# A formal framework for black-box conformance testing of distributed real-time systems

Moez Krichen

Research Unit of Development and
Control of Distributed Applications,
Higher Institute of Computer Science and Multimedia of Sfax,
University of Sfax,
Technopole de Sfax BP 242, 3021, Sfax, Tunisia
E-mail: moez.krichen@redcad.org

**Abstract:** We extend our previous work on model-based conformance testing (Bensalem et al., 2007; Krichen and Tripakis, 2009). We propose a formal framework for black-box conformance testing for distributed real-time systems. Our framework is based on the model of partially-observable, non-deterministic timed automata. A given distributed system can be modelled either as a single timed automaton or a network of timed automata. We recall the definition of the timed input-output conformance relation tioco. We consider two types of tests: analogue-clock tests and digital-clock tests. Our algorithm for generating analogue-clock tests is based on an on-the-fly determinisation of the specification automaton during the execution of the test, which in turn relies on reachability computations. We also provide algorithms for static or on-the-fly generation of digital-clock tests. These tests measure time only with finite-precision digital clocks. Our testing architecture may be either centralised or not.

**Keywords:** black-box; model-based; conformance-testing; real-time; distributed; conformance relation; analogue-clock; digital-clock; centralised; testing architecture.

**Reference** to this paper should be made as follows: Krichen, M. (2012) 'A formal framework for black-box conformance testing of distributed real-time systems', *Int. J. Critical Computer-Based Systems*, Vol. 3, Nos. 1/2, pp.26–43.

**Biographical notes:** Moez Krichen received his PhD in Computer Science from the University of Joseph Fourrier, Grenoble in 2007. He is currently an Associate Professor at the Higher Institute of Computer Science and Multimedia of Sfax (ISIMS). He is a member of the Research Unit on Development and Control of Distributed Applications (REDCAD). His research interests include model-based testing for real-time systems, online monitoring and testing of distributed systems and state identification problems for timed finite state machines (TFSM).

This paper is a revised and expanded version of papers entitled 'A formal framework for conformance testing of distributed real-time systems' and 'Test generation for duration systems' presented at OPODIS 2010, Tozeur, Tunisia, 14–17 December 2010 and VECoS 2007, Algiers, Algeria, 5–6 May 2007.

## 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, thus, can only rely on its observable input/output behaviour. More precisely we are interested in *model-based testing* where the specification is described by a formal model from which a test suite is automatically generated. The obtained tests are then applied to the SUT in order to check the conformance of the SUT with respect to the specification in hand.

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.

Distributed real-time systems correspond to a particular class of real-time systems where the system in hand is made of several interacting components. The communication between these components may be wired, wireless or a mixture of both. In this case, the real-time constraints to consider may be either local (i.e., constraints on a single component) or global (i.e., constraints on the whole system). However, even though a local time constraint may be defined locally at the level of a single component it will generally depend on the interaction with other components of the system. This makes testing distributed real-time systems a harder problem in general.

Monitoring distributed real-time systems can be seen as a particular case of real-time testing. In this case, the tester has no control on the system under test. The SUT is controlled by its environment and by itself. The tester only observes the behaviour of the system and tries to detect possible anomalies with respect to the specification.

We model specifications as timed automata with input, output or unobservable actions (Krichen and Tripakis, 2009). 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. *Abstractions* from low-level details are also used often, to simplify modelling and manage complexity. Such abstractions could, for instance, 'hide' some variables and behaviour, which typically results in non-determinism. 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 Tretmans (1999).

We consider both analogue-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 modelled as a timed automaton. In that way, the user has full control on the assumptions about the execution platform where the tester will execute. 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 *online* (or *on-the-fly*) and *off-line*. Online test generation means that the test is generated essentially during execution. In off-line test, the test is usually represented as a finite tree with PASS/FAIL annotations on the leaves. Online versus off-line test generation is essentially a time versus space trade-off: online 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. All our test-generation methods rely on *symbolic reachability* algorithms similar to those used in timed automata model-checking tools such as Kronos (Daws et al., 1996).

The rest of this paper is organised as follows. Section 2 explains how distributed real-time systems can be modelled by means of timed automata with inputs and outputs. Section 3 describes our testing architecture. Section 4 introduces our testing formal framework, defines analogue and digital-clock tests and presents the test generation methods for the two types of tests. Section 5 reports on some related works. Section 6 presents the conclusions and future work plans.

## 2 Modelling distributed systems using timed automata

### *2.1 Timed labelled transition systems*

Let $\mathsf{R}$ be the set of non-negative reals and $\mathsf{Q}$ the set of non-negative rationals. Given a finite set of *actions* $\mathsf{Act}$, the set $(\mathsf{Act} \cup \mathsf{R})^*$ of all finite-length *real-time sequences* over $\mathsf{Act}$ will be denoted $\mathsf{RT}(\mathsf{Act})$. $\epsilon \in \mathsf{RT}(\mathsf{Act})$ is the empty sequence. Given $\mathsf{Act}' \subseteq \mathsf{Act}$ and $\rho \in \mathsf{RT}(\mathsf{Act})$, $\mathsf{P}_{\mathsf{Act}'}(\rho)$ denotes the *projection* of $\rho$ to $\mathsf{Act}' \cup \mathsf{R}$, obtained by 'erasing' from $\rho$ all actions not in $\mathsf{Act}' \cup \mathsf{R}$. The time spent in a sequence $\rho$, denoted $\mathsf{time}(\rho)$ is the sum of all delays in $\rho$. In the rest of the document, we assume given a set of actions $\mathsf{Act}$, partitioned in two disjoint sets: a set of *input actions* $\mathsf{Act}_{\mathsf{in}}$ and a set of *output actions* $\mathsf{Act}_{\mathsf{out}}$. We also assume there is an *unobservable* action $\tau \notin \mathsf{Act}$. Let $\mathsf{Act}_\tau = \mathsf{Act} \cup \{\tau\}$.

