Software and Systems Modeling manuscript No.
(will be inserted by the editor)

Testing timed systems modeled by Stream X-Machines*

Mercedes G. Merayo[1], Manuel Núñez[1], Robert M. Hierons[2]

[1] Universidad Complutense de Madrid, Spain.
[2] Brunel University, United Kingdom.

The date of receipt and acceptance will be inserted by the editor

Abstract Stream X-machines have been used to specify real systems requiring to represent complex data structures. They are a kind of extended finite state machine using a shared memory to specify communications between the components of systems. In this paper we introduce an extension of the Stream X-machines formalism in order to specify systems that present temporal requirements. On the one hand, we consider that (output) actions take time to be performed. On the other hand, our formalism allows to specify timeouts. Timeouts represent the time a system can wait for the environment to react. Since timeous affect the set of available actions of the system, a relation focusing on the functional behavior of systems, that is, the actions that they can perform, must explicitly take into account the possible timeouts. We also propose a formal testing methodology allowing to systematically test a system with respect to a specification. Finally, we introduce a test derivation algorithm. Given a specification, the derived test suite is sound and complete, that is, a system under test successfully passes the test suite if and only if this system conforms to the specification.

Key words Formal testing – Timed systems – Stream X-machines

1 Introduction

Formal methods allow the analysis of systems and the reasoning about them with mathematical precision and rigor. It is widely recognized that formal methods and testing are complementary techniques (see, for example, [6,31,39,8,14,44,20,48,19]), since their combined use increases the confidence on the correctness of the developed systems. In this context, *formal testing techniques* provide systematic procedures to check systems under test in such a way that the coverage of critical parts/aspects depends less on the intuition of the tester.

Real-time systems are a class of systems subject to timing constraints that are considered critical. Thus, the necessity to ensure the correctness of these systems forces to apply a methodology which considers time requirements. In order to perform this task, several techniques, algorithms, and semantic frameworks have been introduced in the literature. In particular, the testing community has shown a growing interest to take into account time aspects (e.g. [32,12,24,50,47,15,30,7,18, 36,35,45,49]).

The application of formal methods requires to provide languages to formally specify these systems. Even though there exists a myriad of timed extensions of the classical frameworks [46,43,51,37,42,1,17,13,2], most of them specialize only on the time that a system consumes performing operations. However, another kind of time constraint can affect systems: *Timeouts*. A timeout is a specified period of time that will be allowed to elapse in a system waiting for an interaction. If the period ends and no action has been received, then the internal state of the system changes and its reactions to the actions that are received from the environment may be different.

The previous considerations led us to propose a formal testing framework for systems where timeouts are critical [33]. In this work, we introduced an extension of the Stream X-machines formalism in order to specify timeouts. As usual, we assumed that specifications and implementations can be modeled by means of the same formalism, and proposed a formal testing methodology to systematically test a system with respect to a specification. Testing a system requires using a specific notion of correctness. In our framework, we must explicitly take into account the possible timeouts when we establish what it means that an implementation conforms to a specification. That is due to the fact that timeouts can influence the functional behavior of a system, that is, they influence the actions that the system

* This paper represents an extended and improved version of [33]. Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01)

can perform at a given point of time. In addition, we introduced a notion of test that can delay the execution of the implementation and also presented an algorithm to derive sound and complete test suites with respect to the defined implementation relation.

Although a testing methodology that allows to design and check systems with timeouts is very useful, temporal requirements of real-time systems usually involve restrictions over the time that the system spends performing operations. The combination of both kinds of temporal aspects, timeouts and time associated with the performance of actions, in a unique temporal formalism should allow the specifier to define the behavior of systems affected by both kinds of timed restrictions.

This paper represents an extension and continuation of the work initiated in [33] where only timeouts were considered. The inclusion of time associated with the performance of actions requires a suitable extension of the formalism introduced in our previous work. While the definition of the new language is not difficult, mixing both kinds of temporal issues complicates the posterior theoretical analysis. In particular, the definition of timed conformance relations is more involved. These relations still follow the standard pattern: A system is correct with respect to a specification if it does not show any behavior that is forbidden by the specification. However, in addition to *functional* behaviors, that is, the actions that the system can perform, we have to take into account *temporal* behaviors, that is, when these actions can be performed. Moreover, we have to implicitly consider how these two kinds of behaviors affect each other. In fact, the time spent by a system waiting for the environment to react has the capability of affecting the set of available actions of the system. This is so because the passing of time may trigger a change of the internal state of the system due to a timeout. Regarding temporal requirements, our testing methodology will take into account that the system is only responsible for the consumed time while performing actions, not the time that the system waits for a reaction from the environment (as we just indicated, this time has to be considered when evaluating the functional behavior of the system).

In this paper we assume that specifications and implementations can be modeled by means of our extension of Stream X-machines with temporal constraints, and propose a formal testing methodology to systematically test a system with respect to a specification. We introduce a notion of test that includes temporal information and how to test systems that can be represented by using our formalism. We also provide an algorithm to derive test suites from specifications. The main theoretical result of our paper indicates that these test suites have the same distinguishing power as the presented conformance relations. In other words, an implementation successfully passes a test suite iff it is conforming to the specification. Finally, let us remark that in order to have a more expressive formalism, we remove most of the restrictions that are usually assumed in Stream X-machines. For example, we do not impose a bound on the number of states of the system under test. This is required to apply the standard test generation algorithm based on the W-method [11]. More importantly, we do not require that systems have the property of output-distinguishability, requiring that any two different functions cannot generate the same output for a given input and memory value. That is, we allow that systems present non-deterministic behavior (in [33] we restricted ourselves to deterministic behaviors).

The rest of the paper is organized as follows. In order to place our work in context, in the next section we briefly review related work, with an emphasis on the three major research lines to (formally) test timed systems. In Section 3 we introduce our model to represent Stream X-machines with temporal requirements. In Section 4 we introduce the notion of functional and temporal conformance for our framework. In Section 5 we show how our machines can be tested. In Section 6 we give a test derivation algorithm and we prove that the derived test suite is sound and complete. Finally, in Section 7 we present our conclusions.

2 Related Work

We briefly enumerate the main contributions in the field of formal timed testing and classify them in three main categories. First, let us note that our language is based on Stream X-machines, which have been extensively used by the formal testing community. The main advantages of this model are its flexibility and its capabilities to integrate control and data processing. This formalism has been used to specify systems in different areas (e.g. [25, 3, 29, 28]) but, to the best of our knowledge, only our previous work [34, 33] used this formalism to design and test systems that present temporal restrictions. The work [**?**] introduced the notion of time to the X-machine formalism. Nevertheless, the authors only propose a way for representing action duration, neither timeouts or a testing framework are considered.

The standard Stream X-machines test generation algorithm is based on the W-method, introduced by [11] in the context of finite state machines. This method presents some restrictions:

- Specifications and implementations must be deterministic.
- The functions associated with the transitions are supposed to be correctly implemented.
- A number of additional conditions, called *design for test conditions*, hold.

Under these assumptions the set of tests generated by this method is guaranteed to determine the correctness of the system [27, 25]. In further optimizations, this method has been modified to reduce the imposed conditions (e.g. [21, 5, 26, 22]). As we have already explained,

due to the fact that we remove most of the restrictions on machines, we cannot apply the previously mentioned methodologies to our extension of Stream X-machines.

Even though our way to deal with timeouts is completely different to that in *timed automata* [1], it is worth to mention that our notion of timeout is related to having invariants associated with states of a timed automata as described in [47]: Once the value of a clock variable exceeds the invariant, the system produces a prescribed output and enters a new state. In the formalism studied in this paper, we also associate time with the performance of actions.

Regarding testing of temporal requirements, as we mentioned before, there exist already numerous proposals in the literature. The more classical proposals are usually based on timed automata, although recent proposals tend to consider other formalisms such as temporal extensions of labeled transition systems and of finite state machines (e.g. [4,35,36,45,49]). Due to the intrinsical difficulty behind testing timed systems, different approaches to the problem have been exercised, falling into one or more of the following categories.

- Only some behaviors, out of those that are relevant for the correctness of the tested system, are exercised (e.g. [16,9]). In these cases, methods to choose those tests that seem to have a higher capability to find errors are proposed, though they are usually heuristic or are based on restricting the behavior to be tested to some specific *test purposes*.
- A *complete finite* test suite is derived from the specification, that is, if all tests of the finite suite are passed, then the tested system is correct (e.g. [10,47]). Usually, the finiteness of this suite requires to introduce strong assumptions about the system, both to deal with functional requirements (e.g. to assume that the maximal number of states in the implementation is known) and time requirements (e.g. urgency of outputs, discretization of time). In general, the applicability of the derived test suite is not feasible because the number of derived tests is astronomic.
- A complete *infinite* test suite is extracted from the specification (e.g. [32,38,7,36,35]). In this approach, a test derivation algorithm is defined in such a way that, for all system behavior that must be tested before granting correctness, a suitable test is added by the algorithm to the test suite. In this sense, such an algorithm is complete (that is, it provides full fault coverage with respect to the considered testing relation) *in the limit*.

The interest of the work falling in the last category is that, on the one hand, weaker assumptions are required in these methodologies and, on the other hand, it is necessary to have a method to find and construct any required test if we want to *select* some of these tests according to some criteria. That is, methods that are exhaustive in the limit are the basis for other non-exhaustive but more practical methods. The methodology presented in this paper fits into this last category and, consequently, its aim is to provide test suites that are complete in the limit while, in turn, no strong assumptions are required (e.g. about the number of states of the implementation).

3 A Timed Extension of the Stream X-Machine Formalism

In this section we introduce our extension of the classical Stream X-machines model in order to deal with systems that present temporal requirements. We will add new features so that the timed behavior of a system can be properly specified. On the one hand, we consider that output actions take time to be executed. On the other hand, we also consider that a machine can evolve by raising *timeouts*. Intuitively, if after a given amount of time, and depending on the current state, we do not receive an input action, then the machine will change its current state. Thus, we need to add new features to the formalism so that this kind of time constraints can be properly specified. We denote by `Time` the domain to define time values. Let us remark that the theory presented in this paper is independent of the `Time` set since we only require that this is a totally ordered set. During the rest of the paper we will use the following notation.

