' \stackrel{Def}{=} \exists s_1, \cdots, s_n : s = s_1 \stackrel{a_1}{\Rightarrow} s_2 \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} s_n = s'$;
  - $s \stackrel{\sigma}{\Rightarrow} \stackrel{Def}{=} \exists s' : s \stackrel{\sigma}{\Rightarrow} s'$;
  - $s \stackrel{\sigma}{\not\Rightarrow} \stackrel{Def}{=} \nexists s' : s \stackrel{\sigma}{\Rightarrow} s'$.

A sequence of the form $s_0 \xrightarrow{\mu_1} s \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s'$ is called a *run* and a sequence of the form $s_0 \stackrel{a_1}{\Rightarrow} s \stackrel{a_2}{\Rightarrow} \cdots \stackrel{a_n}{\Rightarrow} s'$ an *observable run*.

2.3 Timed Automata

We use timed automata [1] with deadlines to model urgency [43, 9]. A *timed automaton over* 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*;
- 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 a set of clocks to *reset* to zero;
- $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline*;
- $a \in \mathsf{Act}$ is the action.

A timed automaton A defines an infinite TLTS which is denoted L_A. Its states are pairs $s = (q, v)$, where $q \in Q$ and $v : X \to \mathsf{R}$ 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. Discrete transitions are 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 are of the form $(q, v) \xrightarrow{t} (q, v + t)$, where $t \in \mathsf{R}, t > 0$ and there is no edge (q, q'', ψ, r, d, a), such that: (1) either $d = \mathsf{delayable}$ and there exist $0 \leq t_1 < t_2 \leq t$ such that $v + t_1 \models \psi$ and $v + t_2 \not\models \psi$; (2) or $d = \mathsf{eager}$ and $v \models \psi$. Notice that lazy edges do not impact the semantics, they are simply there to denote that an edge is neither delayable, nor eager. The latter two types do impact the semantics. Thus, lazy edges cannot block time progress, whereas delayable and eager edges can.

We will not allow delayable edges with guards of the form $x < c$ and eager edges with guards of the form $x > c$. For the former, there is no *latest* time when the guard is still true. For the latter, there is no *earliest* time when the guard becomes true.

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 $\mathsf{Reach}(A)$.

2.4 Timed Automata with Inputs and Outputs

A *timed automaton with inputs and outputs* (TAIO) is a timed automaton over the partitionned set of actions $\mathsf{Act}_\tau = \mathsf{Act}_{\mathsf{in}} \cup \mathsf{Act}_{\mathsf{out}} \cup \{\tau\}$. For clarity, we will explicitly include inputs and outputs in the definition of a TAIO A and write $(Q, q_0, X, \mathsf{Act}_{\mathsf{in}}, \mathsf{Act}_{\mathsf{out}}, \mathsf{E})$ instead of $(Q, q_0, X, \mathsf{Act}_\tau, \mathsf{E})$.

A TAIO is called *observable* if none of its edges is labeled by τ.

Given a set of inputs $\mathsf{Act}' \subseteq \mathsf{Act}_{\mathsf{in}}$, a TAIO A is called *input-enabled* with respect to Act' if it can accept any input in Act' at any state: $\forall s \in \mathsf{Reach}(A)\,.\,\forall a \in \mathsf{Act}' : s \xrightarrow{a}$. It is simply said to be input-enabled when $\mathsf{Act}' = \mathsf{Act}_{\mathsf{in}}$. A is called lazy-*input* with respect to Act' if the deadlines on all the transitions labeled with input actions in Act' are lazy. It is called lazy-input if it is lazy-input with respect to $\mathsf{Act}_{\mathsf{in}}$. Note that input-enabled does not imply lazy-input in general.

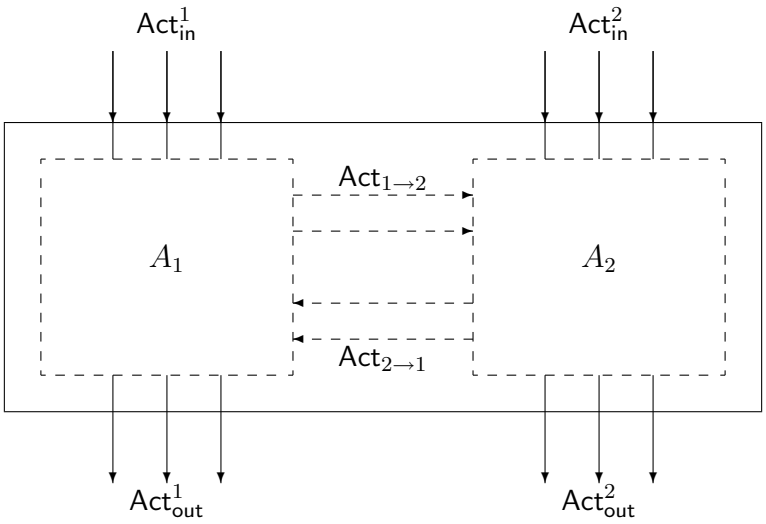


Figure 2: Two interacting TAIO.

A is called *deterministic* if

$$\forall s, s', s'' \in \mathsf{Reach}(A) \,.\, \forall a \in \mathsf{Act}_\tau \,:\, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Rightarrow s' = s''.$$

A is called *non-blocking* if

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

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

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

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

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

$$\mathsf{ObsTTraces}(A) = \{P_{\mathsf{Act}}(\rho) \mid \rho \in \mathsf{RT}(\mathsf{Act}_\tau) \wedge s_0^A \xrightarrow{\rho}\}. \tag{3}$$

The TLTS defined by a TAIO is called a *timed input-output LTS* (TIOLTS). From now on, unless otherwise stated, all the considered TAIO are defined with respect to the same sets $\mathsf{Act}_{\mathsf{in}}$ and $\mathsf{Act}_{\mathsf{out}}$ and unobservable action τ. As for TAIO, a given TIOLTS L is denoted $(S, s_0, \mathsf{Act}_{\mathsf{in}}, \mathsf{Act}_{\mathsf{out}}, T_d, T_t)$ instead of $(S, s_0, \mathsf{Act}_\tau, T_d, T_t)$.

2.5 Parallel composition of TAIO

Most of the time, it is easier to write models in a modular way. That is, to consider models which are the product of some interacting components. For that, we introduce the notion of parallel composition for the case of TAIO.

We are given two TAIO $A_1 = (Q_1, q_0^1, X_1, \mathsf{Act}^1_{\mathsf{in}} \cup \mathsf{Act}_{2\to 1}, \mathsf{Act}^1_{\mathsf{out}} \cup \mathsf{Act}_{1\to 2}, \mathsf{E}_1)$ and $A_2 = (Q_2, q_0^2, X_2, , \mathsf{Act}^2_{\mathsf{in}} \cup \mathsf{Act}_{1\to 2}, \mathsf{Act}^2_{\mathsf{out}} \cup \mathsf{Act}_{2\to 1}, \mathsf{E}_2)$. The pair of TAIO (A_1, A_2) is said to be *compatible* with respect to the pair of action sets $(\mathsf{Act}_{1\to 2}, \mathsf{Act}_{2\to 1})$ if $X_1 \cap X_2 = \emptyset$, the sets $\mathsf{Act}^1_{\mathsf{in}}$, $\mathsf{Act}^1_{\mathsf{out}}$, $\mathsf{Act}^2_{\mathsf{in}}$, $\mathsf{Act}^2_{\mathsf{out}}$, $\mathsf{Act}_{1\to 2}$ and $\mathsf{Act}_{2\to 1}$ are pairwise disjoint and A_i is input-enabled with respect to $\mathsf{Act}_{(3-i)\to i}$, for $i = 1, 2$. This is illustrated in Figure 2. Incoming arrows denote input events and outgoing arrows output events. Solid lines denote

observability and dotted lines non-observability. We also assume that each A_i is input-enabled with respect to $\mathsf{Act}_{(3-i)\to i}$ in order to avoid having time blocked due to internal actions awaiting an input from the other automaton. Notice that this assumption only refers to the internal input actions and not to the external inputs of the product automaton.

The two TAIO synchronise both on time and on their shared common actions $\mathsf{Act}_{1\to 2} \cup \mathsf{Act}_{2\to 1}$. When connected to each other, the interaction between the two TAIO is assumed to be unobservable from outside. We further assume that (A_1, A_2) is compatible with respect to $(\mathsf{Act}_{1\to 2}, \mathsf{Act}_{2\to 1})$.

The parallel composition of A_1 and A_2 is denoted $A_1||A_2$. It is the TAIO $(Q_1 \times Q_2, (q_0^1, q_0^2), X_1 \cup X_2, \mathsf{Act}_{\mathsf{in}}, \mathsf{Act}_{\mathsf{out}}, \mathsf{E})$ such that

$$\mathsf{Act}_{\mathsf{in}} = \bigcup_{i=1,2} \mathsf{Act}_{\mathsf{in}}^i,\ \mathsf{Act}_{\mathsf{out}} = \bigcup_{i=1,2} \mathsf{Act}_{\mathsf{out}}^i$$

and E is the smallest set such that:

- For $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{\mathsf{out}}^1 \cup \{\tau_1\}$:

$$(q_1, q_1', \psi_1, r_1, d_1, a) \in \mathsf{E}_1 \Rightarrow ((q_1, q_2), (q_1', q_2), \psi_1, r_1, d_1, a) \in \mathsf{E}; \tag{4}$$

- For $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{\mathsf{out}}^2 \cup \{\tau_2\}$:

$$(q_2, q_2', \psi_2, r_2, d_2, a) \in \mathsf{E}_2; \Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_2, r_2, d_2, a) \in \mathsf{E}; \tag{5}$$

- For $a \in \mathsf{Act}_{1\to 2}$:

$$(q_1, q_1', \psi_1, r_1, d_1, a) \in \mathsf{E}_1 \wedge (q_2, q_2', \psi_2, r_2, \mathsf{lazy}, a) \in \mathsf{E}_2$$

$$\Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_1, \tau_a) \in \mathsf{E}; \tag{6}$$

- For $a \in \mathsf{Act}_{2\to 1}$:

$$(q_1, q_1', \psi_1, r_1, \mathsf{lazy}, a) \in \mathsf{E}_1 \wedge (q_2, q_2', \psi_2, r_2, d_2, a) \in \mathsf{E}_2$$

$$\Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_2, \tau_a) \in \mathsf{E}. \tag{7}$$

2.6 Parallel composition of TIOLTS

Similarly, it is also useful to define parallel composition over TIOLTS. Given two TIOLTS L_1 and L_2, the corresponding parallel product is denoted $L_1||L_2$. For $i = 1, 2$, $L_i = (S_i, s_0^i, \mathsf{Act}_{\mathsf{in}}^i \cup \mathsf{Act}_{(3-i)\to i}, \mathsf{Act}_{\mathsf{out}}^i \cup \mathsf{Act}_{i\to(3-i)}, T_d^i, T_t^i)$. The sets $\mathsf{Act}_{\mathsf{in}}^1$, $\mathsf{Act}_{\mathsf{out}}^1$, $\mathsf{Act}_{\mathsf{in}}^2$, $\mathsf{Act}_{\mathsf{out}}^2$, $\mathsf{Act}_{1\to 2}$ and $\mathsf{Act}_{2\to 1}$ are pairwise disjoint (as illustrated in Figure 2). The two TIOLTS synchronize on time delays and their common shared actions $\mathsf{Act}_{1\leftrightarrow 2} = \mathsf{Act}_{1\to 2} \cup \mathsf{Act}_{2\to 1}$. The parallel product of the two TIOLTS is

$$L_1||L_2 = (S, (s_0^1, s_0^2), \mathsf{Act}_{\mathsf{in}}, \mathsf{Act}_{\mathsf{out}}, T_d, T_t)$$

such that

$$\mathsf{Act}_{\mathsf{in}} = \bigcup_{i=1,2} \mathsf{Act}_{\mathsf{in}}^i,\ \mathsf{Act}_{\mathsf{out}} = \bigcup_{i=1,2} \mathsf{Act}_{\mathsf{out}}^i$$

and S, T_d and T_t are the smallest sets such that:

- $(s_0^1, s_0^2) \in S$;
- For $(s_1, s_2) \in S$ and $\delta \in \mathsf{R}$:

$$s_1 \xrightarrow{\delta} s_1' \in T_t^1 \wedge s_2 \xrightarrow{\delta} s_2' \in T_t^2 \Rightarrow (s_1', s_2') \in S \wedge (s_1, s_2) \xrightarrow{\delta} (s_1', s_2') \in T_t; \quad (8)$$

- For $(s_1, s_2) \in S$ and $a \in \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{\mathsf{out}}^1 \cup \{\tau_1\}$:

$$s_1 \xrightarrow{a} s_1' \in T_d^1 \Rightarrow (s_1', s_2) \in S \wedge (s_1, s_2) \xrightarrow{a} (s_1', s_2) \in T_d; \quad (9)$$

- For $(s_1, s_2) \in S$ and $a \in \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{\mathsf{out}}^2 \cup \{\tau_2\}$:

$$s_2 \xrightarrow{a} s_2' \in T_d^2 \Rightarrow (s_1, s_2') \in S \wedge (s_1, s_2) \xrightarrow{a} (s_1, s_2') \in T_d; \quad (10)$$

- For $(s_1, s_2) \in S$ and $a \in \mathsf{Act}_{1\leftrightarrow 2}$:

$$s_1 \xrightarrow{a} s_1' \in T_d^1 \wedge s_2 \xrightarrow{a} s_2' \in T_d^2 \Rightarrow (s_1', s_2') \in S \wedge (s_1, s_2) \xrightarrow{\tau_a} (s_1', s_2') \in T_d. \quad (11)$$

It is not difficult to see that from each possible run λ of $L_1||L_2$ it is possible to extract two (unique) timed traces σ_1 and σ_2 of L_1 and L_2, respectively. For example for

$$\lambda = (s_0^1, s_0^2) \xrightarrow{1.5} (s, t) \xrightarrow{\tau_a} (p, q) \xrightarrow{?b} (r, q) \xrightarrow{!c} (r, u)$$

we have $\sigma_1 = 1.5\,?a\,?b$ and $\sigma_2 = 1.5\,!a\,!c$, where $a \in \mathsf{Act}_{2\rightarrow 1}$, $b \in \mathsf{Act}_{\mathsf{in}}^1$ and $c \in \mathsf{Act}_{\mathsf{out}}^2$.

Conversely, the two traces σ_1 and σ_2, in $\mathsf{ObsTTraces}(L_1)$ and $\mathsf{ObsTTraces}(L_2)$ respectively, are said to be *synchronizable* in $L_1||L_2$ if there exists a run λ of $L_1||L_2$ from which the two traces can be extracted. In general, the run from which σ_1 and σ_2 can be extracted may not be unique, due to different possible interleavings . For instance, the two traces σ_1 and σ_2 given above can be also extracted from the run

$$\lambda' = (s_0^1, s_0^2) \xrightarrow{1.5} (s, t) \xrightarrow{\tau_a} (p, q) \xrightarrow{!c} (p, u) \xrightarrow{?b} (r, u).$$

Let L_1' and L_2' be two new TIOLTS. For $i = 1, 2$, L_i' has the same sets of inputs and outputs as L_i. Moreover, L_1' and L_2' synchronize on the same set of actions $\mathsf{Act}_{1\leftrightarrow 2}$ as for L_1 and L_2. Let $\sigma_1 \in \mathsf{ObsTTraces}(L_1) \cap \mathsf{ObsTTraces}(L_1')$, $\sigma_2 \in \mathsf{ObsTTraces}(L_2) \cap \mathsf{ObsTTraces}(L_2')$, λ a run of $L_1||L_2$ and $\sigma \in \mathsf{ObsTTraces}(L_1||L_2)$ the observable timed trace corresponding to λ.

Lemma 1 *If σ_1 and σ_2 are the traces extracted from λ, then σ_1 and σ_2 are synchronizable in $L_1'||L_2'$ and $\sigma \in \mathsf{ObsTTraces}(L_1'||L_2')$.*

Proof Here, we assume, with no loss of generality, that the four TIOLTSs have the same unobservable action τ. Let $\gamma = a_1 \cdots a_n$ be the projection of the run λ to $\mathsf{Act}_{\mathsf{in}} \cup \mathsf{Act}_{\mathsf{out}} \cup \mathsf{Act}_{1\leftrightarrow 2}$. For $i = 1, 2$, since $\sigma_i \in \mathsf{ObsTTraces}(L_i)$, there exists $\gamma_i \in \mathsf{TTraces}(L_i)$ such that σ_i is the projection of γ to Act^i. [1] Thus, there exist $[i_1, \cdots, i_N]$, $[j_1, \cdots, j_N]$, $[k_1, \cdots, k_M]$ and $[l_1, \cdots, l_M]$ such that $\gamma_1 = \tau^{j_1} a_{i_1} \cdots \tau^{j_N} a_{i_N}$ [2] and $\gamma_2 = \tau^{l_1} a_{k_1} \cdots \tau^{l_M} a_{k_M}$. Similarly, there exist $\gamma_1' = \tau^{j_1'} a_{i_1} \cdots \tau^{j_N'} a_{i_N} \in \mathsf{TTraces}(L_1')$ and $\gamma_2' = \tau^{l_1'} a_{k_1} \cdots \tau^{l_M'} a_{k_M} \in \mathsf{TTraces}(L_2')$. Clearly, $\sigma_1 = a_{i_1} \cdots a_{i_N}$ and $\sigma_2 = a_{k_1} \cdots a_{k_M}$. Since σ_1 and σ_2 are extracted from λ, it is possible to synchronize the traces γ_1 and γ_2 in $L_1||L_2$ by considering the trace $\beta = \tau^{r_1} a_1 \cdots \tau^{r_n} a_n \in \mathsf{TTraces}(L_1||L_2)$, where for $p = 1, \cdots, n$: $r_p = j_s + l_t$,

[1] $\mathsf{Act}^i = \mathsf{Act}_{\mathsf{in}}^i \cup \mathsf{Act}_{\mathsf{out}}^i \cup \mathsf{Act}_{1\leftrightarrow 2}$.

[2] τ^{j_1} is the sequence made up of j_1 τ's.

if $a_p = a_{i_s} = a_{k_t}$ appears in both γ_1 and γ_2; $r_p = j_s$, if $a_p = a_{i_s}$ appears only in γ_1; and $r_p = l_t$, if $a_p = a_{k_t}$ appears only in γ_2.

Then in the same way, it is possible to synchronize γ_1' and γ_2' in $L_1'||L_2'$ by considering the trace $\beta' = \tau^{r_1'} a_1 \cdots \tau^{r_n'} a_n \in \mathsf{TTraces}(L_1'||L_2')$, where for $p = 1, \cdots, n$, r_p' is defined similarly to r_p. Hence, we are done. ■

Let $A_1 = (Q_1, q_0^1, X_1, \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{2\to1}, \mathsf{Act}_{\mathsf{out}}^1 \cup \mathsf{Act}_{1\to2}, \mathsf{E}_1)$ and $A_2 = (Q_2, q_0^2, X_2, \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{1\to2}, \mathsf{Act}_{\mathsf{out}}^2 \cup \mathsf{Act}_{2\to1}, \mathsf{E}_2)$ be two TAIO. Then we have the following.

Proposition 1 *If* (A_1, A_2) *is compatible with respect to* $(\mathsf{Act}_{1\to2}, \mathsf{Act}_{2\to1})$ *then*

$$L_{A_1||A_2} = L_{A_1}||L_{A_2}.$$

Proof Clearly, the two TIOLTS are equal iff they have the same initial state and the same set of transitions.

- First, it is not difficult to see that $((q_0^1, \vec{0}), (q_0^2, \vec{0}))$ is the initial state of both $L_{A_1||A_2}$ and $L_{A_1}||L_{A_2}$.
- Let $t = (s_1, s_2) \xrightarrow{a} (s_1', s_2')$ be an arbitrary transition. We abuse notation and write $t \in L_{A_1||A_2}$ when t is a possible transition of $L_{A_1||A_2}$. We prove that

$$t \in L_{A_1||A_2} \Leftrightarrow t \in L_{A_1}||L_{A_2}. \tag{12}$$

  1. If $a \in \mathsf{R}$, t is possible in $A_1||A_2$ iff $s_1 \xrightarrow{a} s_1'$ is possible in A_1 and $s_2 \xrightarrow{a} s_2'$ in A_2. That is because $A_1||A_2$ is defined in a way as to accept a time delay a iff t is accepted by both A_1 and A_2. By definition of $L_{A_1||A_2}$, $t \in L_{A_1||A_2}$ iff t is possible in $A_1||A_2$. By rule (8), $t \in L_{A_1}||L_{A_2}$ iff $s_1 \xrightarrow{a} s_1'$ is possible in A_1 and $s_2 \xrightarrow{a} s_2'$ in A_2. Thus, (12) holds.
  2. If $a \in \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{\mathsf{out}}^1 \cup \{\tau_1\}$, t is possible in $A_1||A_2$ iff $s_1 \xrightarrow{a} s_1'$ is possible in A_1. That is because the edges labeled with a, both in $A_1||A_2$ and A_1, have the same guard, deadline and set of clocks to reset (See rule (4)). By definition of $L_{A_1||A_2}$ and by rule (9), (12) holds.
  3. If $a \in \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{\mathsf{out}}^2 \cup \{\tau_2\}$, t possible in $A_1||A_2$ iff $s_2 \xrightarrow{a} s_2'$ possible in A_2. By the same reasoning as for $a \in \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{\mathsf{out}}^1 \cup \{\tau_1\}$, (12) holds.
  4. If $a = \tau_b$ such that $b \in \mathsf{Act}_{1\leftrightarrow2}$, t is possible in $A_1||A_2$ iff $s_1 \xrightarrow{b} s_1'$ is possible in A_1 and $s_2 \xrightarrow{b} s_2'$ in A_2. That follows immediately from rules (6) and (7). By definition of $L_{A_1||A_2}$ and by rule (11), (12) holds.

■

3 Timed Input-Output 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. As in [45], we assume that the implementation (i.e., the system to be tested) can be modeled as a non-blocking, input-enabled TAIO A_I. Notice that we do not assume that A_I is known, simply that it exists. Input-enabledness 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).

