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

Moez Krichen and Stavros Tripakis

Verimag
Centre Equation, 2, avenue de Vignate, 38610 Gières, France
krichen@imag.fr, tripakis@imag.fr

Abstract. We propose a new framework for black-box conformance testing of real-time systems, based on the model of timed automata. The framework is expressive: it can fully handle partially-observable, non-deterministic timed automata. It also 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. The framework is implementable: tests can be implemented as finite-state machines accessing a finite-precision digital clock. We propose, for this framework, a set of test-generation algorithms with respect to different coverage criteria. We have implemented these algorithms in a prototype tool called TTG. 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.

1 Introduction

Our work targets black-box conformance testing for real-time systems. By "black box" we mean that the tester has no knowledge nor access in the internals of the system under test (SUT), thus, can only rely on its observable input/output behavior. We follow a formal, *model-based* testing approach, in the sense that we assume a formal specification is available and conformance is also defined in a formal way.

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

Many formal testing frameworks use models such as (extended) Mealy machines (e.g., see [11, 14, 16]) or labeled transition systems (e.g., see [18, 8, 13]).

[⋆] Work partially supported by CNRS STIC project "CORTOS" and by IST Network of Excellence "ARTIST2".

These models are not well-suited for real-time systems. In Mealy machines, inputs and outputs are synchronous, which is a reasonable assumption when modeling synchronous hardware, but not when outputs are produced with variable delays, governed by complex timing constraints. In labeled transition systems (LTSs) inputs and outputs are asynchronous. However, there is no explicit modeling of time. In some cases [18], the notion of *quiescence* is used: timeouts are modeled by special δ actions which can be interpreted as "no output will be observed". This is problematic, because timeouts need to be instantiated with concrete values upon testing (e.g., "if nothing happens for 10 seconds, output FAIL"). However, there is no systematic way to derive the timeout values (indeed, durations are not expressed in the specification). Thus, one must rely on empirical, ad-hoc methods.

We advocate an *explicit* specification of timing assumptions and requirements for testing real-time systems. For this, we need a specification model which explicitly talks about time. We opt for the model of *timed automata* (TA) [1]. TA have been established during the past decade as a suitable model for real-time systems. With respect to existing testing methods based on TA (see [12] and references therein) our framework presents two major contributions.

First, the framework is *expressive*: it can handle the full class of *partially-observable*, *non-deterministic* TA. In existing works, only subclasses of the TA model are considered. For example, [17, 9] consider TA with "isolated" and "urgent" outputs, which means that for each input sequence there is a unique output emitted at a precise point in time. A simple specification such as "*when input a is received, output b must be emitted within at most* 10 *time units*" cannot be expressed in this model. Other works use event-recording automata [15] or TA with restricted clock resets [10] or guards [5].

Second, our framework is *implementable*: the tests we generate can be executed by an automatic tester which uses a *digital clock* of finite precision. In most existing works the tester is implicitly assumed to have access to an infinite-precision clock, allowing, for instance, to distinguish between an event observed at time 1 or at time $1+\epsilon$, for ϵ arbitrarily close to 0. An exception is the work [5] where a *digitization* of the TA semantics is used to model the tester clock. Our approach is more general in the sense that the digital-clock model is not "hard-wired" in the test generation algorithm. Rather, it is provided explicitly by the user as a Tick automaton (see Section 2). Tick automata can model not only fixed-step digitization but also skewed or diverging clocks, or any other sampling and interfacing mechanism the tester might use to observe and control the SUT.

In this paper we describe our framework from a methodological point of view (Section 2). We give special emphasis on modeling expressiveness and show through examples that the framework is rich enough to capture assumptions on the environment of the SUT, event-based or variable-based interfaces between the SUT and the tester, delays introduced by such interfaces, digital-clock sampling, and so on. Such *explicit* modeling is important for two reasons. First, it provides the user of the framework with full control on the assumptions made on the testing infrastructure and how these affect the generated tests. Second, it avoids the need for special algorithms (e.g., digitization [5]) in order to treat the above features: the latter simply become part of an extended specification model. Indeed, our test generation method uses standard *symbolic reachability* techniques available in most TA model-checking tools. Also note that symbolic reachability techniques scale much better than testing techniques based on the *region graph* [7, 17].

The framework is accompanied by a prototype tool called TTG (Timed Test Generator), built on top of the IF tool-suite [3]. To control the explosion of the potential number of tests, we have implemented in TTG a set of test selection techniques, among which a set of test generation algorithms with respect to various coverage criteria, such as state, location, action or selected-variable coverage. To illustrate the practical interest of our approach, we have used TTG to generate tests for the well-known Bounded Retransmission Protocol (BRP). The results are described in Section 3 and show that a few tests suffice to cover thousands of reachable symbolic states.