Definition 1 Let $(\texttt{Time}, \leq)$ be a totally ordered set such that $\infty \notin \texttt{Time}$. We assume that for all $a \in \texttt{Time}$ we have $a < \infty$. A tuple of elements $(a_1, a_2 \ldots, a_n)$, with $a_i \in \texttt{Time}$, will be denoted by $\bar{a}$. We say that $\hat{a} = [a_1, a_2)$ is an interval if $a_1, a_2 \in \texttt{Time} \cup \{\infty\}$ and $a_1 \leq a_2$. A tuple of intervals $(\hat{p_1}, \ldots, \hat{p_n})$ will be denoted by $\breve{p}$. Let $\bar{t} = (t_1, \ldots, t_n)$, $\bar{t}' = (t'_1, \ldots, t'_n)$, and $\breve{q} = (\hat{q_1}, \ldots, \hat{q_n})$. We write

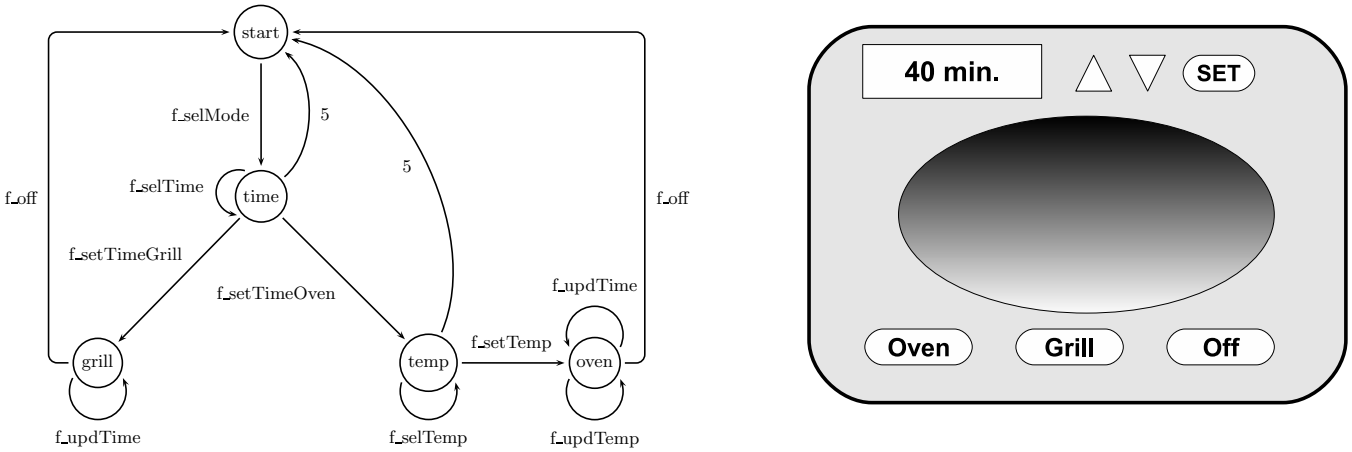
- $\bar{t} \in \breve{q}$ if for all $1 \leq j \leq n$ we have $t_j \in \hat{q_j}$;
- $\bar{t} = \bar{t}'$ if for all $1 \leq j \leq n$ we have $t_j = t'_j$;
- $\bar{t} \leq \bar{t}'$ if for all $1 \leq j \leq n$ we have $t_j \leq t'_j$.

In addition, we introduce the following notation:

- $\sum \bar{t}$ denotes the sum of the elements belonging to the tuple $\bar{t}$, that is, $\sum_{j=1}^{n} t_j$;
- $\pi_i(\bar{t})$, for all $1 \leq i \leq n$, denotes the value t_i.

□

Definition 2 A *Timed Stream X-machine*, in short `TSXM`, is a tuple $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ where I is the set of *input* actions, O is the set of *output* actions, S is a finite set of *states*, M is the *memory*, Φ, called the *type* of X, is a finite set of partial functions, ranging each of them from $M \times I$ to $O \times M \times \texttt{Time}$ called timed processing functions, $F : S \times \Phi \longrightarrow S$ is the *next state function*, $TO : S \longrightarrow S \times (\texttt{Time} \cup \infty)$ is the *timeout*



I = {bgrill,boven,boff,bset,bup,bdown,tickTime,tickTemp}

O = {display(s)| s ∈ Message},
where Message={$25 \cdot n$ ++ "^{o} C"| $0 \leq n \leq 10$} ∪ {$18 \cdot n$ ++ "min."| $0 \leq n \leq 10$}∪
{"Select Time Oven","Select Time Grill", "Select Temperature Oven","Grilling","Cooking","Stop"}

M = Mode×TempMax×TempSel×TempAct×TimeMax×TimeSel×TimeAct
where Mode={OVEN,GRILL}, TempMax={250}, TempSel={$25 \cdot n \mid 0 \leq n \leq 10$}, TempAct={$n \in \mathbb{N} \mid 0 \leq n \leq 250$},
TimeMax={180}, TimeSel={$10 \cdot n \mid 0 \leq n \leq 18$}, TimeAct={$n \in \mathbb{N} \mid 0 \leq n \leq 180$}

m_{in}=(null,250,0,0,180,0,0)

f_selMode(M,boven)	=	(display("select Time Oven"),M[Mode]=OVEN,1)
f_selMode(M,bgrill)	=	(display("select Time Grill"),M[Mode]=GRILL,1)
f_selTime(M[TimeSel]+10<=M[TimeMax],bup)	=	(display(M[TimeSel] ++ "min."),M[TimeSel]=M[TimeSel]+10,1)
f_selTime(M[TimeSel]-10>=0,bdown)	=	(display(M[TimeSel] ++ "min."),M[TimeSel]=M[TimeSel]-10,1)
f_setTimeGrill (M[Mode]=GRILL,bset)	=	(display("Grilling"),M,2)
f_setTimeOven (M[Mode]=OVEN,bset)	=	(display("select Temperature Oven"),M,2)
f_selTemp(M[TempSel]+10<=M[TempMax],bup)	=	(display(M[TempSel] ++ " ^{o}C "),M[TempSel]=M[TempSel]+25,2)
f_selTemp(M[TempSel]-10>=0,bdown)	=	(display(M[TempSel] ++ " ^{o}C "),M[TempSel]=M[TempSel]-25,2)
f_setTemp(M[Mode]=OVEN,bset)	=	(display("Cooking"),M,2)
f_off(M[TimeAct]+1= M[TimeSel],tickTime)	=	(display("Stop"),M,3)
f_updTime(M[TimeAct]+1< M[TimeSel],tickTime)	=	(display(M[TimeSel]-M[TimeAct]++ "min."),M[TimeAct]++,0.1)
f_updTemp(M[TempAct]+1< M[TempSel],tickTemp)	=	(display(M[TempAct]),M[TempAct]++,0.1)

Figure 1 An example of Timed Stream X-machine.

function, $s_{in} \in S$ is the *initial state*, and $m_{in} \in Mem$ is the *initial memory value*.

A *transition* is a tuple (s, ϕ, s') where $s, s' \in S$ are the initial and final states of the transition, and $\phi \in \Phi$ is the timed processing function associated with the transition. A *configuration* in X is a pair (s, m) where $s \in S$ is the current state and $m \in Mem$ is the current memory value.

We say that X is *deterministic* if for all $\phi, \phi' \in \Phi$, if there exists $s \in S$ such that $(s, \phi), (s, \phi') \in dom(F)$, then $\phi = \phi'$ or $dom(\phi) \cap dom(\phi') = \emptyset$. We say that X is *completely specified* if for all $s \in S, m \in M$, and $i \in I$, there exists $\phi \in \Phi$ such that $(m, i) \in dom(\phi)$ and $(s, \phi) \in dom(F)$. □

Intuitively, a TSXM is a state machine where arcs connecting states are labeled by timed processing functions. Each of these functions receives an input and the current memory values and produces an output while may modify the memory. Additionally, a value $t \in$ Time indicates that the lapse between the reception of the input and the performance of the output is equal to t time units.

For each state $s \in S$, the application of the timeout function $TO(s)$ returns a pair (s', t) indicating the time that the machine can remain at the state s waiting for an input, and the state to which the machine evolves if no input is received on time, that is, before t time units pass. We assume that $TO(s) = (s', t)$ implies $s \neq s'$, that is, timeouts always produce a change of the state.

We indicate the absence of a timeout in a given state by setting the corresponding time value to ∞.

As usual, we do not consider a notion of *final* state as in classical automata theory. In fact, in our framework all the states can be considered as *final* since all the sequences that can be performed from the initial state are *valid*. On the contrary, as we will show in Section 5, tests will have two notions of final state, pass and fail, to denote that the sequence performed by the tested system is correct/incorrect.

A `TSXM` is deterministic if for all state s, all input action, and all memory value there is at most one processing function that can be applied. Determinism is a usual restriction when working with Stream X-machines, although recent work tries to diminish this limitation (see [22]). In this paper we do not restrict ourselves to deterministic machines, neither for systems under test nor for specifications. In turn, a `TSXM` is *completely specified* if for all $s \in S, m \in M$, and $i \in I$ there is always a possible transition. As usual, we will consider that systems under test are completely specified so that they will always be able to show a reaction when provided with an input for a given value of the memory.

Next, we show an application of the `TSXM` formalism to the specification of a real system. We will use this specification as a running example to illustrate some of the concepts introduced in the paper.

Example 1 Let us consider the machine depicted in Figure 1 based on a simplification of an electronic oven. In fact, we have only removed some technical details that might distract the reader from the concepts that we would like to illustrate. Thus, the specification that we present is very close to the complete one used to specify the *real* oven.

The oven has two operation modes: Oven and grill. The oven provides a set of buttons (`bgrill`, `boven`, `boff`, `bset`, `bup`, `bdown`) and a screen. The buttons allow the user to access the different functionalities of the oven and correspond to the inputs that can be applied. We consider two additional inputs, `tickTime` and `tickTemp`, that can be emitted by hardware devices embedded in the oven, when the temperature of the oven increases 1^oC or one unit of time passes, respectively. In our system, outputs correspond to the display of different *messages*. We define a data type $Message$ to represent them. We write $display(s)$ to indicate that the message $s \in Message$ is displayed on the screen. We will use "++" to concatenate strings. Finally, the internal memory is a tuple of variables (`Mode`, `TempMax`, `TempSel`, `TempAct`, `TimeMax`, `TimeSel`, `TimeAct`) related to the operation mode of the oven, maximum temperature/time that can be selected, the temperature/time established by the user and the current temperature/time. The initial values of these variables is *null* for `Mode` and 0 for the rest of the variables, except `TempMax` and `TimeMax` that are initialized to 250 and 180, respectively. The specification has five states, being `start` its initial state.

The specification of the oven is depicted in Figure 1. The different transitions are labeled by the processing functions that will be executed considering the current values of the memory and the input applied. All of them follow the pattern:

function_name(mem_input, p_in)=(p_out, mem_upd,time)

In order to present in a compact way the memory values accepted by the functions, the parameter mem_input corresponds to a condition over the variables of the internal memory, that is, the function could be performed only if variables of the internal memory fulfill that condition. For example, if we consider the first occurrence of the processing function f_selTemp, the expression

`M[TempSel]+10<=M[TempMax]`

indicates that this transition will be performed only if the increase of 10 degrees to the selected temperature keeps the value below the maximum allowed temperature. In other words, the expression

f_selTemp(`M[TempSel]+10<=M[TempMax]`,`bup`)

is a shorthand to represent all the expressions

f_selTemp($(x_1, x_2, \ldots, x_7)$,`bup`)

such that $x_1 \in$ `Mode`, $x_2 \in$ `TempMax`, $x_3 \in$ `TempSel`, $x_4 \in$ `TempAct`, $x_5 \in$ `TimeMax`, $x_6 \in$ `TimeSel`, $x_7 \in$ `TimeAct`, and $x_3 + 10 \leq x_2$.