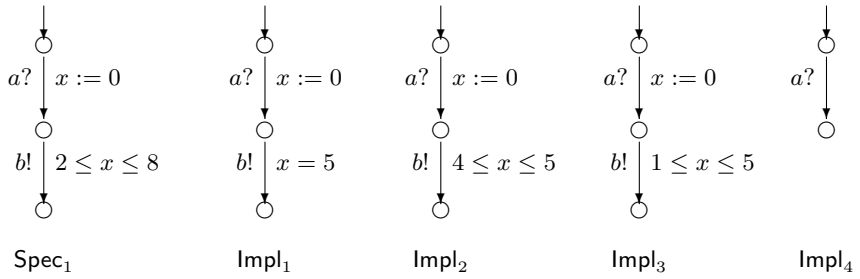


Figure 3: Examples of specifications and implementations.

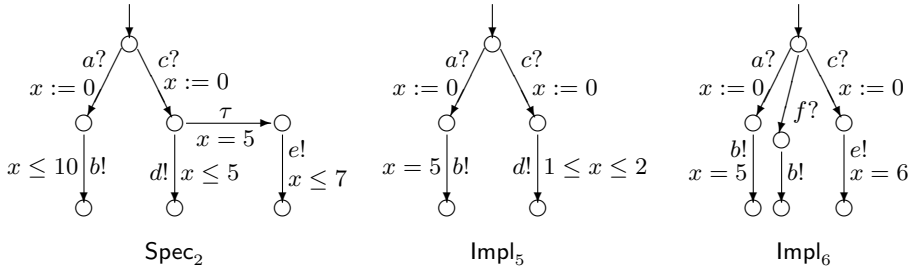


Figure 4: More examples of specifications and implementations.

3.1 Examples

Before we proceed to define the conformance relation, 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 delayable and deadlines of input edges are lazy. In order not to overload the figures, we do not always draw input-enabled (implementation) automata. We assume that implementations *ignore* the missing inputs. This can be modeled by adding self-loop edges covering the missing inputs. Note that, for specifications, which are not assumed to be input-enabled, missing inputs have a different meaning: they correspond to "don't cares", as will be explained below.

Consider the specification Spec_1 shown in Figure 3. 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 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 within 4 to 5 time units. 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.

Now consider specification Spec_2 shown in Figure 4. This specification could be expressed as: "if the first input is a then the system must output b within 10 time units; if the first input is c then the system must either output d within 5 time units or, failing to do that, output e within 7 time units". The second branch of Spec_2 is a typical specification of a timeout. If the "normal" result d does not appear for some time, the system itself must recognize the error and output an error message not much later. None of the four implementations of Figure 3 conform to Spec_2, as they do not react to input c (they ignore it). On the other hand, Impl_5 and Impl_6 of Figure 4 conform to Spec_2. Notice that Impl_6 accepts input f which does not appear in Spec_2. This is an example of a "don't care" input mentioned above. The specification states nothing about the case where this input is

provided. Thus, it imposes no requirements on this case, and the implementation is "free" to behave as it wishes.

3.2 Timed input-output conformance relation: tioco

In this section, we define our conformance relation tioco and we state some of its properties.We first compare specifications with the same set of observable traces and we show that there are equivalent with respect to tioco. Next, we compare between tioco and the timed trace inclusion relation. Also, we show that only lazy-inputs are needed in specifications and how deterministic specifications can be made input-enabled without changing their conformance semantics. In the remaining part of the section we prove the transitivity, undecidability, compositionality of tioco and its stability with respect to decreasing the number of observable actions.

3.2.1 Definition

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

$$A \text{ after } \sigma = \{s \in S_A \mid \exists \rho \in \mathsf{RT}(\mathsf{Act}_\tau) \ : \ s_0^A \xrightarrow{\rho} s \ \wedge \ P_{\mathsf{Act}}(\rho) = \sigma\}. \tag{13}$$

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:

$$\mathsf{elapse}(s) = \{t > 0 \mid \exists \rho \in \mathsf{RT}(\{\tau\}) \ : \ \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}\}. \tag{14}$$

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:

$$\mathsf{out}(s) = \{a \in \mathsf{Act}_{\mathsf{out}} \mid s \xrightarrow{a}\} \cup \mathsf{elapse}(s), \qquad \mathsf{out}(S) = \bigcup_{s \in S} \mathsf{out}(s). \tag{15}$$

The *timed input-output conformance relation*, denoted tioco, is defined as

$$A_I \text{ tioco } A_S \ \textit{ iff } \ \forall \sigma \in \mathsf{ObsTTraces}(A_S) \ : \ \mathsf{out}(A_I \text{ after } \sigma) \subseteq \mathsf{out}(A_S \text{ after } \sigma). \tag{16}$$

The relation states that an implementation A_I conforms to a specification A_S iff for any observable behavior σ of A_S, the set of observable outputs of A_I after any behavior "matching" σ must be a subset of the set of possible observable outputs of A_S. Notice that observable outputs are not only observable output actions but also time delays.

As expected, in the examples above, we have Impl_1 tioco Spec_1 and Impl_2 tioco Spec_1. On the other hand, Impl_3 ~~tioco~~ Spec_1 and Impl_4 ~~tioco~~ Spec_1 because $\mathsf{out}(\mathsf{Impl}_3 \text{ after } a\,1) = (0, 4] \cup \{b\}$ and $\mathsf{out}(\mathsf{Impl}_4 \text{ after } a\,1) = (0, \infty)$, whereas $\mathsf{out}(\mathsf{Spec}_1 \text{ after } a\,1) = (0, 7]$.

We proceed in giving a number of properties of tioco. The first states that specifications that have the same set of observable timed traces are *equivalent* with respect to tioco, in other words, they specify the same requirements.

Lemma 2 *Given two TAIO A_S and A'_S, if* $\mathsf{ObsTTraces}(A_S) = \mathsf{ObsTTraces}(A'_S)$ *then*

$$\forall A_I : A_I \text{ tioco } A_S \Leftrightarrow A_I \text{ tioco } A'_S \ .$$

Proof Let $\sigma \in \mathsf{ObsTTraces}(A_S) = \mathsf{ObsTTraces}(A'_S)$. We claim that $\mathsf{out}(A_S \text{ after } \sigma) = \mathsf{out}(A'_S \text{ after } \sigma)$. Indeed, for any $a \in \mathsf{Act_{out}} \cup \mathsf{R}$, $a \in \mathsf{out}(A_S \text{ after } \sigma) \setminus \mathsf{out}(A'_S \text{ after } \sigma)$ implies $\sigma a \in \mathsf{ObsTTraces}(A_S) \setminus \mathsf{ObsTTraces}(A'_S)$ which contradicts the hypothesis. Thus, for any implementation A_I, $\mathsf{out}(A_I \text{ after } \sigma) \subseteq \mathsf{out}(A_S \text{ after } \sigma)$ iff $\mathsf{out}(A_I \text{ after } \sigma) \subseteq \mathsf{out}(A'_S \text{ after } \sigma)$, and the result follows by definition of tioco. ■

The next lemma relates tioco to observable timed trace inclusion.

Lemma 3 *Consider two TAIO A and B.*

1. $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(B)$ *implies* A tioco B.
2. *If B is input-enabled then A* tioco *B implies* $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(B)$.

Proof

1. Let $\sigma \in \mathsf{ObsTTraces}(B)$ and $a \in \mathsf{out}(A \text{ after } \sigma)$. $a \in \mathsf{out}(A \text{ after } \sigma)$ implies $\sigma a \in \mathsf{ObsTTraces}(A)$. Since $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(B)$, $\sigma a \in \mathsf{ObsTTraces}(B)$. Thus, $a \in \mathsf{out}(B \text{ after } \sigma)$, and $\mathsf{out}(A \text{ after } \sigma) \subseteq \mathsf{out}(B \text{ after } \sigma)$. The result follows by definition of tioco.
2. Suppose there exists $\sigma \in \mathsf{ObsTTraces}(A) \setminus \mathsf{ObsTTraces}(B)$. Thus, there exist $\sigma_1, \sigma_2 \in \mathsf{RT}(\mathsf{Act})$ and $a \in \mathsf{Act} \cup \mathsf{R}$, such that $\sigma = \sigma_1 a \sigma_2$, $\sigma_1 \in \mathsf{ObsTTraces}(B)$ and $\sigma_1 a \notin \mathsf{ObsTTraces}(B)$. If $a \in \mathsf{Act_{in}}$ then $\sigma_1 a \notin \mathsf{ObsTTraces}(B)$ is a contradiction since $\sigma_1 \in \mathsf{ObsTTraces}(B)$ and B is input-enabled. If $a \in \mathsf{Act_{out}} \cup \mathsf{R}$ then we have again a contradiction, since $\sigma_1 \in \mathsf{ObsTTraces}(B)$, $a \in \mathsf{out}(A \text{ after } \sigma_1)$ and A tioco B.

■

Figure 4 gives an example for which the second part of Lemma 3 does not hold when B is not input-enabled. That is, Impl_6 tioco Spec_2 though $\mathsf{ObsTTraces}(\mathsf{Impl}_6) \not\subseteq \mathsf{ObsTTraces}(\mathsf{Spec}_2)$.

3.2.2 Only lazy inputs are needed in specifications

In all our examples so far, input edges of specifications have been annotated with lazy deadlines (and not delayable or eager). This is not a coincidence. As we show in this section, considering only *lazy-input* TAIO is enough for describing all possible (non-blocking) specifications. A lazy-input TAIO is one where every edge labeled with $a \in \mathsf{Act_{in}}$ has deadline lazy. Given a TAIO A, let $\mathsf{Lazy}(A)$ be the TAIO obtained by setting the deadline of every edge of A labeled by an input to lazy.

Lemma 4 *For any non-blocking TAIO A,* $\mathsf{ObsTTraces}(A) = \mathsf{ObsTTraces}(\mathsf{Lazy}(A))$.

Proof It should be clear that $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(\mathsf{Lazy}(A))$, since $\mathsf{Lazy}(A)$ is at least as "permissive" as A (i.e., every transition in the TLTS defined by A is also a transition of the TLTS defined by $\mathsf{Lazy}(A)$). It remains to prove that $\mathsf{ObsTTraces}(\mathsf{Lazy}(A)) \subseteq \mathsf{ObsTTraces}(A)$. Suppose there exists $\sigma \in \mathsf{ObsTTraces}(\mathsf{Lazy}(A)) \setminus \mathsf{ObsTTraces}(A)$. Let $s_0 \xrightarrow{\sigma_1} s_1 \cdots \xrightarrow{\sigma_N} s_N$ a possible run of $\mathsf{Lazy}(A)$ corresponding to the trace σ. Since $\sigma \notin \mathsf{ObsTTraces}(A)$, there must exist some $k \leq N$ such that $s_0 \xrightarrow{\sigma_1} s_1 \cdots \xrightarrow{\sigma_{k-1}} s_{k-1}$ is a possible path in A and $s_{k-1} \not\xrightarrow{\sigma_k}$ in A. Let q and v be the location and the clock valuation, respectively, such that $s_{k-1} = (q, v)$. Depending on the value of σ_k, two cases are possible:

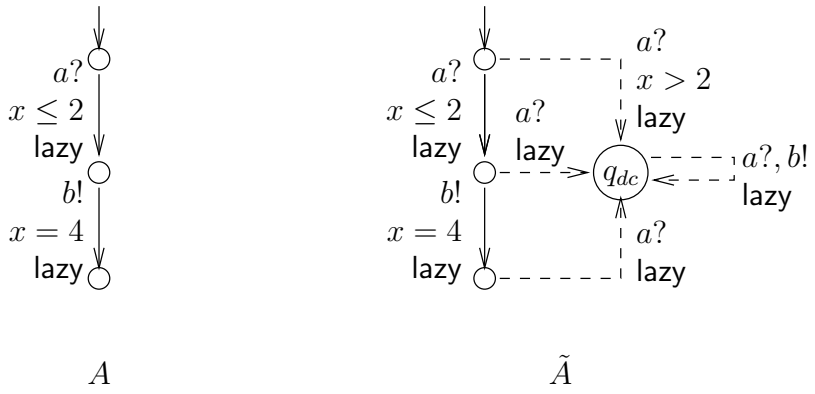


Figure 5: How to transform a deterministic, fully-observable, but not input-enabled specification to an equivalent input-enabled specification.

- $\sigma_k \in \mathsf{Act}_\tau$: By construction, location q has outgoing edges which are labeled with the same actions and have the same deadlines and clocks to reset, both in A and $\mathsf{Lazy}(A)$. Thus for the same valuation v, the discrete transition $s_{k-1} = (q, v) \xrightarrow{\sigma_k} s_k$, possible in $\mathsf{Lazy}(A)$, is also possible in A. Contradiction.
- $\sigma_k \in \mathsf{R}$: The fact that $s_{k-1} \not\xrightarrow{\sigma_k}$, in A, means that there is some delayable or eager outgoing edge e from q which prevents the delay σ_k from elapsing. e cannot be labeled with τ or an output action, since then it would block time in $\mathsf{Lazy}(A)$ as well. Thus, e is labeled with an input action. This implies that at state s_{k-1} time is blocked unless this input action is received, which contradicts the hypothesis that A is non-blocking.

■

From Lemma 2 and Lemma 4, we obtain the following.

Proposition 2 *For any non-blocking TAIO A_S,*

$$\forall A_I : A_I \text{ tioco } A_S \Leftrightarrow A_I \text{ tioco } \mathsf{Lazy}(A_S).$$

3.2.3 Making specifications input-enabled

A deterministic (and fully observable) specification can be made input-enabled 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. More precisely, this transformation is done as follows. Given a TAIO $A = (Q, q_0, X, \mathsf{Act}, \mathsf{E})$, we build the corresponding input-enabled TAIO $\tilde{A} = (\tilde{Q}, q_0, X, \mathsf{Act}, \tilde{\mathsf{E}})$. First, $\tilde{Q} = Q \cup \{q_{dc}\}$ where $q_{dc} \notin Q$ is the "don't care" location. Second,

$$\tilde{E} = E \cup \{(q_{dc}, q_{dc}, \mathsf{true}, \emptyset, \mathsf{lazy}, a) \mid a \in \mathsf{Act}\} \cup \{(q, q_{dc}, \neg\psi, \emptyset, \mathsf{lazy}, a) \mid q \in Q \wedge a \in \mathsf{Act}_{\mathsf{in}}\}$$

such that for each $q \in Q$ and each $a \in \mathsf{Act}_{\mathsf{in}}$, $\psi = \psi_1 \vee \psi_2 \vee \cdots \vee \psi_k$ where ψ_1, ψ_2, ... , ψ_k are the guards of the outgoing edges of q labeled with a. An example showing how this transformation works is given in Figure 5. We transform A to $\tilde{A}$. The TAIO A has only one input (a) and one output (b). The added edges are the dashed ones.

Proposition 3 *Let* Spec *be a deterministic and fully observable TAIO and let* $\widetilde{\mathsf{Spec}}$ *be the input-enabled TAIO corresponding to* Spec *obtained by the transformation given above. For any input-enabled TAIO* Impl, Impl tioco Spec *iff* Impl tioco $\widetilde{\mathsf{Spec}}$.

The proof of the above proposition is based on the following two lemmata.

Lemma 5 $\mathsf{ObsTTraces}(\mathsf{Spec}) \subseteq \mathsf{ObsTTraces}(\widetilde{\mathsf{Spec}})$.

Proof Because $\widetilde{\mathsf{Spec}}$ is obtained by adding edges to Spec and all added edges have deadline lazy. ■

Lemma 6 *Let* $\sigma \in \mathsf{ObsTTraces}(\widetilde{\mathsf{Spec}})$. *If* $\sigma \in \mathsf{ObsTTraces}(\mathsf{Spec})$ *then* $\mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma) = \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$. *Otherwise,* $\mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma) = \mathsf{R} \cup \mathsf{Act}_{\mathsf{out}}$.

Proof If $\sigma \in \mathsf{ObsTTraces}(\mathsf{Spec})$ then the q_{dc} location is not reached yet and $\widetilde{\mathsf{Spec}}$ still has the same behavior as Spec. If $\sigma \notin \mathsf{ObsTTraces}(\mathsf{Spec})$ then the sink location q_{dc} has been reached. Since $\widetilde{\mathsf{Spec}}$ is defined with respect to the same set of outputs $\mathsf{Act}_{\mathsf{out}}$ as Spec, $\mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma) = \mathsf{R} \cup \mathsf{Act}_{\mathsf{out}}$. Hence, $\mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma) = \mathsf{R} \cup \mathsf{Act}_{\mathsf{out}}$ and $\mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$. ■

Now, we give the proof of Proposition 3.

Proof [of Proposition 3]

($\Rightarrow$) We assume that Impl tioco Spec and we prove that Impl tioco $\widetilde{\mathsf{Spec}}$. Let $\sigma \in \mathsf{ObsTTraces}(\widetilde{\mathsf{Spec}})$. If $\sigma \in \mathsf{ObsTTraces}(\mathsf{Spec})$ then by Lemma 6 $\mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma) = \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$. Moreover since Impl tioco Spec we have $\mathsf{out}(\mathsf{Impl}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma)$. So $\mathsf{out}(\mathsf{Impl}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$ and we are done. If $\sigma \notin \mathsf{ObsTTraces}(\mathsf{Spec})$, by Lemma 6 we have $\mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma) = \mathsf{R} \cup \mathsf{Act}_{\mathsf{out}}$. Thus, we clearly have $\mathsf{out}(\mathsf{Impl}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$ and we are done once again.

($\Leftarrow$) We assume that Impl tioco $\widetilde{\mathsf{Spec}}$ and we prove that Impl tioco Spec. Let $\sigma \in \mathsf{ObsTTraces}(\mathsf{Spec})$. By Lemma 5, we have $\sigma \in \mathsf{ObsTTraces}(\widetilde{\mathsf{Spec}})$ By Lemma 6, $\mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma) = \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$. Moreover, we have $\mathsf{out}(\mathsf{Impl}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\widetilde{\mathsf{Spec}}\ \mathsf{after}\ \sigma)$ since Impl tioco $\widetilde{\mathsf{Spec}}$ and $\sigma \in \mathsf{ObsTTraces}(\widetilde{\mathsf{Spec}})$. Thus, $\mathsf{out}(\mathsf{Impl}\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(\mathsf{Spec}\ \mathsf{after}\ \sigma)$ and we are done.

■

Combined with Lemma 3, Proposition 3 implies that for deterministic and fully observable specifications, tioco can be replaced by timed trace inclusion, modulo the above input-enabling transformation. Notice that the proposed transformation is not correct for the case of non-deterministic or partially observable specifications. A simple counter-example is given in Figure 6. The specification Spec has one input a and two outputs b and c. The implementation Impl is input-enabled. [3] We have Impl tioco $\widetilde{\mathsf{Spec}}$ but Impl ~~tioco~~ Spec.

Also note that the determinization of TAIO is undecidable in general [48]. Hence, reducing tioco to timed trace inclusion is not always possible and a specific framework for checking conformance with respect to tioco needs to be established for the case of non-deterministic or partially-observable specifications.

[3] We omit self-loops labeled with a in order not to overload the figure.

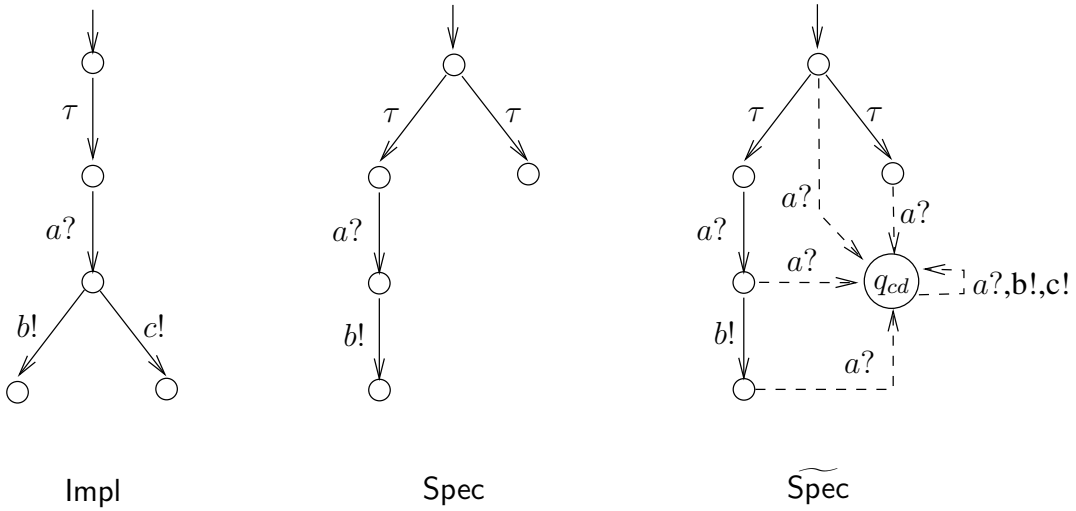


Figure 6: An example showing that the transformation of Figure 5 is incorrect for non-deterministic or partially-observable specifications.

3.2.4 Transitivity

Next we show that tioco is a transitive relation, given the usual assumption that implementations are input-enabled. That is an interesting property of the relation tioco. For instance, it may be helpful in case the specification model is obtained after several refinements.

Proposition 4 *Let A, B and C be three TAIO such that A and B are input-enabled, If A tioco B and B tioco C then A tioco C.*

Proof Let $\sigma \in \mathsf{ObsTTraces}(C)$. Two cases are possible:

- $\sigma \in \mathsf{ObsTTraces}(B)$. From A tioco B and B tioco C we obtain $\mathsf{out}(A \text{ after } \sigma) \subseteq \mathsf{out}(B \text{ after } \sigma)$ and $\mathsf{out}(B \text{ after } \sigma) \subseteq \mathsf{out}(C \text{ after } \sigma)$, thus, $\mathsf{out}(A \text{ after } \sigma) \subseteq \mathsf{out}(C \text{ after } \sigma)$.
- $\sigma \notin \mathsf{ObsTTraces}(B)$. By part 2 of Lemma 3, input-enabledness of B and A tioco B, we get $\sigma \notin \mathsf{ObsTTraces}(A)$. Thus, $\mathsf{out}(A \text{ after } \sigma) = \emptyset \subseteq \mathsf{out}(C \text{ after } \sigma)$.

The result follows by definition of tioco. ■

3.2.5 Undecidability

In this section we show the undecidability of tioco which is indeed a result of only a theoretical interest for instance to check directly (without testing) whether a (known) implementation conforms to its specification.