2 The testing framework

In this section we present the essential features of our testing framework. For more details, the reader is referred to [12]. We also illustrate some methodological aspects of our framework, especially modeling issues regarding environment assumptions and interface conditions between the tester and the SUT.

The model: timed automata with inputs, outputs and unobservable actions To model the specification, we use timed automata (TA) [1]. As the TA model is well-known, we only give a brief overview here. We also present a "pure" TA model without discrete variables and omit discussion on how to compose TA. In practice, these features are essential for ease and clarity of modeling: they are indeed part of our tool, see Section 3.

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

A *timed automaton over* Σ is a tuple $(Q, q_0, X, \Sigma, \mathsf{E})$ where Q is a set of *locations*; $q_0 \in Q$ is the initial location; X is a set of *clocks*; E is a set of *edges*. Each edge is a tuple (q, q', ψ, r, d, a), where $q, q' \in Q$ are the source and destination locations; ψ is the *guard*, a conjunction of constraints of the form $x\#c$, where $x \in X$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$; $r \subseteq X$ is the set of clocks to be *reset*; $d \in \{\mathsf{lazy}, \mathsf{delayable}, \mathsf{eager}\}$ is the *deadline* (lazy deadlines impose no urgency, $\mathsf{delayable}$ means that once enabled the transition must be taken before it becomes disabled and eager means the transition must be taken as soon as it becomes enabled); and $a \in \Sigma$ is the action. We will not allow eager edges with guards of the form $x > c$.

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

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

A *timed automaton with inputs and outputs* (TAIO) is a timed automaton over Σ_τ. A TAIO is called *observable* if none of its edges is labeled by τ. A TAIO A is called *input-complete* if it can accept any input at any state: $\forall s \in \mathsf{Reach}(A) \,.\, \forall a \in \Sigma_{\mathsf{in}} \,.\, s \xrightarrow{a}$. It is called *deterministic* if $\forall s, s', s'' \in \mathsf{Reach}(A) \,.\, \forall a \in \Sigma_\tau \,.\, s \xrightarrow{a} s' \wedge s \xrightarrow{a} s'' \Rightarrow s' = s''$. It is called *non-blocking* if $\forall s \in \mathsf{Reach}(A) \,.\, \forall t \in \mathsf{R} \,.\, \exists \rho \in \mathsf{RT}(\Sigma_{\mathsf{out}} \cup \{\tau\}) \,.\, \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}$. The non-blocking property states that at any state, A can let time pass forever, even if it does not receive any input. This is a sanity property which ensures that a TAIO does not "force" its environment to provide an input by blocking time. The set of *observable timed traces* of A is defined to be $\mathsf{Traces}(A) = \{\Pi_\Sigma(\rho) \mid \rho \in \mathsf{RT}(\Sigma_\tau) \wedge s_0^A \xrightarrow{\rho}\}$.

Specifications, implementations and conformance We assume that the specification of the system to be tested is given as a non-blocking TAIO A_S. We assume that the SUT, also called *implementation*, can be modeled as a non-blocking, input-complete TAIO A_I. Notice that we do not assume that A_I is known, simply that it exists. The assumption of A_S and A_I being non-blocking is natural, since in reality time cannot be blocked. The assumption of A_I being input-complete is also reasonable, since a system usually accepts all inputs at any time, possibly ignoring them or issuing an error message when the input is not valid. Notice that we do not assume, as is often done, that the specification A_S is input-complete. This is because A_S needs to be able to model assumptions on the environment, i.e., restrictions on the inputs, as we show below.

We also do not assume that A_S is deterministic. In fact, A_S may contain unobservable actions. Partially-observable or non-deterministic models often arise in practice. For instance, when specifications are built of many components (Figure 1), internal communication among these components is not observable to the tester (in fact it may simply be an artifact of modeling). This is indeed true in the case of the communication protocol we treat in Section 3. Non-determinism may also result when abstractions are applied to the model in order to reduce its size.

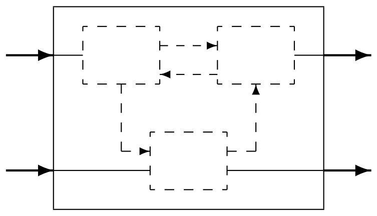


Fig. 1. A specification with internal (unobservable) actions.

The *timed input-output conformance relation*, denoted tioco, requires that after any observable sequence specified in A_S, every possible observable output of A_I (including delays) is also a possible output of A_S. tioco is inspired from its "untimed" counterpart, ioco [18]. The key idea is that time delays, along with output actions, are considered to be observable events. More precisely, define A after σ as the set of all states of A that can be reached by some timed sequence ρ whose projection to observable actions is σ and let $\mathsf{out}(S)$ be the set of all observable events (output actions or delays) that can occur when the automaton is at some state in S. Formally, $A\,\mathsf{after}\,\sigma = \{s \in S_A \mid \exists \rho \in \mathsf{RT}(\Sigma_\tau)\ .\ s_0^A \xrightarrow{\rho} s \wedge \Pi_\Sigma(\rho) = \sigma\}$, $\mathsf{out}(S) = \bigcup_{s \in S} \mathsf{out}(s)$, $\mathsf{out}(s) = \{a \in \Sigma_{\mathsf{out}} \mid s \xrightarrow{a}\} \cup \mathsf{elapse}(s)$ and $\mathsf{elapse}(s) = \{t > 0 \mid \exists \rho \in \mathsf{RT}(\{\tau\})\,.\,\mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}\}$.

