Abstracting Time and Data for Conformance Testing of Real-Time Systems

Wilkerson L. Andrade, Patricia D. L. Machado, Thierry Jéron, Hervé Marchand

Abstracting Time and Data for Conformance Testing of Real-Time Systems

Wilkerson L. Andrade, Patrícia D. L. Machado
Federal University of Campina Grande
Campina Grande, PB, Brazil
{*wilker,patricia*}*@dsc.ufcg.edu.br*

Thierry Jéron, Hervé Marchand
INRIA Rennes - Bretagne Atlantique
Rennes, France
{*thierry.jeron,herve.marchand*}*@inria.fr*

***Abstract*—Current approaches to model-based conformance testing of real-time systems are mostly based either on finite state machines/transition systems or on timed automata. However, most real-time systems manipulate data while being subject to time constraints. The usual solution consists in enumerating data values (in finite domains) while treating time symbolically, thus leading to the classical state explosion problem. This paper proposes a new model of real-time systems as an extension of both symbolic transition systems and timed automata, in order to handle both data and time requirements symbolically. We then adapt the tioco conformance testing theory to deal with this model and describe a test case generation process based on a combination of symbolic execution and constraint solving for the data part and symbolic analysis for timed aspects.**

***Keywords*-Conformance Testing; Real-time Systems; Symbolic Execution; Model-Based Testing; Test Case Generation;**

I. Introduction

The correct behaviour of real-time systems (RTSs) depends not only on the generated results but also on whether the results are generated at the right time-points. For software testing of such RTSs, the main challenges are related to handling time requirements along with the usual difficulties posed by concurrent and reactive systems. In this paper, we focus on conformance testing where the implementation is a black-box whose internal details are unknown, thus the tester can only interact with it through its observable behaviour (inputs and outputs) with test cases derived from a formal specification based on a conformance relation between the implementation and the specification that guides the verdicts of test execution [1].

Since research in this field is very recent, developed techniques and tools are still immature and difficult to use in practice. Most approaches are based on model checking techniques to support test case generation such as [2], [3]. Other approaches extend some classic testing strategies to support the test of RTS, for instance, coverage based test selection [4] and test purposes to select specific scenarios to be verified [5], [6]. Also, some approaches extend finite state machines (FSMs) and their associated methods to deal with time [7], [8].

There are still very few work in this context and most of them use either (variations of) FSMs or (variations of) timed automata (TAs) as the underlying model. However, most approaches to testing real-time systems abstract only time and enumerate data values. This is not suitable when the specification uses large or infinite data domains because data values are enumerated, leading to the state space explosion problem.

In practice, RTSs handle variables and action parameters. Thus, powerful models are needed where variables, action parameters and time are explicitly modelled and these informations are treated in a symbolic way. There is few work whose goal is to provide symbolic approaches to software testing such as [9], [10], [11], [12]. However, none of these approaches take time requirements into account.

The main goal of this work is to propose a new approach for model-based conformance testing of real-time systems, where the system under test (SUT) is modelled through an extension of both symbolic transition systems and timed automata, thus dealing with both data and time requirements. We then adapt the **tioco** conformance testing theory to deal with this new model. Finally, we propose a test case generation process based on test selection using test purposes, which is based on symbolic execution and constraint solving for the data aspects combined with symbolic analysis of timed aspects.

The rest of the paper is structured as follows. Section II discusses the related work. The proposed model is presented in Section III. Section IV introduces the conformance testing theory to deal with the model proposed. The test case generation process is described in Section V and some properties of the generated test cases are discussed in Section VI. Finally, Section VII presents the concluding remarks and future work.

II. Related Work

To our knowledge, most work on model-based testing of real-time systems, either do not deal with data, or enumerate data values in finite domains. For example, Cardell-Oliver [13] proposes a strategy for test generation based on the UPPAAL timed automata [14] and digital clock approximation. As the semantics of a TA is given by a possibly infinite state timed transition system (TLTS), each timed trace is mapped into a set of possible integer-timed trace

interpretations. Thus, symbolic states are used to represent a set of clock valuations but data is not handled.

En-Nouaary *et al.* [7] consider a variant of timed automata with inputs and outputs (TAIOs) with no data. The model considers only urgent transitions, i.e., once enabled, a transition must be taken immediately. This assumption reduces too much the expressiveness of the model. For instance, the following cannot be expressed "when an input is provided, an output must be generated within at most 10 time units".

Li *et al.* [5] propose an approach to property-oriented real-time test case generation but the work only focuses on the specification language, time-enriched statecharts.

Larsen *et al.* [3] propose a tool and the related theory for online testing of real-time systems based on non-deterministic TAIOs. The developed tool (UPPAAL TRON) was implemented by extending the UPPAAL model-checking tool [14]. The tool allows data restricted to finite data domains, which values are enumerated.

Krichen and Tripakis [4] propose a framework for conformance testing of real-time systems where specifications are modelled as non-deterministic TAIOs. They extend the **ioco** conformance relation of [1] into a timed input-output conformance relation (**tioco**), which is defined by considering time delays as observable outputs. Data is not considered. [6] extend the approach by a better treatment of non-determinism and test selection based on test purposes.

Merayo *et al.* [8] propose an extension of finite state machines with only one clock and no data.

Khoumsi [15] combines a real-time testing strategy with a non-real-time symbolic testing strategy. Basically, the approach is in two steps: first, the real-time symbolic model is transformed into an symbolic transition system where the setting and expiring of clocks are represented as actions; second, the symbolic testing strategy presented in [9], [11] is adapted to generate tests. However, the semantics of the proposed model is not formally defined. Additionally, the first step restricts too much the use of clocks and guards leading to a less expressive and flexible specification language.

The work presented in [16], [17] proposes an initial symbolic model-based testing strategy for real-time systems with interruptions. Moreover, it investigates a possible infrastructure that can support test execution in an actual real-time environment. Test cases are designed to run on a real-time operating system named FreeRTOS [18] – a mini-kernel that can be used to develop real-time systems for embedded devices [18].

Timo *et al.* [19] propose a conformance testing strategy for data-flow reactive systems with time constraints based on an extension of TA that uses variables as inputs and outputs. All transitions of the model are urgent, reducing expressiveness, and time is treated with region graphs which may lead to the state space explosion problem.

The problem of symbolically abstracting both time and data is addressed in this paper by an extension of the model defined in [9], [11] that allows to handle time requirements.

III. The Symbolic Model

This section presents a new symbolic model named Timed Input-Output Symbolic Transition System (TIOSTS). This model is an extension of two existing models: Timed Automata with Inputs and Outputs (TAIOs), itself an extension of timed automata [20] with distinguished inputs and outputs, and deadlines to model urgency [21]; and Input-Output Symbolic Transition Systems (IOSTSs) [9]. In other words, a TIOSTS is an automaton with a finite set of locations, variables used to represent the system data, and a finite set of clocks used to represent time evolution. An edge comprises a guard on variables and clocks, an action carrying parameters for the communication with its environment, an assignment of variables, and resets of clocks.