The parameter p_in corresponds to the input accepted and p_out to the message displayed after the internal memory has been updated according to the assignments indicated in the output parameter mem_upd.

Next, we explain the behavior of the oven. The initial state, `start`, corresponds to the point in which the user switches on the oven. At the initial state the user can choose the operation mode by pressing the buttons `boven` or `bgrill`. This interaction produces the execution of the processing function f_selMode, changing the system to the `time` state. The inputs allowed at this state are `bup` and `bdown` to increase/decrease, respectively, the time of cooking. Once the user sets the time, by pressing the `bset` button, depending on the selected operation mode, a transition will be triggered. If the GRILL mode was selected, then the oven will switch on the grill. If the OVEN mode was selected, then the user must establish, using again the buttons `bup` and `bdown`, the temperature. During the cooking phase, either in GRILL or in OVEN modes, the system will periodically receive signals, represented by the input `tickTemp`, that indicate the increment of the temperature. It produces the execution of the processing function f_updTemp until the selected temperature is reached, represented by the condition `M[TempAct]+1 < M[TempSel]`. In addition, the passing of one time unit will be notified by the emission of the input `tickTime`

until the time selected by the user passes. When the selected period of time finishes, that is, `M[TimeAct]+1= M[TimeSel]`, the processing function f_off is performed and the oven switches off.

Overall, the following processes are considered in the system:

- The user chooses the mode: OVEN or GRILL.
- The user selects, by pressing the buttons `bup` or `bdown`, the time/temperature. The temperature can be selected only if the mode OVEN was chosen.
- The temperature/time is set by pushing the button `bset`.
- The oven automatically updates the time for each time unit and displays the passed time.
- The oven automatically updates the temperature, when mode is OVEN, for each degree.
- The oven switches off if the time selected by the user passes.
- The user can switch off the oven at any time.

For the sake of clarity, we have omitted the transitions outgoing from every state and leading to the initial state, corresponding to the functionality that allows the users to switch off the oven at any time by pressing the button `boff`.

There are two states, `time` and `temp`, that have associated a timeout of five time units. This means that if there is no user interaction before five time units, then the system will go back to the initial state and the oven will switch off. □

Next we introduce a partial function that establishes the relation between a pair (memory values, input sequence) and a triple (output sequence, updated memory values, time sequence) produced by the application of a sequence of timed processing functions.

Definition 3 Given a sequence $\bar{\phi} \in \Phi^*$, we consider $\| \bar{\phi} \|$: $Mem \times I^* \to O^* \times Mem \times \texttt{Time}$ to be a partial function inductively defined in the following way:

$$\| \epsilon \| = \{((m,\epsilon),(\epsilon,m,0)) \mid m \in Mem\}$$

$$\| \bar{\phi}\phi \| = \left\{ ((m,\bar{i}i),(\bar{o}o,m',t)) \;\middle|\; \begin{array}{l} \exists\, m'' \in Mem, t'' \in \texttt{Time}: \\ ((m,\bar{i}),(\bar{o},m'',t'')) \in \| \bar{\phi} \| \\ \wedge\, ((m'',i),(o,m',t')) \in \phi \\ \wedge\, t = t' + t'' \end{array} \right\}$$

where ϵ represents the empty sequence and $\bar{x}x$ represents the concatenation of the sequence $\bar{x}$ and the element x. □

In the previous definition we have used the notation $(a,b) \in f$ instead of the more standard $f(a) = b$.

Let us note that when we consider a deterministic `TSXM` each computation from the initial state to any other state is completely determined by the input sequence and the initial memory value. In our setting, this is not the case. This fact complicates the definition of conformance relations. For example, the same sequence of inputs can produce different outputs or the same sequence of outputs but with two different time values. The following notion allows us to compose several transitions possibly preceded by timeouts.

Definition 4 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a `TSXM`. A tuple $(s_0, \phi, s, \hat{t}\,)$ is a *step* of X for the configuration $c_0 = (s_0, m_0)$, if there exist $k \geq 0$ states $s_1, \ldots, s_k \in S$, such that $F(s_k, \phi) = s$, $TO(s_{j-1}) = (s_j, t_j)$ for all $1 \leq j \leq k$, and

$$\hat{t} = \left[\sum_{j=1}^{k} t_j, \sum_{j=1}^{k} t_j + \pi_2(TO(s_k)) \right)$$

We say that a tuple $(\hat{t}_1/i_1/t_{o1}/o_1, \ldots, \hat{t}_r/i_r/t_{or}/o_r)$ is a *timed evolution* of X if there exist r steps of X $(s_{in}, \phi_1, s_1, \hat{t}_1), \ldots, (s_{r-1}, \phi_r, s_r, \hat{t}_r)$ for the configurations $(s_{in}, m_{in}), \ldots, (s_{r-1}, m_{r-1})$, respectively, and there exists $m \in M$ such that $((m_{in}, \bar{i}), (\bar{o}, m, t)) \in \| \bar{\phi} \|$, where $\bar{i} = (i_1, \ldots, i_r)$, $\bar{o} = (o_1, \ldots, o_r)$, and $t = \sum_{j=1}^{r} t_{oj}$. We denote by $\texttt{TEvol}(X)$ the set of timed evolutions of X. In addition, we say that $(\hat{t}_1/i_1/o_1, \ldots, \hat{t}_r/i_r/o_r)$ is a *functional evolution* of M. We denote by $\texttt{FEvol}(X)$ the set of functional evolutions of M. □

Intuitively, a step is a transition preceded by zero or more timeouts. The interval $\hat{t}$ indicates the time values where the input action could be received. An evolution is a sequence of inputs, outputs, and time values corresponding to the transitions of a chain of steps. The first of these steps begins with the initial state of the machine. These steps include the time interval, indicated by the different intervals $\hat{t}_j$, when an input could be accepted. As we will explain later when we introduce our conformance relations, evolutions need to include time information. Specifically, they must contain information related to the triggered timeouts. This is due to the fact that timeouts influence the different timed processing function sequences that `TSXM`s can perform. This information is encoded into the intervals. In addition to the previous information, timed evolutions present the time consumed to execute each output after receiving each input in each step of the evolution.

Along the paper, we will refer to a timed evolution $(\hat{t}_1/i_1/t_{o1}/o_1, \ldots, \hat{t}_r/i_r/t_{or}/o_r)$ as $(\breve{t}, \bar{i}, t_o, \bar{o})$ where $\breve{t} = (\hat{t}_1, \ldots, \hat{t}_r)$, $\bar{i} = (i_1, \ldots, i_r)$, $t_o = \sum_{j=1}^{r} t_{oj}$, and $\bar{o} = (o_1, \ldots, o_r)$. Similarly, $(\breve{t}, \bar{i}, \bar{o})$ will represent the functional evolution associated to the previous timed evolution.

In the next definition we introduce the concept of *instanced evolution*. Intuitively, instanced evolutions are constructed from evolutions by instantiating to a concrete value each timeout, given by an interval, of the evolution.

Definition 5 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a `TSXM` and $e = (\breve{t}, \bar{i}, t_o, \bar{o})$ be a *timed evolution* of X. We

say that the tuple $(\bar{t}, \bar{i}, t_o, \bar{o})$ is an *instanced timed evolution* of e if $\bar{t} \in \breve{t}$. In addition, we say that the tuple $(\bar{t}, \bar{i}, \bar{o})$ is an *instanced functional evolution* of e.

We denote by $\texttt{InsTEvol}(X)$ the set of instanced temporal evolutions of X and by $\texttt{InsFEvol}(X)$ the set of instanced functional evolutions of X. □

The next example illustrates the concepts of step and (instanced) evolutions.

Example 2 Let us consider the TSXM presented in Example 1. For example, (start, f_selMode, time, $[0, \infty)$) represents a step where no timeouts precede the corresponding transition. The processing function f_selMode can be performed at any time if the buttons boven or bgrill are pressed by the user. Let us note that these are the only inputs accepted by the function f_selMode. Another example where no timeouts precede the transition is the step (time, f_selTime, time, $[0, 5)$). The processing function f_selTime can be performed before five time units pass, indicated by the interval $[0, 5)$, if an input belonging to its domain, that is, bup and bdown, is received. Another step, (time, f_selMode, time, $[5, \infty)$), is built from the timeout associated to the time state and the transition outgoing from start. The step represents that if the machine is at time state and after 5 time units no input is received, then the timeout associated with that state will be triggered and the state will change to start. After this, the machine can accept an input that belongs to the domain of the processing function f_selMode at any time. Similarly, we can obtain the step (temp, f_selMode, time, $[5, \infty)$), using the timeout corresponding to the temp state and the transition outgoing from start.

The following is an example of functional evolution built from two steps:

$$\left(\begin{array}{l} [0,\infty)/\texttt{boven}/\text{“select Time Oven”}, \\ [0,5)/\texttt{bup}/\text{“10 min.”}, \\ [0,5)/\texttt{bup}/\text{“20 min.”} \end{array}\right)$$

If we consider this functional evolution we have that these tuples

$$\left(\begin{array}{l} 15/\texttt{boven}/\text{“select Time Oven”}, \\ 2/\texttt{bup}/\text{“10 min.”}, \\ 3/\texttt{bup}/\text{“20 min.”} \end{array}\right)$$

and

$$\left(\begin{array}{l} 1200/\texttt{boven}/\text{“select Time Oven”}, \\ 1/\texttt{bup}/\text{“10 min.”}, \\ 4/\texttt{bup}/\text{“20 min.”} \end{array}\right)$$

are two of its instanced evolutions.

Let us note that a tuple built with the first step followed by more than 18 occurrences of the second step is not an evolution. That is due to the fact that after the user presses 18 times the button bup the values of the memory would not belong to the domain of the function f_selTime. □

Given a TSXM, we can establish another relation between the input sequences applied to the machine and the output sequences produced by it. This relation is given by the execution of a sequence of processing functions, from the initial state of the machine, that allows to obtain an output sequence in response to an input sequence. In our formalism, we will require to extend this relation to consider temporal requirements. On the one hand, we need to deal with the specified timeouts to take into account the time values when the machine receives an interaction from the environment. On the other hand, we must consider the time that the machine needs to perform the processing functions.