Then, A_I conforms to A_S, denoted A_I tioco A_S, if

$$\forall \sigma \in \mathsf{Traces}(A_S)\ .\ \mathsf{out}(A_I\ \mathsf{after}\ \sigma) \subseteq \mathsf{out}(A_S\ \mathsf{after}\ \sigma). \tag{1}$$

Figure 2 shows an example of a specification Spec_1, which 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".[1] Thus, this specification requires that the output b is not emitted neither too early nor too late. Implementations Impl_1 and Impl_2 conform to Spec_1. Impl_1 produces b exactly 5 time units after reception of a. Impl_2 produces b sometime in the interval $[4, 5]$. Implementations Impl_3 and Impl_4 do not conform to Spec_1. Impl_3 may produce a b after 1 time unit, which is too early. Impl_4 fails to produce a b at all. Formally, letting $\sigma = a\,1$, we have $\mathsf{out}(\sigma(\mathsf{Impl}_3)) = (0, 4] \cup \{b\}$ and $\mathsf{out}(\sigma(\mathsf{Impl}_4)) = (0, \infty)$, whereas $\mathsf{out}(\sigma(\mathsf{Spec}_1)) = (0, 7]$. The last example shows that "doing nothing" is not an option for the SUT, since doing nothing is equivalent to letting time pass, resulting in a tester timeout when the deadline for producing an output is reached. This example also illustrates how our framework handles timeouts in a seamless way, without the need of modeling artifacts such as quiescence [18].

[1] Unless otherwise mentioned, deadlines of output edges are delayable and deadlines of input edges are lazy. In order not to overload the figures, we do not always draw input-complete automata. We assume that implementations ignore the missing inputs (this can be modeled by adding self-loop edges covering these inputs).

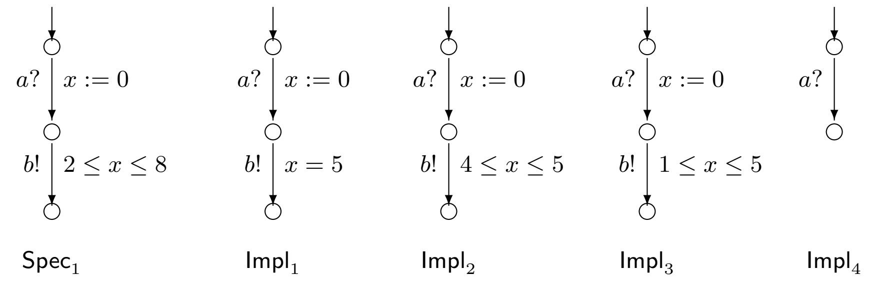


Fig. 2. Examples of specification and implementations.

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 3 shows how this can be done. The specification can be modeled compositionally, in two parts: one part modeling the environment (assumptions) and another part the nominal behavior of the SUT in this environment (requirements). In this case, the interactions between the two components are not unobservable, but are exported as inputs and outputs of the global specification. A simple example of such a situation is shown in Figure 3. 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-complete. In the above example, the second arrive input cannot be accepted until at least 20 time units have elapsed since the first arrive.

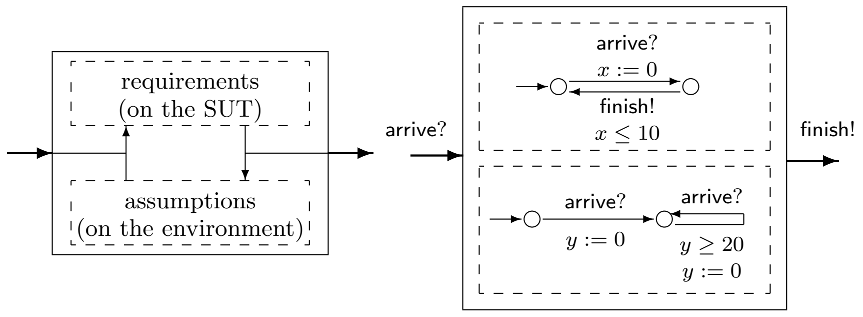


Fig. 3. Specification including assumptions on the environment: generic scheme (left) and example of a task scheduler (right).