A. Syntax of TIOSTS

A TIOSTS is formally described as follows:

Definition 3.1 (TIOSTS): A TIOSTS consists in a tuple $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$, where:

- V is a finite set of typed variables;
- P is a finite set of parameters. For $x \in V \cup P$, $type(x)$ denotes the type of x;
- Θ is the initial condition, a predicate with variables in V;
- L is a finite, non-empty set of locations and $l^0 \in L$ is the initial location;
- $\Sigma = \Sigma^? \cup \Sigma^! \cup \Sigma^\tau$ is a non-empty, finite alphabet, which is the disjoint union of a set $\Sigma^?$ of input actions, a set $\Sigma^!$ of output actions, and a set Σ^τ of internal actions. Each action $a \in \Sigma$ has a signature $sig(a) = \langle p_1, ..., p_n \rangle$, that is a tuple of distinct parameters. The signature of internal actions is the empty tuple;
- C is a finite set of clocks;
- $\mathcal{T}$ is a finite set of transitions. Each transition $t \in \mathcal{T}$ is a tuple $\langle l, a, G, A, y, l' \rangle$, where:
  - $l \in L$ is the origin location of the transition,
  - $a \in \Sigma$ is the action,
  - $G = G^D \wedge G^C$ is the guard, where G^D is a predicate over variables in $V \cup sig(a)$[1] and G^C is a clock constraint over C defined as a conjunction of constraints of the form $\alpha \# c$, where $\alpha \in C$, c is an integer constant and $\# \in \{<, \leq, =, \geq, >\}$,
  - $A = A^D \cup A^C$ is the assignment of the transition. For each variable $x \in V$ there is exactly one assignment in A^D, of the form $x := A^{D^x}$, where A^{D^x} is an expression on $V \cup sig(a)$. $A^C \subseteq C$ is the set of clocks to be reset,
  - $y \in \{$lazy, delayable, eager$\}$ is the deadline of the transition,

[1] G^D is assumed to be expressed in a theory in which satisfiability is decidable.

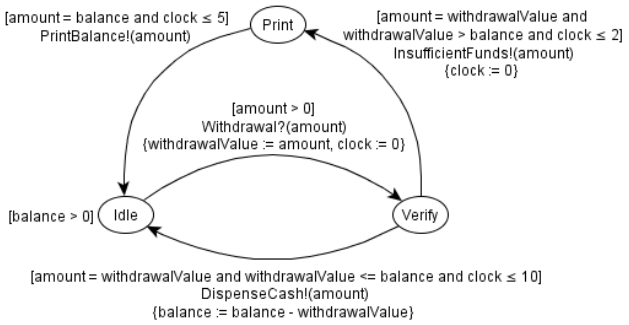

Figure 1. TIOSTS Example

– $l' \in L$ is the destination location of the transition.

◇

We intuitively explain the different notions of the TIOSTS model through the example shown in Figure 1 that models a withdrawal transaction in an ATM system. The transaction has a precondition (the initial condition) that says that the current *balance* must be strictly positive. Initially, the system is in the *Idle* location where it expects the *Withdrawal* input carrying a strictly positive integer parameter *amount* that is saved into the variable *withdrawalValue* with the clock set to zero when the transition is taken. Then, as the value of *withdrawalValue* is less than or equal to the *balance* and the time represented by *clock* is less than or equal to 10 time units, the ATM system dispenses the cash through the output *DispenseCash* carrying the parameter *amount* (the guard $amount = withdrawalValue$ and $withdrawalValue <= balance$ means "choose a value for the parameter *amount* that, with the value of the variable *withdrawalValue*, satisfies the guard"), the variable *balance* is decreased by the withdrawn value, and the system returns to *Idle*. Otherwise, if the account does not have sufficient funds, the system must emit the invalid withdrawal value through the output *InsufficientFunds* carrying the parameter *amount* when *clock* is at most 2 (the guard $amount = withdrawalValue$ and $withdrawalValue > balance$ has a similar meaning to the previous guard), and reset the clock to zero again. Finally, the current balance is emitted through the output *PrintBalance* when *clock* is at most 5 (the guard $amount = balance$ means "choose a value for the parameter *amount* such that it is equal to the value of the parameter *balance*"), and the system returns to *Idle*.

Guards on transitions indicate when they are enabled. However transitions can be forced using deadlines used to model urgency [21]. Transitions are annotated with one of the following three deadlines: *lazy*, *delayable*, and *eager*. The *lazy* deadline imposes no urgency to the transition to be taken, *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. In the figures, when not specified, the deadline of transitions with output actions is assumed to be *delayable* and the deadline of transitions with input actions is assumed to be *lazy*.

B. Semantics of TIOSTS

The semantics of a TIOSTS $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is described by a Timed Input-Output Labelled Transition Systems (TIOLTS). Intuitively, the TIOLTS states explore the sets of locations, of *valuations* of variables V and clocks C, while transitions explore the sets of actions Σ associated with parameters values P. A *valuation* of the variables in V is a mapping ν which maps every variable $x \in V$ to a value $\nu(x)$ in the domain of x. Valuations of parameters P are defined similarly. Let $\mathcal{V}$ denote the set of valuations of the variables V and let Γ denote the set of valuations of the parameters P. Let the function $\psi : C \to \mathbb{R}^{\geq 0}$ denote a clock valuation. We note $\overline{0}$ the valuation that assigns 0 to all clocks.

Considering $\nu \in \mathcal{V}$ and $\gamma \in \Gamma$, for an expression E involving a subset of $V \cup P$, we denote by $E(\nu, \gamma)$ the value obtained by evaluating the result of substituting in E each variable by its value according to ν and each parameter by its value according to γ.

Definition 3.2 (TIOLTS semantics of a TIOSTS): The semantics of a TIOSTS $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is a TIOLTS $[\![W]\!] = \langle S, S^0, Act, T \rangle$, defined as follows:

- $S = L \times \mathcal{V} \times (C \to \mathbb{R}^{\geq 0})$ is the set of states of the form $s = \langle l, \nu, \psi \rangle$ where $l \in L$ is a location, $\nu \in \mathcal{V}$ is a specific valuation for all variables V, and ψ is a clock valuation;
- $S^0 = \{\langle l^0, \nu, \psi \rangle \mid \Theta(\nu) = true, \overline{0}\}$ is the set of initial states;
- $Act = \Lambda \cup D$ is the set of actions, where $\Lambda = \{\langle a, \gamma \rangle \mid a \in \Sigma, \gamma \in \Gamma_{sig(a)}\}$ is the set of discrete actions and $D = \mathbb{R}^{\geq 0}$ is the set of time-elapsing actions. Λ is partitioned into the sets $\Lambda^?$ of input actions, $\Lambda^!$ of output actions, and Λ^τ of internal actions;
- T is the transition relation defined as follows: (1) transitions with discrete actions are of the form $\langle l, \nu, \psi \rangle \overset{\langle a, \gamma \rangle}{\to} (l', \nu', \psi')$, where the system moves from $\langle l, \nu, \psi \rangle$ to $\langle l', \nu', \psi' \rangle$ through an action $\langle a, \gamma \rangle$ if there is a transition $t : \langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ such that G evaluates to $true$, $\nu' = A^D(\nu, \gamma)$, and $\psi' = A^C(\psi)$; (2) transitions with time-elapsing actions are of the form $(l, \nu, \psi) \overset{d}{\to} (l, \nu, \psi + d)$ for all $d \in D$ considering that the deadlines do not block time progress. Once the lazy deadline is used only to denote the absence of deadlines, lazy transitions cannot block time progress. A delayable transition can block time progress if there exist $0 \leq d_1 < d_2 \leq d$ such that $\psi + d_1 \models G^C$ and $\psi + d_2 \not\models G^C$, whereas an eager transition can block time progress if $\psi \models G^C$.