A *timed labelled transition system* (TLTS) over $\mathsf{Act}$ 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 $t \in \mathsf{R}$.

$T_t$ 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 ($S$, $s_0$, Act, $T_d$, $T_t$) is said to be r*ational-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

$$s \xrightarrow{\mu} s' \stackrel{Def}{=} (s, \mu, s') \in T_d \bigcup T_t;$$

$$s \xrightarrow{\mu} \stackrel{Def}{=} \exists s' : s \xrightarrow{\mu} s';$$

$$s \overset{\mu}{\nrightarrow} \stackrel{Def}{=} \nexists s' : s \xrightarrow{\mu} s';$$

$$s \xrightarrow{\mu_1 \cdots \mu_n} s' \stackrel{Def}{=} \exists s_1, \cdots, s_n : s = s_1 \xrightarrow{\mu_1} s_2 \xrightarrow{\mu_2} \cdots \xrightarrow{\mu_n} s_n = s';$$

$$s \xrightarrow{\rho} \stackrel{Def}{=} \exists s' : s \xrightarrow{\rho} s';$$

$$s \overset{\rho}{\nrightarrow} \stackrel{Def}{=} \nexists s' : s \xrightarrow{\rho} s'.$$

- Observable transitions

$$s \overset{\epsilon}{\Rightarrow} s' \stackrel{Def}{=} s = s' \text{ or } s \overset{\tau \cdots \tau}{\rightarrow} s';$$

$$s \overset{a}{\Rightarrow} s' \stackrel{Def}{=} \exists s_1, s_2 : s \overset{\epsilon}{\Rightarrow} s_1 \xrightarrow{a} s_2 \overset{\epsilon}{\Rightarrow} s';$$

$$s \overset{a}{\Rightarrow} \stackrel{Def}{=} \exists s' : s \overset{a}{\Rightarrow} s';$$

$$s \overset{a}{\not\Rightarrow} \stackrel{Def}{=} \nexists s' : s \overset{a}{\Rightarrow} s';$$

$$s \overset{a_1 \cdots a_n}{\Rightarrow} s' \stackrel{Def}{=} \exists s_1, \cdots, s_n : s = s_1 \overset{a_1}{\Rightarrow} s_2 \overset{a_2}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n = s';$$

$$s \overset{\sigma}{\Rightarrow} \stackrel{Def}{=} \exists s' : s \overset{\sigma}{\Rightarrow} s';$$

$$s \overset{\sigma}{\not\Rightarrow} \stackrel{Def}{=} \nexists s' : s \overset{\sigma}{\Rightarrow} s'.$$

## *2.2 Timed automata with inputs and outputs*

We use timed automata (Alur and Dill, 1994) with deadlines to model urgency (Sifakis and Yovine, 1996; Bornot et al., 1998). A *timed automaton over* Act is a tuple $A = (Q, q_0, X, \mathsf{Act}, \mathsf{E})$, where:

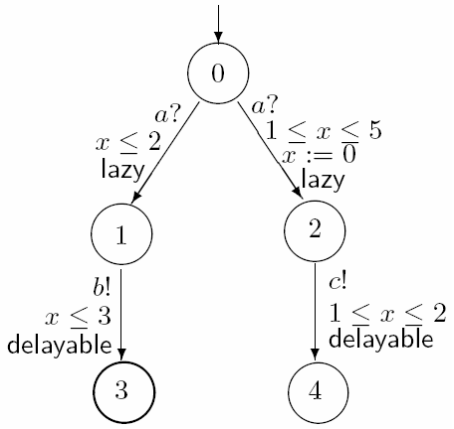
- Q is a finite set of locations
- $q_0 \in Q$ is the initial location
- $X$ is a finite set of *clocks*
- E is a finite set of *edges*.

Each edge is a tuple ($q$, $q'$, $\psi$, $r$, $d$, $a$), where:

- $q, q' \in Q$ are the source and destination locations
- $\psi$ is the *guard*, a conjunction of constraints of the form
- $x\#c$, where $x \in X$, $c$ is an integer constant
- $\# \in \{<, \leq, =, \geq, >\}$
- $r \subseteq X$ is a set of clocks to *reset* to zero
- $d \in \{\text{lazy}, \text{delayable}, \text{eager}\}$ is the *deadline*
- $a \in \text{Act}$ is the action.

An example of a TAIO is shown in Figure 1. This TAIO has five locations, one input-action, two output-actions and one clock. The symbols '?' and '!' are used to distinguish between inputs and outputs respectively.