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 can be modeled using a "begin" action for the point in time where a variable changes its value to the value that is of interest and an "end" action for the moment where the variable changes to a different value. For example, assume x is an input variable and y an output variable. Consider the requirement "y will be updated at most 10 time units after x is updated". Notice that x is updated by the environment (or the tester) while y is updated by the SUT. Thus, update_x can be introduced as an input action and update_y as an output action. The specification can be modeled as a TA similar to the one for Spec_1 of Figure 2, with a replaced by update_x and b replaced by update_y.

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 below). 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.

Modeling interfacing delays As a last example of modeling methodology, we show how to model interfacing delays between the tester and the SUT. This can again be done by composing the specification with "delay automata", as shown in Figure 4. 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.

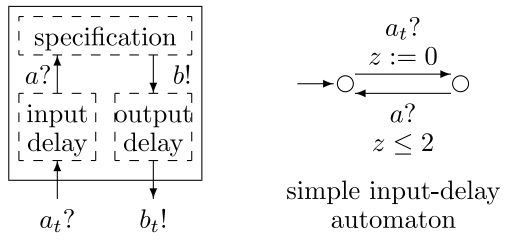


Fig. 4. Specification composed with interface-delay automata.

Digital-clock test generation 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, if the guard $1 \leq x \leq 5$ of SUT Impl_3 of Figure 2 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.

Note that generating digital-clock tests does *not* mean that we discretize time: the specification still has a continuous-time semantics. It is the tester which "samples" this semantics with a digital clock. Also note that the tests we generate are both *sound* and *precise* with respect to tioco. Intuitively, soundness means that if the tester announces "fail" then the SUT is indeed non-conforming w.r.t. tioco. The test is precise in the sense that the tester announces "fail" *as soon as possible*: as soon as the observations the tester disposes of permit to conclude that the SUT is non-conforming, the tester will announce "fail". It may, however, be the case that the observations do not permit such a conclusion to be made: this situation occurs, for instance, when a faulty behavior gives the same digital-clock observation as a non-faulty behavior.

The tester's digital clock is modeled as a Tick automaton, which is a special TAIO with a single output action tick. Three possible Tick automata are shown in Figure 5. 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).

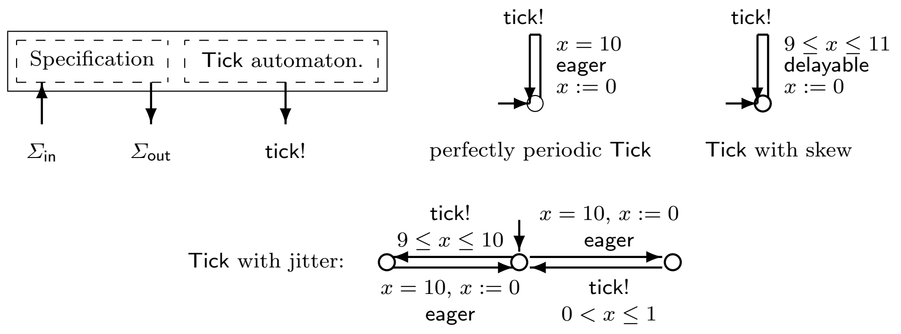


Fig. 5. Extending the specification with a tester clock model and possible such models.

Once a Tick automaton is chosen, it is composed with the specification automaton A_S as shown in Figure 5. This yields a new TAIO, denoted A_S^{Tick}, which has as inputs the inputs of A_S and as outputs the outputs of A_S plus the new output 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).

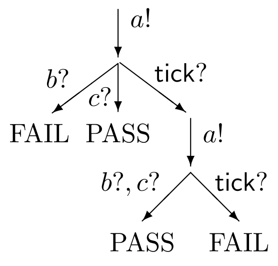


Fig. 6. A digital-clock test represented as a finite tree.

Test generation is done based on the extended specification, A_S^{Tick}.[2] A test is represented as a finite tree like the one shown in Figure 6 and is generated using an algorithm similar to the one presented in [18]. Nodes of the tree are either input nodes (where the tester issues an input to the SUT) or output nodes (where the tester awaits for an output from the SUT or for the next tick of its own clock). Leaves are marked "pass" or "fail" indicating conformance or not. Each node of the tree corresponds to a set of states of A_S^{Tick}. Such sets 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*) [2, 6]. Computing successor nodes is also done symbolically, using a *bounded-time reachability analysis* for timed automata, as shown in [19, 12].

Test selection with respect to coverage criteria At each point during test generation, the generation algorithm has a number of choices to make: stop the test or continue, wait for an output or issue an input, which of the possible inputs,

[2] The extended specification may also include other automata to model environment assumptions, interface delays, etc., as shown previously.

etc. There are different ways to resolve these choices. For instance: *interactively* (the user guides the test generation), *randomly* (the algorithm takes decisions at random), *exhaustively* (generate all possible tests, up to a given depth provided by the user), or guided by some *coverage criterion*. Our tool implements all these choices.[3] We briefly elaborate on the last one.