Definition 6 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TSXM. The *timed function computed* by X

$$f_X^{\mathcal{T}} : \ I^* \times \texttt{Time}^* \rightarrow \ O^* \times \ \texttt{Time}$$

is defined as follows:

$$f_X^{\mathcal{T}} = \left\{ ((\bar{i}, \bar{t}), (\bar{o}, t_o)) \,\middle|\, (\bar{t}, \bar{i}, t_o, \bar{o}) \in \texttt{InsTEvol}(X) \right\}$$

The *function computed* by X

$$f_X : I^* \times \texttt{Time}^* \rightarrow O^*$$

is defined as follows:

$$f_X = \left\{ ((\bar{i}, \bar{t}), \bar{o}) \,\middle|\, (\bar{t}, \bar{i}, \bar{o}) \in \texttt{InsFEvol}(X) \right\}$$

□

Along the paper we will refer to a tuple $((\bar{i}, \bar{t}), \bar{o})$ as $(\bar{t}, \sigma)$, where $\sigma = (i_1/o_1, \ldots, i_r/o_r)$. Respectively, the tuple $(\bar{t}, \sigma, t_o)$ will represent $((\bar{i}, \bar{t}), (\bar{o}, t_o))$. The following result shows that for all the elements belonging to the timed function computed by a TSXM there exists a sequence of processing functions that computes, in the specified time, the corresponding inputs/outputs sequence.

Lemma 1 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TSXM. We have that

$$((\bar{i}, \bar{t}), (\bar{o}, t_o)) \in f_X^{\mathcal{T}}$$

$$\Downarrow$$

$$\exists\, \bar{\phi} \in \Phi^*, m' \in M : ((m_{in}, \bar{i}), (\bar{o}, m', t_o)) \in \| \ \bar{\phi} \ \|$$

Proof: Let $\bar{i} = (i_1, \ldots, i_r)$, $\bar{o} = (o_1, \ldots, o_r)$, and $\bar{t} = (t_1, \ldots, t_r)$. Since $((\bar{i}, \bar{t}), (\bar{o}, t_o)) \in f_X^{\mathcal{T}}$ we have $(\bar{t}, \bar{i}, t_o, \bar{o}) \in \texttt{InsTEvol}(X)$. Thus, there exists a timed evolution $e = (\breve{t}, \bar{i}, t_o, \bar{o})$ such that $\bar{t} \in \breve{t} = (\hat{t}_1, \ldots, \hat{t}_r)$. By Definition 4, there exist r steps $(s_{in}, \phi_1, s_1, \hat{t}_1), \ldots, (s_{r-1}, \phi_r, s_r, \hat{t}_r)$ of X and $m \in M$ such that $((m_{in}, \bar{i}), (\bar{o}, m, t_o)) \in \| \ \bar{\phi} \ \|$, where $\bar{\phi} = (\phi_1, \ldots, \phi_r)$ and $t_o = \sum_{j=1}^{r} t_j$. □

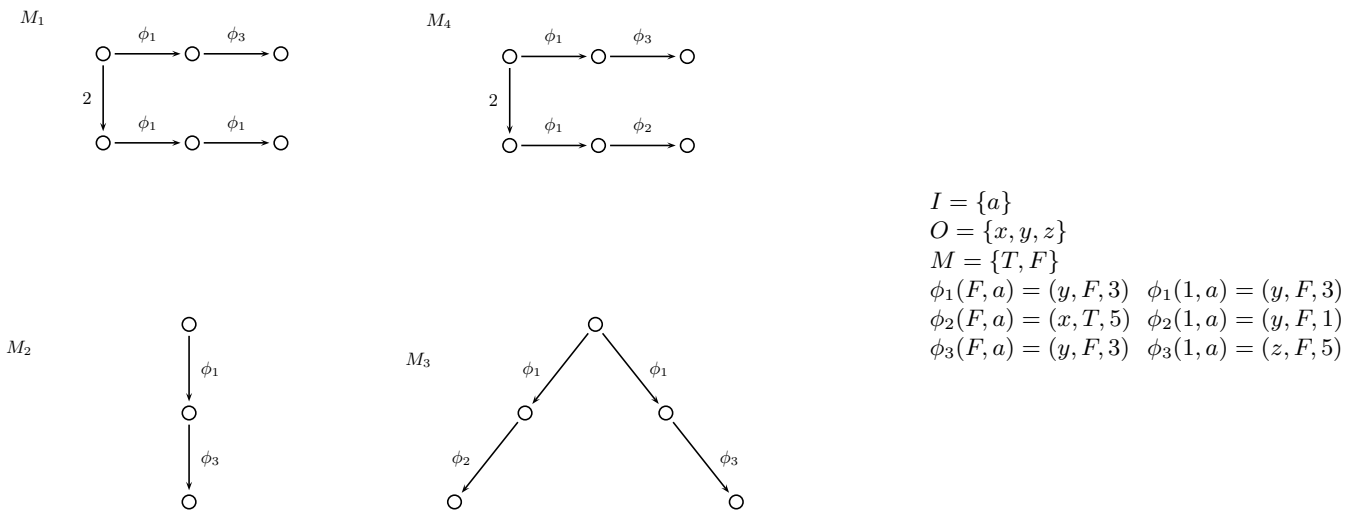


Figure 2 Examples of functional conformance.

4 Timed Conformance Relations

In order to properly define how to test a system against a specification it is necessary to state what it means for a system to conform to a specification. Even though there are different perspectives to consider what a good implementation is, there is an agreement on correctness if we consider only *functional behavior*, that is, abstracting the time that actions take to be performed. Regarding the performance of usual inputs and outputs, an implementation should not *invent* behaviors when inputs specified in the specification are applied to it. In other words, the *relevant* evolutions of the implementation must be contained in those corresponding to the specification. Thus, all our implementation relations follow the same pattern: An implementation I *conforms* to a specification S if for all possible evolution of S the outputs that the implementation I may perform after a given input are a subset of those of the specification. Let us note that the time spent by a system waiting for the environment to react has the capability of affecting the set of available outputs of the system. This is because this time may trigger a change of the state by raising one or more timeouts. So, our functional conformance relation must explicitly take into account the maximal time the system may stay in each state. This time is given by the *timeout* of the state. Thus, we require that the implementation always complies, in a certain manner, with the timeouts established by the specification.

Additionally, we need to describe what means for a system to be *timely* correct with respect to a specification. It would be reasonable to require that any sequence of the implementation has the same delay as the one specified by the specification. However, in our context, the notion of correctness has several possible definitions. In fact, in the case of timed systems, what is a good implementation is less precise than in untimed systems. For example, one may consider that a system I is a good implementation of a specification S if I takes the same time to perform its tasks as the specification S while another could consider that the implementation has to be always/sometimes faster. Thus, for each of these considerations, we will define a different conformance relation, being the users of our framework who will have to decide which implementation relation better suites their idea of what a good implementation is.

In our framework, specifications will be modeled by timed Stream X-machines. Let us note that we consider black-box testing, that is, a system will be tested against a specification without having information about the internal structure of the system. The only knowledge about the system under test (in short, SUT) must be inferred from the outputs observed when tests interact with the system. As usual, we also consider that the SUT is described by using a `TSXM` having the same sets of inputs and outputs as the specification. We also assume that SUTs are completely specified (see Definition 2). This is a usual condition in (formal) testing to ensure that the SUT will be always able to answer to any stimulus provided by the tester. Let us remind that we do not restrict the machines to be deterministic. Thus, both implementations and specifications may present non-deterministic behavior, increasing the expressive power of the framework. Next, we introduce the conformance relation $\texttt{conf}_f$.

Definition 7 Let S and I be two `TSXM`s. We say that I *functionally conforms* to S, denoted by $I\ \texttt{conf}_f\ S$, if for all $e = (\breve{t}, \bar{i}, \bar{o}) \in \texttt{FEvol}(S)$, with $\bar{o} = (o_1, \ldots, o_{r-1}, o_r)$, we have that for all $\bar{t} \in \breve{t}$ and $\bar{o'} = (o_1, \ldots, o_{r-1}, o'_r)$

$$e' = ((\bar{i}, \bar{t}), \bar{o'}) \in f_I \Longrightarrow e' \in f_S$$

being f_S and f_I the functions computed by S and I, respectively, as introduced in Definition 6. □

Let us note that if the specification is completely specified then we can remove the condition "for all $e = (\check{t}, \bar{i}, \bar{o}) \in \texttt{FEvol}(S)$, with $\bar{o} = (o_1, \ldots, o_{r-1}, o_r)$".

In the following example we illustrate our notion of functional conformance.

Example 3 Let us consider the schematic machines depicted in Figure 2. These diagrams represent simplified TSXMs. We consider the following notation: A transition with a label t indicates that a timeout will be applied at time t, that is, if after t time units no input is received then the timeout is executed. We consider that the initial state of the machine is the upper-left one and that the initial value of the only variable appearing in the memory is equal to F.

Let us note that the behavior of M_1 and M_2 is exactly the same regardless of the timeout presented in M_1. The only sequences of inputs/outputs that can be produced by M_1 and M_2 are a/y and $a/y, a/y$. We can say the same regarding the conformance of M_2 with respect to M_1. We also have that M_1 conforms to M_3. This is due to the fact that we only consider the sequences of inputs/outputs that can be performed by M_1, that is, a/y and $a/y, a/y$, and both of them can be generated by M_3. If we consider M_2 instead of M_1 then the same considerations apply: M_2 conforms to M_3.

Next, we show how the availability of outputs affects functional conformance. We have that M_3 does not conform to either M_1 or to M_2. Let us note that if a is offered in M_1 or M_2 after executing a/y then only y can be produced. However, M_3 can produce this output as well as x, which is forbidden by M_1 and M_2.

Similarly, we have that M_3 does not conform to M_4. The function ϕ_2 is allowed by M_4 after two consecutive applications of the input a only if the first application has been performed after 3 time units have passed. Otherwise, the only function allowed will be ϕ_3. However, M_3 could perform either ϕ_3 or ϕ_2 at any time, depending on which transition has been triggered after the first input has been applied. Thus, if we consider M_4, wait three time units, apply a, obtaining y, and then apply again the input a, then only x can be produced. However, M_3 can produce this output as well as y, which is forbidden by M_4. □

In the following definition we present our *timed* conformance relations. We have several possibilities. In order to concentrate on the presentation of the considered relations, in this paper we introduce only three relations, but other relations could be defined by following the same pattern as [35], where we gave ten relations for a notion of timed Finite State Machine. Informally, the $\texttt{conf}_a$ relation (conforms *always*) holds if for all instanced timed evolution e of the implementation we have that if its associated instanced functional evolution is an instanced functional evolution of the specification, then e is also an instanced timed evolution of the specification. In the $\texttt{conf}_w$ relation (conformance in the *worst* case) the implementation is forced, for each instanced timed evolution fulfilling the previous conditions, to be faster than the slowest instance of the same evolution in the specification. The $\texttt{conf}_b$ relation (conforms in the *best* case) is similar but taking into account only the fastest instance of the specification.