Proposition 5 *Checking* tioco *is undecidable.*

Proof We reduce the timed trace inclusion problem for timed automata which is known to be undecidable [1] to the problem of checking tioco. Let A and B be two TA over the set of actions Act. The timed trace inclusion problem consists in checking whether $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(B)$. Let $\mathsf{Act}_{\mathsf{out}} = \mathsf{Act}$, i.e., $\mathsf{Act}_{\mathsf{in}} = \emptyset$. Then, both A

and B are input-enabled. By part 2 of Lemma 3, $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(B)$ iff A tioco B. ■

It is worth noting that the undecidability of tioco is not a problem for black-box testing: since the implementation A_I is unknown, we cannot check conformance directly, anyway.

3.2.6 Compositionality

We prove that tioco is compositional as well. That is, if we succeed to check conformance of modules with respect to their models, then the product of the several modules conforms to the product of the models. That makes the task easier since checking conformance at the module levels is likely to be easier and cheaper than checking conformance at the whole product level.

Let A_1, A_1', A_2 and A_2' be four TAIO such that, for $i = 1, 2$, A_i and A_i' have the same sets of inputs and outputs, as shown in Figure 2. Suppose that all four automata are input-enabled with respect to their respective sets of inputs. Furthermore, suppose that A_1 and A_2 are compatible with respect to $(\mathsf{Act}_{1\to 2}, \mathsf{Act}_{2\to 1})$, and so are A_1' and A_2'. Then, we have the following compositionality result.

Proposition 6 *If* A_1' tioco A_1 *and* A_2' tioco A_2 *then* $A_1'||A_2'$ tioco $A_1||A_2$.

Proof Observe that both $A_1||A_2$ and $A_1'||A_2'$ have the same set of inputs $\mathsf{Act}_{\mathsf{in}} = \mathsf{Act}^1_{\mathsf{in}} \cup \mathsf{Act}^2_{\mathsf{in}}$ and set of outputs $\mathsf{Act}_{\mathsf{out}} = \mathsf{Act}^1_{\mathsf{out}} \cup \mathsf{Act}^2_{\mathsf{out}}$.

- We first prove that $A_1||A_2$ is input-enabled with respect to $\mathsf{Act}^1_{\mathsf{in}} \cup \mathsf{Act}^2_{\mathsf{in}}$. A state s of $A_1||A_2$ is a pair (s_1, s_2) where s_i is a state of A_i for $i = 1, 2$. By assumption, each A_i is input-enabled with respect to $\mathsf{Act}^i_{\mathsf{in}}$. Thus for each $a \in \mathsf{Act}^i_{\mathsf{in}}$, $s_i \xrightarrow{a}$. By (4) and (5), for each $a \in \mathsf{Act}^1_{\mathsf{in}} \cup \mathsf{Act}^2_{\mathsf{in}}$, $s \xrightarrow{a}$.
- By the same reasoning, $A_1'||A_2'$ is input-enabled with respect to $\mathsf{Act}^1_{\mathsf{in}} \cup \mathsf{Act}^2_{\mathsf{in}}$.
- Now, we show that $A_1'||A_2'$ tioco $A_1||A_2$. By Lemma 3, it suffices to prove that $\mathsf{ObsTTraces}(A_1'||A_2') \subseteq \mathsf{ObsTTraces}(A_1||A_2)$. Let $\mathsf{Act} = \mathsf{Act}_{\mathsf{in}} \cup \mathsf{Act}_{\mathsf{out}}$, $\mathsf{Act}_{1\leftrightarrow 2} = \mathsf{Act}_{1\to 2} \cup \mathsf{Act}_{2\to 1}$ and $\sigma \in \mathsf{ObsTTraces}(A_1'||A_2')$. Since $\mathsf{Act}_{1\leftrightarrow 2} \cup \{\tau\}$ are internal unobservable actions of $A_1'||A_2'$, there exists $\gamma \in \mathsf{TTraces}(A_1'||A_2')$ such that $P_{\mathsf{R}\cup\mathsf{Act}}(\gamma) = \sigma$. For $i = 1, 2$, let $\mathsf{Act}_i = (\mathsf{Act}^i_{\mathsf{in}} \cup \mathsf{Act}^i_{\mathsf{out}} \cup \mathsf{Act}_{1\leftrightarrow 2})$ (i.e., the observable actions of A_i) and $\sigma_i = P_{\mathsf{R}\cup\mathsf{Act}_i}(\gamma)$. Then $\sigma_i \in \mathsf{ObsTTraces}(A_i')$. By part 2 of Lemma 3, input-enabledness of A_i and A_i' and the assumption A_i' tioco A_i, we get $\sigma_i \in \mathsf{ObsTTraces}(A_i)$. By Lemma 1, σ_1 and σ_2 are synchronizable in $A_1||A_2$ and $\sigma \in \mathsf{ObsTTraces}(A_1||A_2)$.

■

Note that the above result does not generally hold for the case of non input-enabled TAIO. A counter-example is given in Figure 7. We consider four TAIO A_1, A_2, A_1' and A_2'. TAIO A_2 and A_2' are the same. The action a (dashed arrows in the figure) is shared between A_1 and A_2, as well as between A_1' and A_2'. That is, $\mathsf{Act}^1_{\mathsf{in}} = \{c\}$, $\mathsf{Act}^1_{\mathsf{out}} = \{d, e\}$, $\mathsf{Act}_{2\to 1} = \{a\}$ and $\mathsf{Act}^2_{\mathsf{in}} = \mathsf{Act}^2_{\mathsf{out}} = \mathsf{Act}_{1\to 2} = \emptyset$. The two TAIO A_1 and A_1' are input-enabled with respect to $\{a\}$. A_1 is not input-enabled with respect to $\{c\}$. The guards of the transitions of all the automata are equal to true with deadline lazy. We clearly have A_2' tioco A_2 since $A_2' = A_2$. It is also not difficult to see that A_1' tioco A_1. The figure also shows the two product automata $A_1||A_2$ and $A_1'||A_2'$. After receiving input

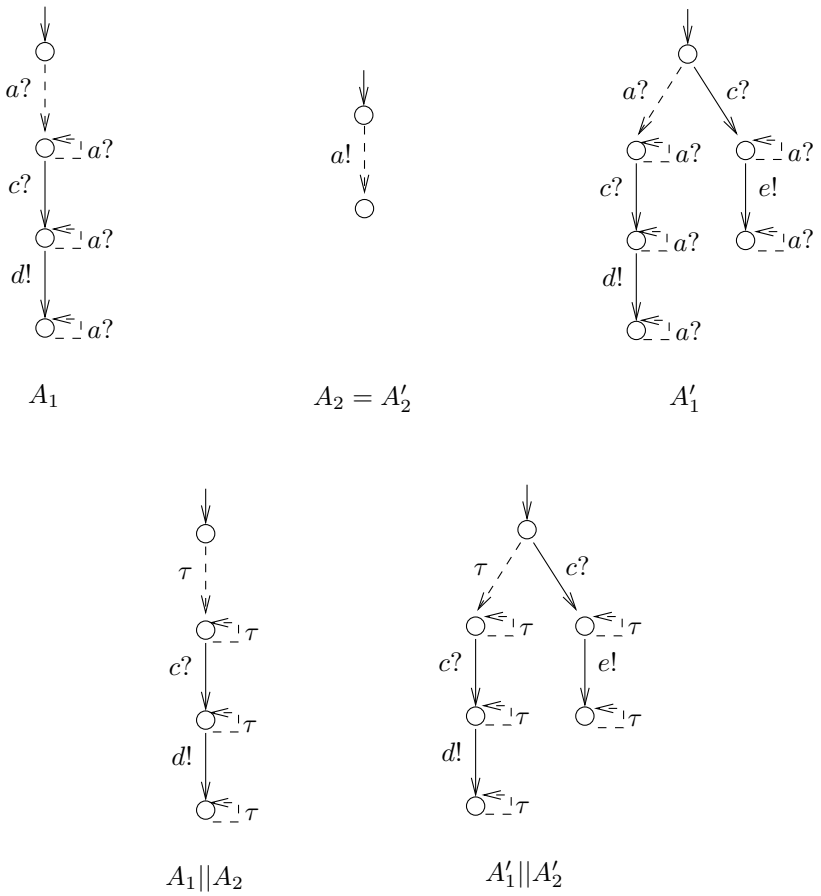


Figure 7: A counter example showing that tioco is not compositional for the case of non-input-enabled TAIO.

c, $A'_1||A'_2$ may generate either output d or e while $A_1||A_2$ may generate only d. Thus, $A'_1||A'_2 \not\mathsf{tioco} A_1||A_2$.

3.2.7 Decreasing the number of observable actions

In this section, we prove the stability of tioco with respect to the decreasing of the number of observable actions. That is under the input-enabledness assumption, if one succeeds to prove the conformance of the SUT to the considered specification with respect to some given set of observable actions then conformance still holds with respect to any smaller set of observable actions. For example, if due to any reason some observable action is no longer accessible (i.e., no longer observable), then that does not affect conformance.

Given a TAIO A and an observable action $a \in \mathsf{Act}$, we denote by $A_{[\tau/a]}$ the TAIO obtained from A by replacing action a, anywhere it appears, by τ. We have the following result.

Proposition 7 *Given two input-enabled TAIO A and A' and an observable action $a \in \mathsf{Act}$, if A' tioco A then $A'_{[\tau/a]}$ tioco $A_{[\tau/a]}$.*

Proof We first prove that $A_{[\tau/a]}$ is input-enabled. Let $\mathsf{Act}^a_{\mathsf{in}}$ be the set of inputs of $A_{[\tau/a]}$ and $A'_{[\tau/a]}$. $\mathsf{Act}^a_{\mathsf{in}} \subseteq \mathsf{Act}_{\mathsf{in}}$ since either $\mathsf{Act}^a_{\mathsf{in}} = \mathsf{Act}_{\mathsf{in}}$ (if a is an output) or $\mathsf{Act}^a_{\mathsf{in}} = \mathsf{Act}_{\mathsf{in}} \setminus \{a\}$ (if a is an input). $A_{[\tau/a]}$ has exactly the same set of states as A. A is input-enabled, and the result follows. In the same manner, we prove that $A'_{[\tau/a]}$ is input-enabled.

Now, by Lemma 3, it suffices to prove that $\mathsf{ObsTTraces}(A'_{[\tau/a]}) \subseteq \mathsf{ObsTTraces}(A_{[\tau/a]})$. Let $\sigma \in \mathsf{ObsTTraces}(A'_{[\tau/a]})$. There exists a trace $\sigma' \in \mathsf{ObsTTraces}(A')$ such that $\sigma = P_{\mathsf{R}\cup\mathsf{Act}\setminus\{a\}}(\sigma')$. Since A and A' are input-enabled and A' tioco A, $\sigma' \in \mathsf{ObsTTraces}(A)$ (by Lemma 3). Since $\sigma = P_{\mathsf{R}\cup\mathsf{Act}\setminus\{a\}}(\sigma')$, $\sigma \in \mathsf{ObsTTraces}(A_{[\tau/a]})$. ■

The above result is not valid for non-input-enabled TAIO, in general. We use the counter-example of Figure 7. We consider the two TAIO A_1 and A'_1. As already mentioned, A'_1 tioco A_1. It is easy to see that $A_{1[\tau/a]} = A_1||A_2$ and $A'_{1[\tau/a]} = A'_1||A'_2$. So, clearly $A'_{1[\tau/a]} \not\mathsf{tioco} A_{1[\tau/a]}$.

3.3 Comparison of tioco with other conformance relations

Different conformance relations have been used in the literature in the context of timed testing. We review some of these and compare them to tioco.

[44] define conformance as timed bisimulation (TB), which in their case reduces to timed trace equivalence (TTE), since determinism is assumed. [41] define conformance using a must/may preorder (MMP). None of $\mathsf{Impl}_1, \mathsf{Impl}_2$ conform to Spec_1 with respect to TB, TTE or MMP. We believe that this is too strict.[4]

[33, 29] define conformance as timed trace inclusion (TTI). As stated in Lemma 3, TTI is generally stricter than tioco: tioco allows an implementation to accept inputs not accepted by the specification, whereas TTI does not. For instance in Figure 4, we have Impl_6 tioco Spec_2 though $\mathsf{ObsTTraces}(\mathsf{Impl}_6) \not\subseteq \mathsf{ObsTTraces}(\mathsf{Spec}_2)$. As also stated in Lemma 3, when the specification is input-enabled, tioco and TTI are equivalent.

Finally, as stated in Proposition 3, TTI may replace tioco in the case of deterministic and fully-observable specifications, modulo an input-enabling transformation. Note,

[4] It should be noted, however, that the issue does not arise in [44] because outputs are assumed to be urgent, thus, Spec_1 cannot be expressed.

however, that when this transformation is performed care must be taken to instruct the test generation algorithm not to explore the added "don't care" inputs, so that it does not generate useless tests. We opt for tioco, which avoids these complications in a simple way.

Next we compare tioco with two other conformance relations proposed in the literature.

3.3.1 Comparison with the relativized timed conformance relation

In [38], the *relativized timed conformation relation*, or rtioco, is defined. It is "relativized" in the sense that it compares the implementation $\mathcal{I}$ and the specification $\mathcal{S}$ with respect to some given environment $\mathcal{E}$. Both $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{E}$ are given as TIOLTS. $\mathcal{S}$ and $\mathcal{I}$ are assumed to be input-enabled with respect to $\mathsf{Act}_{\mathsf{in}}$; and $\mathcal{E}$ input-enabled with respect to $\mathsf{Act}_{\mathsf{out}}$. $\mathcal{S}$, $\mathcal{I}$ and $\mathcal{E}$ are also non-blocking. For comparing $\mathcal{S}$ and $\mathcal{I}$, the first step consists in making the parallel composition of each of them with $\mathcal{E}$. The used parallel composition is slightly different from the one we propose. To avoid confusion, we denote it $||_r$. What is different with $||_r$ is that it does not hide the actions on which the two TIOLTS synchronize (i.e., in Condition (11) the action a remains observable after synchronization). Formally, rtioco is defined as follows:

$$\mathcal{I}\ \mathsf{rtioco}_{\mathcal{E}}\ \mathcal{S}\ \text{iff}\ \forall \sigma \in \mathsf{ObsTTraces}(\mathcal{E}) : \mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}((\mathcal{S}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma),$$

where the function $\mathsf{ObsTTraces}(\cdot)$ is extended in a natural way to the case of TIOLTS. It follows that $\mathsf{ObsTTraces}(\mathcal{S}\,||_r\,\mathcal{E}) = \mathsf{ObsTTraces}(\mathcal{S}) \cap \mathsf{ObsTTraces}(\mathcal{E})$ and similarly $\mathsf{ObsTTraces}(\mathcal{I}\,||_r\,\mathcal{E}) = \mathsf{ObsTTraces}(\mathcal{I}) \cap \mathsf{ObsTTraces}(\mathcal{E})$.

To be able to compare rtioco and tioco, we extend the conformance relation tioco in a natural way to the case of TIOLTS. Then we have the following result.

Proposition 8 *Let $\mathcal{S}$ and $\mathcal{I}$ be two input-enabled and non-blocking TLTS. Furthermore, let $\mathcal{E}$ be an environment of $\mathcal{S}$ given as an input-enabled and non-blocking TLTS. Then we have*

$$\mathcal{I}\ \mathsf{rtioco}_{\mathcal{E}}\ \mathcal{S} \ \Leftrightarrow\ (\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{tioco}\ (\mathcal{S}\,||_r\,\mathcal{E}).$$

Proof

($\Rightarrow$) Let $\sigma \in \mathsf{ObsTTraces}(\mathcal{S}\,||_r\,\mathcal{E})$. Since $\mathsf{ObsTTraces}(\mathcal{S}\,||_r\,\mathcal{E}) = \mathsf{ObsTTraces}(\mathcal{S}) \cap \mathsf{ObsTTraces}(\mathcal{E})$, $\sigma \in \mathsf{ObsTTraces}(\mathcal{E})$. $\mathcal{I}\ \mathsf{rtioco}_{\mathcal{E}}\ \mathcal{S}$ implies $\mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma)$.

($\Leftarrow$) Let $\sigma \in \mathsf{ObsTTraces}(\mathcal{E})$. Two cases are possible:

- $\sigma \in \mathsf{ObsTTraces}(\mathcal{S})$. $\mathsf{ObsTTraces}(\mathcal{S}\,||_r\,\mathcal{E}) = \mathsf{ObsTTraces}(\mathcal{S}) \cap \mathsf{ObsTTraces}(\mathcal{E})$ implies $\sigma \in \mathsf{ObsTTraces}(\mathcal{S}\,||_r\,\mathcal{E})$. $(\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{tioco}\ (\mathcal{S}\,||_r\,\mathcal{E})$ implies $\mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma)$.
- $\sigma \notin \mathsf{ObsTTraces}(\mathcal{S})$. Thus there exist $\sigma' \in \mathsf{RT}(\mathsf{Act})$ and $b \in \mathsf{R} \cup \mathsf{Act}$ such that: $\sigma' b$ is a prefix of σ, $\sigma' \in \mathsf{ObsTTraces}(\mathcal{S})$ and $\sigma' b \notin \mathsf{ObsTTraces}(\mathcal{S})$. Since $\mathcal{S}$ is input-enabled we deduce that $b \in \mathsf{R} \cup \mathsf{Act}_{\mathsf{out}}$. Since $(\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{tioco}\ (\mathcal{S}\,||_r\,\mathcal{E})$, $\sigma' \in \mathsf{ObsTTraces}(\mathcal{E}) \cap \mathsf{ObsTTraces}(\mathcal{S})$ and $b \notin \mathsf{out}((\mathcal{S}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma')$, we deduce that $b \notin \mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma')$ either. The latter means that $\sigma' b \notin \mathsf{ObsTTraces}(\mathcal{I})$ which, in turn, means that $\sigma \notin \mathsf{ObsTTraces}(\mathcal{I})$ either. So, $\mathsf{out}((\mathcal{I}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma) = \mathsf{out}((\mathcal{S}\,||_r\,\mathcal{E})\ \mathsf{after}\ \sigma) = \emptyset$ and we are done.

■

The above result shows that tioco can capture rtioco simply by modeling the assumptions on the environment and the requirements from the system-under-test separately, and then taking their composition (See also Section 3.4.1 below).

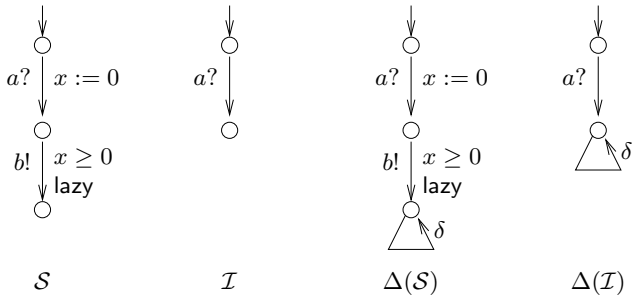


Figure 8: $\mathcal{I}$ tioco $\mathcal{S}$ but $\mathcal{I} \not\sqsubseteq^{M}_{tioco} \mathcal{S}$, for any M.

3.3.2 Comparison with the conformance relation $\sqsubseteq_{tioco}$

Another conformance relation, denoted $\sqsubseteq_{tioco}$, is introduced in [17]. The main goal of this work is to propose a testing framework which extends the notion of quiescence to the case of timed systems. The relation $\sqsubseteq_{tioco}$ bears a lot of similarity with tioco too. It is defined with respect to TIOLTS. The considered TLTS are assumed to be non-blocking and input-enabled. [5] Given two TLTS $\mathcal{S}$ the specification and $\mathcal{I}$ the implementation, the first step for comparing $\mathcal{S}$ and $\mathcal{I}$ consists in identifying the *quiescent* states of both of them. A given state s of $\mathcal{S}$ is said to be quiescent if $\forall t \in \mathsf{R} : \mathsf{out}(s \,\mathsf{after}\, t) = \mathsf{R}$ (i.e., starting from s no discrete output can be generated if no input is received). For each detected quiescent state s, a self loop $s \xrightarrow{\delta} s$ is added to the corresponding TLTS. Thus the action δ models the fact that no output must be generated. The new obtained TLTS are denoted $\Delta(\mathcal{S})$ and $\Delta(\mathcal{I})$, respectively. The relation $\sqsubseteq_{tioco}$ is defined with respect to an arbitrary duration M. Given a duration M, we let $\mathsf{ObsTTraces}_M(\mathcal{S}) = \mathsf{ObsTTraces}(\Delta(\mathcal{S})) \cap (\mathsf{R} \cdot \mathsf{Act} \cup \{M\delta\})^*$. Given a state s and a set of states S, we let

$$\mathsf{out}_M(s) = \{tb \in \mathsf{R} \cdot \mathsf{Act}_{\mathsf{in}} \mid s \overset{tb}{\Rightarrow}\} \cup \{M\delta \mid s \overset{M\delta}{\Rightarrow}\};\ \mathsf{out}_M(S) = \bigcup_{s \in S} \mathsf{out}_M(s).$$

Then, the conformance relation $\sqsubseteq_{tioco}$ with respect to M, denoted $\sqsubseteq^{M}_{tioco}$, is defined as follows

$$\mathcal{I} \sqsubseteq^{M}_{tioco} \mathcal{S} \text{ iff } \forall \sigma \in \mathsf{ObsTTraces}_M(\mathcal{S}) : \mathsf{out}_M(\Delta(\mathcal{I}) \,\mathsf{after}\, \sigma) \subseteq \mathsf{out}_M(\Delta(\mathcal{S}) \,\mathsf{after}\, \sigma).$$

$\sqsubseteq_{tioco}$ and tioco are different. It is possible to find many examples where there is conformance with respect to $\sqsubseteq_{tioco}$ but not with respect to tioco. A simple such example is given in Figure 8. For simplicity, both $\mathcal{S}$, $\mathcal{I}$, $\Delta(\mathcal{S})$ and $\Delta(\mathcal{I})$ are given as TAIO and not TLTS. For this example, it is not difficult to see that $\mathcal{I}$ tioco $\mathcal{S}$. However for any possible value of M we clearly have $\mathcal{I} \not\sqsubseteq^{M}_{tioco} \mathcal{S}$, since $\Delta(\mathcal{I})$ produces δ after receiving input a while $\Delta(\mathcal{S})$ does not.

Now, we check the other direction. That is, we assume we are given $\mathcal{S}, \mathcal{I}$ and a duration M such that $\mathcal{I} \sqsubseteq^{M}_{tioco} \mathcal{S}$ and we want to know whether $\mathcal{I}$ tioco $\mathcal{S}$ holds. For that, we first introduce the following intermediary result.