At the moment, we consider simple coverage criteria such as *state*, *location* or *edge* coverage, aiming to cover, respectively, all reachable states, locations or edges of the specification. The state and location criteria are based on the fact that each node of a digital test tree corresponds to a set of states (thus also a set of locations) of the specification. Therefore, each such node "covers" the corresponding set. The edge criterion is similarly based on the fact that each edge of the test tree corresponds to a set of transitions (thus also a set of edges) of the specification.

We also consider simple variations of the above criteria. For instance, *input/output action* coverage seeks to cover reachable input/output actions of the specification. In a context of extended TA such as those used by our tool, we can also define *partial* state coverage, with respect to a subset of the variables making up the state space. It should be noted that some of these criteria subsume others. For instance, achieving state coverage implies location coverage, partial state coverage and action coverage.

We now briefly describe the test-generation algorithm w.r.t. a coverage criterion. The algorithm starts by choosing at random a point p in the space to be covered (e.g., a location for location coverage, a symbolic state for state coverage, etc.). Then a reachability algorithm is run on the product A_S^{Tick} in order to find a discrete path reaching the point p to be covered. Note that, since we consider coverage only for reachable states (or locations, or actions, etc.) the point is reachable, thus, a path exists. Also note that this path is labeled only with observable (input or output) actions. Once the path is found, it is extended into a test tree: this is done by completing all nodes in the path whose outgoing edge is labeled with an output action or tick, by the remaining outputs. This is the first generated test which covers not only p but other points as well (e.g., all locations encountered in the test tree). The algorithm proceeds by choosing a new uncovered point and repeating the above process, until all points are covered. This algorithm has been implemented in our tool TTG, described below.

3 Tool and case study

The TTG tool We have built a prototype test-generation tool, called TTG (Timed Test Generator), on top of the IF environment [3]. 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

[3] Another way to select tests is using a *test purpose*. This approach is taken, for instance, in the TGV tool [8].

includes a simulator, a model checker and a connection to the untimed test generator TGV [8]. 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.

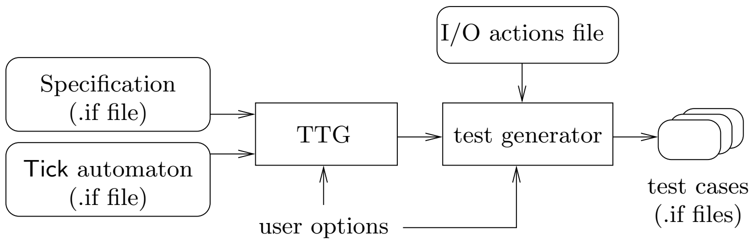


Fig. 7. The TTG tool.

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* (user-guided); *random*; *exhaustive* up to a user-defined depth; or *coverage* with respect to a criterion among *state*, *location*, *action* or *partial state*. TTG generates an executable which will perform the test generation when run. The executable takes additional options (e.g., depth) and generates one or more tests, depending on the chosen mode. The tests are output in IF language.

Case study: the Bounded Retransmission Protocol The Bounded Retransmission Protocol (BRP) is a protocol for transmitting files over an unreliable (lossy) medium. The architecture of the protocol is shown in Figure 8. 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 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.

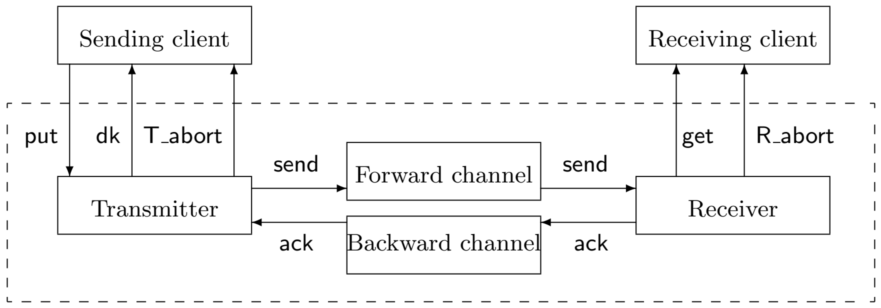


Fig. 8. The BRP specification and interfaces

Here, we use the BRP model developed in [4]. The model has been initially developed in SDL, then automatically translated in IF. The model is shown in Figure 9.[4] 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".[5] 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:

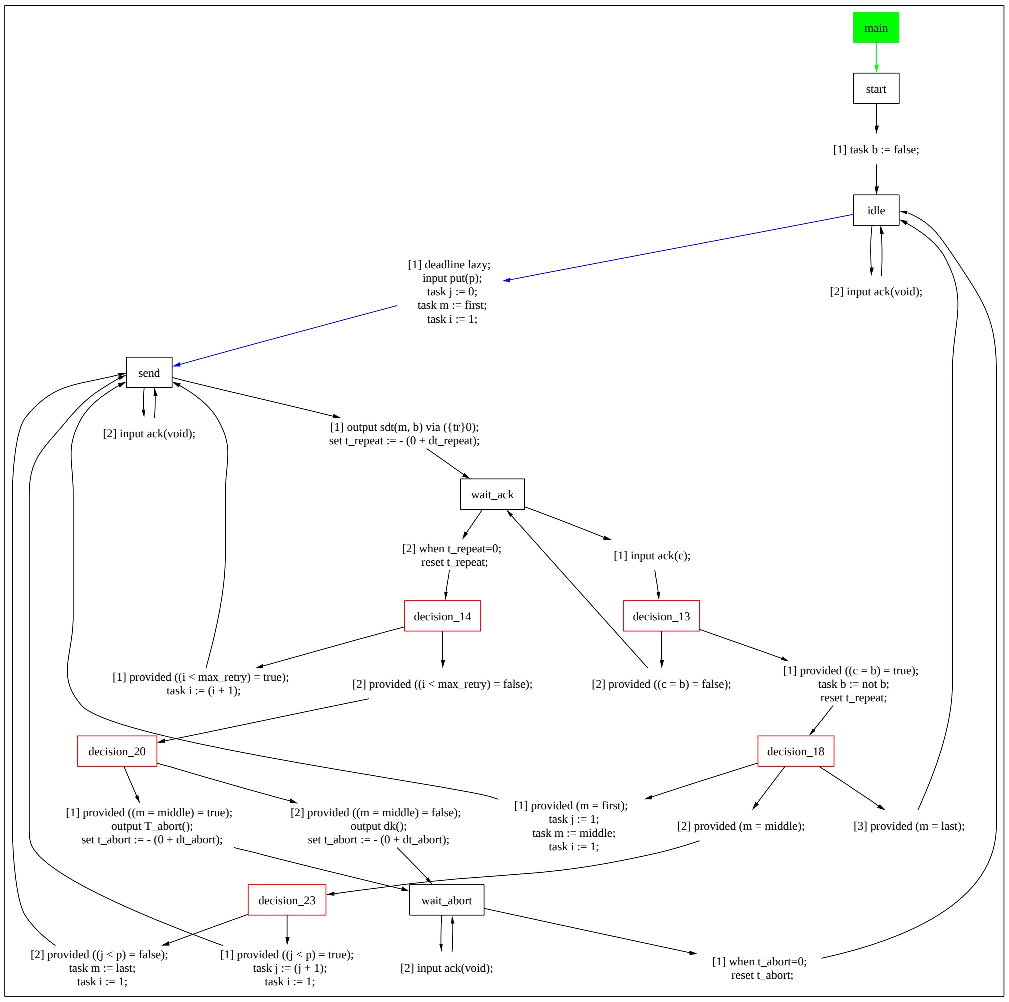
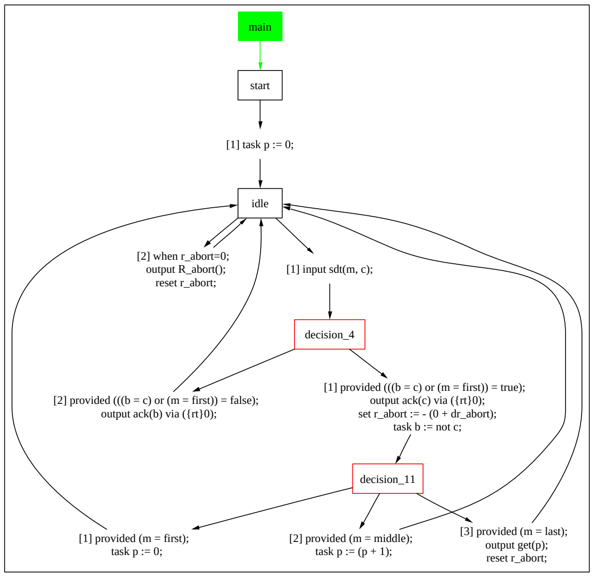
$$\mathsf{p} = 2,\ \ \mathsf{max\_retry} = 4,\ \ \mathsf{dt\_repeat} = 2,\ \ \mathsf{dt\_abort} = 15,\ \ \mathsf{dr\_abort} = 13.$$

For testing, we view the four components enclosed in dashed square in Figure 8 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).

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 1. The criteria used are: (reachable) configurations, locations, actions, and the values of the five discrete variables of the model, namely, m, b, c, i, j. A configuration corresponds to an entire symbolic state and includes a vector of locations and values of variables for each automaton, plus a DBM

[4] The automata have been drawn automatically using the `if2eps` tool by Marius Bozga. The model of BRP that we use in this paper can be found in the IF web page: `http://www-verimag.imag.fr/∼async/IF/` under "examples".