Definition 8 Let X be a TSXM and $t_o \in \texttt{Time}$. Given $e = (\bar{t}, \sigma) \in f_X$, we denote by $e \trianglerighteq t_o$ the triple $(\bar{t}, \sigma, t_o)$.

Let S and I be two TSXMs. Our *timed conformance relations* are defined as follows:

- *(always)* $I \,\texttt{conf}_a\, S$ iff $I \,\texttt{conf}_f\, S$ and for all $e \in f_I \cap f_S$ we have that for all $t \in \texttt{Time}$:

$$e \trianglerighteq t \in f_I^{\mathcal{T}} \Longrightarrow e \trianglerighteq t \in f_S^{\mathcal{T}}$$

- *(best)* $I \,\texttt{conf}_b\, S$ iff $I \,\texttt{conf}_f\, S$ and for all $e \in f_I \cap f_S$ we have that for all $t \in \texttt{Time}$:

$$e \trianglerighteq t \in f_I^{\mathcal{T}}$$

$$\Downarrow$$

$$\forall\, t' \in \texttt{Time} : \left(e \trianglerighteq t' \in f_S^{\mathcal{T}} \Longrightarrow t \leq t'\right)$$

- *(worst)* $I \,\texttt{conf}_w\, S$ iff $I \,\texttt{conf}_f\, S$ and for all $e \in f_I \cap f_S$ we have that for all $t \in \texttt{Time}$:

$$e \trianglerighteq t \in f_I^{\mathcal{T}}$$

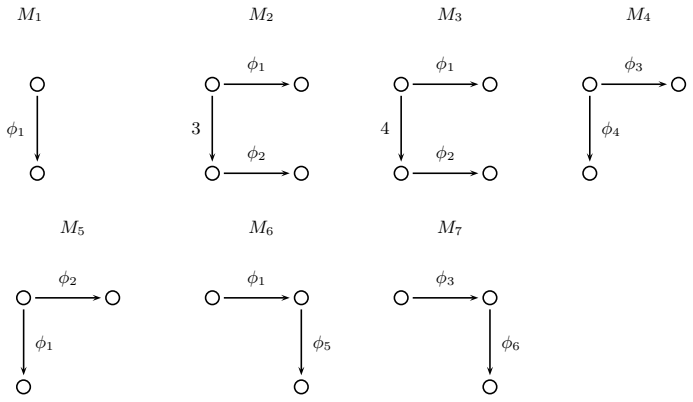
$$\Downarrow$$

$$\exists\, t' \in \texttt{Time} : \left(e \trianglerighteq t' \in f_S^{\mathcal{T}} \ \wedge\ t \leq t'\right)$$

□

Next we will show several examples to illustrate the previous conformance relations.

Example 4 Let us consider the systems depicted in Figure 3. If we consider the machines M_2 and M_1 we have that none of the relations given in Definition 8 hold between them. This is due to the existence of *different time values to perform output actions*. Let us note that M_2 may take 3 time units to perform the output x if it receives the input a after 3 time units, $(3/a/3/x)$, while M_1 only needs 2 time units, $(3/a/2/x)$. Since M_2 is, in *any* case, slower than M_1 for these sequences of inputs/outputs, no conformance relation where M_2 is the SUT and M_1 is the specification holds. On the contrary, if M_1 is the SUT and M_2 is the specification then all the relations, except $\texttt{conf}_a$, hold. First, let us remark that the only sequence that both machines can perform is a/x. Despite the fact that M_1 does not take the same time values as M_2 to perform this sequence, its time is smaller than (in the case that the input is received after the timeout is triggered) or equal to (in the case that the input is received before the timeout is triggered) the time values corresponding to M_2.



$I = \{a, b\}$
$O = \{x\}$
$M = \{0, 1\}$
$\phi_1(a, 0) = (x, 1, 2)$ $\phi_2(a, 0) = (x, 1, 3)$
$\phi_3(a, 0) = (x, 1, 4)$ $\phi_4(a, 0) = (x, 1, 1)$
$\phi_5(b, 1) = (x, 1, 4)$ $\phi_6(b, 1) = (x, 1, 3)$

Figure 3 Examples of timed conformance.

As we have seen, reducing the time consumed by actions can produce that a certain TSXM conforms to another one with respect to some of the available conformance relations. Even though the requirements on timeouts are *strict*, sometimes having different timeouts can also produce the same effect.

Let us consider M_2 and M_3. In this case, the lack of conformance is produced by the fact that the machines have *different timeouts*. Most input/output sequences in M_2 and M_3 take the same time values to be performed. However, there is an exception: The sequence including the timeout 3. In M_2 we have $(3/a/3/x)$ but in M_3 we have $(3/a/2/x)$. This is so because after 3 time units pass the state does not change yet in M_3. Hence, we have that all relations, except $\mathtt{conf}_a$, hold but no conformance relation where M_2 is the SUT and M_3 is the specification holds.

Let us consider an example where SUTs and specifications can spend different time values in executing sequences. We consider M_4 and M_5. Both M_4 and M_5 can execute a/x by spending an amount of time that the other one cannot take. So, we do not have $M_4\, \mathtt{conf}_a\, M_5$. The worst time values to execute a/x in M_4 and M_5 are 4 and 3, respectively, while the best time values are 1 and 2, respectively. The worst time of M_4 is not better than either the worst or the best time of M_5. Thus, we have neither $M_4\, \mathtt{conf}_w\, M_5$ nor $M_4\, \mathtt{conf}_b\, M_5$. On the contrary, the worst time of M_5 is better than the worst one of M_4 but worse than the best one of M_4. Hence, $M_5\, \mathtt{conf}_w\, M_4$ holds but $M_5\, \mathtt{conf}_b\, M_4$ does not.

Let us consider M_6 and M_7. M_6 performs both available sequences (a/x and $a/x, b/x$) faster than M_7: In M_6 they spend 2 and 6 time units, respectively, while these time values are 4 and 7, respectively, in M_7. So, we have $M_6\, \mathtt{conf}_w\, M_7$ and $M_6\, \mathtt{conf}_b\, M_7$. Let us remark that none of these relations holds if we exchange the roles of both machines. □

Theorem 1 Let I and S be two TSXMs. The relations given in Definition 8 are related as follows:

$$I\ \mathtt{conf}_a\ S \Rightarrow I\ \mathtt{conf}_w\ S \Leftarrow I\ \mathtt{conf}_b\ S$$

Proof: Let us note that the condition about functional conformance is the same in all the definitions. So, we only need to take into account the conditions on time values appearing in the second clause of the corresponding relations.

If $I\ \mathtt{conf}_a\ S$ then we have that for all $e \in f_I \cap f_S$ and $t \in \mathtt{Time}$, if $e \trianglerighteq t \in f_I^{\mathcal{T}}$ then $e \trianglerighteq t \in f_S^{\mathcal{T}}$. So, there exists $t' = t \in \mathtt{Time}$ such that $e \trianglerighteq t' \in f_S^{\mathcal{T}}$ and we have $I\ \mathtt{conf}_w\ S$.

If $I\ \mathtt{conf}_b\ S$ then for all $e \in f_I \cap f_S$ and all $t \in \mathtt{Time}$, if $e \trianglerighteq t \in f_I^{\mathcal{T}}$ then all $t' \in \mathtt{Time}$ such that $e \trianglerighteq t' \in f_S^{\mathcal{T}}$ fulfills $t \leq t'$. Therefore, in particular, there exists $t' \in \mathtt{Time}$ such that $e \trianglerighteq t' \in f_S^{\mathcal{T}}$ and $t \leq t'$ and we conclude $I\ \mathtt{conf}_w\ S$. □

It is interesting to note that if specifications are restricted to be deterministic then the relations $\mathtt{conf}_b$ and $\mathtt{conf}_w$ would coincide, but they would be still different from the $\mathtt{conf}_a$ relation. This is so because $\mathtt{conf}_a$ requires that time values in the implementation coincide with those in the specification, while other relations require that time values in the implementation are *less than or equal to* those of the specification.

The hierarchy of relations induced by Theorem 1 allows us to compare implementations in the following way: I_1 is *preferable* to I_2 to implement S if the former meets with S a *stricter* relation.

Definition 9 Let I_1, I_2, S be TSXMs and $\mathtt{conf}_x$ and $\mathtt{conf}_y$ be timed conformance relations such that $I_1\ \mathtt{conf}_x\ S$, $I_2\ \mathtt{conf}_y\ S$, $\mathtt{conf}_x \Rightarrow \mathtt{conf}_y$. In this case, we say that I_1 is *preferred* to I_2 to implement S. □

5 Definition and Application of Tests

Tests are sequences of inputs to be applied to a SUT. Once an output is received we have to check whether it belongs to the set of expected ones or not. In addition to checking the functional behavior of the SUT, tests have also to detect whether wrong timed behaviors appear. Thus, tests have to include capabilities to deal with the

time that the system spends performing the specified actions and timeouts. On the one hand, we will include *time stamps*. The time values recorded from the SUT while applying the test will be compared with the ones indicated in the test. Each time stamp will contain a set of *time values* for each instanced timed evolution. On the other hand, tests will include *delays* before offering input actions. The purpose of delays is to induce timeouts in the tested machine. In this way, we may indirectly check whether the timeouts imposed by the specification are reflected in the SUT by offering input actions after a specific delay. Let us note that we cannot observe when the SUT takes a timeout. However, it is still possible to check the SUT behavior after different delays. Thus, our tests have a more complex structure than being a simple sequence of inputs (and its corresponding expected outputs).

Definition 10 We say that a *test* is a tuple represented by $T = (S, I, O, Tr, s_0, S_I, S_O, S_F, S_P, C, W)$ where S is the set of states, I and O are disjoint sets of inputs and outputs, respectively, $Tr \subseteq S \times (I \cup O) \times S$ is the transition relation, $s_0 \in S$ is the initial state, and the sets $S_I, S_O, S_F, S_P \subseteq S$ are a partition of S. The transition relation and the sets of states fulfill the following conditions:

- S_I is the set of *input* states. We have that $s_0 \in S_I$. For all input state $s \in S_I$ there exists a unique outgoing transition $(s, a, s') \in Tr$. For this transition we have that $a \in I$ and $s' \in S_O$.
- S_O is the set of *output* states. For all output state $s \in S_O$ we have that for all $o \in O$ there exists a unique state s' such that $(s, o, s') \in Tr$. In this case, $s' \notin S_O$. Moreover, there do not exist $i \in I$ and $s' \in S$ such that $(s, i, s') \in Tr$.
- S_F and S_P are the sets of *fail* and *pass* states, respectively. We say that these states are *terminal*. Thus, for all state $s \in S_F \cup S_P$ we have that there do not exist $a \in I \cup O$ and $s' \in S$ such that $(s, a, s') \in Tr$.