**Figure 1** An example of a TAIO




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 $a \in \text{Act}$ and there is an edge $(q, q', \psi, r, d, a)$, such that $v$ satisfies $\psi$ and $v'$ is obtained by resetting to zero all clocks in $r$ and leaving the others unchanged. Timed transitions are of the form $(q,v) \xrightarrow{t} (q, v+t)$, where $t \in \mathsf{R}$, $t > 0$ and there is no edge $(q, q'', \psi, r, d, a)$, such that:

1 either $d = \text{delayable}$ and there exist $0 \leq t_1 < t_2 \leq t$ such that $v + t_1 \vDash \psi$ and $v + t_2 \nvDash \psi$

2 or $d = \text{eager}$ and $v \vDash \psi$.

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 $\mathsf{Reach}(A)$. A *timed automaton with inputs and outputs* (TAIO) is a timed automaton over the partitioned 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 labelled by $\tau$. Given a set of inputs $\mathsf{Act}' \subseteq \mathsf{Act}_{\mathsf{in}}$, a TAIO $A$ is called *input-enabled* with respect to $\mathsf{Act}'$ if it can accept any input in $\mathsf{Act}'$ 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 *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 $\forall s \in \mathsf{Reach}(A).\ \forall t \in \mathsf{R}.\ \exists \rho \in \mathsf{RT}\left(\mathsf{Act}_{\mathsf{out}} \cup \{\tau\}\right) : \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho}$. 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

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

The set of *observable timed traces* of A is:

$$\mathsf{ObsTTraces}(A) = \left\{ \mathsf{P}_{\mathsf{Act}}(\rho) \mid \rho \in \mathsf{RT}\left(\mathsf{Act}_\tau\right) \wedge s_0^A \xrightarrow{\rho} \right\}..$$

The TLTS defined by a TAIO is called a *timed input-output LTS* (TIOLTS).

## *2.3 Parallel composition of TAIO*

We are given two TAIO $A_1 = \left(Q_1, q_0^1, X_1, \mathsf{Act}_{\mathsf{out}}^1 \cup \mathsf{Act}_{1\to 2}, \mathsf{E}_1\right)$ and $A_2 = \left(Q_2, q_0^2, X_2, \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{1\to 2}, \mathsf{Act}_{\mathsf{out}}^2 \cup \mathsf{Act}_{2\to 1}, \mathsf{E}_2\right)$. 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 $\mathsf{Act}_{\mathsf{in}}^1, \mathsf{Act}_{\mathsf{out}}^1, \mathsf{Act}_{\mathsf{in}}^2, \mathsf{Act}_{\mathsf{out}}^2, \mathsf{Act}_{1\to 2}$ and $\mathsf{Act}_{2\to 1}$ are pairwise disjoint. 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. 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 $\left(Q_1 \times Q_2, \left(q_0^1, q_0^2\right), X_1 \cup X_2, \mathsf{Act}_{\mathsf{in}}, \mathsf{Act}_{\mathsf{out}}, \mathsf{E}\right)$ such that $\mathsf{Act}_{\mathsf{in}} = \mathsf{Act}_{\mathsf{in}} \cup \mathsf{Act}_{\mathsf{in}}^2$, $\mathsf{Act}_{\mathsf{out}} = \mathsf{Act}_{\mathsf{out}}^1 \cup \mathsf{Act}_{\mathsf{out}}^2$ and $E$ is the smallest set such that:

- for $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \mathsf{Act}_{\mathsf{in}}^1 \cup \mathsf{Act}_{\mathsf{out}}^1 \cup \{\tau_1\}$ :
  $(q_1, q_1', \psi_1, r_1, d_1, a) \in \mathsf{E}_1 \Rightarrow ((q_1, q_2), (q_1', q_2), \psi_1, r_1, d_1, a) \in \mathsf{E}$

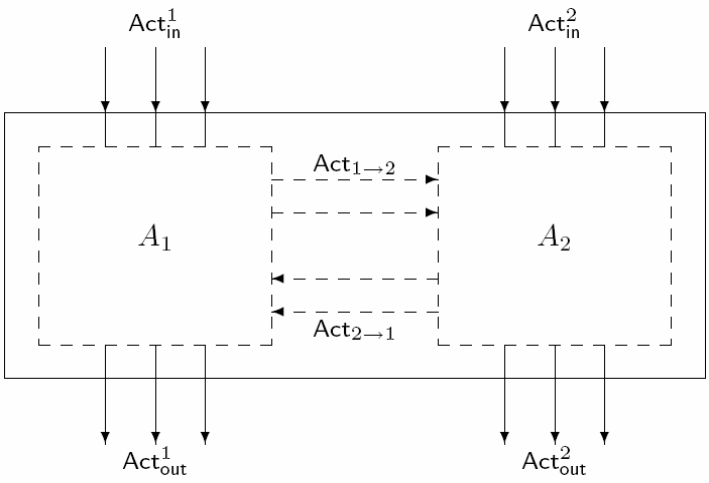
- for $(q_1, q_2) \in Q_1 \times Q_2$ and $a \in \mathsf{Act}_{\mathsf{in}}^2 \cup \mathsf{Act}_{\mathsf{out}}^2 \cup \{\tau_2\}$ :
  $(q_2, q_2', \psi_2, r_2, d_2, a) \in \mathsf{E}_2 \Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_2, r_2, d_2, a) \in \mathsf{E}$