◇

Remark 3.1: As in [4], delayable transitions with guards of the form $\alpha < c$ are not allowed because there is no latest

time so that the guard is still true. Also, eager transitions with guards of the form $\alpha > c$ are not allowed because there is no earliest time so that the guard becomes true. Moreover, the deadline of transitions with internal actions is assumed to be *eager*. $\diamond$

Most notions and properties of TIOSTS are defined in terms of their underlying TIOLTS semantics (Definition 3.2). Then, consider $s, s', s_i \in S$; $\tau_i \in \Lambda^\tau$; $\omega, \omega_i \in Act$; and $a, a_i \in (Act \backslash \Lambda^\tau)$. Moreover, let $\rho \in Act^*$ be a sequence of discrete actions and time-elapsing actions, and $\sigma \in (Act \backslash \Lambda^\tau)^*$ be a sequence of visible discrete and time-elapsing actions. $\epsilon \in Act^*$ is the empty sequence. The sum of all delays spent in a sequence of actions ρ (respectively σ) is denoted by time(ρ) (respectively by time(σ)). For example, time(ϵ) $= 0$ and time($2.5\ a?\ 0.5\ x!$) $= 3.0$.

Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS whose semantics is defined by the TIOLTS $[\![W]\!] = \langle S, S^0, Act, T \rangle$. We write $s \xrightarrow{\omega} s'$ for $(s, w, s') \in T$, $s \xrightarrow{\omega}$ for $\exists s' : s \xrightarrow{\omega} s'$. Let $s \xrightarrow{\omega_1 \ldots \omega_n} s' \triangleq \exists s_0, ..., s_n : s = s_0 \xrightarrow{\omega_1} s_1 \xrightarrow{\omega_2} \cdots \xrightarrow{\omega_n} s_n = s'$ be an *execution*. We also write $s \xrightarrow{\rho}$ for $\exists s' : s \xrightarrow{\rho} s'$. *Traces*$(s) \triangleq \{\rho \in Act^* \mid s \xrightarrow{\rho}\}$ describes the set of sequences of discrete and time-elapsing actions fireable from s. The set of fireable actions from s is defined by $\Omega(s) \triangleq \{\omega \in Act \mid s \xrightarrow{\omega}\}$. $Out(s) \triangleq \Omega(s) \cap (\Lambda^! \cup D)$ is the set of all output events (including time-elapsing actions) fireable from s. The definition of $Out(s)$ can be extended for sets of states: for $P \subseteq S$ we have $Out(P) \triangleq \bigcup_{s \in P} Out(s)$.

The $\Rightarrow$ relation is used to denote the observable behaviour. Given $s, s' \in S$, $d \in \mathbb{R}^{\geq 0}$ and $a \in \Lambda^! \cup \Lambda^?$, we have $s \overset{d}{\Rightarrow} s'$ whenever $\exists \rho \in (\Lambda^\tau \cup D)^*$ such that $s \xrightarrow{\rho} s'$ and $time(\rho) = d$, whereas we have $s \overset{a}{\Rightarrow} s'$ whenever $\exists \rho_1, \rho_2 \in (\Lambda^\tau)^*$, $s_1, s_2 \in S$ such that $s \xrightarrow{\rho_1} s_1 \xrightarrow{a} s_2 \xrightarrow{\rho_2} s'$. Given $a_1, \cdots a_n \in (Act \setminus \Lambda^\tau)^*$, an *observable execution* is defined as $s \overset{a_1 \ldots a_n}{\Rightarrow} s' \triangleq \exists s_0, ..., s_n : s = s_0 \overset{a_1}{\Rightarrow} s_1 \overset{a_2}{\Rightarrow} \cdots \overset{a_n}{\Rightarrow} s_n = s'$. For $a \in Act \setminus \Lambda^\tau$ we also define $s \overset{a}{\Rightarrow} \triangleq \exists s' : s \overset{a}{\Rightarrow} s'$ and for $\sigma \in (Act \setminus \Lambda^\tau)^*$, $s \overset{\sigma}{\Rightarrow} \triangleq \exists s' : s \overset{\sigma}{\Rightarrow} s'$. *ObservableTraces*$(s) \triangleq \{\sigma \in (Act \setminus \Lambda^\tau)^* \mid s \overset{\sigma}{\Rightarrow}\}$ describes the set of sequences of observable and time-elapsing actions fireable from s. Finally, the set of sequences of observable behaviours fireable from the initial state of a TIOSTS W is defined by *ObservableTraces*$(W) \triangleq$ *ObservableTraces*(S_0).

The set s *after* $\sigma \triangleq \{s' \in S \mid s \overset{\sigma}{\Rightarrow} s'\}$ is the set of states reachable from s after the execution of σ, and P *after* $\sigma \triangleq \bigcup_{s \in P} s$ *after* σ is the set of states reachable from the set P after the execution of σ.

Subclasses of TIOSTS. Let $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ be a TIOSTS and $[\![W]\!] = \langle S, S^0, Act, T \rangle$ its associated TIOLTS. W is *complete* if it can accept any action at any state, i.e., $\forall s \in S, b \in \Lambda : s \xrightarrow{b}$. On the other hand, W is *input-complete* if it can accept any input action at any state, possibly after internal actions, i.e., $\forall s \in S, b \in \Lambda^? : s \overset{b}{\Rightarrow}$. W is said to be a *lazy-action* TIOSTS if the deadlines of all transitions are lazy. W is said to be *non-blocking* when it does not block time [4]: $\forall s \in S_0$ *after* ρ, $\forall d \in \mathbb{R}^{\geq 0}$, $\exists \rho' \in (\Lambda^! \cup \Lambda^\tau \cup D)^*$: time$(\rho') = d \wedge s \xrightarrow{\rho'}$. W is said to be *deterministic* if these conditions are satisfied [9], [11]: (1) $\Lambda^\tau = \emptyset$ (i.e. there is no internal actions); (2) $|S_0| = 1$, (3) for all $l \in L$ and for each pair of distinct transitions with origin in l carrying the same action a, i.e., $t_1 : \langle l, a, G_1, A_1, y_1, l'_1 \rangle$ and $t_2 : \langle l, a, G_2, A_2, y_2, l'_2 \rangle$, the guards G_1 and G_2 are mutually exclusive (i.e., $G_1 \wedge G_2$ is unsatisfiable).