We have two functions: $W : S_I \longrightarrow \texttt{Time}$ associates delays with input states while $C : S_P \longrightarrow \mathcal{P}(\texttt{Time})$ associates time stamps with pass states. We say that a test T is *valid* if the graph induced by T is a tree with root at the initial state s_0. In the rest of the paper we consider only valid tests.

Let $\sigma = i_1/o_1, \ldots, i_r/o_r$. We write $T \stackrel{\sigma}{\Longrightarrow} s^T$ if $s^T \in S_F \cup S_P$ and there exist states $s_{12}, s_{21}, s_{22}, \ldots s_{r1}, s_{r2} \in S$ such that $\{(s_0, i_1, s_{12}), (s_{r2}, o_r, s^T)\} \subseteq Tr$, for all $2 \leq j \leq r$ we have $(s_{j1}, i_j, s_{j2}) \in Tr$, and for all $1 \leq j \leq r-1$ we have $(s_{j2}, o_j, s_{(j+1)1}) \in Tr$.

Let T be a test, $\sigma = i_1/o_1, \ldots, i_r/o_r$, s^T be a state of T, and $\overline{t} \in \texttt{Time}^r$. We write $T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ if $T \stackrel{\sigma}{\Longrightarrow} s^T$, $t_1 = W(s_0)$ and for all $1 < j \leq r$ we have $t_j = W(s_{j1})$. □

In Figure 4 we show a graphical representation of some tests. Let us remark that $T \stackrel{\sigma}{\Longrightarrow} s^T$, and its variant $T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$, imply that s^T is a terminal state. Next we define the application of a test suite to an implementation. We say that the test suite $\mathcal{T}$ is *passed* if for all test belonging to the suite we have that the terminal states reached by the composition of implementation and test are *pass* states. Besides, we give different timing conditions in a similar way to what we did for implementation relations.

Definition 11 Let I be a TSXM, T be a valid test, s^T be a state of T, $\sigma = (\bar{i}, \bar{o})$, where $\bar{i} = (i_1, \ldots, i_r)$ and $\bar{o} = (o_1, \ldots, o_r)$, and $\overline{t} = (t_1, \ldots, t_r)$. We write $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ if $T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ and $(\bar{t}, \sigma) \in f_I$. We write $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}, t_o}} s^T$ if $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ and $(\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$.

Let $\mathcal{T}$ be a test suite and $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$. We define $\texttt{Fin\_State}(e, \mathcal{T})$ as the set of states

$$\{s^T \mid \exists T \in \mathcal{T} : I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}, t_o}} s^T\}$$

Let I be a TSXM and $\mathcal{T}$ be a test suite. We define the following timed conformance relations:

- I *passes* $\mathcal{T}$, denoted by $\texttt{pass}(I, \mathcal{T})$, if for all $T \in \mathcal{T}$ there do not exist σ, s^T, $\overline{t}$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow_{\overline{t}}} s^T$ and $s^T \in S_F$.
- I *passes always in time* $\mathcal{T}$ if $\texttt{pass}(I, \mathcal{T})$ and for all $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$ and all $s^T \in \texttt{Fin\_State}(e, \mathcal{T})$, we have $t_o \in C(s^T)$.
- I *passes in the best time* $\mathcal{T}$ if $\texttt{pass}(I, \mathcal{T})$ and for all $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$, all $s^T \in \texttt{Fin\_State}(e, \mathcal{T})$, and all $t_c \in C(s^T)$, we have $t_o \leq t_c$ holds.
- I *passes in the worst time* $\mathcal{T}$ if $\texttt{pass}(I, \mathcal{T})$ and for all $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$ and all $s^T \in \texttt{Fin\_State}(e, \mathcal{T})$, there exists $t_c \in C(s^T)$ such that $t_o \leq t_c$ holds.

□

6 Derivation of Sound and Complete Test Suites

In this section we present an algorithm to derive tests from specifications. The idea underlying our algorithm consists in traversing the specification in order to get all the possible input/output sequences in an appropriate way. First, we introduce some additional notation. The following functions will be used in the forthcoming derivation algorithm.

Definition 12 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TSXM. The function $\texttt{out}(s, i, m)$ computes the set of outputs associated with those transitions that can be executed from s after receiving the input i, and assuming that the value of the memory is given by m.

$$\texttt{out}(s, i, m) = \left\{ o \;\middle|\; \begin{array}{l} \exists\, t \in \texttt{Time}, \phi \in \Phi, m' \in M : \\ ((m, i), (o, m', t)) \in \phi \\ \wedge\ (s, \phi) \in dom(F) \end{array} \right\}$$

The function $\texttt{afterTO}(s, t, m, t_0)$ computes the state that will be reached in X if we start in the state s assuming that the value of the memory is given by m and

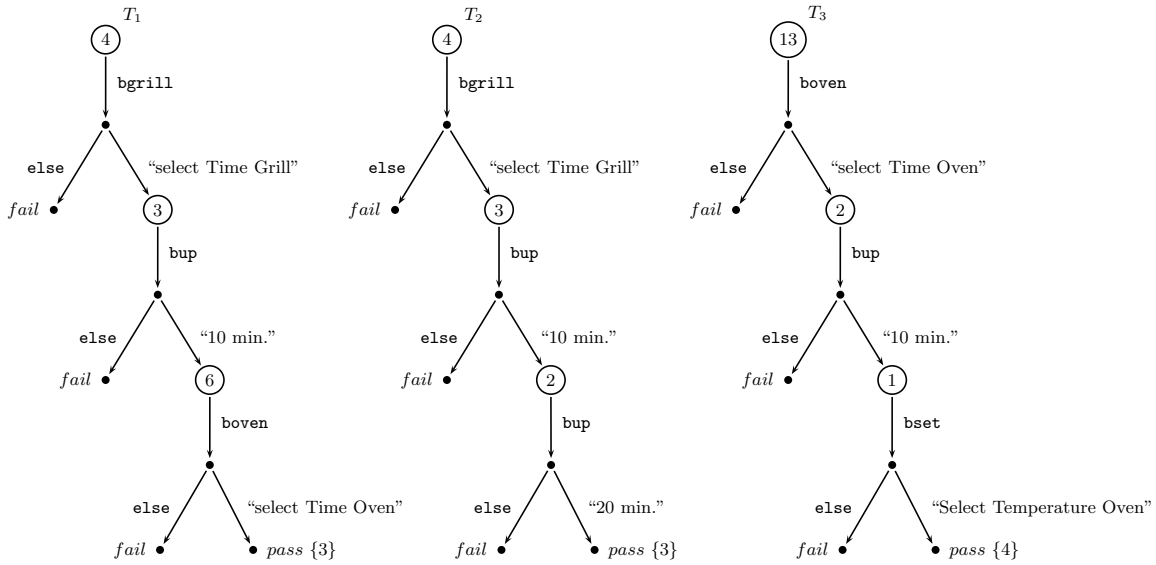


Figure 4 Examples of Tests.

t time units pass without receiving an input. The time needed to perform previous actions is given by t_0.

$$\texttt{afterTO}(s,t,m,t_0) = \begin{cases} (s,m,t_0) & \pi_2(TO(s)) > t \\ \texttt{afterTO}(\pi_1(TO(s)), t-\pi_2(TO(s)), m, t_0) & \text{otherwise} \end{cases}$$

The function $\texttt{after}(s,i,o,m,t)$ is used to compute all the states that can be reached from a state s after receiving the input i, producing the output o, for a value of the memory m, and assuming that t is the amount of time spent to perform previous actions. In addition, this function also returns the new value of the memory and the updated time consumed since the system started its performance.

$$\texttt{after}(s,i,o,m,t) = \left\{ (s',m',t') \;\middle|\; \begin{array}{l} \exists\ \phi \in \Phi : \\ ((m,i),(o,m',t'')) \in \phi \\ \wedge\ F(s,\phi) = s' \ \wedge\ t' = t + t'' \end{array} \right\}$$

The previously defined functions can be extended in the natural way to deal with sets:

$$\begin{aligned} \texttt{out}(G,i) &= \textstyle\bigcup_{(s,m)\in G} \texttt{out}(s,i,m) \\ \texttt{afterTO}(D,t) &= \{\texttt{afterTO}(s,t,m,t_o) \mid (s,m,t_o) \in D\} \\ \texttt{after}(D,i,o) &= \textstyle\bigcup_{(s,m,t)\in D} \texttt{after}(s,i,o,m,t) \end{aligned}$$

□

Let us recall that $TO(s)$ denotes the timeout associated with the state s, that is, a pair containing the reached state after the performance of the timeout and when the timeout will be triggered.

The algorithm to derive tests from a specification is given in Figure 5. This algorithm is non-deterministic and its application generates a single test. By considering all the possible non-deterministic choices in the algorithm we extract a full test suite from the specification. Let us remark that this set will be, in general, infinite. For a given specification S, we denote this test suite by $\texttt{tests}(S)$. Essentially, our algorithm consists in traversing the specification S in all the possible ways. Next, we explain how the algorithm works. The *set of pending situations* D contains triples (s,m,t) denoting the state that could have been reached in the transversal of the specification, the corresponding value of the memory, and the time registered up to this point. The auxiliary set S_{aux} keeps pairs (D, s^T) associating a state of the test, that is, s^T, with a set of pending situations, that is, D. Specifically, it indicates that we did not complete the state s^T of the test and the possible situations for completing it with information from the specification are given by the set D. Let us remark that D is a set of situations, instead of a single one, due to the non-determinism that can appear in the specification. Let us review the different steps of the algorithm. The set S_{aux} initially contains a tuple with the initial state of the specification, the initial value of the memory, zero time units, corresponding to the only possible pending situation, and the initial state of the test. For each tuple belonging to S_{aux} we have two possibilities. The first possibility simply indicates that the state of the test under construction becomes a pass state (case 1 of the algorithm). If the second possibility is chosen, then it must

Input: A specification $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$
Output: A test case $T = (S', I, O, Tr, s_{tin}, S_I, S_O, S_F, S_P, W, C)$.

Initialization:

- $S' := \{s_{in}\}, Tr := S_I := S_O := S_F := S_P := \emptyset$.
- $S_{aux} := \{(\{(s_{in}, m_{in}, 0)\}, s_{tin})\}$.

Cases: Choose one of the following two options until $S_{aux} = \emptyset$.