Lemma 7 *Let $\mathcal{S}$ be a non-blocking TLTS and S a set of states of $\mathcal{S}$.*

1. *For $b \in \mathsf{Act}_{\mathsf{out}}$: $b \in \mathsf{out}(S) \Leftrightarrow 0b \in \mathsf{out}_M(S)$.*

[5] In [17], the authors use the term "no forced inputs" instead of "non-blocking". Moreover, the notion of input-enabledness they introduce is slightly less strict than ours.

2. *For* $t \in \mathsf{R}$: $t \in \mathsf{out}(S) \Leftrightarrow M\delta \in \mathsf{out}_M(S)$ *or* $\exists t'b \in \mathsf{out}_M(S) \cap (\mathsf{R} \cdot \mathsf{Act}_{\mathsf{out}}) : t \leq t'$.

Proof

1. This first point is obvious. It follows immediately from the definitions of "out" and "out_M".
2. ($\Rightarrow$) We assume that $t \in \mathsf{out}(S)$. Let $s \in S$ such that $\exists s' : s \stackrel{t}{\Rightarrow} s'$. If s' is quiescent then we clearly have $M\delta \in \mathsf{out}_M(s)$. In case s' is not quiescent, since $\mathcal{S}$ is non-blocking then there must exist $t''b \in \mathsf{R} \cdot \mathsf{Act}_{\mathsf{out}}$ such that $s' \stackrel{t''b}{\Rightarrow}$. Thus, we only need to consider $t' = t + t''$ and we are done.

   ($\Leftarrow$) This direction is obvious.

■

Now, we give the following result.

Proposition 9 *Given two non-blocking and input-enabled TLTS* $\mathcal{S}$ *and* $\mathcal{I}$ *and a duration* M. *If* $\mathcal{I} \sqsubseteq^M_{tioco} \mathcal{S}$ *then* $\mathcal{I}$ tioco $\mathcal{S}$.

Proof Let $\sigma \in \mathsf{ObsTTraces}(\mathcal{S})$. Since $\mathsf{ObsTTraces}(\mathcal{S}) \subseteq \mathsf{ObsTTraces}_M(\mathcal{S})$ then $\sigma \in \mathsf{ObsTTraces}_M(\mathcal{S})$ too. By definition of $\Delta(\mathcal{S})$, it is not difficult to see that $\mathcal{S}$ after $\sigma = \Delta(\mathcal{S})$ after σ. Similarly, we have $\mathcal{I}$ after $\sigma = \Delta(\mathcal{I})$ after σ, as well. Let $b \in \mathsf{out}(\mathcal{I}$ after $\sigma)$. Two cases are possible then. Either $b \in \mathsf{Act}_{\mathsf{out}}$ or $b \in \mathsf{R}$.

- For $b \in \mathsf{Act}_{\mathsf{out}}$: By Lemma 7, we know that $0b \in \mathsf{out}_M(\Delta(\mathcal{I})$ after $\sigma)$. Moreover since $\mathcal{I} \sqsubseteq^M_{tioco} \mathcal{S}$, $0b \in \mathsf{out}_M(\Delta(\mathcal{S})$ after $\sigma)$, too . Then once again by Lemma 7, we have $b \in \mathsf{out}(\mathcal{S}$ after $\sigma)$ and we are done.
- For $b \in \mathsf{R}$: Since $\mathcal{I}$ after $\sigma = \Delta(\mathcal{I})$ after σ, $b \in \mathsf{out}(\Delta(\mathcal{I})$ after $\sigma)$. By point 2 of Lemma 7, $M\delta \in \mathsf{out}_M(\Delta(\mathcal{I})$ after $\sigma)$ or $\exists b'c \in \mathsf{out}_M(\Delta(\mathcal{I})$ after $\sigma) \cap (\mathsf{R} \cdot \mathsf{Act}_{\mathsf{out}}) : b \leq b'$. Since $\mathcal{I} \sqsubseteq^M_{tioco} \mathcal{S}$, $\mathsf{out}_M(\Delta(\mathcal{I})$ after $\sigma) \subseteq \mathsf{out}_M(\Delta(\mathcal{S})$ after $\sigma)$. Thus, $M\delta \in \mathsf{out}_M(\Delta(\mathcal{S})$ after $\sigma)$ or $\exists b'c \in \mathsf{out}_M(\Delta(\mathcal{S})$ after $\sigma) \cap (\mathsf{R} \cdot \mathsf{Act}_{\mathsf{out}}) : b \leq b'$. Then by point 2 of Lemma 7, we have $b \in \mathsf{out}(\mathcal{S}$ after $\sigma) = \mathsf{out}(\Delta(\mathcal{S})$ after $\sigma)$ and we are done.

■

In practice, it is clearly infeasible to check whether the implementation really produced a δ action or not (i.e, whether we have waited enough time or not). To alleviate this problem, the authors of [17] make the extra-assumption that the implementation $\mathcal{I}$ is M-quiescent. $\mathcal{I}$ is said to be M-quiescent if for any state s of $\mathcal{I}$ any state in (s after M) is quiescent. Intuitively this means that M is an upper bound on the "reactivity" of $\mathcal{I}$: if $\mathcal{I}$ does not react within M, it is assumed that it will never react.

For an extensive discussion of various untimed conformance relations, see [46].

3.4 Modelling issues

The main goal of this section is to illustrate some methodological aspects of our framework. We show how it is possible to alleviate modeling issues regarding environment assumptions and interface conditions between the tester and the SUT using the timed automaton model on which our framework is based.

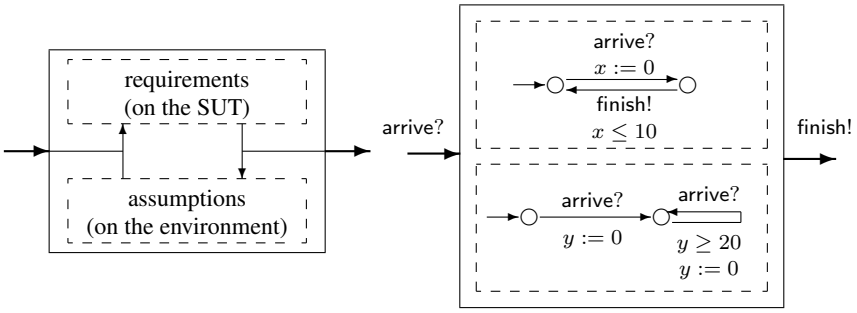


Figure 9: Specification including assumptions on the environment: generic scheme (left) and example of a task scheduler (right).

3.4.1 Modeling assumptions on the environment

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 model of specification. Figure 9 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. A simple example of such a situation is shown in Figure 9. The specification expresses 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". Notice that environment assumptions generally make the specification non-input-enabled. In the above example, the second arrive input cannot be accepted until at least 20 time units have elapsed since the first arrive. In order to keep the specification non-blocking, as it should be, the urgency of inputs must be set to lazy.

3.4.2 Modeling input/output variables

The TA model we have presented uses the notion of input/output *actions*, implying an event-based interface between the tester and the SUT. In practice, many systems communicate with the external world using input/output *variables*. We now show how to model such situations 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 corresponds to the amount of time during which the variable keeps the same value. It 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 y is an input variable and z an output variable. Consider the requirement "z will be updated at most 10 time units after y is updated". Notice that y is updated by the environment (or the tester) while z is updated by the SUT. Thus, update_y can be introduced as an input action and update_z as an output action. The specification can be modeled as a TA similar to the one for Spec_1 of Figure 3, with a replaced by update_y and b replaced by update_z (in this case the guard $2 \leq x \leq 8$ must also be changed into $x \leq 10$).

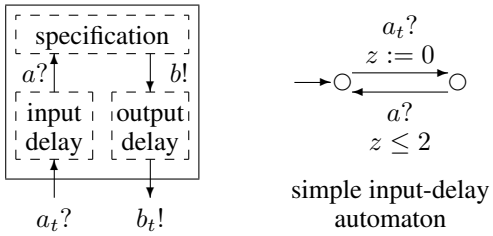


Figure 10: Specification composed with interface-delay automata.

This simplistic way of modeling supposes that updates are immediately perceived (by the SUT or by the tester) when they occur. This is obviously 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 in order to generate digital-clock tests (see Section 4.2). The Tick automaton models in some sense 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. More elaborate interfaces (e.g., event handlers with buffering, and so on) can also be modeled, as long as they can be expressed as (extended) timed automata.

3.4.3 Modeling interfacing delays

As a last example of modeling methodology, we show how to model interfacing delays between the tester and the SUT. That is the amount of time needed for messages to be transmitted between the SUT and the environment. This can again be done by composing the specification with "delay automata", as shown in Figure 10. 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. The point is that, as mentioned above, such elaborate interfaces can all be modeled explicitly. Thus, the user has full control on how the assumptions made on the tester equipment affect the generated tests.

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 [28]). *Analog-clock* tests can measure precisely the delay between two observed actions and can emit an input at any point in time. 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 11 corresponds to the tester emitting a, upon execution of the test. *Digital-clock* tests can only count how many "ticks" of a digital 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. 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 [39]), 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 [44, 29] generate non-adaptive test *sequences*.

4.1 Analog-clock tests

Analog-clock tests can be represented as either total functions or TAIO.

4.1.1 Analog-clock tests as total functions

An analog-clock test for a specification A_S over Act_τ is a total function

$$T : \mathsf{RT}(\mathsf{Act}) \to \mathsf{Act}_{\mathsf{in}} \cup \{\mathsf{wait}, \mathsf{pass}, \mathsf{fail}\}. \tag{17}$$

$T(\rho)$ specifies the action the tester must take once it observes ρ. If $T(\rho) = a \in \mathsf{Act}_{\mathsf{in}}$ then the tester emits input a. If $T(\rho) = \mathsf{wait}$ 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:

$$\exists t \in \mathsf{R} \ : \ \forall \rho \in \mathsf{RT}(\mathsf{Act}) \ : \ \mathsf{time}(\rho) > t \Rightarrow T(\rho) \in \{\mathsf{pass}, \mathsf{fail}\} \tag{18}$$

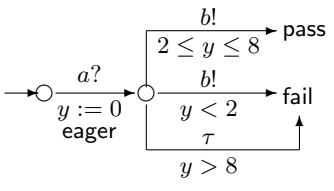
$$\forall \rho \in \mathsf{RT}(\mathsf{Act}) \ : \ T(\rho) \in \{\mathsf{pass}, \mathsf{fail}\} \Rightarrow \forall \rho' \in \mathsf{RT}(\mathsf{Act}) \ : \ T(\rho \cdot \rho') = T(\rho) \tag{19}$$

Condition (18) states that the test reaches a verdict in bounded time t (called the *completion time* of the test). Condition (19) 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 TIOLTS defined by T. The states of this TIOLTS are sequences $\rho \in \mathsf{RT}(\mathsf{Act})$. The initial state is ϵ. 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 $\forall t' \leq t : T(\rho) = \mathsf{wait}$. 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 ρ such that $T(\rho) = \mathsf{pass}$ are "collapsed" into a single sink state pass, and similarly with fail. We require that states of this TIOLTS are non-blocking as in Condition (1), unless pass or fail is reached.

4.1.2 Analog-clock tests as TA

Analog-clock tests can sometimes be represented as TAIO.[6] For example, the test defined in the right part of Figure 11 can be equivalently represented by the TAIO shown in the

[6]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.



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

Figure 11: Analog-clock test represented as a TAIO or a function.

left part. Function T is partially defined in the figure. The remaining cases are covered by the suffix-closure property of pass/fail − Condition (19). For instance, $T(a?\,9\,b!) = \mathsf{fail}$, because $T(a?\,9) = \mathsf{fail}$.

4.1.3 Execution of an analog-clock test

The execution of the test T on the implementation A_I can be defined as the *parallel composition* of the TIOLTS defined by T and A_I, with the usual *synchronization* rules for transitions carrying the same label. We will denote the product TIOLTS 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 $\mathcal{T}$.

4.1.4 Correctness requirements

We say that an analog-clock test suite $\mathcal{T}$ is *sound with respect to* A_S if

$$\forall A_I \ : \ A_I \text{ tioco } A_S \Rightarrow A_I \text{ passes } \mathcal{T}.$$

We say that $\mathcal{T}$ is *complete with respect to* A_S if

$$\forall A_I \ : \ A_I \text{ passes } \mathcal{T} \Rightarrow A_I \text{ tioco } A_S.$$

Soundness is a minimal correctness requirement. It 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 6). 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 [32]. The test shown in Figure 11 is sound and strict with respect to Spec_1 of Figure 3. Consider an arbitrary implementation A_I such that A_I passes T. Let T' be the TA obtained from the test by removing the node fail and its incoming edges. Since A_I passes T, $\mathsf{ObsTTraces}(A_I \| T) \subseteq \mathsf{ObsTTraces}(T')$. Moreover clearly,

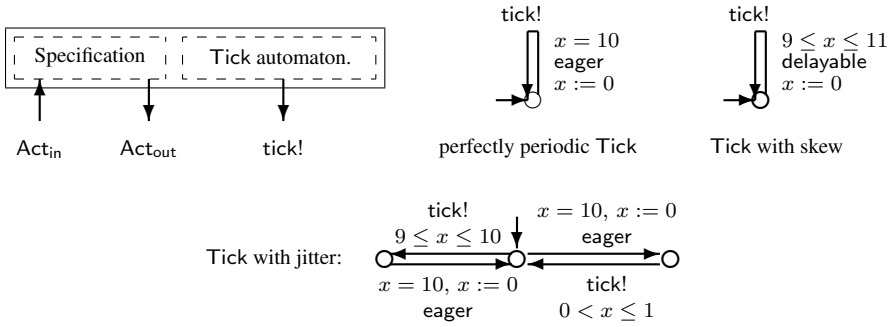


Figure 12: Extending the specification with a tester clock model and possible such models.

$\mathsf{ObsTTraces}(T') \subseteq \mathsf{ObsTTraces}(A_S)$. Thus, $\mathsf{ObsTTraces}(A_I \| T) \subseteq \mathsf{ObsTTraces}(A_S)$. By Lemma 3, $A_I \| T$ tioco A_S. 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 Act_τ and let tick be a new output action, not in Act_τ. A digital-clock test for A_S is a total function

$$D : (\mathsf{Act} \cup \{\mathsf{tick}\})^* \rightarrow \mathsf{Act}_{\mathsf{in}} \cup \{\mathsf{wait}, \mathsf{pass}, \mathsf{fail}\}. \tag{20}$$

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 that the tester's digital clock is modeled as a Tick automaton, which is a special TAIO with a single output action tick. 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 $[n, n+1)$, for the case of a periodic digital-clock with one time unit period.

The digital-clock can be either periodic or not. Three possible Tick automata are shown in Figure 12. The first models a perfectly periodic clock with period equal to 10 time units: in this case, the n-th tick occurs precisely at time $10n$. The second automaton models a clock with "skew": in this case, the n-th tick may occur anywhere in the interval $[9n, 11n]$. The third automaton models a clock with "jitter": in this case, the n-th tick may occur anywhere in the interval $[10n - 1, 10n + 1]$. Notice that this automaton contains unobservable transitions (the ones with deadline eager).

The above examples show different models of digital clocks. How realistic the model of a digital clock is depends on the digital clock itself as well as the application. What we have aimed to show is that our framework allows the user to make this decision explicitly, instead of relying on implicit (unrealistic) assumptions encoded in the framework.

Validity conditions similar to those for analog-clock apply to digital-clock. A digital-clock test D defines a TIOLTS with states in $(\mathsf{Act} \cup \{\mathsf{tick}\})^*$ and labels in $\mathsf{Act} \cup \{\mathsf{tick}\} \cup \mathsf{R}$. Given state π, if $D(\pi) = \mathsf{wait}$ then π has a self-loop transition labeled with t, for all $t \in \mathsf{R}$. The reason such transitions are missing from states such that $D(\pi) = a \in \mathsf{Act}_{\mathsf{in}} \cup \{\mathsf{pass}, \mathsf{fail}\}$ is that we assume that the digital-clock test emits a immediately after the last event in π is observed.

4.2.1 Execution of a digital-clock test

The execution of a digital-clock test is defined by forming the parallel product of three TIOLTSs, namely, the ones of the test D, the implementation A_I, and the Tick automaton. 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 $A_I \| \mathsf{Tick} \| D$.

4.2.2 Correctness requirements

Given a set of digital-clock tests $\mathcal{D}$ and a Tick automaton, $\mathcal{D}$ is said to be sound with respect to A_S and Tick if

$$\forall A_I \ : \ A_I \ \mathsf{tioco} \ A_S \Rightarrow A_I \ \mathsf{passes} \ \mathcal{D}.$$

We say that $\mathcal{D}$ is complete with respect to A_S and Tick if

$$\forall A_I \ : \ A_I \ \mathsf{passes} \ \mathcal{D} \Rightarrow A_I \ \mathsf{tioco} \ A_S.$$

We say that a test D is strict with respect to A_S and Tick if

$$\forall A_I \ : \ A_I \ \mathsf{passes} \ D \Rightarrow A_I \| \mathsf{Tick} \| D \ \mathsf{tioco} \ A_S \| \mathsf{Tick}.$$

In this case for the relation conformance tioco between $A_I \| \mathsf{Tick} \| D$ and $A_S \| \mathsf{Tick}$, the action tick must be considered as an output.

Digital-clock tests 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 $1+\epsilon$ before the next tick happens. If the output is b, in both cases the tester will observe $\mathsf{tick}\, b\, \mathsf{tick}$. Thus the faulty behavior gives the same digital-clock observation as the non-faulty one and the tester will announce pass in both cases.

A weaker notion of strictness may be introduced for the case of digital-clock testing. It will be weaker in the sense that a digital-clock test cannot be as strict as an analog-clock one due to its limited observation capability. The formal definition of digital-clock strictness can be based on the use of an untimed conformance relation instead of tioco and the inclusion of the Tick automaton in the definition. Providing such a formal definition is beyond the scope of this paper.

5 Test Generation

We adapt the untimed test generation algorithm of [45]. 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 Figure 13. 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 $Pass$.

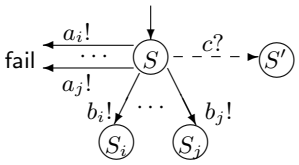


Figure 13: Generic test-generation scheme.

Two features of the above algorithm are worth noting. First, the algorithm is only partially specified. One may say the algorithm is "non-deterministic". 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 6.

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 directly 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 [47]. 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 or *on-line*, meaning that the test is generated at the same time it is executed.

More precisely, the tester 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:

$$\mathsf{dsucc}(S,a) = \{s' \mid \exists s \in S \ : \ s \xrightarrow{a} s'\} \tag{21}$$

$$\mathsf{tsucc}(S,t) = \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \ : \ \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho} s'\} \tag{22}$$

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 ρ 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 [47]. In fact the sets S are generally *dense* due to the continuous state-space of the clocks. The sets are represented *symbolically* using simple constraints on clocks. For instance, the constraint $1 \leq x \leq 2 \wedge x = y$ represents the fact that clock x has some value within $[1,2]$ and clock y is equal to x. The constraints are implemented

using a matrix data structure called DBM (*difference bound matrix*) [7, 24]. Computing successor nodes is also done symbolically, using a *bounded-time reachability analysis* for timed automata, as shown in [47, 34].

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 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.

On-line analog-clock test generation is performed by Algorithm 1. The algorithm keeps running as long as no non-conformance is detected. At any time the tester can stop testing and declare pass.

The algorithm uses the following notation. Given a nonempty set X, $\mathsf{pick}(X)$ chooses randomly an element in X. Given a set of states S, $\mathsf{valid\_inputs}(S)$ is defined as the set of valid inputs at S, that is: $\{a \in \mathsf{Act_{in}} | \mathsf{dsucc}(\mathsf{tsucc}(S, 0), a) \neq \emptyset\}$.

Notice that for practical reasons, we assume that the SUT is a rational-delay TLTS and the clock x of Algorithm 1 ranges over rational values. Thus, it is possible to consider again the DBM structure for symbolic successor computation.

```
1  S ← tsucc({s_0^{A_S}}, 0);
2  while(true)
3    x ← 0; /* x is a clock measuring elapsing time */
4    await(output b is received at x < T or x = T)
5    if (b received at x)
6      S ← dsucc(tsucc(S, x), b);
7    else
8      S ← tsucc(S, T);
9    endif;
10   if (S = ∅)
11     announce fail;
12     exit ;
13   endif;
14   if (valid_inputs(S) ≠ ∅)
15     i ← pick({0, 1}); /* 0 to send an input and 1 to continue observation */
16   endif;
17   if (i = 0)
18     a ← pick(valid_inputs(S));
19     S ← dsucc(S, a);
20   endif;
21 endwhile;
```

Algorithm 1: On-the-fly analog-clock test generation.

5.1.1 Soundness, strictness and completeness

Next we prove that Algorithm 1 is sound.

Proposition 10 *If verdict* fail *is observed while executing Algorithm 1, then* A_I ~~tioco~~ A_S.

Proof Let $\sigma = a_0 a_1 \cdots a_n \in \mathsf{RT}(Act)$ the trace corresponding to the interaction between the tester and A_I from the starting of the algorithm until the announcement of fail. Let $\sigma_{n-1} = a_0 a_1 \cdots a_{n-1}$. According to the algorithm, $a_n \in \mathsf{R} \cup \mathsf{Act_{out}}$. Thus, $\mathsf{out}(A_I \text{ after } \sigma_{n-1}) \neq \emptyset$. It contains at least a_n. Since fail is declared, $\mathsf{out}(A_S \text{ after } \sigma_{n-1}) = \emptyset$. Hence, $\mathsf{out}(A_I \text{ after } \sigma_{n-1}) \not\subseteq \mathsf{out}(A_S \text{ after } \sigma_{n-1})$ and A_I ~~tioco~~ A_S. ■

We prove that Algorithm 1 is strict too.