C. Synchronous Product of TIOSTS

The synchronous product of two TIOSTSs W_1 and W_2 is an important operation used in both property oriented testing and conformance testing. This operation is used in the former for identifying behaviours of the specification accepted or rejected by a particular property (e.g., W_1 could be a specification and W_2 could be a test purpose). On the other hand, for conformance testing, this operation is used for modelling the synchronous execution of a test case on an implementation (e.g., W_1 is a test case and W_2 is a SUT). The synchronous product operation requires compatibility between W_1 and W_2, i.e., W_1 and W_2 must share the same sets of input and output actions from the same signature, with same set of parameters, and have no variables, internal actions, or clocks in common.

Definition 3.3 (Compatible TIOSTS): The TIOSTSs $W_i = \langle V_i, P_i, \Theta_i, L_i, l_i^0, \Sigma_i, C_i, \mathcal{T}_i \rangle$ $(i = 1, 2)$ are compatible if $V_1 \cap V_2 = \emptyset, P_1 = P_2, \Sigma_1^? = \Sigma_2^?, \Sigma_1^! = \Sigma_2^!, \Sigma_1^\tau \cap \Sigma_2^\tau = \emptyset$, and $C_1 \cap C_2 = \emptyset$. $\diamond$

Given the ordering $lazy < delayable < eager$ on deadlines and two deadlines y_1, y_2, $op(y_1, y_2) = (y_2$ if $y_1 < y_2$ and y_1 otherwise) is an operation which computes the resulting deadline in the synchronous product operation by keeping the most restrictive one.

Given two compatible TIOSTSs, Definition 3.4 formally describes the synchronous product between them.

Definition 3.4 (Synchronous Product): The synchronous product of two compatible TIOSTSs W_1 and W_2 is denoted by $SP = W_1 \parallel W_2$. SP is the TIOSTS $\langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ defined by: $V = V_1 \cup V_2, P = P_1 = P_2, \Theta = \Theta_1 \wedge \Theta_2, L = L_1 \times L_2, l^0 = \langle l_1^0, l_2^0 \rangle, \Sigma^? = \Sigma_1^? = \Sigma_2^?, \Sigma^! = \Sigma_1^! = \Sigma_2^!, \Sigma^\tau = \Sigma_1^\tau \cup \Sigma_2^\tau$, and $C = C_1 \cup C_2$. The set $\mathcal{T}$ is the smallest set such that:

1) For $a \in \Sigma_1^\tau$ and $l_2 \in L_2$: $\langle l_1, a, G_1, A_1, y_1, l'_1 \rangle \in \mathcal{T}_1 \Rightarrow \langle \langle l_1, l_2 \rangle, a, G_1, A_1, y_1, \langle l'_1, l_2 \rangle \rangle \in \mathcal{T}$;
2) For $a \in \Sigma_2^\tau$ and $l_1 \in L_1$: $\langle l_2, a, G_2, A_2, y_2, l'_2 \rangle \in \mathcal{T}_2 \Rightarrow \langle \langle l_1, l_2 \rangle, a, G_2, A_2, y_2, \langle l_1, l'_2 \rangle \rangle \in \mathcal{T}$;
3) For $a \in \Sigma^? \cup \Sigma^!$: $\langle l_1, a, G_1, A_1, y_1, l'_1 \rangle \in \mathcal{T}_1 \wedge \langle l_2, a, G_2, A_2, y_2, l'_2 \rangle \in \mathcal{T}_2 \Rightarrow$ $\langle \langle l_1, l_2 \rangle, a, G_1 \wedge G_2, A_1 \cup A_2, op(y_1, y_2), \langle l'_1, l'_2 \rangle \rangle \in \mathcal{T}$.

$\diamond$

Rules 1 and 2 say that the execution of internal actions can occur independently, while Rule 3 describes the synchronization of W_1 and W_2 through observable actions.

IV. Conformance Testing with TIOSTS

Conformance testing relates a specification with an implementation through a conformance relation, which is checked by the execution of test cases, possibly selected according to a test purpose [1]. These concepts are defined as follows.

The specification is a formal model of the SUT given as a non-blocking TIOSTS $\mathcal{S}$. We are considering specifications of software systems that do not force input actions, i.e., the system cannot block because an input action was not provided by the environment.

The implementation is a physical software system running on a real-time environment (e.g., a real-time operating system). In order to reason about conformance, it is assumed that its semantics can be modelled by a formal object. We assume here that it is modelled by a TIOLTS $\mathcal{I}$. The notions and properties of TIOLTSs are defined in [4]. Moreover, the implementation is assumed to be input-complete, non-blocking, and has the same interface (input and output actions with their signatures) as the specification $\mathcal{S}$. These assumptions are called *test hypotheses*.

A Test Case is used to check the conformance between the specification and its implementation. It is here defined as a TIOSTS TC as follows:

Definition 4.1 (Test Case): A test case is a deterministic, input-complete TIOSTS $TC = \langle V_{TC}, P_{TC}, \Theta_{TC}, L_{TC}, l^0_{TC}, \Sigma_{TC}, C_{TC}, \mathcal{T}_{TC} \rangle$, where $\Sigma^?_{TC} = \Sigma^!_{\mathcal{S}}$ and $\Sigma^!_{TC} = \Sigma^?_{\mathcal{S}}$ (actions are mirrored w.r.t. S), equipped with three disjoint sets of locations *Pass*, *Fail*, and *Inconclusive*. $\diamond$

Intuitively, when the location $Fail$ is reached, it means rejection, the location $Pass$ means that some targeted behaviour has been reached (this will be clarified later) and $Inconclusive$ means that targeted behaviours cannot be reached anymore.

Conformance Relation. We consider the conformance relation **tioco** defined by Krichen and Tripakis in [4]. Informally, an implementation conforms to a specification for **tioco** if and only if, after any trace of the specification, any output action (including time-elapsing actions) that the implementation provides after this trace is an output action that the specification may also provide.

Definition 4.2 (tioco): An implementation $\mathcal{I}$ conforms to a specification $\mathcal{S}$ for **tioco**, denoted by $\mathcal{I}$ **tioco** $\mathcal{S}$, iff $\forall \sigma \in$ *ObservableTraces*$(\mathcal{S})$, $Out(\mathcal{I} \text{ after } \sigma) \subseteq Out(\mathcal{S} \text{ after } \sigma)$. $\diamond$

V. Test Case Generation Process

The test generation process describes how test cases are derived from specifications according to the conformance relation. For simplicity, we assume that the specification $\mathcal{S}$ is deterministic and non-blocking. However, it is possible to deal with non-determinism, under some assumptions, for both data [22] and time [6]. We will consider the selection of test cases by test purposes. We introduce this notion here.