1. If $(D, s^T) \in S_{aux}$ then perform:
   (a) $S_P := S_P \cup \{s^T\}$.
   (b) $C(s^T) := \{t \mid (s, m, t) \in D\}$.
   (c) $S_{aux} := S_{aux} - \{(D, s^T)\}$.
2. If $S_{aux} = \{(D, s^T)\}$ and there exist $t_d \in \texttt{Time}, i \in I$ such that $\texttt{out}(S_M, i) \neq \emptyset$, where $S_M = \{(s, m) \mid (s, m, t) \in \texttt{afterTO}(D, t_d)\}$, then perform:
   (a) Choose $t_d \in \texttt{Time}$ and $i \in I$ fulfilling the previous conditions.
   (b) $D := \texttt{afterTO}(D, t_d)$.
   (c) $S_M := \{(s, m) \mid (s, m, t) \in D\}$.
   (d) $S_{aux} := \emptyset$.
   (e) Consider a fresh state $s' \notin S'$ and let $S' := S' \cup \{s'\}$.
   (f) $S_I := S_I \cup \{s^T\}$; $S_O := S_O \cup \{s'\}$; $Tr := Tr \cup \{(s^T, i, s')\}$.
   (g) $W(s^T) := t_d$.
   (h) For all $o \in O$ do
   - Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
   - $Tr := Tr \cup \{(s', o, s'')\}$
   - if $o \in \texttt{out}(S_M, i)$ then perform:
     $D' := \texttt{after}(D, i, o)$; $S_{aux} := S_{aux} \cup \{(D', s'')\}$.
     else $S_F := S_F \cup \{s''\}$.

Figure 5 Derivation of test cases from a specification.

be checked that there exists a delay t_d and an input i such that the specification can perform at least one output after applying the input i after the delay t_d (this is formalized in the side condition associated with the second inductive case). If this is the case, then we update the time values by considering that this delay is applied (step 2.b of the algorithm). Next, we generate an input transition in the test labeled by i and having as delay t_d (steps 2.e–g of the algorithm).

Then, the whole sets of outputs is considered to generate a new transition in the test for each of these outputs (step 2.h of the algorithm). If the output is not expected by the specification then a transition leading to a failing state is created. In order to simplify the representation, we can simulate all the transitions leading to a fail state simply by using an *else* branch. For expected outputs we create a transition with the corresponding output action. Finally, we update and add the appropriate tuple to the set S_{aux}, that is, the new pending situations after traversing the transitions corresponding to the input i and each of the expected outputs.

Example 5 Next we show how our test generation algorithm works. In Figure 4 we present some tests derived from the specification presented in Figure 1. We suppose that the initial memory value of the system is `(null,250,0,0,180,0,0)`.

We will explain the details to produce T_1. In order to generate the test T_1, a delay of 4 time units is applied in the step 2.a of the algorithm and the input `bgrill` is chosen. A transition labelled by this input is generated in the test. Since the initial state of the specification has associated timeout ∞, the oven will remain in the initial state. Then, the processing function f_selMode is considered in order to determine the expected output and the new memory value. The processing function f_selMode produces the output "select Time Grill" that is displayed at the screen and the memory value changes to `(GRILL,250,0,0,180,0,0)`. Thus, a transition for the output "select Time Grill" is created in the test (step 2.h of the algorithm). Moreover, another transition leading to a fail state is created for the rest of the outputs, that we represent by the label `else`. After this, a delay of 3 time units is established for the new input state and the input `bup` is selected. The only processing function that accepts the input `bup` is f_selTime that will change the memory to `(GRILL,250,0,0,180,10,0)` and emit the message "`10 min.`". After this, the corresponding transition for the rest of the outputs, leading to a fail state, is created in test. Next, a delay of 6 time units triggers the timeout associated with the *time* state and the machine changes the state to *start*. The input `boven` is selected. The appropriate transitions are created, and finally step 1 of the algorithm is applied in or-

der to conclude the generation of this test. Only one pass state is created. The attached set contains the time values associated with the different possibilities to perform the complete input/output sequence. Let us remind that we only consider the lapses, extracted from the specification, between reception of inputs and performance of outputs. Thus, time passing due to delays of the test is not taken into account to compute the values belonging to this set. In our case we have only one value since our specification needs always three time units, because all these lapses are equal to one time unit, to perform the corresponding sequence.

The difference between the tests T_1 and T_2 lies in the delays that have been considered before applying the third input: 6 and 2 time units, respectively. This fact implies that for the test T_2 the output "`20 min.`", produced by the function f_selTime, leads to a pass state.

The test T_3 corresponds to the selection of the OVEN mode. The processing function s_selMode displays the message "select Time Oven" and updates the internal memory to `(OVEN,250,0,0,180,0,0)`. After the user selects the time and presses the button `bset`, the only transition whose domain fits with the memory values is the one labelled by the function f_selTemp. Then, the message "select Temperature Oven" is displayed and the machine changes to the `temp` state. □

Let us note that finite tests are constructed simply by considering a step where the second case is not applied.

Let us comment on the *finiteness* of our algorithm. If we do not impose any restriction on the implementation (e.g. a bound on the number of states) we cannot determine some important information such as the maximal length of the sequences that the implementation can perform. In other words, we would need a *coverage criterion* to generate a finite test suite. Since we do not assume any criteria, all we can do is to say that the derived test suite is the (possibly infinite) suite that would allow us to prove completeness. Obviously, one can impose restrictions such as "generate n tests" or "generate all the tests with m inputs" and *completeness* will be obtained up to that coverage criterion.

Moreover, if we assume a bound n on the number of states of the SUT, and we consider specification having at most m states, then we can adapt [23] to the current framework to conclude that we can restrict ourselves, under certain conditions, to tests having at most $m \cdot n + 1$ inputs. The next result relates, for a specification S and an implementation I, conformance relations and application of test suites. Let us remark that the existence of different instances of the same timed evolution in the specification is the reason why only some tests (and for some time values) are forced to be passed by the implementation (e.g. sometimes we only need the *fastest/slowest* test). Specifically, we take those tests matching the requirements of the specific conformance relation. In this sense, the result holds because the temporal conditions required to conform to the specification and to pass the test suite are in fact the same.

Theorem 2 Let S and I be two `TSXM`s. We have that:

- I $\mathtt{conf}_a$ S iff I passes always in time $\mathtt{tests}(S)$.
- I $\mathtt{conf}_w$ S iff I passes in the worst time $\mathtt{tests}(S)$.
- I $\mathtt{conf}_b$ S iff I passes in the best time $\mathtt{tests}(S)$.

Proof: We will prove the first result. The technique is similar for the other two ones. First, let us show that I passes always in time $\mathtt{tests}(S)$ implies I $\mathtt{conf}_a$ S. We will use the contrapositive, that is, we will suppose that I $\mathtt{conf}_a$ S does not hold and we will prove that I does not pass $\mathtt{tests}(S)$ in the *always* sense. If I $\mathtt{conf}_a$ S does not hold, then we have two possibilities:

- Either I $\mathtt{conf}_f$ S does not hold, or
- there exists an instanced timed evolution e such that $e \in f_I^{\mathcal{T}}$ but $e \notin f_S^{\mathcal{T}}$.

Let us consider the first case, that is, we suppose that I $\mathtt{conf}_f$ S does not hold. Then, there exist $e = (\bar{t}, \sigma)$, with $\sigma = (i_1/o_1, \ldots, /i_r/o_r)$ and $\bar{t} = (t_1, \ldots, t_r)$, and $e' = (\bar{t}, \sigma')$, with $\sigma' = (i_1/o_1, \ldots, /i_r/o'_r)$, being $r \geq 1$ and such that $e \in f_S$, $e' \in f_I$, and $e' \notin f_S$. We will show that there exists a test $T \in \mathtt{tests}(S)$ such that $T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$, with $s^T \in S_P$, and $T \stackrel{\sigma'}{\Longrightarrow}_{\bar{t}} u^T$, with $u^T \in S_F$. By constructing such a test T we obtain $I \parallel T \stackrel{\sigma'}{\Longrightarrow}_{\bar{t}} u^T$, for a fail state u^T. Thus, we conclude S does not pass $\mathtt{tests}(S)$. We will build this test T by applying the algorithm given in Figure 5. The different non-deterministic choices underlying the algorithm will be resolved in the following way:

1. for $1 \leq j \leq r$ do
   (a) Apply case 2 for the input action i_j and assign the time value t_j to t_d.
   (b) Apply case 1 to all the elements $(D, s^T) \in S_{aux}$ that are obtained by processing an output different to o_j.
   endfor
2. Apply the first case to the last element $(D, s^T) \in S_{aux}$.

Let us remark that Step 1.a corresponds to continue the construction of the test in the state reached by the transition labeled by o_{j-1} (in the case of $j = 1$ we mean the initial state of the test). Let us also note that the *spine* of the constructed test is the sequence σ.

Since $e' \notin f_S$, the last application of the second case for the output o'_r generates a test T such that $T \stackrel{\sigma'}{\Longrightarrow}_{\bar{t}} u^T$, with $u^T \in S_F$ (else branch of the third item of 2.f). Since $e' \in f_I$ we have that $I \parallel T \stackrel{\sigma'}{\Longrightarrow}_{\bar{t}} u^T$. Given the fact that $T \in \mathtt{tests}(S)$ we deduce that $\mathtt{pass}(I, \mathtt{tests}(S))$ does not hold. Thus, we conclude I does not pass $\mathtt{tests}(S)$ in the always sense.

Let us suppose now that I $\mathtt{conf}_a$ S does not hold because there exists $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$ such that $(\bar{t}, \sigma) \in f_S$ and $e \notin f_S^{\mathcal{T}}$. Since $(\bar{t}, \sigma) \in f_S$, let us consider the

test T we built before. We have again $T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$, with $s^T \in S_P$. The time stamps associated with the state s^T are generated by considering all the possible time values in which σ could be performed in S by considering the delays contained in $\bar{t}$. Besides, since $e \in f_I^{\mathcal{T}}$, we also have $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\bar{t},to} s^T$. Thus, if $e \notin f_S^{\mathcal{T}}$ then $t_o \notin C(s^T)$. We conclude I does not pass $\texttt{tests}(S)$ in the always sense.

Let us show now that $I\ \texttt{conf}_a\ S$ implies I passes $\texttt{tests}(S)$ in the always sense. Again by contrapositive, we will assume that I does not pass $\texttt{tests}(S)$ in the always sense and we will conclude that $I\ \texttt{conf}_a\ S$ does not hold. If I does not pass $\texttt{tests}(S)$ in the always sense, then we have two possibilities:

- Either $\texttt{pass}(I, \texttt{tests}(S))$ does not hold, or
- there exist an instanced timed evolution $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$ and a state $s^T \in \texttt{Fin\_State}(e, \texttt{tests}(S))$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\bar{t},t_o} s^T$, with $s^T \in S_P$ and $t_o \notin C(s^T)$.