Proposition 11 *Let* A_I *be a possible implementation and* T *a test generated by Algorithm 1:*

$$A_I || T \text{ ~~tioco~~ } A_S \;\Rightarrow\; A_I \text{ fail } T.$$

Proof Since Algorithm 1 works online, T is a simple trace $\sigma = a_0 a_1 \cdots a_n \in \mathsf{RT}(Act)$. Since, $A_I||T$ ~~tioco~~ A_S, there exists a prefix $\sigma' = a_0 a_1 \cdots a_k$ of σ such that $a_{k+1} \in \mathsf{out}(A_I||T \text{ after } \sigma')$ and $a_{k+1} \notin \mathsf{out}(A_S \text{ after } \sigma')$. Thus, $(A_S \text{ after } a_0 a_1 \cdots a_{k+1}) = \emptyset$. Then by Algorithm 1, the execution of $a_0 a_1 \cdots a_{k+1}$ must lead to verdict fail and we are done. ■

Algorithm 1 is parameterized with the time elapse T. Thus, the tester waits at most T time units before announcing that the timeout occurs if this duration is not accepted by the specification.

Furthermore, Algorithm 1 is "complete" in the sense that, for any non-conforming implementation given as a rational TLTS A_I, there exists an execution of the algorithm that detects non-conformance of this implementation.

Proposition 12 *Let* A_I *be a rational-delay TLTS. If* A_I ~~tioco~~ A_S *then there exists an execution of Algorithmm 1 that announces* fail.

Proof If A_I ~~tioco~~ A_S then there exists $\sigma \in (\mathsf{Q} \cup \mathsf{Act})^* \cap \mathsf{ObsTTraces}(A_S)$ and $a \in \mathsf{Q} \cup \mathsf{Act_{out}}$ such that $a \in \mathsf{out}(A_I \text{ after } \sigma)$ and $a \notin \mathsf{out}(A_S \text{ after } \sigma)$. Since $\mathsf{out}(A_I \text{ after } \sigma) \neq \emptyset$, $\sigma \in \mathsf{ObsTTraces}(A_I)$ as well. By induction on the length of σ, it is easy to show that the tester and A_I may interact together to produce σ. We assume $\sigma = a_0 a_1 \cdots a_n$. For $i = 1, \cdots, n$:

- If $a_i \in \mathsf{Q}$, the algorithm will force pick( ) to choose this duration. Since a_i is accepted by both A_S and A_I, the tester continues running the algorithm and does not announce fail.
- If $a_i \in \mathsf{Act_{in}}$, since a_i is accepted by A_S the algorithm will detect that $\mathsf{valid\_inputs}(S) \neq \emptyset$ and will force the tester to send immediately a_i to A_I.
- If $a_i \in \mathsf{Act_{out}}$, the algorithm decides to wait until A_I generates an output. Since a_i is accepted by A_I, the implementation will output a_i. The tester receives it immediately and does not announce fail since a_i is accepted by A_S as well.

After generating σ, the tester will wait for an output to be generated by the implementation. If $a \in \mathsf{Q}$, the tester calculates $S' = \mathsf{tsucc}(S, a)$, where S is the current estimation of the tester. If $a \in \mathsf{Act_{out}}$, it calculates $S' = \mathsf{dsucc}(S, a)$. In both cases $S' = \emptyset$ since $a \notin \mathsf{out}(A_S \text{ after } \sigma)$. Thus, the tester announces immediately fail and the non-conformance is detected.

■

5.2 Generating digital-clock tests

The conformance relation tioco is "ideal" in the sense that it captures non-conformance of a SUT at an infinite-precision time-measuring level. For instance in the case of one-time-unit periodic digital-clock, if the guard $1 \leq x \leq 5$ of SUT Impl_3 of Figure 3 was replaced by $1.9 \leq x \leq 5$ then Impl_3 would still be non-conforming. In fact, the same would be true if the guard was replaced by $2 - \epsilon \leq x \leq 5$, for any $\epsilon > 0$. It is reasonable to define tioco in such an "ideal" way, since we do not want conformance to depend on implementation details such as tester equipment. On the other hand, the tester's time-observation capabilities are limited in practice: testers only dispose of a finite-precision digital clock (a counter) and cannot distinguish among observations which elude their clock precision. Our framework takes this limitation into account. First, we allow the user to explicitly model the assumptions on the tester's digital clock. Second, we generate tests with respect to this model.

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-line or off-line. This is a matter of a space/time trade-off. The on-line method does not require space to store the generated tests. On the other hand, a test computed on-line has a longer reaction time than a test which has been computed off-line, because the former takes more time to compute the next state of the test. Independently of which option we choose, we proceed as follows.

We first form the product $A_S^{\mathsf{Tick}} = A_S \| \mathsf{Tick}$.The extended specification A_S^{Tick} may also include other automata to model environment assumptions, interface delays, etc., as shown previously. This yields a new TAIO which has as inputs the inputs of A_S and as outputs the outputs of A_S plus the new output tick of Tick. Notice that A_S and Tick do not synchronize on any discrete transitions, they only synchronize in time (time elapses at the same rate for both). We define the set of *observable discrete traces* of A_S with respect to Tick to be:

$$\mathsf{ObsDTraces}(A_S^{\mathsf{Tick}}) = \{P_{\mathsf{Act}\cup\{\mathsf{tick}\}}(\sigma) \mid \sigma \in \mathsf{ObsTTraces}(A_S^{\mathsf{Tick}})\}. \tag{23}$$

For example for the 10 time-unit periodic Tick automaton of Figure 12, if the trace $\sigma = a\,11\,b\,9\,c\,10 \in \mathsf{ObsTTraces}(A_S)$ then $\lambda = a\,\mathsf{tick}\,b\,\mathsf{tick}\,c\,\mathsf{tick} \in \mathsf{ObsDTraces}(A_S^{\mathsf{Tick}})$. The trace λ is called the *digitization* of σ (with respect to Tick).

We also define the following operator on A_S^{Tick}:

$$\mathsf{usucc}(S) = \{s' \mid \exists s \in S \,.\, \exists \rho \in \mathsf{RT}(\{\tau\}) \;:\; s \xrightarrow{\rho} s'\}. \tag{24}$$

$\mathsf{usucc}(S)$ contains all states which can be reached by some state in S via a sequence ρ which contains no observable actions. Notice that, by construction of A_S^{Tick}, the duration of ρ is bounded: since tick is observable and has to occur after a bounded duration.

Finally, we apply the generic test-generation scheme presented above. The root of the test tree is defined to be $S_0 = \{s_0^{A_S^{\mathsf{Tick}}}\}$. 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 $S \xrightarrow{a} \mathsf{fail}$. For this first possible choice, the node S is said to be an *output node*. 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 $S \xrightarrow{b} S''$. For this second choice, S is said to be an *input node*. 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.

Off-line digital-clock test generation is performed by Algorithm 2. We use the same notation as in Algorithm 1. It is worth noticing that Algorithm 2 may produce a test tree of infinite depth. To avoid this we can force the test generator to go to "case($i = 2$)" when the depth of the test becomes too big. The choices "case($i = 0$)" and "case($i = 1$)" correspond to the possibilities of considering the current node S as an output or an input node, respectively.

---

1 $S \leftarrow \{s_0^{A_S^{\mathsf{Tick}}}\}$;
2 $T \leftarrow$ the one-node tree with root S;
3 **while**(true)
4 **foreach**(leaf S of T distinct from $Pass$ and $Fail$)
5 **if**(valid_inputs(S) $\neq \emptyset$)
6 $i \leftarrow$ pick($\{0, 1, 2\}$);
7 **else** $i \leftarrow$ pick($\{1, 2\}$);
8 **endif**;
9 **case**($i = 0$) :
10 $b \leftarrow$ pick(valid_inputs(S));
11 $S' \leftarrow$ dsucc(tsucc($S, 0$), b);
12 **append** edge $S \xrightarrow{b} S'$ to T;
13 **case**($i = 1$) :
14 **foreach**($a \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$)
15 $S' \leftarrow$ dsucc(usucc(S), a);
16 **if**($S' \neq \emptyset$)
17 **append** edge $S \xrightarrow{a} S'$ to T;
18 **else append** edge $S \xrightarrow{a} Fail$ to T;
19 **endif**;
20 **endforeach**;
21 **case**($i = 2$) : **replace** S with $Pass$ in T;
22 **endforeach**;
23 **endwhile**;

Algorithm 2: Off-line digital-clock test generation.

It is not difficult to see that Algorithm 2 can be transformed in a straightforward way to an on-line digital-clock test generation algorithm. In that case, there will be no longer any need for storing the considered test. The tester will observe the outputs generated by the SUT and the tick-actions generated by the digital clock. From time to time, the tester will decide to send inputs to the SUT. The choices of the tester are made non-deterministically. As soon as the tester reaches an empty set, it will announce fail and stop testing.

5.2.1 Soundness and completeness

Now, we prove that Algorithm 2 generates only sound tests.

Proposition 13 *Let T be a test generated by Algorithm 2. If verdict* fail *is observed while applying the test-case T on A_I, then A_I ~~tioco~~ A_S.*

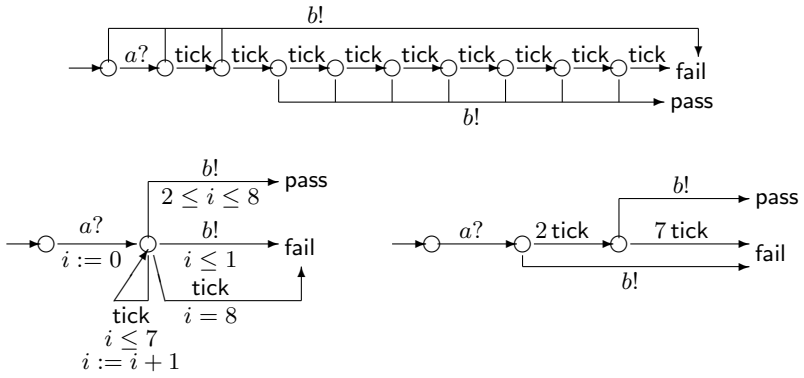


Figure 14: A digital-clock test (top) and two alternative representations (bottom).

Proof Let $\sigma = a_0 a_1 \cdots a_n \in (\mathsf{Act} \cup \{\mathsf{tick}\})^*$ the trace corresponding to the interaction between the tester and A_I from the starting of of T until the announcement of fail. Let $\sigma_{n-1} = a_0 \cdots a_{n-1}$. According to the algorithm $a_n \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$. Since no fail is announced during the occurrence of σ_{n-1}, the set S of possible reachable states after executing σ_{n-1} is non-empty. However since the execution of a_n leads to fail, $S' = \mathsf{dsucc}(\mathsf{usucc}(S), a) = \emptyset$. Moreover since the interaction between the tester and A_I is instantaneous, there exists $\lambda \in \mathsf{ObsTTraces}(A_I)$ such that σ_{n-1} is the digitization of λ. Two cases are possible:

- $\lambda \in \mathsf{ObsTTraces}(A_S)$: After the execution of λ a time delay t will elapse before a_n is observed.
  - If $t \notin \mathsf{out}(A_S \,\mathsf{after}\, \lambda)$: Since $t \in \mathsf{out}(A_I \,\mathsf{after}\, \lambda)$, we deduce that $A_I \not\mathsf{tioco}\, A_S$.
  - If $t \in \mathsf{out}(A_S \,\mathsf{after}\, \lambda)$: $a_n = \mathsf{tick}$ can not be true since: (1) tick is the action with the highest priority and (2) if tick occurred in $A_I || \mathsf{Tick}$ after t elapses then it will do so in $A_S || \mathsf{Tick}$ as well. The latter will be a contradiction with $S' = \emptyset$. Thus, $a_n \in \mathsf{Act}_{\mathsf{out}}$. Then for the trace λt, we have $a_n \in \mathsf{out}(A_I \,\mathsf{after}\, \lambda t)$. $S' = \emptyset$ implies that $a_n \notin \mathsf{out}(A_S \,\mathsf{after}\, \lambda t)$. Hence, $A_I \not\mathsf{tioco}\, A_S$.
- $\lambda \notin \mathsf{ObsTTraces}(A_S)$: Let $\lambda' \in \mathsf{RT}(\mathsf{Act})$ and $b \in \mathsf{Act} \cup \mathsf{R}$ such that $\lambda' b$ is a prefix of λ, $\lambda' \in \mathsf{ObsTTraces}(A_S)$ and $\lambda' b \notin \mathsf{ObsTTraces}(A_S)$. If $b \in \mathsf{Act}_{\mathsf{in}}$, that will be a contradiction with the fact that Algorithm 2 chooses only valid input-actions (i.e., actions which are in valid_inputs( )). Thus, $b \in \mathsf{Act}_{\mathsf{out}} \cup \mathsf{R}$ and we are done.

■

Algorithm 2 is not complete in general. That is due to the unprecision of digital-clocks. For example, consider a periodic tick-automaton with period 2 time units (that is, ticks are generated at times $0, 2, 4, ...$). Suppose the specification states that output a should be emitted no later than time 3. A tester that observes tick tick a must accept the sequence as conforming, since a may have been emitted anywhere in the interval $[2, 4]$. Thus, it *might* have been emitted before time 3, and in order for the tester to be sound, it must not announce fail. Of course, this means that a non-conforming implementation that emits a at time 3.5 would remain undetected.

As in [38], we can prove completeness of the algorithm under some extra assumptions. The considered assumptions are mainly based on the so-called *digitization techniques* [28].

5.2.2 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 [31] 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 14 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 (sometimes also called *Kolmogorov complexity*). 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 14 is shown in the bottom-right of the figure.

5.3 Generating TA testers: the monitor case

The motivation for representing analog-clock tests as deterministic timed automata arises from the fact that on-line testing requires a time-efficient reachability algorithm, since the tester must be able to react to the SUT in real-time.

The problem of generating an analog-clock test which is a TAIO can be anything from trivial to undecidable, depending on its precise definition. If we require a test which is only sound, then the problem is trivial, because a test always announcing "pass" is sound. On the other hand, if we require a test which is also complete, when such a test exists,[7] then we can show that the problem is undecidable, by reducing the timed automata determinization problem [48].

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

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

5.3.1 "One-clock determinization" of TA

For pedagogical reasons, we first explain our technique for plain timed automata, which can be seen as TAIO with an empty set of input actions. For such an automaton A, the technique amounts to determinizing A "as best as possible", given that we can only use one

[7] It may not exist because the specification is non-deterministic whereas the test has to be deterministic.

clock. Formally, the deterministic counterpart of A, denoted A_{mon}, will accept a superset of $\mathsf{TTraces}(A)$. Notice that A may contain unobservable actions and non-determinism. Viewing A as the specification, A_{mon} is a *monitor* for A.

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

5.3.2 The equivalence relation "$\sim_S^a$"

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

$$\mathsf{E}_a(S, u) = \{e \in \mathsf{E}_a \mid \exists s \in S : y(s) = u \wedge s \models e\}.^{8} \tag{25}$$

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

$$u_1 \sim_S^a u_2 \quad \text{iff} \quad \mathsf{E}_a(S, u_1) = \mathsf{E}_a(S, u_2). \tag{26}$$

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

Let us illustrate the meaning of $\sim_S^a$ with the example shown in Figure 15. We assume that $S = (q, -2 \leq x - y \leq 1)$ and that q has two outgoing edges e_1 and e_2 labeled a, with guards $\phi_1 \equiv x \leq 3$ and $\phi_2 \equiv x \geq 2$, respectively. Then, $\sim_S^a$ induces three equivalence classes, namely, $y < 1$, $1 \leq y \leq 5$ and $y > 5$. Indeed, given the assumption $-2 \leq x - y \leq 1$, $y < 1$ implies $x < 2$. Thus, when $y < 1$ we know that ϕ_2 does not hold, therefore, e_2 is not enabled. Similarly, when $y > 5$ we know that e_1 is not enabled. When $1 \leq y \leq 5$, both e_1 and e_2 may be enabled. It is important to note that *not all* states in S for which $1 \leq y \leq 5$ satisfy ϕ_1, and similarly for ϕ_2. However, given our information on y, we cannot be sure. Thus, we need to include both e_1 and e_2 in the set of possible enabled edges, given the constraint $1 \leq y \leq 5$.

5.3.3 Monitor construction

We now explain the construction of the monitor automaton A_{mon}. A location of A_{mon} is associated with a set of extended states of A, $S \subseteq S_A^y$. For each action a, for each equivalence class ψ in the (coarsest) partition induced by $\sim_S^a$, A_{mon} has an edge $e = (S, S', \psi, \{y\}, \mathsf{lazy}, a)$, where the destination location S' is computed as follows:

$$S' = \mathsf{succ}(S \cap \psi, a) \tag{27}$$

[8] For $s = (q, v)$ and $e = (q', q'', \psi, r, d, a)$, $s \models e$ is a shortcut for $q = q'$ and $v \models \psi$.

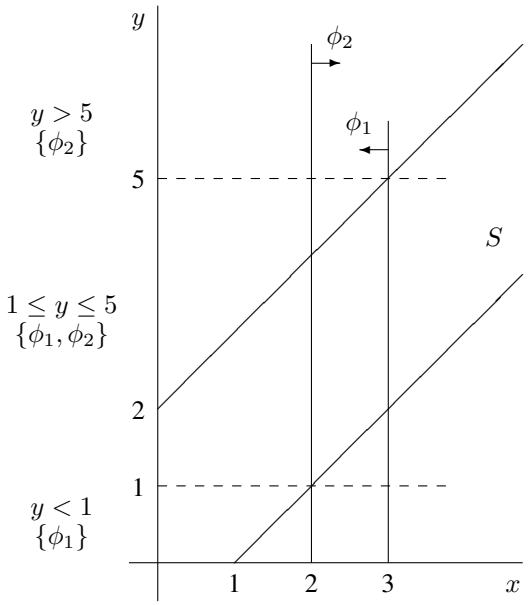


Figure 15: Illustration of the $\sim_S^a$ equivalence.

where

$$\mathsf{succ}(S \cap \psi, a) = \mathsf{usucc}(\mathsf{dsucc}(S \cap \psi, a)) \tag{28}$$

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

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

The above algorithm is essentially a *subset construction* for A, with the addition that clock y is used to infer knowledge about states that A can possibly be in. The construction relies on repeating two basic steps: (a) computing the partition induced by equivalences $\sim_S^a$, and (b) computing successor locations S' using reachability. We show how step (a) can be implemented below. As for step (b), standard symbolic reachability techniques, coupled with so-called *extrapolation abstractions* can be used to ensure that the number of possible locations of A_{mon} remains finite [23, 4, 10].

5.3.4 Computing the coarsest partition induced by "$\sim_S^a$"

A simple algorithm for computing the coarsest partition induced by $\sim_S^a$ is the following. Given a constraint ψ on clock y, let $\mathsf{E}_a^{S,\psi} = \{e \in \mathsf{E}_a \mid S \cap (\psi \wedge \mathsf{guard}(e)) \neq \emptyset\}$, where $\mathsf{guard}(e)$ is the guard of edge e. $\mathsf{E}_a^{S,\psi}$ contains all edges labeled a whose guards may be satisfied by a state in S where y lies in the interval ψ. In other words, $\mathsf{E}_a^{S,\psi}$ is the union of $\mathsf{E}_a(S,u)$ over all values u satisfying ψ.

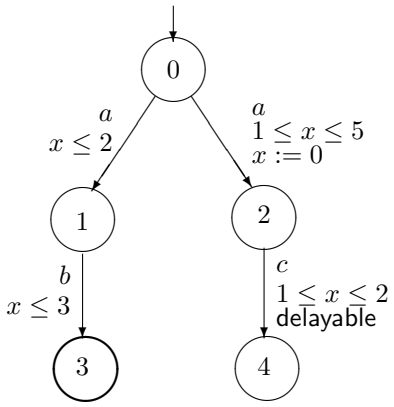


Figure 16: A non-deterministic timed automaton.

Now, let K be the greatest constant appearing in a constraint defining S or a guard of an edge in E_a. For each ψ in the set of intervals

$$\{[0,0],(0,1),[1,1],(1,2),...,[K,K],(K,\infty)\},$$

compute $\mathsf{E}_a^{S,\psi}$. For this, the condition $S \cap (\psi \wedge \mathsf{guard}(e)) \neq \emptyset$ needs to be checked. This can be done symbolically, using standard techniques and data structures such as DBMs [24]. Once $\mathsf{E}_a^{S,\psi}$ is computed for all intervals ψ, the coarsest partition is obtained by "merging" (i.e., taking the union of) intervals having the same set $\mathsf{E}_a^{S,\psi}$. For the example of Figure 15, $\mathsf{E}_a^{S,y<1} = \{e_1\}$, $\mathsf{E}_a^{S,1\leq y\leq 5} = \{e_1, e_2\}$ and $\mathsf{E}_a^{S,y>5} = \{e_2\}$. Notice that the correctness of the above algorithm relies on the fact that all values in an interval $(i, i+1)$ are equivalent, and the same is true for the interval (K, ∞). This is because constraints only have integer constants.

5.3.5 Accepting and rejecting states

It remains to define the accepting and rejecting states of A_{mon}. Given $S \subseteq S_A^y$, let $S_{/y}$ be the projection of S on clock y, that is, $S_{/y} = \{u \in \mathsf{R} \mid \exists s \in S : y(s) = u\}$. Then, all states $(S, S_{/y})$ of A_{mon} are accepting, provided $S \neq \emptyset$. The rest of the states are rejecting.

The different steps for constructing the monitor automaton are given in Algorithm 3. It is worth noting that the latter bears similarity with the common reachability algorithm.

5.3.6 Example of monitor construction

Let us give an example illustrating the construction of A_{mon}. Consider the non-deterministic timed automaton shown in Figure 16. All its edges are lazy, except the one from location 2 to location 4, which is delayable. Its one-clock monitor automaton is shown in Figure 17. Not all locations and edges of the monitor are shown, in order not to overload the figure. In particular, the empty location and all edges leading to it are not shown. For instance, there is an edge labeled a with guard $y > 5$ from the initial location to the empty location, since a is not accepted if it arrives after 5 time units from start. Apart from the empty location, the rejecting states of the monitor are at location $S = (2, x = y \leq 2)$ and for $y > 2$ (notice that $S_{/y} = y \leq 2$). This is because c must be received at most 2 time units after a, in order to be accepted. Note that there are no such rejecting states at location $S' = (1, 1 \leq x - y \leq 2) \cup (2, x = y \leq 2)$. This is because the monitor does not know

```
1  S_0 ← usucc({s_0^{A^y}});
2  list_1 ← {S_0};
3  list_2 ← ∅;
4  T ← the one node TA the initial location of which is S_0;
5  compute the rejecting states corresponding to S_0;
6  while(list_1 ≠ ∅)
7    S ← pick(list_1);
8    foreach(a ∈ Act_out)
9      P ← the coarsest partition induced by ∼_S^a;
10     foreach(ψ ∈ P)
11       S' ← succ(S ∩ ψ, a);
12       if (S' ≠ ∅)
13         append edge (S, S', ψ, {y}, lazy, a) to T;
14         compute the rejecting states corresponding to S';
15         if (S' ∉ list_2) list_1 ← list_1 ∪ {S'};
16         endif;
17       else append edge (S, fail, ψ, {y}, lazy, a) to T;
18       endif;
19     endforeach;
20   endforeach;
21   list_1 ← list_1 \ {S};
22   list_2 ← list_2 ∪ {S};
23 endwhile;
```

Algorithm 3: One-clock determinization of TA.