- for $a \in \mathsf{Act}_{1\to 2}$: $(q_1, q_1', \psi_1, r_1, d_1, a) \in \mathsf{E}_1 \wedge (q_2, q_2', \psi_2, r_2, \mathsf{lazy}, a) \in \mathsf{E}_2 \Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_1, \tau_a) \in \mathsf{E}$

- for $a \in \mathsf{Act}_{2\to 1}$: $(q_1, q_1', \psi_1, r_1, \mathsf{lazy}, a) \in \mathsf{E}_1 \wedge (q_2, q_2', \psi_2, r_2, d_2, a) \in \mathsf{E}_2 \Rightarrow ((q_1, q_2), (q_1, q_2'), \psi_1 \wedge \psi_2, r_1 \cup r_2, d_2, \tau_a) \in \mathsf{E}.$

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

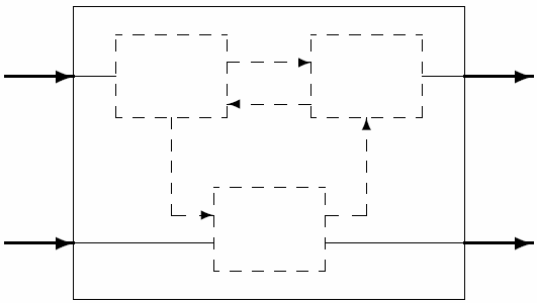
**Figure 2** Two interacting TAIO




## *2.4 Modelling distributed real-time systems*

Specifications are usually built in a *compositional* way, from many components. This greatly simplifies modelling. Figure 3 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).

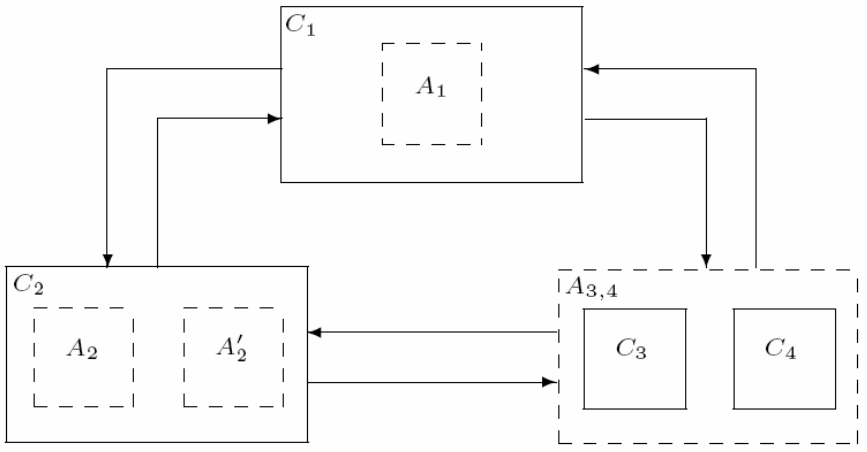
**Figure 3** A compositional specification with internal (unobservable) actions



Notice that a compositional specification does not require the SUT to be implemented following the same structure. Composition is merely a way of modelling the specification. Thus, a given distributed system can be modelled either as a single timed automaton or as a network of timed automata. To model our system, we may associate one timed automaton per component or not. Moreover a single timed automaton may be used to model the behaviour of the composition of several interacting components. Thus, the total number of timed automata used for modelling and the number of components of the system to model may not be the same.

For instance, the distributed system shown in Figure 4 consists of four components, namely $C_1$, $C_2$, $C_3$ and $C_4$. The behaviour of component $C_1$ (solid line box) is modelled by TAIO $A_1$ (dotted box). The behaviour of $C_2$ is modelled by TAIO $A_2 \| A_2'$ obtained by the parallel composition of the two TAIO $A_2$ and $A_2'$. Finally, the subsystem made of the two components $C_3$ and $C_4$ is modelled by TAIO $A_{3,4}$. That is we do not dispose, in this case, of the model of each of the components $C_3$ or $C_4$ separately. Rather, we dispose of a model of the subsystem they make.

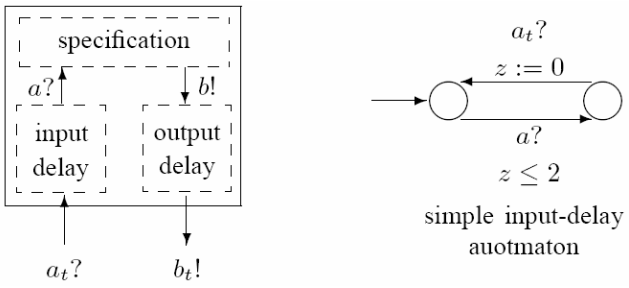
**Figure 4** An example of the structure of a real-time distributed system modelled using a network of (four) timed automata



### 2.5 Modelling issues

The main goal of this section is to illustrate some methodological aspects of our framework. We show how to model interfacing delays between two components of the SUT or also the 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 5. 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 two time units until it is perceived by the SUT.