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


main
start
[1] task b := false;
idle
[1] deadline lazy; input put(p); task j := 0; task m := first; task i := 1;
[2] input ack(void);
send
[2] input ack(void);
[1] output sdt(m, b) via ({tr}0); set t_repeat := - (0 + dt_repeat);
wait_ack
[2] when t_repeat=0; reset t_repeat;
[1] input ack(c);
decision_14
decision_13
[1] provided ((i < max_retry) = true); task i := (i + 1);
[2] provided ((i < max_retry) = false);
[2] provided ((c = b) = false);
[1] provided ((c = b) = true); task b := not b; reset t_repeat;
decision_20
decision_18
[1] provided ((m = middle) = true); output T_abort(); set t_abort := - (0 + dt_abort);
[2] provided ((m = middle) = false); output dk(); set t_abort := - (0 + dt_abort);
[1] provided (m = first); task j := 1; task m := middle; task i := 1;
[2] provided (m = middle);
[3] provided (m = last);
decision_23
wait_abort
[2] provided ((j < p) = false); task m := last; task i := 1;
[1] provided ((j < p) = true); task j := (j + 1); task i := 1;
[2] input ack(void);
[1] when t_abort=0; reset t_abort;

main
start
[1] task p := 0;
idle
[2] when r_abort=0; output R_abort(); reset r_abort;
[1] input sdt(m, c);
decision_4
[2] provided (((b = c) or (m = first)) = false); output ack(b) via ({rt}0);
[1] provided (((b = c) or (m = first)) = true); output ack(c) via ({rt}0); set r_abort := - (0 + dr_abort); task b := not c;
decision_11
[1] provided (m = first); task p := 0;
[2] provided (m = middle); task p := (p + 1);
[3] provided (m = last); output get(p); reset r_abort;
Fig. 9. Transmitter (up) and Receiver (down).

representing symbolically the set of clock states. Thus, this criterion is the same as state coverage discussed in Section 2.

Column "size" shows the number of elements to be covered. Thus, there are 14687 reachable configurations in total[6], 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. 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. Notice that the configuration criterion requires 24 tests whereas all other criteria can be covered with just one test. Column "depth" shows the depth of the generated tests (i.e., the length of the longest path from the root to a leaf). For the configuration criterion, the depth varies between 6 and 53. 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 3105 configurations, which amounts to approximately 21% of the total number of configurations.

criterion used	size	time (sec)	# of tests	depth	coverage of other criteria							
					config.	locations	actions	m	b	c	i	j
config.	14687	400	24	6 - 53	100%							
locations	4	37	1	12	21%	100%	100%	100%	100%	100%	25%	100%
actions	7	76	1	43	36%	100%	100%	100%	100%	100%	25%	100%
m	3	17	1	2	1%	75%	50%	100%	100%	100%	25%	100%
b	2	16	1	2	1%	75%	50%	100%	100%	100%	25%	100%
c	2	17	1	2	1%	75%	50%	100%	100%	100%	25%	100%
i	4	35	1	9	20%	75%	50%	100%	100%	100%	100%	100%
j	3	16	1	2	1%	75%	50%	100%	100%	100%	25%	100%

Table 1. Test generation results for the BRP case study

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 2, 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

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

of coverage is achieved: for instance, 18% configuration coverage for 371 tests at depth 6.

depth	time	# of	coverage of other criteria							
	(sec)	tests	config.	locations	actions	m	b	c	i	j
1	17	2	0.2%	75%	33%	100%	100%	100%	25%	100%
2	17	3	2%	75%	50%	100%	100%	100%	50%	100%
3	22	5	5%	75%	50%	100%	100%	100%	50%	100%
4	39	11	8%	75%	50%	100%	100%	100%	75%	100%
5	168	41	14%	75%	50%	100%	100%	100%	75%	100%
6	1677	371	18%	75%	50%	100%	100%	100%	100%	100%

Table 2. Exhaustive test suites for the BRP case study

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 1 the largest test depth is 53. 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) 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 10.

4 Summary and future work

We have proposed a testing framework for real-time systems based on partially-observable, non-deterministic timed-automata specifications and on digital-clock tests. To our knowledge, this is the first framework that can fully handle such specifications and such tests. We showed that, through appropriate modeling, assumptions on the environment and the interface between the tester and SUT can be captured in the framework in a seamless way, without need for extra notions or algorithms. We also reported on a recent implementation of a test generation algorithm with respect to coverage criteria and experimental results obtained for the Bounded Retransmission Protocol. These results show that a few tests suffice to cover thousands of reachable symbolic states of the specification. We are currently studying alternative notions of coverage and methods to generate *minimal* test suites (without redundant tests). We are also examining how to adapt other testing problems than conformance, for instance, fault detection or state identification [14], to the timed setting.