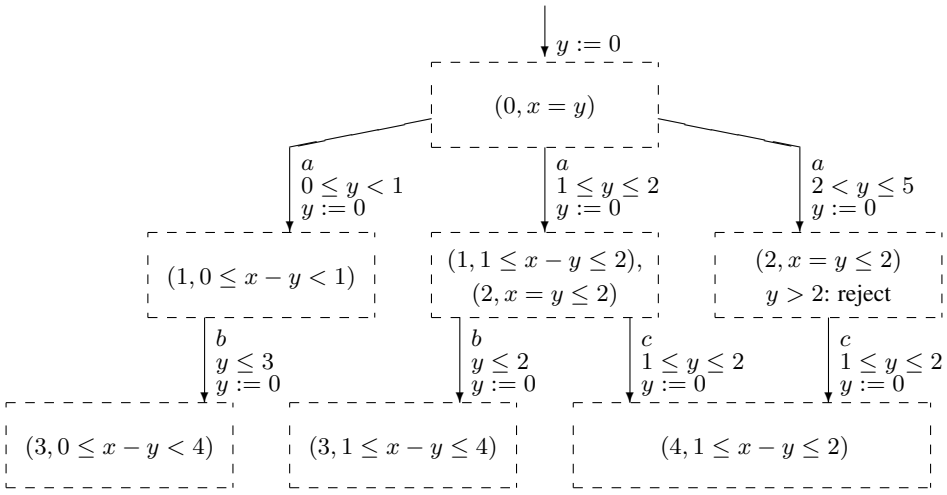


Figure 17: The one-clock deterministic monitor of the automaton of Figure 16.

whether the original automaton is at location 1 or 2, and there is no urgency at location 1. Indeed, $S'_{/y} = \mathit{true}$.

5.3.7 The use of extrapolation techniques

A question that arises is the termination of Algorithm 3. Using the operator usucc may lead to an infinite set of states (zones) [23]. Moreover, there are cases where the exact set of reachable states is not representable as finite unions of zones [10]. The second problem can be avoided only by restricting the class of timed automata considered to *diagonal-free* automata, that is, without guards of the form $x - y \le c$ [10]. To alleviate the first problem, we use the so-called *extrapolation abstractions* [23, 4, 10]. These abstractions result in a finite state space, thus ensuring termination of reachability-based algorithms.

Extrapolation abstractions rely on the maximal integer constants used in guards of the timed automaton in question. In our case, this raises an issue when determining the constant that is to bound the space of the monitor clock y. Indeed, this clock does not "participate" in any guard a-priori (before the construction of the monitor) thus there is no reference constant to use. We can show that increasing the maximal constant for y amounts to increasing the observational power of the monitor. In fact, there are cases where there is no "optimal" monitor: the greater the maximal constant allowed for y is, the more precise A_{mon} will be, in the sense of how "close" the language of A_{mon} is to the language of A.

An example is shown in Figure 18. The TAIO shown in the figure can produce a single output a at any time k, where $k \in \{1, 2, ...\}$. It can be seen that for any such k, a monitor able to compare y to constants up to k is "less accurate" than a monitor able to compare y to constants up to $k + 1$. Indeed, the former cannot distinguish between $a!$ happening precisely at time k or at time strictly greater than k, while the latter can.

Another observation to be made about Algorithm 3 is the following.[9] There are cases where it is possible to build more precise monitor automata by using partitions which are finer than those induced by the equivalence relations "$\sim^a_S$", used in Algorithm 3. We use the example given in Figure 19 to illustrate this. It can be checked that by applying Algorithm 3 on the TA A, we obtain the monitor automaton A_{mon}.Note that though A

[9] This issue has been pointed to us by Fabrice Chevalier and Patricia Bouyer.

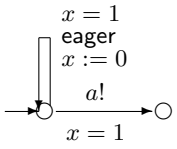


Figure 18: A TAIO which can produce $a!$ at times 1, 2, 3, $\cdots$.

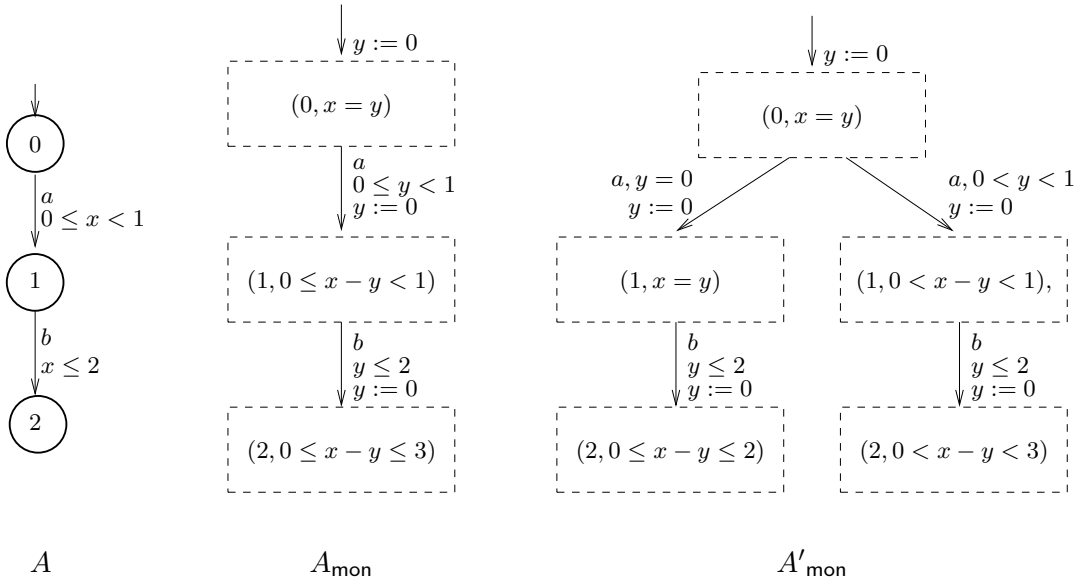


Figure 19: Computing a more precise monitor automaton: i.e., A'_{mon} more precise than A_{mon}.

is deterministic and has only one clock the two TA A and A_{mon} are distinct from each other. This is due the fact that clock x of TA A is not reset after each transition as the clock y is supposed to be. The initial location of A_{mon} is labeled with the set of states $S = (0, x = y)$. For the output action a, we consider the relation equivalence $\sim_S^a$. The coarsest partition with respect to $\sim_S^a$ is made up of the two intervals $0 \le y < 1$ and $1 \le y$. The first intervall is shown in the figure while the second is hidden since it leads to the *Fail*-location. The monitor automaton A'_{mon} is obtained by considering a finer partition made up of the intervals $y = 0$, $0 < y < 1$ and $1 \le y$. That is the only change we bring to Algorithm 3. All the other steps remain unchanged. It is not difficult to check that $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(A_{\mathsf{mon}})$ and $\mathsf{ObsTTraces}(A) \subseteq \mathsf{ObsTTraces}(A'_{\mathsf{mon}})$. Thus, both A_{mon} and A'_{mon} are monitor automata for A. However, A'_{mon} is more precise than A_{mon} since $\mathsf{ObsTTraces}(A'_{\mathsf{mon}}) \subset \mathsf{ObsTTraces}(A_{\mathsf{mon}})$. This inclusion is strict since for all t, such that $2 < t < 3$, the traces "$0\,a\,t\,b$" are in $\mathsf{ObsTTraces}(A_{\mathsf{mon}})$ but not in $\mathsf{ObsTTraces}(A'_{\mathsf{mon}})$.

In general, a "forward" algorithm such as the one we propose above cannot determine the partition that is sufficiently fine for precision purposes, thus, it has to rely on the finest possible partition, namely, the region graph. This is the approach taken in [11]. Our method opts for monitors that have weaker observational power, however, are much more efficient to construct and store.

5.4 Generating TA testers: the general case

We now consider the general case of TAIO with both input and output actions. In this case, the monitor becomes a tester, since it must supply inputs to the SUT. Formally, the tester is an analog-clock test TAIO, denoted A_{test}, as defined in Section 4.

5.4.1 Input and output locations

As for the case of digital-clock tests, an analog-clock test TAIO has two distinct types of locations, namely *input locations* and *output locations*. The outgoing edges from an input location are all labeled with output actions. When the tester is occupying such a location, it just waits for outputs coming from the implementation and reacts accordingly. For the second type of locations, each location must have exactly one outgoing edge labeled with an input action and all the other edges must be labeled with output actions. For each output location, the tester has an input action that it must send to the implementation at a precise timing. When the tester occupies some output location, it waits till the time for sending the corresponding input happens. If an output is received before that time the tester reacts accordingly. Otherwise, it will send the specific input action to the implementation at the precise chosen time and continues the execution of the test strategy accordingly.

5.4.2 Generation algorithm

The algorithm for constructing A_{test} is a generalization of the algorithm for building A_{mon}. As with A_{mon}, each location of A_{test} is a set $S \subseteq S^y_A$. The choice of marking a location as input or output is made by the algorithm non-deterministically. For locations marked as input, their outgoing edges are computed as shown in the previous section, using the equivalence $\sim^a_S$, where, in this case, $a \in \mathsf{Act}_{\mathsf{out}}$.

For output-locations, in order to mark a location S as output, A must have an edge e labeled with $a \in \mathsf{Act}_{\mathsf{in}}$, such that $S \cap \mathsf{guard}(e) \neq \emptyset$. If S is indeed marked as output, then one of the above edges and a rational value u are chosen, such that $\exists s \in S : y(s) = u \wedge s \models \mathsf{guard}(e)$ (by the above condition, such a u exists). Then, an edge $(S, S', y = u, \mathsf{eager}, a)$ is added to A_{test}, where S' is computed as shown in the previous section. Notice that the deadline of the edge is eager: this is because we want the output to be emitted urgently, at a precise point in time. Also note that if we cannot find an integer value u satisfying the above condition, then we pick a rational value and multiply at the end of the construction all constants in the automaton with a sufficiently large constant to make them integer. For the interval $0 \leq y < u$, the outgoing edges labeled with input actions are computed exactly as for the case of input-locations. The only difference is that we compute a partition of the interval $[0, u)$ instead of the whole set of reals R.

The states of A_{test} are defined to be either accepting or rejecting, as with A_{mon}. Rejecting states correspond to the tester emitting a "fail" verdict. On the other hand, there is no specific point in time where the tester emits a "pass". Indeed, the execution of the test can go on as long as the tester remains in an accepting state. The user can stop the test when he/she is tired of waiting.

The different steps for constructing a test TAIO are given in Algorithm 4. In the algorithm, $\mathsf{valid\_inp\_edges}(S)$ denotes the set of edges $\{e \mid S \cap \mathsf{guard}(e) \neq \emptyset\}$ labeled with input actions.

```
1  S_0 ← usucc({s_0^{A^y}});
2  list_1 ← {S_0}; list_2 ← ∅;
3  T ← the one-location TAIO with initial location S_0;
4  while(list_1 ≠ ∅)
5    S ← pick(list_1);
6    if(valid_inp_edges(S) ≠ ∅)
7      i ← pick({0, 1});
8    else i ← 1;
9    endif;
10   if(i = 0)
11     e ← pick(valid_inp_edges(S));
12     u ← a rational value s.t ∃s ∈ S : y(s) = u ∧ s ⊨ guard(e);
13     b ← the label of e;
14     S' ← succ(S ∩ (y = u), b);
15     append edge (S, S', y = u, {y}, eager, b) to T;
16     if(S' ∉ list_2)
17       list_1 ← list_1 ∪ {S'};
18     endif;
19   else u ← ∞;
20   endif;
21   foreach(a ∈ Act_out)
22     P ← the coarsest partition of [0, u) induced by ∼_S^a;
23     foreach(ψ ∈ P)
24       S' ← succ(S ∩ ψ, a);
25       if(S' ≠ ∅)
26         append edge (S, S', ψ, {y}, lazy, a) to T;
27         if(S' ∉ list_2)
28           list_1 ← list_1 ∪ {S'};
29         endif;
30       else append edge (S, fail, ψ, {y}, lazy, a) to T;
31       endif;
32     endforeach;
33   endforeach;
34   list_1 ← list_1 \ {S}; list_2 ← list_2 ∪ {S};
35 endwhile;
36 compute the rejecting states of all the input-locations of T;
```

Algorithm 4: A test TAIO generation.

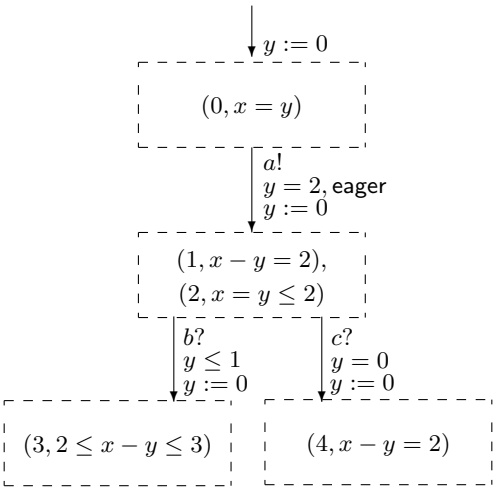


Figure 20: A possible analog-clock test TAIO for of the automaton of Figure 16 considered as a TAIO with input (a) and outputs (b and c).

5.4.3 Example of test TAIO construction

An example A_{test} of an analog-clock test TAIO is given in Figure 20.This test TAIO corresponds to the automaton of Figure 16. For the purpose of this example, the latter is considered here as a TAIO with input a and outputs b and c. We write $a!$, $b?$ and $c?$ (instead of $a?$, $b!$ and $c!$) since, with respect to the tester, a is an output and b and c are inputs. The initial location of A_{test} is an output-location and the two others are input-locations. If the tester receives any input action during the time interval $0 \leq y < 2$, a fail verdict is emitted and the test is stopped. Otherwise, if no input is received within this interval, the tester produces action a exactly at time $y = 2$, resets clock y and then waits for inputs. Then, only three possible behaviors are accepted by the tester: (1) time elapse, (2) receiving input b within interval $0 \leq y \leq 1$, (3) receiving input c exactly at time $y = 0$. If one of these behaviors is observed, the test may be stopped at any time and a pass verdict is emitted. If any other behavior is observed the test must be stopped and a fail verdict must be emitted.

5.4.4 Soundness and completeness

Next, we discuss the soundness and completeness of Algorithm 4. First, we prove that the algorithm is sound. Let T be a test TAIO generated by Algorithm 4.

Proposition 14 *If verdict* fail *is observed while applying T on A_I, then A_I* ~~tioco~~ *A_S.*

Proof Let $\sigma = a_0 a_1 \cdots a_n \in \mathsf{RT}(\mathsf{Act})$ the trace corresponding to the interaction between the tester and A_I from the starting of of T until the announcement of fail. Let $\sigma_{n-1} = a_0 \cdots a_{n-1}$. According to the algorithm, $a_n \in \mathsf{Act}_{\mathsf{out}} \cup \mathsf{R}$. Two cases are possible:

- $\sigma_{n-1} \in \mathsf{ObsTTraces}(A_S)$: With no-restriction, it is possible to assume that $a_{n-2} \in \mathsf{Act}$ and $a_{n-1} \in \mathsf{R}$.[10] Let v be the node of T reached after the execution of $\sigma_{n-2} = a_0 a_1 \cdots a_{n-2}$ and S the set of states associated with v. Clearly, A_s after $\sigma_{n-2} \subseteq S$.

[10] we can insert "0" between a_{n-1} and a_n if that is not the case.

- If $a_n \in \mathsf{R}$: Since fail is announced, the duration $a_{n-1} + a_n$ is not accepted within v and leads by the way to a rejecting state. That is $a_{n-1} + a_n \notin \mathsf{out}(A_S \,\mathsf{after}\, \sigma_{n-2})$. Hence, $A_I \not\mathsf{tioco}\, A_S$.
- If $a_n \in \mathsf{Act_{out}}$: Let ψ the element of the current partition $\mathcal{P}$ induced by $\sim_S^{a_n}$ such that $y = a_{n-1} \in \psi$. Since fail is announced, $\mathsf{succ}(s \cap \psi, a_n) = \emptyset$. Thus, $a_n \notin \mathsf{out}(A_S \,\mathsf{after}\, \sigma_{n-1})$ and by the way $A_I \not\mathsf{tioco}\, A_S$.

- $\sigma_{n-1} \notin \mathsf{ObsTTraces}(A_S)$: Let $\sigma' = a_0 a_1 \cdots a_k$ such that $k < n-1$, $\sigma' \in \mathsf{ObsTTraces}(A_S)$ and $\sigma' a_{k+1} \notin \mathsf{ObsTTraces}(A_S)$. If $a_{k+1} \in \mathsf{Act_{in}}$, that will be a contradiction with the fact that Algorithm 2 chooses only valid input-edges (i.e., edges which are in valid_inp_edges( )). Thus, $a_{k+1} \in \mathsf{Act_{out}} \cup \mathsf{R}$ and we are done.

■

Algorithm 4 is not complete in general. The example given in Figure 19 shows that depending on the kind of partitions we use there are mistakes that we can detect and others that we can not.

6 Coverage

As already mentioned in Section 5.2, the digital-clock test generation algorithm takes as input the extended specification model A_S^{tick} and generates a test in the form of a tree. Nodes of the tree correspond to sets of states of A_S^{tick}. Nodes are marked as input or output. This algorithm is only partially specified. It must be completed by specifying a policy for marking nodes as input or output, for choosing which of the possible outputs to emit and for choosing when to stop the test. One way is to resolve these choices randomly. This may not be satisfactory when some completeness guarantees are required or when repetitions must be avoided as much as possible. Another possibility is to generate an *exhaustive* test suite up to a depth k specified by the user. This approach suffers from the *explosion* problem, since the number of tests is generally exponential in k.

To remedy the above problems, many approaches have been proposed for generating test suites with respect to a given *coverage criterion*. Different coverage criteria have been proposed for software, such as statement coverage, branch coverage, and so on [40]. In the TA case existing methods attempt to cover either finite abstractions of the state space (e.g., the region graph [44] or a time-abstracting quotient graph [41]) or structural elements of the specification such as edges or locations [29]. In [8], a method for generating test cases from coverage criteria is proposed. The coverage criteria are encoded as *observer automata*.

Here, we propose a new technique for covering states, locations or edges of the specification. As mentioned in the introduction, we cannot use the technique of [29] because it relies on the assumption that outputs in the specification are urgent and isolated.

Our technique relies on the concept of *observable graph*.

6.1 The observable graph

The observable graph OG of the composed automaton A_S^{Tick} is generated as follows. The initial node of the graph is $S_0 = \{s \mid \exists \rho \in \mathsf{RT}(\{\tau\}) : s_0^{A_S^{\mathsf{Tick}}} \xrightarrow{\rho} s\}$. For each generated node S and each $a \in \mathsf{Act} \cup \{\mathsf{tick}\}$, a successor node S' is generated and an edge $S \xrightarrow{a} S'$ is added to the graph. Extrapolation abstractions can be used here as well, to ensure that

the graph remains finite. The way for constructing the observable graph OG is given in Algorithm 5.

```
1  S_0 ← usucc({s_0^{A_S^Tick}});
2  list_1 ← {S_0};
3  list_2 ← ∅;
4  OG ← the one-node graph with initial node S_0;
5  while(list_1 ≠ ∅)
6    S ← pick(list_1);
7    foreach(a ∈ Act_in ∪ Act_out ∪ {tick})
8      if (a ∈ Act_in)
9        S' ← dsucc(tsucc(S, 0), a);
10     else S' ← dsucc(usucc(S), a);
11     endif;
12     if (S' ≠ ∅)
13       append edge S -a-> S' to OG;
14       if (S' ∉ list_2)
15         list_1 ← list_1 ∪ {S'};
16       endif;
17     endif;
18   endforeach;
19   list_1 ← list_1 \ {S};
20   list_2 ← list_2 ∪ {S};
21 endwhile;
```

Algorithm 5: Construction of the observable graph OG.

6.1.1 Coverage criteria

Every node of OG corresponds to a set of states S of A_S^{Tick}. We say that the node *covers* S. On the other hand, every static test-tree is essentially a sub-graph of OG. We say that such a test covers the union of all sets of states covered by its nodes. We say that a set of tests (or *test suite*) achieves *state coverage* if every reachable state of A_S is covered by some test in the suite. Unreachable states of A_S can be ignored, since they play no role regarding conformance.

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

Every edge of OG can be associated to a set of edges of A_S. In particular, an edge $S \xrightarrow{a} S'$ will be associated to all edges which are visited during the reachability algorithm

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

We also define *action coverage* as follows. If a given edge $S \xrightarrow{a} S'$ is reachable then the corresponding observable action a is said to be reachable as well. Action coverage is achieved if all the reachable observable actions are covered by the considered test suite. Clearly, action coverage is weaker than edge coverage.

Note that these coverage criteria, the way they are defined, should be interpreted merely as a way to "guide" the test generation algorithm. We do not claim that a test suite achieving, say, global location coverage, indeed guarantees such coverage *during execution*. This cannot be guaranteed, simply because execution is partly controlled by the system under test. The latter decides which outputs to emit to the tester, therefore, it also implicitly decides which parts of the nodes of the test-tree will be "covered". One could, of course, define also a notion of "execution coverage", book-keeping the locations of the specification covered during execution of the tests. In this case, however, it cannot be guaranteed that all locations will be covered, because the implementation may simply not allow to reach some parts of the specification state-space, as already said.

6.2 Generation algorithm

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

6.2.1 Generating a location-covering suite

While there are reachable locations not covered, the algorithm picks such a location, say q. Next, it picks a node v of OG associated with q (such a node exists since q is reachable) and finds a path in OG from the initial node to v. Then, it extends this path into a test-tree as explained in Section 6.2.4 below. 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. The different steps for generating a location-covering suite are given in Algorithm 6. Notice that the algorithm is essentially an AND/OR search in a finite graph, AND nodes being input nodes and OR nodes being output nodes.