**Figure 5** Specification composed with interface-delay automata



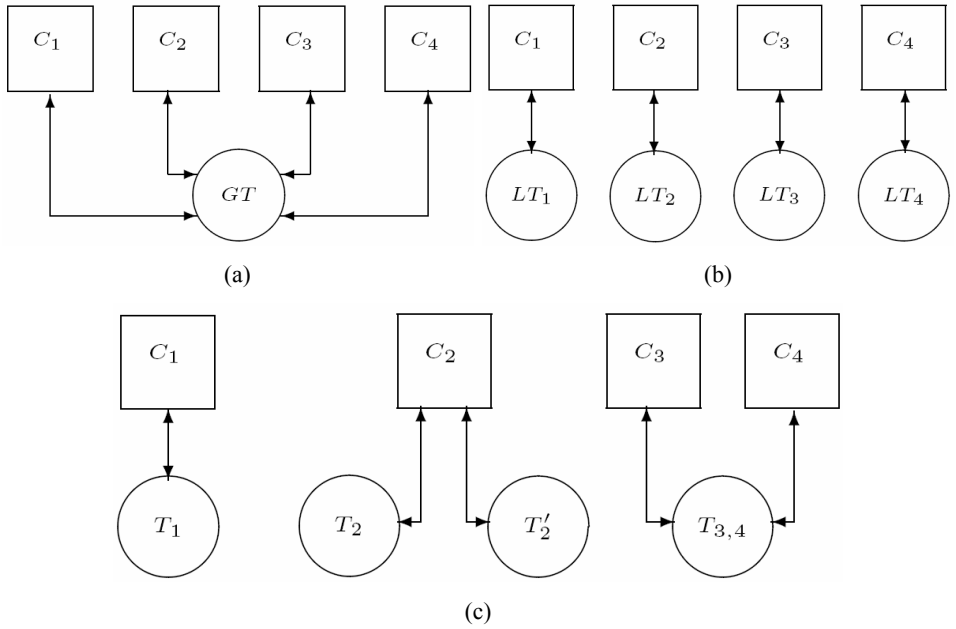

Notice that this automaton does not allow a new input to be produced while the previous one is still in 'transit'. For this, a more complicated automaton is necessary, which buffers input events. The point is that such elaborate interfaces can all be modelled explicitly. Thus, the user has full control on how the assumptions made on the tester equipment affect the generated tests. Also notice that it is possible to use the TAIO formalism to model broadcasting from one component of the SUT to other components.

## 3 Testing architecture

The role of a tester consists in interacting with the SUT in order to execute the available test cases and then to observe the response of the SUT due to this excitation. In case of monitoring, the role of the tester is limited to observe the behaviour of the SUT and to decide whether the generated behaviour is accepted or not. In case of online testing, the tester may be also in charge of generating test cases (i.e., deriving them from the specification) on-the-fly while testing.

A given tester may be global or local. We may either associate only one tester with the whole system under test [e.g., Figure 6(a)] or associate one tester with each component of the system [e.g., Figure 6(b)]. These are two extreme situations for a possible testing architecture which may be referred to as *global-tester-based architecture* and *local-tester-based architecture*, respectively. In this work, we propose a more general testing architecture named *hybrid architecture* which allows both extreme situations. That is in our framework, a given component of the SUT may be connected to more than one testers simultaneously. On the other hand, a given tester may be associated with one or several components of the SUT as well.

**Figure 6** Different possible testing architectures for distributed real-time systems, (a) global-tester-based architecture (b) local-tester-based architecture (c) hybrid architecture




For instance, a possible hybrid testing architecture for the distributed real-time system of Figure 4 is depicted in Figure 6(c). Accordingly, one tester $T_1$ is associated with component $C_1$; two testers $T_2$ and $T_2'$ are associated with $C_2$; and a same tester $T_{3,4}$ is associated with the two components $C_3$ and $C_4$.

It is worth noting that the testing architecture proposed for this example is not unique. Also note that there is no correlation between the way the SUT is modelled and the testing architecture to adopt. In other terms, it is not necessary to associate a tester with each TAIO appearing in the model of the SUT. For instance, the number of testers to use may exceed the number of TAIO appearing in the model and vice versa. In our example, we have chosen to test component $C_2$ using two testers namely $T_2$ and $T_2'$. This may be justifiable by the fact that in this case we are interested in checking the conformance of $C_2$ with respect to two distinct and independent properties (one encoded by $A_2$ and the other one by $A_2'$ for instance). So each tester has a separate job to achieve independently from the other tester.

In some other situations, it would have been possible to test $C_2$ by means of only one tester even though the behaviour of the component has been specified as the parallel product of two TAIO $A_2 \| A_2'$ for instance. In that case, the situation will be quite similar to the case of $C_1$.

For the case of $C_3$ and $C_4$ while executing a test case, tester $T_{3,4}$ must know which inputs it should send to $C_3$ and which ones to $C_4$. In addition, the tester should combine the obtained responses from the two components to deduce whether the executed test has been successful or not.

In general, the different testers may need to communicate with each other to exchange messages about their respective observations about the SUT. Since we are dealing with real-time systems, these exchanged messages should contain the instants at which these observations are made. Moreover since each tester has its own local clock, a phase of clock synchronisation is needed between the clocks of the testers.

## 4 Formal testing framework

We now describe our testing framework. We assume that the specification of the system to be tested is given as a non-blocking TAIO $A_S$. We assume that the implementation (i.e., the system to be tested) can be modelled 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).