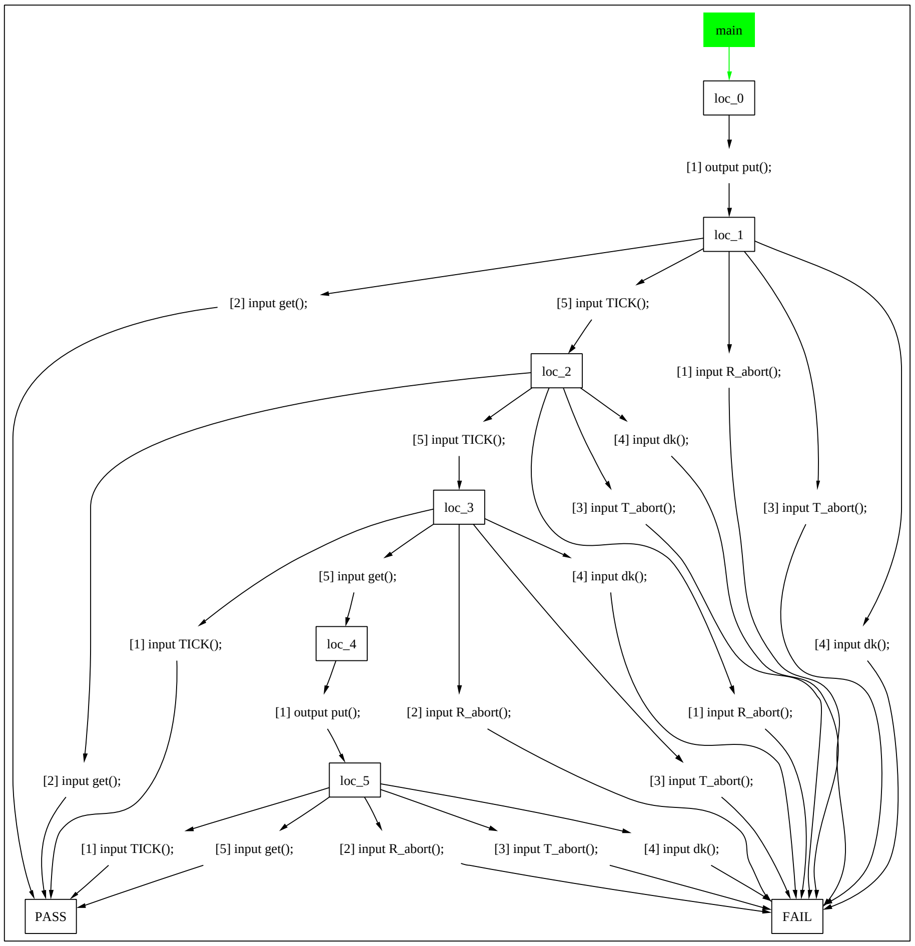


Fig. 10. A test generated by TTG for the BRP case study

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. B. Berthomieu and M. Menasche. An enumerative approach for analyzing time Petri nets. *IFIP Congress Series*, 9:41–46, 1983.
3. 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.
4. 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.
5. R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, 2000.
6. D.L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, volume 407 of *LNCS*, pages 197–212. Springer–Verlag, 1989.
7. A. En-Nouaary, R. Dssouli, F. Khendek, and A. Elqortobi. Timed test cases generation based on state characterization technique. In *RTSS'98*. IEEE, 1998.
8. 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.
9. A. Hessel, K. Larsen, B. Nielsen, P. Pettersson, and A. Skou. Time-optimal real-time test case generation using UPPAAL. In *FATES'03*, 2003.
10. A. Khoumsi, T. Jéron, and H. Marchand. Test cases generation for nondeterministic real-time systems. In *FATES'03*, 2003.
11. Z. Kohavi. *Switching and finite automata theory, 2nd ed.* McGraw-Hill, 1978.
12. 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.
13. V. Kuliamin, A. Petrenko, N. Pakoulin, A. Kossatchev, and I. Bourdonov. Integration of functional and timed testing of real-time and concurrent systems. In *Ershov Memorial Conference*, volume 2890 of *LNCS*. Springer, 2003.
14. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - A survey. *Proceedings of the IEEE*, 84:1090–1126, 1996.
15. B. Nielsen and A. Skou. Automated test generation from timed automata. In *TACAS'01.* LNCS 2031, Springer, 2001.
16. A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in EFSM testing. *IEEE Trans. Software Eng.*, 30(1), 2004.
17. J. Springintveld, F. Vaandrager, and P. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254, 2001.
18. J. Tretmans. Testing concurrent systems: A formal approach. In J.C.M Baeten and S. Mauw, editors, *CONCUR'99 – 10^{th} Int. Conference on Concurrency Theory*, volume 1664 of *LNCS*, pages 46–65. Springer-Verlag, 1999.
19. 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.