6.2.2 Generating a state-covering suite

A state-covering suite can be extracted in a similar way. If some state s is not covered, we first find a node v of OG covering s. Then we extract a test including v as above. Notice that this test will cover not only s, but a set of states containing s. It will at least cover the region in which s belongs. This guarantees that the algorithm terminates with a finite test suite, even though the set of states is infinite.

```
1  list ← the set of reachable locations;
2  T ← ∅;
3  while(list ≠ ∅)
4     q ← pick(list);
5     v ← a node of OG associated with q;
6     σ ← a path in OG from the initial node to v;
7     T ← the extension of σ into a test tree ;
8     list ← list \ the set of locations covered by T;
9     T ← T ∪ {T};
10 endwhile;
```

Algorithm 6: Generation of a location-covering suite $\mathcal{T}$.

6.2.3 Generating an edge-covering suite

The algorithm is similar for edge coverage, with the difference that instead of finding a path reaching a target node of OG, the algorithm finds a path reaching a target edge (the so-far uncovered edge).

6.2.4 Extending a path into a test-tree

This can be done by completing the path with the missing edges, labeled with tester inputs. Let $v \xrightarrow{a} v'$ be an edge in the considered path such that $a \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\}$. For $a_i \in \mathsf{Act}_{\mathsf{out}} \cup \{\mathsf{tick}\} \setminus \{a\}$, if $v \xrightarrow{a_i} v_i$ is an edge of the observable graph then an edge $v \xrightarrow{a_i} \mathsf{pass}$ is added to the path. Otherwise, edge $v \xrightarrow{b} \mathsf{fail}$ is added to it. In general, it is a good idea to continue extending the test-tree in this way. This is because, using such a policy, a single test will cover as many locations as possible. An example of how to extend a path into a test-tree is given in Figure 21. We are given the observable graph OG and a path σ of it. The figure shows the test-tree T into which the path σ is extended. In this example, the considered system has two inputs a, b and three outputs c, d, e. Since a is an input action then we can omit the other outgoing edges from the initial node of the path σ ($c!$ and $b?$). That is why these edges do not appear in T. For the second location of σ, the outgoing edge is labeled with an output action. For this location, all output actions must appear in T. The actions tick, c and e lead to $Pass$ since they appear in OG and d to $Fail$ since it does not appear in OG (i.e., it is not an expected output according to the specification).

6.2.5 Finiteness of the number of obtained tests and complexity

It is not difficult to see that for every reachable state of A_S there exists a node S of OG covering this state, and similarly for locations and edges. Thus, covering all nodes and edges in OG suffices to achieve coverage for each of the three criteria above. Since OG is finite, a finite number of tests suffices to achieve coverage, thus, the algorithm terminates. The worst-case complexity of the algorithm is polynomial in the size of OG. Indeed, finding a node (or edge) of OG associated with a location (or edge) of A_S is linear. Finding a path in OG and extending the path into a test-tree is also linear. These steps are performed at most as many times as there are nodes in OG.

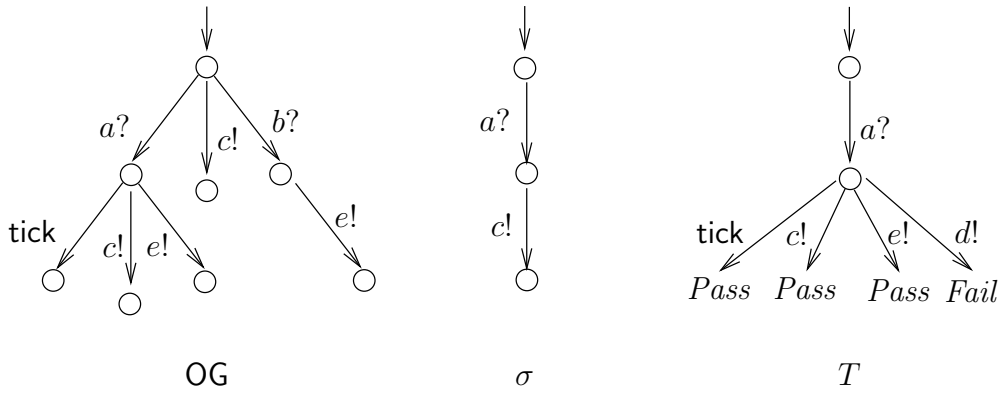


Figure 21: An example of how to extend a path σ of the observable graph OG into a test-tree T.

6.2.6 A limitation of the generation algorithm

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. Clearly, a non-minimal suite contains some redundant tests, and one would like to remove such tests. On the other hand, one would also like to have a *minimal-number* suite, that is, a minimal suite with as few tests as possible. Notice that minimal does not imply minimal-number: the first is a sort of "local optimum" while the second is a "global optimum". In general minimal or minimal-number suites are not unique. Moreover, adding a new test to the suite may result in making one or more previously generated tests redundant. Studying efficient methods of generating minimal or minimal-number test suites is beyond the scope of this paper.

7 Tool and Case Studies

7.1 TTG

We have built a prototype test-generation tool, called TTG, on top of the IF environment [12]. 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 [26]. TTG is implemented independently from TGV. TTG is written in C++ and uses the basic libraries of IF for parsing and symbolic reachability of timed automata with deadlines.

Only digital-clock tests are generated by TTG at this point. As shown in Figure 22, TTG takes as inputs the specification and Tick automata, written in IF language, as well as a set of user options specifying the test-generation mode. There are four modes:

- *Interactive*: the user guides the test generation algorithm, resolving the non deterministic points (whether to issue an output or wait for an input, which output if many are possible, when to stop generating the test, etc.).
- *Random*: the non-deterministic points are resolved randomly.

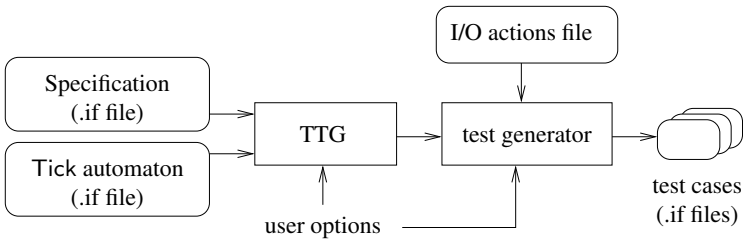


Figure 22: The TTG tool.

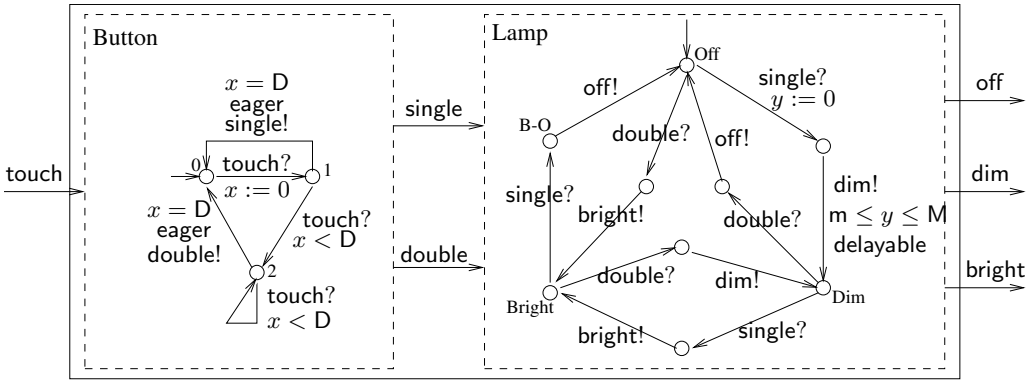


Figure 23: A lighting device.

- *Exhaustive*: all possible tests are generated up to a user-defined depth.
- *Coverage*: a set of tests that achieves a user-defined coverage criterion is generated. The criteria implemented currently are: *state*, *location*, *action* or *partial state*(i.e., coverage of the variables appearing in IF model).

In fact, TTG does not generate the tests itself. Instead, it generates an executable program, the "test generator" depicted in Figure 22. It is this program that generates the tests (one or more tests, depending on the chosen mode). The test generator has additional options: for instance, for exhaustive test generation the user specifies the desired test depth when running the test generator, not when running TTG. The tests are output in the IF language.

In the rest of this section, we present two small case studies treated with TTG.

7.2 A lighting switch

This case study is a modification of the one presented in [29]. The (modified) specification is shown in Figure 23. 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 are 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

depth	time	# of	coverage of other criteria			
	(sec)	tests	config.	local loc.	global loc.	actions
1	0.03	2	3%	30%	7%	25%
2	0.03	4	7%	42%	19%	25%
3	0.03	8	12%	50%	26%	25%
4	0.04	16	18%	50%	33%	25%
5	0.08	28	22%	58%	37%	50%
6	0.14	57	38%	75%	52%	75%
7	0.22	101	50%	92%	74%	75%
8	0.35	176	63%	100%	93%	75%
9	0.62	306	77%	100%	100%	75%
10	1.14	533	91%	100%	100%	100%
11	1.86	928	98%	100%	100%	100%
12	3.52	1611	100%			

Table 1: Exhaustive test generation results for the lighting-device case study.

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).

7.2.1 Automatic test generation with TTG

We have used TTG to generate digital-clock tests for the above specification, with parameter set $\mathsf{D} = 1, \mathsf{m} = 1, \mathsf{M} = 2$ and with respect to the perfectly one-time-unit periodic Ticker. The results of the exhaustive generation for various depth levels are shown in Table 1. Depth levels are ranging from 1 to 12. Column "depth" shows the depth of the generated tests (i.e., the length of the longest path from the root to a leaf). Column "time" shows the time in seconds taken by TTG to generate a test suite with respect to the corresponding coverage criterion. Column "# of tests" shows the number of tests in the suite. The remaining columns show the coverage percentage for the different considered criteria.

Notice that these are the sets of all possible tests up to the specified depth: no test selection is performed. It is also worth noticing that in order to perform the complete coverage of all the considered criteria, we need to consider the exhaustive test-suite of depth 12 which is made of 1611 test cases.

We have also used TTG to perform test selection for this case-study with respect to the several considered criteria. The obtained results are shown in Table 2. Column "size" shows the number of elements to be covered. TTG succeeds to achieve all the coverage criteria but the action coverage criterion. It succeeds only to generate a test suite made of 25 test cases with depth ranging form 5 to 12 and with a coverage rate of 75%.

One of the tests generated by TTG is shown in Figure 24. The drawing has been produced automatically using the `if2eps` tool written by Marius Bozga, which is based on the dot/graphviz utility (http://www.graphviz.org).

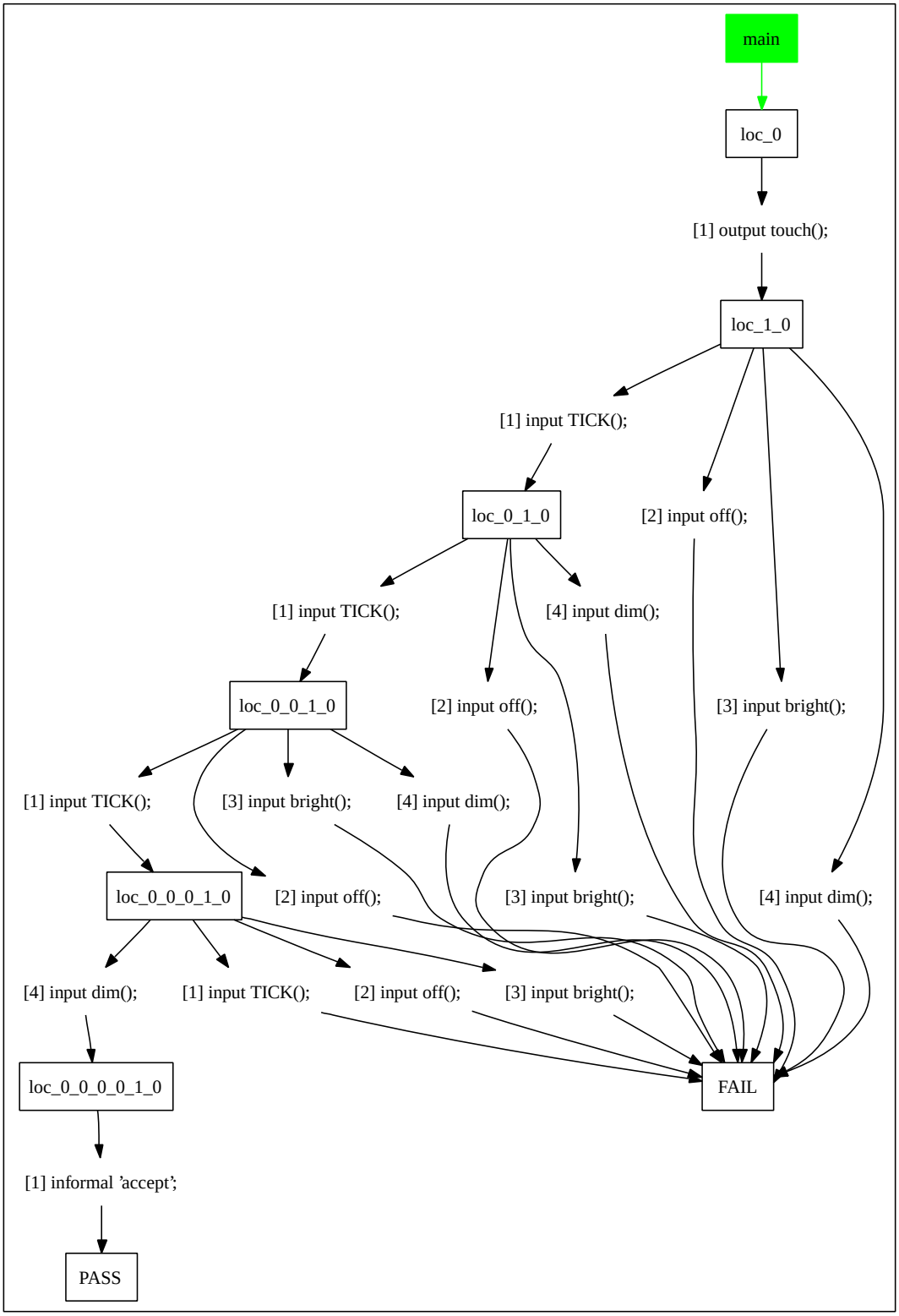


Figure 24: A test generated automatically by TTG.

criterion used	size	time (sec)	# of tests	depth	coverage of other criteria			
					config.	local loc.	global loc.	actions
config.	175	0.2	25	5 -12	100%	100%	100%	75%
local loc.	12	0.12	11	10 -12	67%	100%	89%	75%
global loc.	27	0.18	22	5 -12	91%	100%	100%	75%
actions	4	0.2	25	5 -12	100%	100%	100%	75%

Table 2: Test generation results for the lighting-device case study.

7.2.2 Comparison with manually generated tests

In this section we show that it is possible to reduce the number of tests generated by TTG for achieving coverage with respect to the several considered criteria. However the manually generated tests are likely to be of bigger size.

Consider, for instance, the two tests shown in Figure 25. In order not to overload the figure, each node of the tests is labeled only with the set of corresponding global locations; states are omitted. Also, for output nodes we only draw the outgoing edges which do not lead to FAIL. For example, node (2,Off) of the leftmost test has three outgoing edges labeled off?, dim?, bright? and leading to FAIL. Also, to save space, we draw the tree as a DAG (directed acyclic graph).

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

The two tests do not achieve global location (and, consequently, neither state) coverage. For example, location (1,O-B) is not covered. However, 18 out of 30 global locations are covered. For covering the rest, it is possible either to generate more tests or to extend one of the two tests above. For instance, we can append the rightmost test at the end of the leftmost one. Also, in order to cover location (1,O-B), say, we can consider node (0,O-B) of the leftmost test as an output node instead of an input node (issuing the only possible output, touch!) and keep the remaining part of the test unchanged. Doing this, we can obtain a single test of depth 41 which achieves global location coverage. Alternatively, a suite of 8 tests of lengths smaller than those of Figure 25 suffices to achieve global location coverage. This suite can be generated by the algorithm of Section 6. Notice that the depth of the leftmost test of Figure 25 is 19. Generating an exhaustive test suite up to this depth would be infeasible due to explosion.

7.3 The bounded retransmission protocol

The Bounded Retransmission Protocol (BRP) [27] is a protocol for transmitting files over an unreliable (lossy) medium. The architecture of the protocol is shown in Figure 26. The protocol is implemented by the Transmitter and the Receiver. The users of the protocol are the Sending and Receiving clients. The medium is modeled by the Forward and Backward channels. Upon receiving a file from the Sending client (action put), the Transmitter fragments the file into packets and sends each packet to the Receiver (action send), awaiting an acknowledgment for each packet sent (action ack). If a timeout occurs without receiving an acknowledgment, the Transmitter resends the packet, up to a maximum number of

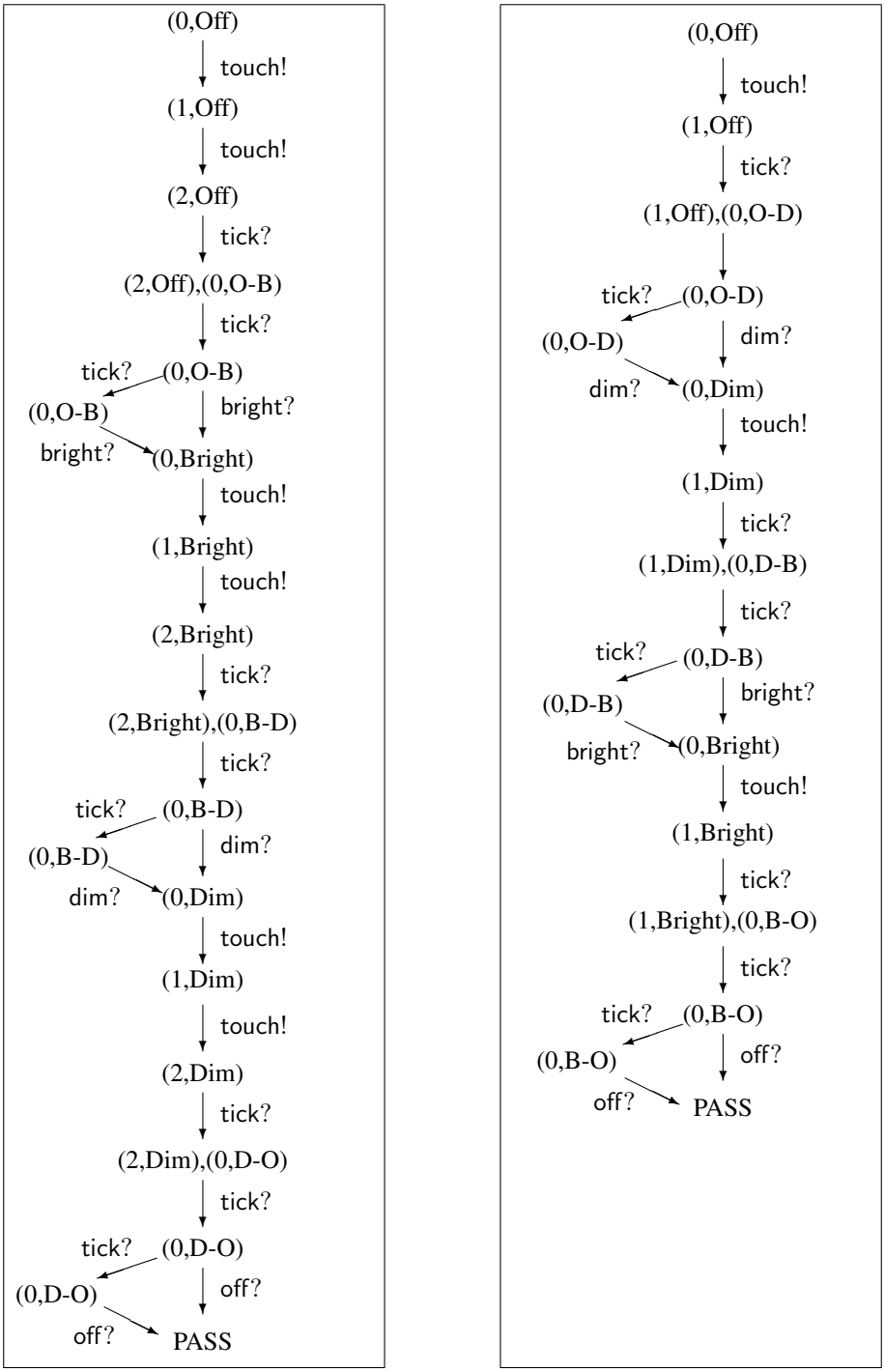


Figure 25: Two digital-clock tests covering most global locations of the specification of Figure 23.

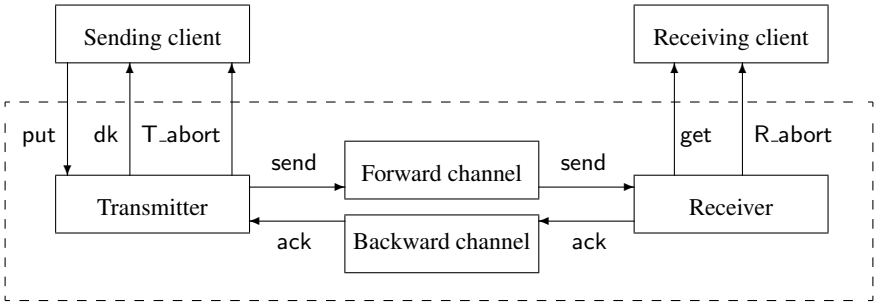


Figure 26: The BRP specification and interfaces

retrials. At the end, if the file is transmitted successfully the Transmitter does not output anything to the Sending client. Otherwise, the Transmitter responds either with "abort" (action T_abort) if the packet that failed was a "middle" one, or with "don't know" (action dk) if the packet was the first or last one (in this case the file may or may not be received at the other end). In case of success, the Receiving client receives the file (action get). In case the Receiver does not hear from the Transmitter for some time, it outputs R_abort to the Receiving client.