A test Purpose describes some desired behaviours that we wish to check on the implementation during the test campaign. They are used to select test cases in order to check specific scenarios. In our setting, a test purpose is a particular TIOSTS TP formally described as follows:

Definition 5.1 (Test Purpose): Given a specification TIOSTS $\mathcal{S}$ with action alphabet Σ, a test purpose is a deterministic, complete, lazy-action TIOSTS $TP = \langle V_{TP}, P_{TP}, \Theta_{TP}, L_{TP}, l^0_{TP}, \Sigma_{TP}, C_{TP}, \mathcal{T}_{TP} \rangle$, equipped with a special set of locations *Accept* $\subseteq L_{TP}$ such that all transitions leaving these locations are self-loops[2]. Moreover TP has to be compatible with $\mathcal{S}$ thus $\Sigma_{TP} = \Sigma$. $\diamond$

Complete test purposes are needed to ensure that the runs of a specification are not restricted before they are accepted (if ever). *Accept* locations are used to indicate that the expected scenario modelled by the test purpose has been fulfilled. Figure 2 presents an example of a test purpose for the withdrawal transaction system. It is used to select the scenarios where the user successfully performs a withdrawal transaction.

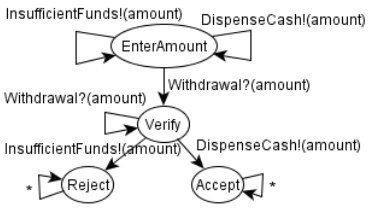

Figure 2. Test Purpose for the specification of Figure 1

The test case generation process starts with the specification $\mathcal{S}$ of the SUT $\mathcal{I}$ and a test purpose TP. The specification of $\mathcal{I}$ is combined with TP through the computation of the synchronous product (Definition 3.3). Figure 3 shows the synchronous products obtained from specification of Figure 1 and the test purpose of Figure 2. Then, the resulting TIOSTS model is symbolically executed to identify and select possible traces leading to an *Accept* location. Finally, the selected trace is translated into a test case.

The remainder of this section describes the following steps: symbolic execution, test case selection and translation.

A. Symbolic Execution

Symbolic execution is a technique for analysing programs based on symbolic values as input rather than concrete values [23], [24]. Symbolic execution techniques were used by Gaston *et al.* [25] and Jöbstl *et al.* [26] for test generation for untimed systems. We here extend the work proposed by Jöbstl *et al.* [26] to TIOSTS models to deal with time.

The main idea is to symbolically execute TIOSTS models using the same technique used for symbolically executing programs. Thus, all possible traces are identified using

[2]One can also consider another set of locations $Reject$ that can be used to discard all other scenarios where the system does not exhibit the desired behaviour.

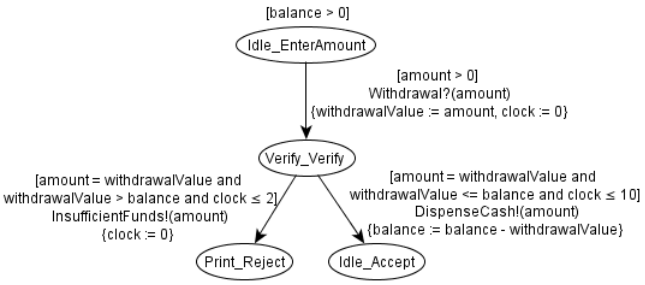

Figure 3. Synchronous Product Example

symbolic values instead of concrete values for action parameters and variables of the model, avoiding the state space explosion problem w.r.t. the data part since data values are not enumerated. The resulting traces are represented as a zone-based symbolic execution tree (Definition 5.4), whose nodes are zone-based symbolic extended states (Definition 5.2) and edges are symbolic actions (Definition 5.3).

Definition 5.2 (Zone-Based Symbolic Extended State): A zone-based symbolic extended state (ZSES) of a TIOSTS $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$ is a tuple $\eta = \langle l, \pi, \varphi, Z \rangle$, where: (1) $l \in L$ is a location of W; (2) π is a path condition, i.e. a Boolean expression representing a data guard; (3) φ is a mapping from variables and action parameters to their symbolic values; (4) Z is a zone representing the solution set of a clock constraint. $\diamond$

Symbolically executing a TIOSTS implies that data and time must be taken into account. As in [26], path conditions are checked using constraint solving. However, our definition of states differs from [26] because zones are used to check the reachability of states w.r.t. time requirements: a state is reachable if its path condition π is satisfiable and its zone Z is not empty. Zones provide an efficient symbolic representation of time requirements, avoiding the state space explosion problem w.r.t. time part. Furthermore, ZSESs are connected through transitions labelled by symbolic actions (Definition 5.3).

Definition 5.3 (Symbolic Action): A symbolic action is a tuple $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$, where: (1) $a \in \Sigma$ is the corresponding action in the TIOSTS; (2) μ_{sa} is a list of unique identifiers denoting the action parameters of sa; (3) φ_{sa} is a mapping from the original action parameter names to the unique identifiers in μ_{sa}. $\diamond$

We are now ready to define zone-based symbolic execution trees:

Definition 5.4 (Zone-Based Symbolic Execution Tree): A zone-based symbolic execution tree (ZSET) is a deterministic, connected graph with no cycles represented by a tuple $\langle S, \mathrm{SA}, \eta^0, T \rangle$, where: (1) S is a finite set of zone-based symbolic extended states; (2) SA is a finite set of symbolic actions; (3) $\eta^0 \in S$ is the initial zone-based symbolic extended state; (4) T is a finite set of transitions. Each transition $t \in T$ is a tuple $\langle \eta, sa, \eta' \rangle$, where: (4.1) $\eta \in S$ is the origin state, (4.2) $sa \in \mathrm{SA}$ is the symbolic action, (4.3) $\eta' \in S$ is the destination state. $\diamond$

A ZSET is deterministic if $\forall \eta, \eta', \eta'' \in S,\ \forall sa \in \mathrm{SA} : \langle \eta, sa, \eta' \rangle \in T \wedge \langle \eta, sa, \eta'' \rangle \in T \Rightarrow \eta' = \eta''$.

Algorithms for symbolically executing symbolic transition systems have been proposed by Gaston *et al.* [25] and Jöbstl *et al.* [26]. However, as they do not deal with time, a new algorithm is presented in this work. Algorithm 1 is an extended version of the one proposed by Jöbstl *et al.* [26]. It requires two parameters: TIOSTS W is the model to be symbolically executed and *ZSET* is the resulting zone-based symbolic execution tree. Firstly, a unique symbolic value is generated for each variable of V and each action parameter of P (Line 2). In Line 3, the first state η^0 of *ZSET* is defined considering the initial location of W, the initial condition of W as first path condition, the mapping defined in Line 2, and the initial clock zone (i.e., all clocks set to zero). Once defined, the first state η^0 is added to *ZSET* (Line 4).