### *4.1 Timed input-output conformance relation: tioco*

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 $\sigma$ is the set of all states of $A$ that can be reached by some timed sequence $\rho$ whose projection to observable actions is $\sigma$. Formally:

$$A \text{ after } \sigma = \left\{ s \in S_A \middle| \exists \rho \in \mathsf{RT}\left(\mathsf{Act}_\tau\right) : s_0^A \xrightarrow{\rho} s \wedge \mathsf{P}_{\mathsf{Act}}(\rho) = \sigma \right\}.$$

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:

$$\mathsf{elapse}(s) = \left\{ t > 0 \middle| \exists \rho \in \mathsf{RT}(\{\tau\}) : \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho} \right\}.$$

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:

$$\mathsf{out}(s) = \left\{ a \in \mathsf{Act}_{\mathsf{out}} \middle| s \xrightarrow{a} \right\} \cup \mathsf{elapse}(s)$$

and

$$\mathsf{out}(s) = \bigcup_{s \in S} \mathsf{out}(s).$$

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

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

The relation states that $A_I$ conforms to $A_S$ iff for any observable behaviour $\sigma$ of $A_S$, the set of observable outputs of $A_I$ after $\sigma$ must be a subset of the set of possible observable outputs of $A_S$.

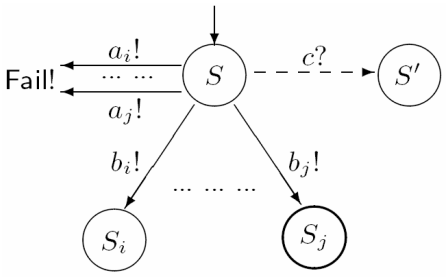
Let $A_1$, $A_1'$, $A_2$ and $A_2'$ be four TAIO such that, for $i$ = 1, 2, $A_i$ and $A_i'$ have the same sets of inputs and outputs, as shown in Figure 2. Suppose that all four automata are input-enabled with respect to their respective sets of inputs. Furthermore, suppose that $A_1$ and $A_2$ are compatible with respect to ($\mathsf{Act}_{1\to2}$, $\mathsf{Act}_{2\to1}$), and so are $A_1'$ and $A_2'$. Then, we have the following compositionality result (Krichen and Tripakis, 2009).

Proposition 4.1 If $A_1'$ $\mathsf{tioco}$ $A_1$ and $A_2'$ $\mathsf{tioco}$ $A_2$ then $A_1' \| A_2'$ $\mathsf{tioco}$ $A_1 \| A_2$.

## *4.2 Test generation principle*

We adapt the untimed test generation algorithm of Tretmans (1999). 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 7. For all *illegal* outputs $a_i$ 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. At any node, the algorithm may decide to stop the test and label this node as Pass.

**Figure 7** Generic test-generation scheme




## *4.3 Generating analogue-clock tests*

We use the idea of Tripakis (2002). We represent an analogue-clock test as an *algorithm*. The latter essentially performs subset construction on the specification automaton, during the execution of the test. Thus, our analogue-clock testing method can be classified as on-the-fly or *online*, 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. We define the following operators:

$$\mathsf{dsucc}(S,a) = \left\{ s' \middle| \exists s \in S : s \xrightarrow{a} s' \right\}$$

and

$$\mathsf{tsucc}(S,t) = \left\{ s' \middle| \exists s \in S \;.\; \exists \rho \in \mathsf{RT}(\{\tau\}) : \mathsf{time}(\rho) = t \wedge s \xrightarrow{\rho} s' \right\}$$

where $a \in \mathsf{Act}$ and $t \in \mathsf{R}$. The two operators can be implemented using standard data structures for symbolic representation of the state space (Berthomieu and Menasche, 1983; Dill, 1989) and simple modifications of reachability algorithms for timed automata (Tripakis, 2002).

**Algorithm 1** On-the-fly analogue-clock test generation

```
1   S ← tsucc({s0^As}, 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)  S ← dsucc(tsucc(S, x), b);
6       else  S ← tsucc(S, T);
7       endif;
8       if(S = ∅) announce Fail and exit;
9       endif;
10      if(valid_inputs(S) ≠ ∅)
11          i ← pick({0,1});/* 0 to send an input and 1 to continue observation */
12      endif;
13      if(i = 0)
14          a ← pick(valid_inputs(S));
15          S ← dsucc(S, a);
16      endif;
17  endwhile;
```

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, ten 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.

Online analogue-clock test generation is performed by Algorithm 1. 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: $\{a \in \mathsf{Act}_{\mathsf{in}} \mid \mathsf{dsucc}(\mathsf{tsucc}(S, 0), a) \neq \emptyset\}$. 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.

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.

### *4.4 Generating digital-clock tests*

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 as a particular TAIO Tick. 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 online or off-line which is a matter of a space/time trade-off.