7.3.1 IF model of BRP

Here, we use the BRP model developed in [13]. The model has been initially developed in SDL, then automatically translated to IF.[11] The model is shown in Figure 27. States in red (labeled "decision_...") are *transient* states, meaning that time does not elapse and the automaton moves through these states without being interrupted by other concurrent automata. The Transmitter has two clocks, "t_repeat" and "t_abort", and the Receiver one clock, "r_abort".[12] The keyword "when" preceeds a clock guard and "provided" precedes a guard on discrete variables. Keyword "task" is for assignments. The model is parameterized by five parameters: p, the number of packets in a file; max_retry, the maximum number of retries in sending a packet (after timeout); dt_repeat, the timeout delay; dt_abort, the time the Transmitter waits before outputting T_abort; dr_abort, the time the Receiver waits before outputting R_abort. The values used in our case study are:

$$\mathsf{p} = 2, \quad \mathsf{max\_retry} = 4, \quad \mathsf{dt\_repeat} = 2, \quad \mathsf{dt\_abort} = 15, \quad \mathsf{dr\_abort} = 13.$$

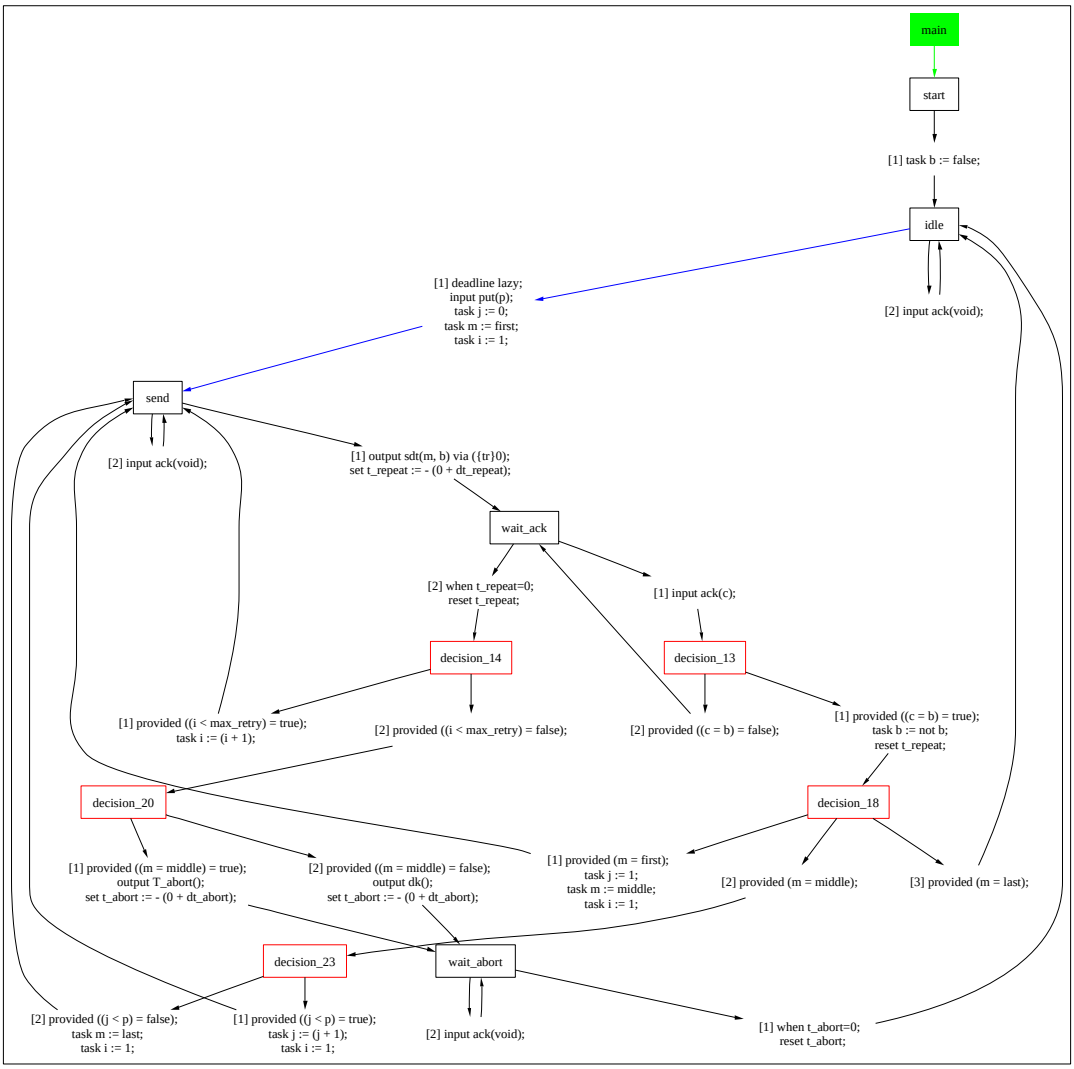
For testing, we view the four components enclosed in dashed square in Figure 26 as the BRP specification. The Sending and Receiving clients play the role of the environment, but they are not explicitly modeled, i.e., no assumptions are made on the environment. The interface of the SUT with its environment is captured by actions put (input) and get, dk, T_abort, R_abort (outputs).

7.3.2 Automatic test generation using TTG

Using TTG, we generate tests for the perfectly periodic Tick automaton with clock period equal to 1, with respect to various coverage criteria. The results are shown in Table 3. The

[11] The model of BRP can be found in the IF web page: http://www-verimag.imag.fr/ async/IF/ under "examples".

[12] The clocks are reset to a negative value and count upwards. This is not an essential difference with the TA model presented earlier.



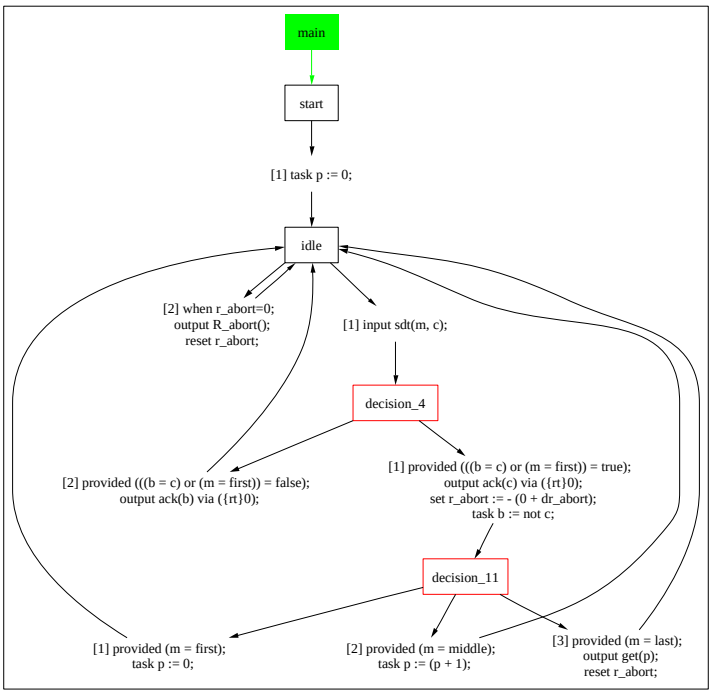


Figure 27: BRP Transmitter (up) and Receiver (down).

criterion used	size	time (sec)	# of tests	depth	coverage of other criteria							
					config.	locations	actions	m	b	c	i	j
config.	5808	25	9	22 -53	100%							
locations	4	3	1	13	25%	100%	60%	100%	100%	100%	100%	100%
actions	5	5	1	44	40%	100%	100%	100%	100%	100%	100%	100%
m	3	2	1	2	2%	75%	40%	100%	100%	100%	25%	100%
b	2	2	1	2	2%	75%	40%	100%	100%	100%	25%	100%
c	2	2	1	2	2%	75%	40%	100%	100%	100%	25%	100%
i	4	3	1	10	23%	75%	40%	100%	100%	100%	100%	100%
j	3	2	1	2	2%	75%	40%	100%	100%	100%	25%	100%

Table 3: Test generation results for the BRP case study.

criteria used are: state, locations, actions, and the values of the five discrete variables of the model, namely, m, b, c, i, j. For the case study, we will use the term *configuration*, instead of "state", for measuring coverage. A configuration corresponds to an entire "symbolic state" and includes a vector of locations and values of variables for each automaton, plus a DBM representing symbolically the set of clock states. Thus, a configuration is a set of TA states. We can count configurations, but we cannot count TA states.

Thus, there are 5808 reachable configurations in total[13], there are 4 global locations (we do not count transient locations) and 6 actions (the 5 input/output actions plus tick). Variables b and c are booleans (they encode the *alternating bit* for the Transmitter and Receiver, respectively). Variable m takes three possible values (beginning, middle or end of file). Variable i takes four possible values, from 1 to max_retry. Variable j takes three possible values, from 0 to p. Notice that the configuration criterion requires 9 tests whereas all other criteria can be covered with just one test.

For the configuration criterion, the depth varies between 21 and 52. The rest of the columns show the percentage of coverage of the other criteria by the test suite generated for the given criterion. For example, the test covering the four global locations also covers 1421 configurations, which amounts to approximately 25% of the total number of configurations.

Perhaps the most interesting finding from the above experiments is that a relatively small number of tests suffices to cover all reachable configurations of the specification (in fact, we cover the states of the product automaton A_S^{Tick}). It is worth comparing this number to the number of tests generated with the "exhaustive up to given depth" option. As shown in Table 4, the size of exhaustive test suite grows too large even for relatively small depths. The table also shows the percentage of the above criteria covered by the exhaustive test suite. It can be seen that even though the number of tests is large, only a small percentage of coverage is achieved: for instance, 21% configuration coverage for 222 tests at depth 7.

Sometimes not only the number of tests but also their size is important. By looking at our test generation algorithm, where a test is obtained by completing a path, we can say that the size of a test is essentially its depth. As one can see from Table 3 the largest test depth is 52. This can be explained as follows. In our implementation we use the following heuristic to choose which configuration to cover next: we pick a configuration which is "far" from the initial one, that is, at a large depth. The expectation is to cover as many configurations as possible with every new test. Thus, this heuristic tends to favor the generation of fewer but "longer" tests. Obviously, a different approach is to favor "shorter" (but perhaps more)

[13] The forward and backward channels are modeled by lossy FIFO buffers. These buffers remain bounded because reception of messages are eager.

depth	time	# of	coverage of other criteria							
	(sec)	tests	config.	locations	actions	m	b	c	i	j
1	2	2	1%	75%	20%	100%	100%	100%	25%	100%
2	2	3	2%	75%	40%	100%	100%	100%	25%	100%
3	2	5	5%	75%	40%	100%	100%	100%	50%	100%
4	2	8	7%	75%	40%	100%	100%	100%	50%	100%
5	5	18	12%	75%	40%	100%	100%	100%	75%	100%
6	13	42	14%	75%	40%	100%	100%	100%	75%	100%
7	79	222	21%	75%	40%	100%	100%	100%	100%	100%

Table 4: Exhaustive test suites for the BRP case study.

tests. This can be done by changing the heuristic to pick configurations which are "close" to the initial one.

A test generated by TTG for the configuration coverage option is shown in Figure 28.

8 Conclusions and Perspectives

We have proposed a testing framework for real-time systems based on partially-observable, non-deterministic timed-automata and a formal conformance relation called tioco. We have provided algorithms to generate analog-clock or digital-clock tests in an on-line or off-line fashion. We have shown how the number of generated tests can be reduced using coverage criteria. Finally, we reported on a prototype test-generation tool and two case-studies.

We believe that this work opens a number of interesting perspectives. First, we have only touched upon the problem of test execution. Building an actual test harness is particularly challenging in a real-time context, where timing accuracies are critical. Digital-clock tests are crucial in this respect, since they allow to formally define the assumptions on the accuracy of the tester's clocks and take it into account during test generation.

The topic of coverage also deserves to be studied in more depth in a real-time context. Our notions of coverage mainly extend well-known notions developed for software. One could imagine alternative notions that exploit some knowledge of our domain, in particular with respect to timing constraints and their topology. Also notice that we are applying coverage at the specification level whereas usually in software it is applied for the system-under-test (when the latter is "white-box"). Methods to generate minimal test suites (without redundant tests) should also be examined.

Combining coverage with on-line test execution is another issue that seems to be little explored. The problem is related to choosing online tester outputs and output times. Many heuristics can be applied to resolve such choices, but an additional problem is how to manage these choices across the execution of the entire test suite, using some appropriate book-keeping techniques.

Another perspective is to study other testing problems in a real-real setting, except the conformance testing problem. Classic testing problems include state identification problems [39]. Some preliminary work has been done in this direction but many problems remain open [37].

Another direction is to use the techniques developed in this paper for testing in other contexts, for instance, controller synthesis.

Acknowledgments We would like to thank Fabrice Chevalier and Patricia Bouyer for their comments on monitor generation. We are grateful to Marius Bozga for his help

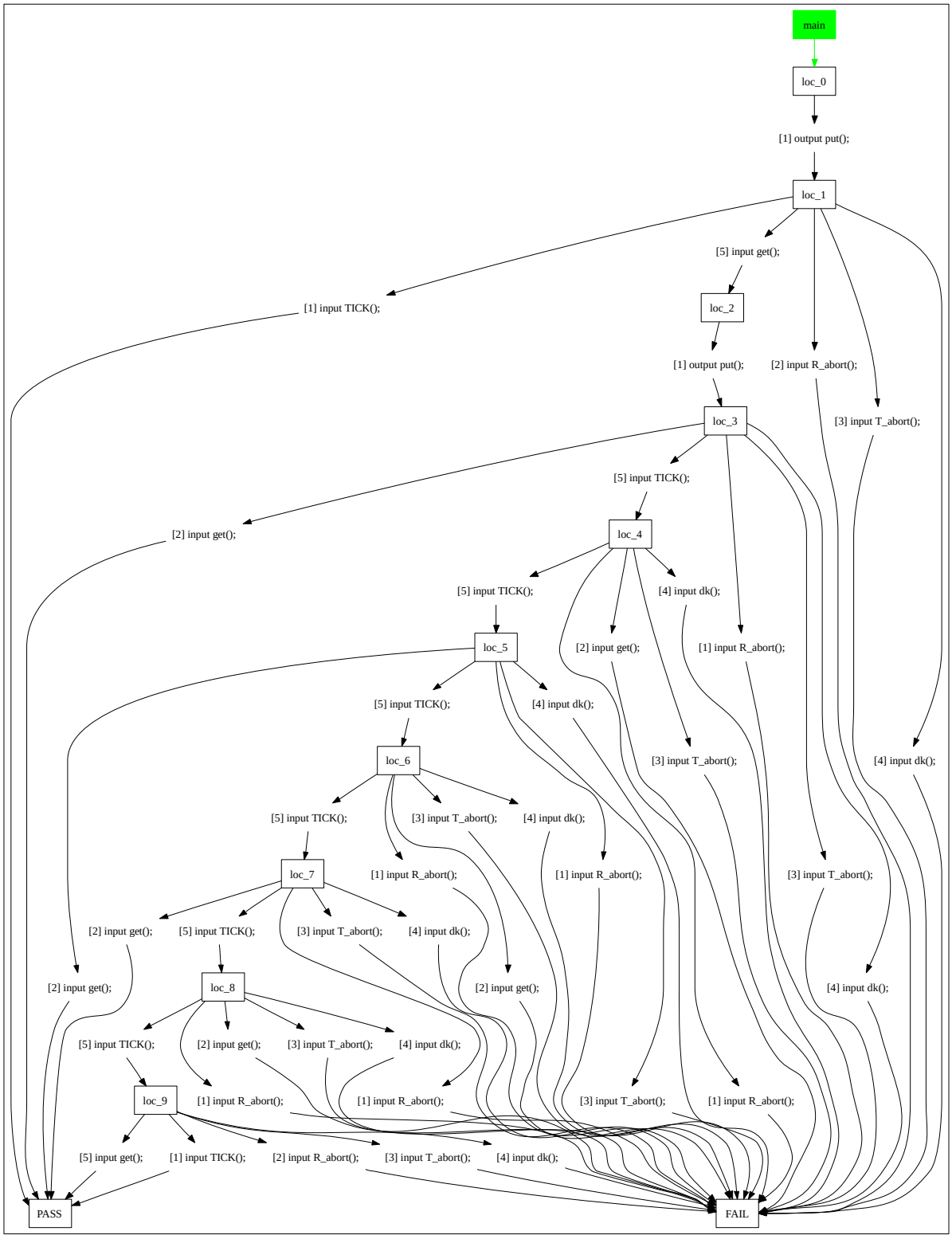


Figure 28: A test generated by TTG for the BRP case study (The one ensuring the coverage of parameter i).

with the implementation of TTG on top of IF. We also express our gratitude to Mohamed Dergueche for his contributions to the implementation of TTG.

References

[1] R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994. 6, 7, 11, 21

[2] R. Alur, L. Fix, and T. Henzinger. A determinizable class of timed automata. In *CAV'94*, volume 818 of *LNCS*. Springer, 1994. 6, 41

[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. 6, 34

[4] J. Bengtsson and W. Yi. On clock difference constraints and termination in reachability analysis of timed automata. In *ICFEM'03*, volume 2885 of *LNCS*. Springer, 2003. 43, 45

[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)*, volume 113 of *ENTCS*, pages 23–43. Elsevier, 2005. 9

[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. 7

[7] B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9:41–46, 1983. 35

[8] J. Blom, A. Hessel, B. Jonsson, and P. Pettersson. Specifying and generating test cases using observer automata. In *In Proc. of Formal Approaches to Software Testing*, pages 125–139, 2004. 51

[9] S. Bornot, J. Sifakis, and S. Tripakis. Modeling urgency in timed systems. In *Compositionality*, volume 1536 of *LNCS*. Springer, 1998. 11

[10] P. Bouyer. Forward analysis of updatable timed automata. *Formal Methods in System Design*, 24(3):281–320, 2004. 43, 45

[11] P. Bouyer, F. Chevalier, and D. D'Souza. Fault diagnosis using timed automata. In *FoSSaCS'05*, volume 3441 of *LNCS*, pages 219–233. Springer, 2005. 46

[12] M. Bozga, J.C. Fernandez, L. Ghirvu, S. Graf, J.P. Krimm, and L. Mounier. IF: a validation environment for timed asynchronous systems. In *Proc. CAV'00*, volume 1855 of *LNCS*. Springer, 2000. 9, 54

[13] M. Bozga, S. Graf, A. Kerbrat, L. Mounier, I. Ober, and D. Vincent. SDL for Real-Time: What is Missing? In *Proceedings of SAM'00: 2nd Workshop on SDL and MSC (Grenoble, France)*, pages 108–122. IMAG, June 2000. 60

[14] V. Braberman, M. Felder, and M. Marre. Testing timing behavior of real-time software. In *International Software Quality Week*, 1997. 6

[15] 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. 9

[16] E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *MOVEP 2000*, volume 2067 of *LNCS*. Springer, 2001. 6

[17] L. Briones and E. Brinksma. A test generation framework for quiescent real-time systems. In *FATES'04*, volume 3395 of *LNCS*. Springer, 2004. 6, 26, 27

[18] R. Cardell-Oliver. Conformance test experiments for distributed real-time systems. In *ISSTA'02*. ACM Press, 2002. 6

[19] T.S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(1), 1978. 6

[20] D. Clarke, T. Jéron, V. Rusu, and E. Zinovieva. STG: A symbolic test generation tool. In *TACAS'02*, volume 2280 of *LNCS*. Springer, 2002. 6

[21] 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. 6

[22] 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. 9

[23] C. Daws and S. Tripakis. Model checking of real-time reachability properties using abstractions. In *Tools and Algorithms for the Construction and Analysis of Systems '98, Lisbon, Portugal*, volume 1384 of *LNCS*. Springer-Verlag, 1998. 43, 45

[24] D. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer–Verlag, 1989. 35, 43

[25] A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS'98*. IEEE, 1998. 6

[26] 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*, volume 1102 of *LNCS*. Springer, 1996. 6, 8, 55

[27] Jan Friso Groote and Jaco van de Pol. A bounded retransmission protocol for large data packets. In *Algebraic Methodology and Software Technology*, pages 536–550, 1996. 9, 60

[28] T. Henzinger, Z. Manna, and A. Pnueli. What good are digital clocks? In *ICALP'92*, LNCS 623, 1992. 30, 40

[29] A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal real-time test case generation using UPPAAL. In *FATES'03*, 2003. 6, 24, 30, 51, 56

[30] 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. 6

[31] 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. 40

[32] C. Jard, T. Jéron, and P. Morel. Verification of test suites. In *TestCom 2000*, 2000. 32

[33] A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES'03*, 2003. 6, 24

[34] 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. 7, 35

[35] M. Krichen and S. Tripakis. Real-time testing with timed automata testers and coverage criteria. In *Formal Techniques, Modelling and Analysis of Timed and Fault Tolerant Systems (FORMATS-FTRTFT'04)*, volume 3253 of *LNCS*. Springer, 2004. 7

[36] M. Krichen and S. Tripakis. An expressive and implementable formal framework for testing real-time systems. In *The 17th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom'05)*, volume 3502 of *LNCS*. Springer, 2005. 7, 9

[37] M. Krichen and S. Tripakis. State identification problems for timed automata. In *The 17th IFIP Intl. Conf. on Testing of Communicating Systems (TestCom'05)*, volume 3502 of *LNCS*. Springer, 2005. 65

[38] K. Larsen, M. Mikucionis, and B. Nielsen. Online Testing of Real-time Systems using Uppaal. In *FATES'04*, volume 3395 of *LNCS*. Springer, 2004. 6, 25, 40

[39] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996. 6, 30, 65

[40] G.J. Myers. *The art of software testing*. Wiley, 1979. 9, 50

[41] B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS'01*. LNCS 2031, Springer, 2001. 6, 24, 50

[42] J. Peleska. Formal methods for test automation - hard real-time testing of controllers for the airbus aircraft family. In *IDPT'02*, 2002. 6

[43] J. Sifakis and S. Yovine. Compositional specification of timed systems. In *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, volume 1046 of *LNCS*. Spinger-Verlag, 1996. 11

[44] J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254, 2001. 6, 24, 30, 50

[45] J. Tretmans. Testing concurrent systems: A formal approach. In *CONCUR'99*, volume 1664 of *LNCS*. Springer, 1999. 6, 7, 15, 33

[46] J. Tretmans. Testing techniques. Lecture notes, University of Twente, The Netherlands, 2002. 6, 27

[47] 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. 8, 34, 35

[48] S. Tripakis. Folk theorems on the determinization and minimization of timed automata. In *Formal Modeling and Analysis of Timed Systems (FORMATS'03)*, volume 2791 of *LNCS*. Springer, 2004. 7, 20, 40