The state η^0 is added to the set of states to be visited (Line 5). As long as there are unvisited states (Line 6), the algorithm picks and remove some state η from *Unvisited* (Line 7). The ZSES η refers to a location l of W and the loop in Line 8 processes all transitions from l.

The symbolic action sa is computed from the action a, attributing unique symbolic values for every parameter of a (Line 9) and mapping the original action parameter names to the defined symbolic values (Line 10). Once the symbolic action has been defined (Line 11), the target state η' is computed in the next step. Thus, the path condition π' for η' is defined (Line 12) as the conjunction of π with the guard

Algorithm 1. Symbolic Execution of $W = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$

```
1  symbolicExecution(TIOSTS W, ZSET ZSET){
2    φ⁰ ← map of variables of V ∪ P to symbolic values
3    η⁰ ← ⟨l⁰, Θ, φ⁰, Z⁰⟩
4    addState(ZSET, η⁰)
5    Unvisited ← {η⁰}
6    while Unvisited ≠ ∅ do
7      pick and remove some η = ⟨l, π, φ, Z⟩ from Unvisited
8      for all ⟨l, a, G, A, y, l'⟩ ∈ T do
9        μ_sa ← list of unique symb. values for every param. of a
10       φ_sa ← map of action parameters to symbolic values
11       sa ← ⟨a, μ_sa, φ_sa⟩
12       π' ← π ∧ φ(φ_sa(G^D))
13       φ' ← φ ∘ φ_sa ∘ A^D
14       Z' ← [A^C ← 0](G^C ∩ Z⃗)
15       η' = ⟨l', π', φ', Z'⟩
16       if (isReachable(η') ∧ ¬(upperBoundReached(l')) ∧
                                 η' ⋢ η''  ∀η'' ∈ ZSET) then
17         Unvisited ← Unvisited ∪ {η'}
18         addState(ZSET, η')
19         addTransition(ZSET, ⟨η, sa, η'⟩)
20       end if
21     end for
22   end while
23 }
```

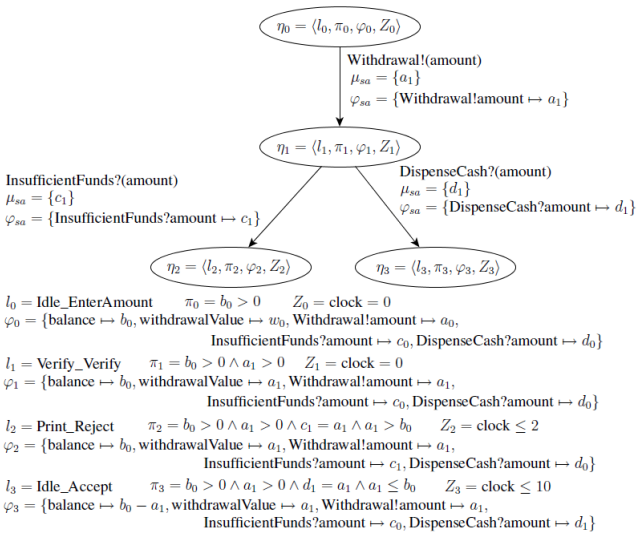

Figure 4. Zone-Based Symb. Execution Tree of the TIOSTS of Figure 3

G^D (i.e., the data guard of G) considering the mappings φ and φ_{sa}. The mapping φ' is defined through $\varphi \circ \varphi_{sa} \circ A^D$ (Line 13), where A^D represents data assignments of A and $\circ$ denotes function composition.

Z' is defined in Line 14. The successor of Z is defined by letting time elapse ($\vec{Z}$), taking the intersection with the clock guard G^C, and finally updating the values of clocks that are reset (i.e., clocks in A^C).

Once π', φ', and Z' have been defined, the target state η' is created in Line 15. Finally, η' is added to the set of states to be visited (Line 17) and a new transition labelled by sa connecting η to η' is added to *ZSET* (Lines 18 and 19), if the following conditions are satisfied (Line 16): (1) The state η' is reachable, that is, the path condition π' is satisfiable and the zone Z' is not empty; (2) The number of ZSESs in the current path that correspond to the location l' does not exceed a certain bound. This checking is needed to avoid infinite ZSETs in the case where there are loops in the specification whose number of iterations depends on values assigned to parameters and variables [26]; (3) $\eta' \not\subseteq \eta'' \quad \forall \eta'' \in$ *ZSET* according to Definition 5.5, where the state inclusion of Gaston *et al.* [25] was extended to deal with zones. Figure 4 presents the ZSET obtained from the symbolic execution of TIOSTS of Figure 3.

Definition 5.5 (ZSES comparison): Let $\eta = \langle l, \pi, \varphi, Z \rangle$ and $\eta' = \langle l', \pi', \varphi', Z' \rangle$ be two zone-based symbolic extended states. ZSES η' is included in ZSES η, that is, $\eta' \subseteq \eta$, if and only if: (1) $l' = l$; (2) $(\pi' \wedge \bigwedge_{x \in A^D}(x = \varphi'(x))) \Rightarrow (\pi \wedge \bigwedge_{x \in A^D}(x = \varphi(x)))$ is a tautology, where A^D represents data assignments of the TIOSTS; (3) $Z' \subseteq Z$. $\diamond$

B. Test Case Selection

Once all possible traces have been identified by symbolic execution, the next step is to select a test case by choosing a trace that leads to an *Accept* state. For this, it is necessary to select a subtree of the generated ZSET called test tree.

The strategy used for the selection of the test tree is the same proposed by Jöbstl *et al.* [26], which is similar to the strategy of the TGV tool [27]. The idea is to select one reachable *Accept* state and perform a backward traversal to the root ZSES. Finally, a forward traversal is performed in order to extend the selected path to a test tree by adding missing inputs that are allowed by the specification. These missing inputs are possible outputs of the SUT and they are important to avoid fail verdicts on outputs allowed by the specification. In this case, the verdict is **Inconclusive**. Note that the forward traversal ensures the controllability of the generated test tree (i.e. test cases do not have the choice between inputs and outputs, or between several outputs).

The test tree from the ZSET in Figure 4 is the same ZSET since there is only one path leading to an *Accept* state and the addition of missing inputs leads to the whole ZSET.

C. Test Tree Transformation

The last step of the test case generation process consists in translating the selected test tree $TT = \langle S, \text{SA}, \eta^0, T \rangle$ into a test case TIOSTS $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T} \rangle$.

The data of *TC* (i.e. $V \cup P$) is defined by symbolic values of *ZSET*. As in [26], the symbolic values are considered as variables and parameters of the test case. Let $\eta^0 = \langle l^0, \pi^0, \varphi^0, Z^0 \rangle$ be the initial state of *ZSET*, then the initial condition of *TC* is π^0, the set of locations is S, the initial location is η^0, the alphabet is $\bigcup_{\langle a, \mu_{sa}, \varphi_{sa} \rangle \in SA} a$, and the set of clocks is the same as the synchronous product SP, that is, C_{SP}.