**Algorithm 2** Off-line digital-clock test generation

```
1   S ← {s0^(A_S^Tick)} and ← the one-node tree with root S;
2   while(true)
3       foreach(leaf S of D distinct from Pass and Fail)
4           if(valid_inputs(S) ≠ ∅) i ← pick({0, 1, 2});
5           else i ← pick({1, 2});
6           endif;
7           case(i = 0):
8               b ← pick(valid_inputs(S)) S′ ← dsucc(tsucc(S, 0), b);
9               append edge S -b-> S′ to D;
10          case(i = 1):
11              foreach(a ∈ Act_out ∪ {tick})
12                  S′ ← dsucc(tsucc(S),a);
13                  if(S ≠ ∅) append edge S -a-> S′ to D;
14                  else append edge S -a-> Fail to D;
15                  endif;
16              endforeach;
17          case(i = 2) : replace S with Pass in D;
18      endforeach;
19  endwhile;
```

We form the product $A_S^{\mathsf{Tick}} = A_S \| \mathsf{Tick}$. 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. We define the following operator on $A_S^{\mathsf{Tick}}$ :

$$\mathsf{usucc}(S,t) = \left\{ s' \mid \exists s \in S \;.\; \exists \rho \in \mathsf{RT}(\{\tau\}) : \wedge s \xrightarrow{\rho} s' \right\}$$

and we apply the generic test-generation scheme presented above. Off-line digital-clock test generation is performed by Algorithm 2. We use the same notation as in Algorithm 1.

*D* denotes the digital-clock test the algorithm generates. Algorithm 2 can be transformed in a straightforward way to an online digital-clock test generation algorithm.

## 5 Related work

Classical testing frameworks are based on Mealy machines (e.g., see Chow, 1978; Lee and Yannakakis, 1996) or finite labelled transition systems (e.g., see Tretmans, 2002; Brinksma and Tretmans, 2001; Fernandez et al., 1996; Belinfante et al., 1999; Clarke et al., 2002). These formalisms are not well-suited for modelling real-time systems. In Mealy machines, inputs and outputs are synchronous, which is a reasonable assumption when modelling 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 modelled by special $\delta$ actions (Tretmans, 1999) 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 ten 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 number of methods for testing real-time systems based on variants of the model of timed automata (or other similar models such as timed Petri nets) have been proposed (e.g., see Braberman et al., 1997; Clarke and Lee, 1997; En-Nouaary et al., 1998; Higashino et al., 1999; Nielsen and Skou, 2001; Springintveld et al., 2001; Peleska, 2002; Cardell-Oliver, 2002; Khoumsi et al., 2003; Hessel et al., 2003; Larsen et al., 2004; Briones and Brinksma, 2004). 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, Springintveld et al. (2001) and Hesse et al. (2003) consider TA where outputs are *isolated* and *urgent*. 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. The second limitation is that only analogy-clock tests are considered in the works above.

The authors of Thane and Hansson (2001) provided a technique for finding all the possible execution scenarios for a distributed real-time system with preemption and jitter. The benefit of this testing strategy is that it allows any testing technique for sequential software to be used to test distributed real-time systems. In Thane and Hansson (1999), this technique was extended to handle interrupts. Khoumsi (2001) proposed a model for specifying distributed real-time systems, a simple distributed test architecture and a procedure for distributing global test sequences. In Cavalli et al. (2008), the authors presented two different tools to test distributed systems with time constraints. The first one allows to automatically generate test cases based on model-based active testing techniques. Whereas the second tool is based on passive testing approach to check that the collected system traces respect a set of formal properties called invariants.

A new technique for testing that a distributed real-time system satisfies a formal timed automata specification was introduced in Rachel (2002). The paper outlines how to write test specifications in the language of Uppaal timed automata, how to translate those specifications into program code for executing the tests, and describes the results of test experiments on a distributed real-time system with limited hardware and software

resources. Garousi (2008) presented a modified testing methodology which can be used to stress test systems when the timing information of messages is imprecise or unpredictable. Siddiquee and En-Nouaary (2006) proposed both a centralised architecture and a distributed architecture for the execution of test cases on distributed real-time systems.

## 6 Conclusions and future work

In this work, we proposed a formal framework for testing distributed real-time systems. Our framework is based on the model of timed automata with inputs and outputs. The later allows high expressivity for real-time systems since it allows non-determinism, partial-observability and parallel composition. We recalled the definition of the timed conformance relation tioco. We proposed a hybrid testing architecture which combines classical centralised and decentralised architectures. We also provided techniques for deriving analogue-clock and digital-clock tests from the specification of the SUT.

An important contribution in this work was to use a rich formalism to model distributed real-time systems. An second important contribution was trying to make decorrelation between the number of components of the system under test, the number of entities used to model this system and finally the number of testers used for test.

Many extensions are possible for this work. First, we need selection techniques-based on coverage criteria- to guide test generation and to reduce the number of generated tests. We may also combine off-line and online testing within the same testing architecture in order to better balance the space/time trade-off. As a future work direction, we are intending to implement our testing methodology in the context of distributed software architectures.

## References

Alur, R. and Dill, D. (1994) 'A theory of timed automata', *Theoretical Computer Science*, Vol. 126, No. 2, pp.183–235.

Belinfante, A., Feenstra, J., de Vries, R.G., Tretmans, J., Goga, N., Feijs, L., Mauw, S. and Heerink, L. (1999) 'Formal test automation: a simple experiment', in *12th Int. Workshop on Testing of Communicating Systems*, Kluwer.

Bensalem, S., Krichen, M., Majdoub, L., Robbana, R. and Tripakis, S. (2007) 'Test generation for duration systems', in *First International Workshop on Verification and Evaluation of Computer and Communication Systems (VECoS 2007)*, Algiers, Algeria, 5–6 May, BCS.

Berthomieu, B. and Menasche, M. (1983) 'An enumerative approach for analyzing time Petri nets', *IFIP Congress Series*, Vol. 9, pp.41–46.