First, let us assume that I does not pass $\texttt{tests}(S)$ in the always sense because $\texttt{pass}(I, \texttt{tests}(S))$ does not hold. Thus, there exists a test $T \in \texttt{tests}(S)$, $s^T \in S_F$, $\sigma = (i_1/o_1, \ldots, i_r/o_r)$, and $\bar{t}$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$. Therefore, $T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$. According to our derivation algorithm, a branch of a derived test leads to a fail state only if its associated output action is not expected in the specification. Thus, $e = (\bar{t}, \sigma) \notin f_S$. Let us note that our algorithm allows to create a fail state only as the result of the application of the second inductive case. One of the premises of this inductive case is $\texttt{out}(S_M, i) \neq \emptyset$, that is, the specification is allowed to perform some output actions after the reception of the corresponding input. Thus, there exists an output action o'_r and a sequence $\sigma' = (i_1/o_1, \ldots, i_{r-1}/o_{r-1}, i_r/o'_r)$ such that $e' = (\bar{t}, \sigma') \in f_S$. Given the fact that $e \in f_I$, $e \notin f_S$, and $e' \in f_S$, we have that $I\ \texttt{conf}_f\ S$ does not hold. We conclude $I\ \texttt{conf}_a\ S$ does not hold.

Let us suppose now that I does not pass $\texttt{tests}(S)$ in the always sense because there exist $e = (\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$, $T \in \mathcal{T}$, and a state $s^T \in \texttt{Fin\_State}(e, \texttt{tests}(S))$ of T such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\bar{t},t_o} s^T$, with $s^T \in S_P$ and $t_o \notin C(s^T)$. Since $s^T \in S_P$ we deduce $(\bar{t}, \sigma) \in f_S$. Besides, by considering that $t_o \notin C(s^T)$, we deduce $(\bar{t}, \sigma, t_o) \notin f_S^{\mathcal{T}}$. Finally, by using $(\bar{t}, \sigma, t_o) \in f_I^{\mathcal{T}}$, we conclude that $I\ \texttt{conf}_a\ S$ does not hold. □

7 Conclusions and future work

We have presented a formal specification and testing framework for timed systems where timeouts are critical and the lapse between the reception of inputs and the performance of outputs is quantified. The model introduced for specifying the systems is a suitable extension of the classical Stream X-machines formalism. It is important to remark that in this paper we remove the usual limitations considering that implementations and/or specifications are deterministic. This fact is relevant since it complicates the posterior theoretical development. Three conformance relations have been introduced to define correctness of a system with respect to a specification. We have introduced a notion of test that can delay the execution of the tested system as well as to compare the time that the tested system needs to perform a certain action with a set of time values. Finally, we have presented an algorithm to derive sound and complete test suites with respect to the three conformance relations presented in this paper.

In terms of future work, we have already identified two separate lines. First, we would like to take this work as a first step, together with [34], to define a testing theory for systems presenting timeouts and using stochastic time, instead of fix time, to quantify the time that pass between receiving an input and performing an output. The second line is to study different methods for reducing the size of the generated test suite. We plan to consider the adaption of *state counting* [41,40], to deal with conformance, recently introduced in [23] and adapt it to the context of TSXMs, more specifically, to its stochastic extension.

Acknowledgments

We would like to thank the anonymous reviewers of [33] for their useful comments that have helped to improve the quality of this paper.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. J.C.M. Baeten and C.A. Middelburg. *Process algebra with timing*. EATCS Monograph. Springer, 2002.
3. J. Barnard. COMX: A design methodology using communicating X–machines. *Information and Software Technology*, 40(5–6):271–280, 1998.
4. S.S. Batth, E. Rodrigues Vieira, A. Cavalli, and M.Ü. Uyar. Specification of timed EFSM fault models in SDL. In *27th IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'07, LNCS 4574*, pages 50–65. Springer, 2007.
5. K. Bogdanov, M. Holcombe, F. Ipate, L. Seed, and S. Vanak. Testing methods for X-machines: a review. *Formal Aspects of Computing*, 18:3–30, 2006.
6. B.S. Bosik and M.Ü. Uyar. Finite state machine based formal methods in protocol conformance testing. *Computer Networks & ISDN Systems*, 22:7–33, 1991.
7. L. Brandán Briones and E. Brinksma. Testing real-time multi input-output systems. In *7th Int. Conf. on Formal Engineering Methods, ICFEM'05, LNCS 3785*, pages 264–279. Springer, 2005.
8. E. Brinksma and J. Tretmans. Testing transition systems: An annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 187–195. Springer, 2001.

9. R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, 2000.
10. R. Cardell-Oliver and T. Glover. A practical and complete algorithm for testing real-time systems. In *5th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT'98, LNCS 1486*, pages 251–260. Springer, 1998.
11. T.S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.
12. D. Clarke and I. Lee. Automatic generation of tests for timing constraints from requirements. In *3rd Workshop on Object-Oriented Real-Time Dependable Systems, WORDS'97*, pages 199–206. IEEE Computer Society Press, 1997.
13. J. Davies and S. Schneider. A brief history of timed CSP. *Theoretical Computer Science*, 138:243–271, 1995.
14. K. El-Fakih, N. Yevtushenko, and G. von Bochmann. FSM-based incremental conformance testing methods. *IEEE Transactions on Software Engineering*, 30(7):425–436, 2004.
15. A. En-Nouaary, R. Dssouli, and F. Khendek. Timed Wp-method: Testing real time systems. *IEEE Transactions on Software Engineering*, 28(11):1024–1039, 2002.
16. H. Fouchal, E. Petitjean, and S. Salva. An user-oriented testing of real time systems. In *IEEE Workshop on Real-Time Embedded Systems, RTES'01*. IEEE Computer Society Press, 2001.
17. M. Hennessy and T. Regan. A process algebra for timed systems. *Information and Computation*, 117(2):221–239, 1995.
18. A. Hessel, K.G. Larsen, M. Mikucionis, B. Nielsen, P. Pettersson, and A. Skou. Testing real-time systems using UPPAAL. In *Formal Methods and Testing, LNCS 4949*, pages 77–117. Springer, 2008.
19. R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys*, 41(2), 2009.
20. R.M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.
21. R.M. Hierons and M. Harman. Testing conformance of a deterministic implementation to a non-deterministic stream X-machine. *Theoretical Computer Science*, 323(1–3):191–233, 2004.
22. R.M. Hierons and F. Ipate. Testing a deterministic implementation against a non-controllable non-deterministic stream X-machine. *Formal Aspects of Computing*, 20(6):597–617, 2008.
23. R.M. Hierons, M.G. Merayo, and M. Núñez. Testing from a stochastic timed system with a fault model. *Journal of Logic and Algebraic Programming*, 78(2):98–115, 2009.
24. T. Higashino, A. Nakata, K. Taniguchi, and A. Cavalli. Generating test cases for a timed I/O automaton model. In *12th Int. Workshop on Testing of Communicating Systems, IWTCS'99*, pages 197–214. Kluwer Academic Publishers, 1999.
25. M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer, 1998.
26. F. Ipate. Testing against a non-controllable stream X-machine using state counting. *Theoretical Computer Science*, 353(1):291–316, 2006.
27. F. Ipate and M. Holcombe. An integration testing method that is proved to find all faults. *International Journal of Computer Mathematics*, 63(3-4):159–178, 1997.
28. P. Kefalas, G. Eleftherakis, and E. Kehris. Communicating X-machines: a practical approach for formal and modular specification of large systems. *Information and Software Technology*, 45(5):269–280, 2003.
29. E. Kehris, G. Eleftherakis, and P. Kefalas. Using X–machines to model and test discrete event simulation programs. In N. Mastorakis, editor, *Systems and Control: Theory and Applications*, pages 163–171. World Scientific and Engineering Society Press, 2000.
30. M. Krichen and S. Tripakis. An expressive and implementable formal framework for testing real-time systems. In *17th Int. Conf. on Testing of Communicating Systems, TestCom'05, LNCS 3502*, pages 209–225. Springer, 2005.
31. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
32. D. Mandrioli, S. Morasca, and A. Morzenti. Generating test cases for real time systems from logic specifications. *ACM Transactions on Computer Systems*, 13(4):356–398, 1995.
33. M.G. Merayo, R.M. Hierons, and M. Núñez. Extending stream X-machines to specify and test systems with timeouts. In *6th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM'08*, pages 201–210. IEEE Computer Society Press, 2008.
34. M.G. Merayo and M. Núñez. Testing conformance on stochastic stream X-machines. In *5th IEEE Int. Conf. on Software Engineering and Formal Methods, SEFM'07*, pages 227–236. IEEE Computer Society Press, 2007.
35. M.G. Merayo, M. Núñez, and I. Rodríguez. Extending EFSMs to specify and test timed systems with action durations and timeouts. *IEEE Transactions on Computers*, 57(6):835–848, 2008.
36. M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.
37. X. Nicollin and J. Sifakis. An overview and synthesis on timed process algebras. In *3rd Int. Conf. on Computer Aided Verification, CAV'91, LNCS 575*, pages 376–398. Springer, 1991.
38. J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.
39. A. Petrenko. Fault model-driven test derivation from finite state models: Annotated bibliography. In *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*, pages 196–205. Springer, 2001.
40. A. Petrenko, N. Yevtushenko, and G. von Bochmann. Testing deterministic implementations from their nondeterministic FSM specifications. In *9th IFIP Workshop on Testing of Communicating Systems, IWTCS'96*, pages 125–140. Chapman & Hall, 1996.
41. A. Petrenko, N. Yevtushenko, A.V. Lebedev, and A. Das. Nondeterministic state machines in protocol conformance testing. In *6th IFIP Workshop on Protocol*

Test Systems, IWPTS'93, pages 363–378. North Holland, 1993.
42. J. Quemada, D. de Frutos, and A. Azcorra. TIC: A TImed Calculus. *Formal Aspects of Computing*, 5:224–252, 1993.
43. G.M. Reed and A.W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science*, 58:249–261, 1988.
44. I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.
45. J. Schmaltz and J. Tretmans. On conformance testing for timed systems. In *6th Int. Conf. on Formal Modeling and Analysis of Timed Systems, FORMATS'08, LNCS 5215*, pages 250–264. Springer, 2008.
46. J. Sifakis. Use of Petri nets for performance evaluation. In *3rd Int. Symposium on Measuring, Modelling and Evaluating Computer Systems*, pages 75–93. North-Holland, 1977.
47. J. Springintveld, F. Vaandrager, and P.R. D'Argenio. Testing timed automata. *Theoretical Computer Science*, 254(1-2):225–257, 2001. Previously appeared as Technical Report CTIT-97-17, University of Twente, 1997.
48. J. Tretmans. Model based testing with labelled transition systems. In *Formal Methods and Testing, LNCS 4949*, pages 1–38. Springer, 2008.
49. M.Ü. Uyar, S.S. Batth, Y. Wang, and M.A. Fecko. Algorithms for modeling a class of single timing faults in communication protocols. *IEEE Transactions on Computers*, 57(2):274–288, 2008.
50. M.Ü. Uyar, M.A. Fecko, A.S. Sethi, and P.D. Amar. Testing protocols modeled as FSMs with timing parameters. *Computer Networks*, 31(18):1967–1998, 1999.
51. W. Yi. CCS+ Time = an interleaving model for real time systems. In *18th Int. Colloquium on Automata, Languages and Programming, ICALP'91, LNCS 510*, pages 217–228. Springer, 1991.