All transitions $\langle \eta, sa, \eta' \rangle \in T$ of *ZSET* are analysed. Each transition of *ZSET* leads to the creation of a new transition $\langle l, a, G, A, y, l' \rangle \in \mathcal{T}$ in the test case. Thus, the source location is η, the action of the new transition is the action of $sa = \langle a, \mu_{sa}, \varphi_{sa} \rangle$ with parameters of μ_{sa}, the conjunction of the path condition of η' with clock guards associated with a in *SP* is the guard, the assignments are defined based on clock resets associated with a in *SP*, the deadline is the same as the one associated with a in *SP*, and the target location is η'.

Figure 5 presents the test case obtained from the ZSET of Figure 4. It starts by performing a withdrawal transaction and resetting the clock to zero. Then it expects to receive the money. If the expected money is dispensed in at most 10 time units, the verdict is **Pass**, i.e., the implementation is in conformance with the specification and the test purpose. If the ATM system indicates insufficient funds in at most 2 time units, the verdict is **Inconclusive** (i.e. the implementation conforms to the specification but the desired behaviour was not observed). Finally, if either an unspecified input is observed or a time requirement is not met, the verdict is **Fail**.

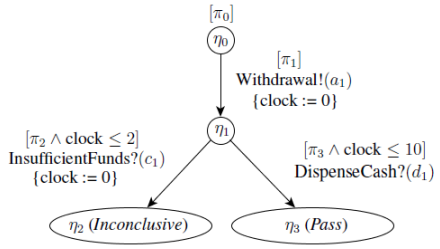

Figure 5. Test Case Obtained from the ZSET of Figure 4

VI. Properties of the Test Cases

This section comments on properties of the test cases generated by the process presented in Section V. The generated test cases are considered as a mechanism for guiding the execution of the implementation. Thus, the conformance checking is performed in an offline way. Firstly, the implementation is executed, guided by test cases, and all information needed to check the conformance (e.g., input actions, responses, and time) are logged into a file. This is important in real-time environments to reduce the number of processes and consequently avoid introduction of noise in the results.

The logged information is an *observable trace* (defined in Subsection III-B). For the ATM system example, an observable trace of a scenario where a withdrawal transaction is successfully done in 5 time units could be represented by $\sigma = 0$ *Withdrawal*?(100) 5 *DispenseCash*!(100). Moreover, it is not difficult to see that σ is a TIOLTS.

Let $[\![TC]\!] = \langle S, S^0, Act, T\rangle$ be the TIOLTS semantics of the test case $TC = \langle V, P, \Theta, L, l^0, \Sigma, C, \mathcal{T}\rangle$. Thus, an observable trace of $\mathcal{I}$ can be checked with respect to the test case through the TIOLTS parallel composition defined by Krichen and Tripakis [4]. In this case, each trace $\sigma \in \mathit{Traces}([\![TC]\!] \;||\; \mathit{ObservableTraces}(\mathcal{I}))$ is associated with one of the following scenarios: (1) if all outputs of TC are executed and all inputs are observed on time, then the resulting verdict is **Pass**, that is, $\mathit{verdict}(\sigma) = \textbf{Pass} \triangleq S^0 \textit{ after } \sigma \subseteq \mathit{Pass}$; (2) if, at any moment, any unspecified input is observed by the test case or some time requirement is not met, the conformance checking is stopped and the resulting verdict is **Fail**, that is, $\mathit{verdict}(\sigma) = \textbf{Fail} \triangleq S^0 \textit{ after } \sigma \subseteq \mathit{Fail}$; (3) we denote $\mathit{verdict}(\sigma) = \textbf{Inconclusive} \triangleq S^0 \textit{ after } \sigma \subseteq \mathit{Inconclusive}$ for two situations: if $\mathcal{I}$, at any moment, blocks or spends a lot of time to emit an output; and if the outputs of $\mathcal{I}$ are specified by $\mathcal{S}$ but the behaviour specified by a test purpose is not exhibited.

Given these scenarios, the rejection of $\mathcal{I}$ by a test case TC is formally defined as follows:

Definition 6.1 (may reject): TC *may reject* $\mathcal{I} \triangleq \exists \sigma \in \mathit{Traces}([\![TC]\!] \;||\; \mathit{ObservableTraces}(\mathcal{I})) : \mathit{verdict}(\sigma) = \textbf{Fail}$. $\diamond$

The following definition formally relates **tioco** to the verdicts considering some properties of test cases and test suites.

Definition 6.2 (Soundness and Exhaustiveness): A test case TC is *sound* for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}, \mathcal{I}$ **tioco** $\mathcal{S} \Rightarrow \neg(TC$ *may reject* $\mathcal{I})$. A test suite is *sound* if all its test cases are *sound* and it is *exhaustive* for $\mathcal{S}$ and **tioco** if $\forall \mathcal{I}$, $\neg(\mathcal{I}$ **tioco** $\mathcal{S}) \Rightarrow \exists TC : TC$ *may reject* $\mathcal{I}$. Finally, a test suite is *complete* if it is both *sound* and *exhaustive*. $\diamond$

Informally, a test suite is *sound* whether correct implementations are never rejected. On the other hand, a test suite is *exhaustive* if all non-conforming implementations are rejected. A test suite that can identify all conforming and non-conforming implementations is called complete. Since a complete test suite is a very strong requirement for practical testing, sound test suites are more commonly accepted. In this context, the test cases generated by our approach have the properties stated in Theorem 6.1.

Theorem 6.1: For every specification $\mathcal{S}$, all test suites generated by our approach are sound. Moreover, test generation can be considered as being exhaustive in the following sense: for each non-conformant implementation, one can design a test purpose such that a test case generated from this test purpose may reject the implementation. $\diamond$

The proofs of Theorem 6.1 are only discussed here for the sake of space. For soundness, we need to prove that if a test case TC may reject $\mathcal{I}$ (implementing the specification $\mathcal{S}$), then $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$. In this case, we only need to prove that a **Fail** verdict only occur if $\mathcal{I}$ emits an unspecified output or some time requirement is not met. In our approach, test cases are generated based on symbolic execution of specifications. This approach allows to identify all possible traces of a specification. Thus, the unique case where a **Fail** verdict occurs is exactly when $\mathcal{I}$ emits an unexpected output or some time requirements is not satisfied. For exhaustiveness, we need to prove that for every non-conforming $\mathcal{I}$ there is a test purpose TP and a way of generating a test case TC from $\mathcal{S}$ and TP, such that TC may reject $\mathcal{I}$. Given that $\neg(\mathcal{I}$ **tioco** $\mathcal{S})$, then there is a trace σ of $\mathcal{S}$ such that an output of $\mathcal{I}$ after σ is not allowed by $\mathcal{S}$. In this case, a TP can be defined based on σ and used to generate test cases where $\mathcal{I}$ may be rejected.