Bornot, S., Sifakis, J. and Tripakis, S. (1998) 'Modeling urgency in timed systems', in *Compositionality*, Vol. 1536 of LNCS, Springer.

Braberman, V., Felder, M. and Marre, M. (1997) 'Testing timing behavior of real-time software', in *International Software Quality Week*.

Brinksma, E. and Tretmans, J. (2001) 'Testing transition systems: an annotated bibliography', in *MOVEP 2000*, Vol. 2067 of LNCS, Springer.

Briones, L. and Brinksma, E. (2004) 'A test generation framework for quiescent real-time systems', in *FATES'04*, Vol. 3395 of LNCS, Springer.

Cardell-Oliver, R. (2002) 'Conformance test experiments for distributed real-time systems', in *ISSTA'02*, ACM Press.

Cavalli, A.R., Oca, E.M.D., Mallouli, W. and Lallali, M. (2008) 'Two complementary tools for the formal testing of distributed systems with time constraints', in *DS-RT '08*, pp.315–318, IEEE Computer Society, Washington, DC, USA.

Chow, T.S. (1978) 'Testing software design modeled by finite-state machines', *IEEE Transactions on Software Engineering*, Vol. 4, Nos. 1–2, pp.178–187.

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

Clarke, D., Jéron, T., Rusu, V. and Zinovieva, E. (2002) 'STG: a symbolic test generation tool', in *TACAS'02*, Vol. 2280 of LNCS, Springer.

Daws, C., Olivero, A., Tripakis, S. and Yovine, S. (1996) 'The tool Kronos', in *Hybrid Systems III, Verification and Control*, Vol. 1066 of LNCS, pp.208–219, Springer-Verlag.

Dill, D. (1989) 'Timing assumptions and verification of finite-state concurrent systems', in J. Sifakis (Ed.): *Automatic Verification Methods for Finite State Systems*, Vol. 407 of LNCS, pp.197–212, Springer–Verlag.

En-Nouaary, A., Dssouli, R., Khendek, F. and Elqortobi, A. (1998) 'Timed test cases generation based on state characterization technique', in *RTSS'98*, IEEE.

Fernandez, J.C., Jard, C., Jéron, T. and Viho, G. (1996) 'Using on-the-fly verification techniques for the generation of test suites', in *CAV'96*, Vol. 1102 of LNCS, Springer.

Garousi, V. (2008) 'Traffic-aware stress testing of distributed real-time systems based on UML models in the presence of time uncertainty', in *ICST*, pp.92–101, IEEE Computer Society.

Hessel, A., Larsen, K., Nielsen, B., Pettersson, P. and Skou, A. (2003) 'Time-optimal real-time test case generation using UPPAAL', in *FATES'03*.

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

Khoumsi, A. (2001) 'Testing distributed real time systems using a distributed test architecture', in *ISCC*, pp.648–654, IEEE Computer Society.

Khoumsi, A., Jéron, T. and Marchand, H. (2003) 'Test cases generation for nondeterministic real-time systems', in *FATES'03*.

Krichen, M. and Tripakis, S. (2009) 'Conformance testing for real-time systems', *Formal Methods in System Design*, Vol. 34, No. 3, pp.238–304.

Larsen, K., Mikucionis, M. and Nielsen, B. (2004) 'Online testing of real-time systems using UPPAAL', in *FATES'04*, Vol. 3395 of LNCS, Springer.

Lee, D. and Yannakakis, M. (1996) 'Principles and methods of testing finite state machines – a survey', *Proceedings of the IEEE*, Vol. 84, pp.1090–1126.

Nielsen, B. and Skou, A. (2001) 'Automated test generation from timed automata', in *TACAS'01*, LNCS 2031, Springer.

Peleska, J. (2002) 'Formal methods for test automation – hard real-time testing of controllers for the airbus aircraft family', in *IDPT'02*.

Rachel, C.O. (2002) 'Conformance test experiments for distributed real-time systems', in *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ACM, New York, NY, USA, pp.159–163.

Siddiquee, S.H. and En-Nouaary, A. (2006) 'Two architectures for testing distributed real-time systems', in *Information and Communication Technologies, ICTTA '06*, 2nd, Vol. 2, pp.3388–3393.

Sifakis, J. and Yovine, S. (1996) 'Compositional specification of timed systems', in *13th Annual Symposium on Theoretical Aspects of Computer Science, STACS'96*, Vol. 1046 of LNCS, Spinger-Verlag.

Springintveld, J., Vaandrager, F. and D'Argenio, P. (2001) 'Testing timed automata', *Theoretical Computer Science*, Vol. 254, Nos. 1–2, pp.225–257.

Thane, H. and Hansson, H. (1999) 'Handling interrupts in testing of distributed real-time systems', in *RTCSA'99*, IEEE Computer Society, Washington, DC, USA, p.450.

Thane, H. and Hansson, H. (2001) 'Testing distributed real-time systems', *Journal of Microprocessors and Microsystems*, Elsevier, Vol. 24, pp.463–478.

Tretmans, J. (1999) 'Testing concurrent systems: a formal approach', in *CONCUR'99*, Vol. 1664 of LNCS, Springer.

Tretmans, J. (2002) 'Testing techniques', Lecture notes, University of Twente, The Netherlands.

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