VII. Concluding Remarks

This paper presents an approach to conformance testing of real-time systems based on the use of a symbolic model that abstracts both time and data in order to broadening the application of conformance testing in this field. It also introduces a conformance testing theory to deal with the model proposed and describes how test cases can be generated.

A tool is being developed to support the proposed approach. In order to check the satisfiability of path conditions and verify state inclusion w.r.t. data we are using the CVC3

SMT Solver[3]. In this case, our approach is limited to the types supported by this solver such as Boolean, integer, real, arrays, records, etc. All operations related to zones used in Algorithm 1 are provided by UPPAAL DBM Library[4]. The same library is used to verify the state inclusion w.r.t. zones.

Furthermore, a test architecture based on the FreeRTOS environment is being developed [16], [17]. As future work, we plan to integrate the test case generation tool with this test architecture in order to provide a complete environment to generate and execute test cases.

ACKNOWLEDGMENT

This work is part of an international cooperation supported by the INRIA (Equipe Associée TReaTiES). This work was supported by the National Institute of Science and Technology for Software Engineering (INES[5]), funded by CNPq, grant 573964/2008-4.

REFERENCES

[1] J. Tretmans, "Testing concurrent systems: A formal approach," in *CONCUR'99: Proc. of the 10th Int. Conf. on Concurrency Theory*. Springer, 1999, pp. 46–65.

[2] A. Hessel, K. G. Larsen, B. Nielsen, P. Pettersson, and A. Skou, "Time-optimal real-time test case generation using UPPAAL," in *FATES'03*, ser. LNCS. Springer, 2004, vol. 2931, pp. 114–130.

[3] K. Larsen, M. Mikucionis, and B. Nielsen, "Online testing of real-time systems using UPPAAL," in *FATES'04*, ser. LNCS, vol. 3395. Springer, 2005, pp. 79–94.

[4] M. Krichen and S. Tripakis, "Conformance testing for real-time systems," *Form. Methods Syst. Des.*, vol. 34, no. 3, pp. 238–304, 2009.

[5] S. Li, J. Wang, W. Dong, and Z.-C. Qi, "Property-oriented testing of real-time systems," in *APSEC'04: Proc. of the 11th Asia-Pacific Software Engineering Conference*. IEEE Computer Society, 2004, pp. 358–365.

[6] N. Bertrand, T. Jéron, A. Stainer, and M. Krichen, "Off-line test selection with test purposes for non-deterministic timed automata," in *TACAS 2011*, march 2011, to appear.

[7] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed wp-method: Testing real-time systems," *IEEE Trans. Softw. Eng.*, vol. 28, no. 11, pp. 1023–1038, 2002.

[8] M. Merayo, M. Núñez, and I. Rodríguez, "Extending EFSMs to specify and test timed systems with action durations and time-outs," *IEEE Trans. Comput.*, vol. 57, no. 6, pp. 835–844, 2008.

[9] V. Rusu, L. du Bousquet, and T. Jéron, "An approach to symbolic test generation," in *IFM'00: Proc. of the Second Int. Conf. on Integrated Formal Methods*. Springer, 2000, pp. 338–357.

[10] G. Lestiennes and M.-C. Gaudel, "Testing processes from formal specifications with inputs, outputs and data types," in *ISSRE'02*. IEEE Computer Society, 2002, p. 3.

[11] B. Jeannet, T. Jéron, V. Rusu, and E. Zinovieva, "Symbolic test selection based on approximate analysis," in *TACAS'05*, ser. LNCS, vol. 3440, 2005, pp. 349–364.

[12] L. Frantzen, J. Tretmans, and T. Willemse, "A Symbolic Framework for Model-Based Testing," in *FATES/RV 2006*, ser. LNCS, no. 4262. Springer, 2006, pp. 40–54.

[13] R. Cardell-Oliver, "Conformance tests for real-time systems with timed automata specifications," *Formal Aspects of Computing*, vol. 12, no. 5, pp. 350–371, Dec. 2000.

[14] K. G. Larsen, P. Pettersson, and W. Yi, "UPPAAL in a nutshell," *Int. J. Softw. Tools Technol. Transfer*, vol. 1, no. 1, pp. 134–152, 1997.

[15] A. Khoumsi, "Complete test graph synthesis for symbolic real-time systems," *ENTCS*, vol. 130, pp. 79–100, 2005.

[16] W. L. Andrade, P. D. L. Machado, E. L. G. Alves, and D. R. Almeida, "Test case generation of embedded real-time systems with interruptions for freertos," in *Formal Methods: Foundations and Applications*, ser. LNCS, vol. 5902. Springer, 2009, pp. 54–69.

[17] A. Q. Macedo, W. L. Andrade, D. R. Almeida, and P. D. L. Machado, "Automating test case execution for real-time embedded systems," in *ICTSS'10*, 2010, pp. 37–42, Short Paper.

[18] The FreeRTOS.org Project, "FreeRTOS," http://www.freertos.org.

[19] O. N. Timo, H. Marchand, and A. Rollet, "Automatic test generation for data-flow reactive systems with time constraints," in *ICTSS'10*, 2010, pp. 25–30, Short Paper.

[20] R. Alur and D. L. Dill, "A theory of timed automata," *Theor. Comput. Sci.*, vol. 126, no. 2, pp. 183–235, 1994.

[21] S. Bornot, J. Sifakis, and S. Tripakis, "Modeling urgency in timed systems," in *Compositionality: The Significant Difference*, ser. LNCS. Springer, 1998, vol. 1536, pp. 264–279.

[22] T. Jéron, H. Marchand, and V. Rusu, "Symbolic determinisation of extended automata," in *Proc. of the 4th IFIP Int. Conf. on Theor. Comput. Sci.*, ser. IFIP. Springer, 2006, vol. 209, pp. 197–212.

[23] J. C. King, "A new approach to program testing," in *Proc. of the Int. Conf. on Reliable software*. ACM, 1975, pp. 228–233.

[24] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Trans. Softw. Eng.*, vol. 2, no. 3, pp. 215–222, 1976.

[25] C. Gaston, P. Le Gall, N. Rapin, and A. Touil, "Symbolic execution techniques for test purpose definition," in *Testing of Communicating Systems*, ser. LNCS. Springer, 2006, vol. 3964, pp. 1–18.

[26] E. Jöbstl, M. Weiglhofer, B. K. Aichernig, and F. Wotawa, "When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving," in *ICST'10*. IEEE Computer Society, 2010, pp. 479–488.

[27] C. Jard and T. Jéron, "TGV: theory, principles and algorithms: A tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems," *Int. J. Softw. Tools Technol. Transfer*, vol. 7, no. 4, pp. 297–315, 2005.

[3] http://www.cs.nyu.edu/acsys/cvc3

[4] http://www.cs.aau.dk/~adavid/UDBM

[5] http://www.ines.org.br