2008 Sixth IEEE International Conference on Software Engineering and Formal Methods

Extending Stream X-Machines to specify and test systems with timeouts*

Mercedes G. Merayo
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain
mgmerayo@fdi.ucm.es

Robert M. Hierons
Department of Information Systems and Computing
Brunel University, Uxbridge, Middlesex, UB8 3PH United Kingdom
rob.hierons@brunel.ac.uk

Manuel Núñez
Departamento de Sistemas Informáticos y Computación
Universidad Complutense de Madrid, Madrid, Spain
mn@sip.ucm.es

Abstract

Stream X-machines are a kind of extended finite state machine used to specify real systems where communication between the components is modeled by using a shared memory. In this paper we introduce an extension of the Stream X-machines formalism in order to specify delays/timeouts. 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. So, a relation focusing on functional aspects 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.

1 Introduction

Real-time can be found in several branches of the industrial world, such as aeronautics, medicine, and transportation, where reliability is a must since human lives are involved. In order to improve the reliability of the developed systems, it is necessary to apply formal methodologies that consider time requirements. In most models, temporal requirements usually refer to time that a system consumes for 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 with it; if the period ends and no action has been produced, the state of the system changes and its reactions to the actions that are received from the environment may be different.

Formal methods allow us not only to represent with precision the systems that we are going to study but also to reason about them with mathematical precision and rigor. It is widely recognized that formal methods and testing are complementary techniques ([22, 30, 5, 31, 12]) since they help to check the correctness of systems and provide a framework for testing. The application of formal testing techniques requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behavior. While the relevant aspects of some systems only concern *what* they do, in some other systems it is equally relevant *how* they do what they do. 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. [23, 9, 15, 32, 10, 28, 21, 4, 26]).

One approach to formally specify systems is to use X-machines [16]. They can be seen as a form of finite state machine where the transitions are labeled with relations over a basic data set. This set of relations is called the

*Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01) and the Marie Curie project MRTN-CT-2003-505121/TAROT.

978-0-7695-3437-4/08 $25.00 © 2008 IEEE
DOI 10.1109/SEFM.2008.15

type of the machine and represents the operations that can perform. Several classes of X-machines have been defined and studied, in particular, Stream X-machines. Stream X-machines have an internal memory and the input, the current state, and the current value of the internal memory determine the next state, the output and how the memory is updated. The main advantages of using this model for representing systems is that it is flexible and allows the integration of control and data processing. This formalism has been used to specify systems in different areas ([17, 2, 20, 19]) and several testing techniques have been developed.

The standard Stream X-machines test generation algorithm is based on the W-method introduced by [8] in the context of finite state machines. The method presents some restrictions: (a) specification and implementation must be deterministic (b) the functions associated with the transitions are correctly implemented and (c) it assumes that certain 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 [18, 17]. This integration of specification and test generation is one of the most significant benefits of using deterministic stream X-machines in software development. Since this method was developed, it has been extended for reducing the imposed conditions (for a survey see [3]).

In this paper we study systems presenting timeouts. In order to concentrate on the special features introduced by timeouts, we do not explicitly consider other time aspects. For example, it is usual to associate time with the performance of actions. Nevertheless, once we have a model where timeouts are included it is not technically difficult to extend it to consider also time durations. We consider a suitable extension of the classical Stream X-machine model to allow a specifier to explicitly denote timeouts of a system, taking advantage of the power that this model provides. Moreover, 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 (required to apply [8]). 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.

We assume that specifications and implementations can be modeled by means of Stream X-machines with timeouts, and propose a formal testing methodology to systematically test a system with respect to a specification. Since we remove most of the restrictions on machines, we cannot apply the classical testing framework for Stream X-machines. Specifically, we consider the testing framework [25] in the context of the the machines presented in this paper. Testing a system requires using a specific notion of correctness. We have to consider that the sequences of inputs/outpus produced by the system under test are allowed in the specification, that is, the implementation does not *invent* anything for those inputs that are *specified* in the specification. In addition, we have to take into account the possible timeouts. For example, a sequence of inputs/outputs could be accepted only after different timeouts have been triggered.

The rest of the paper is organized as follows. In Section 2 we introduce our model to represent Stream X-machines with timeouts. In Section 3 we introduce a notion of conformance for our framework. In Section 4 we show how our machines can be tested. In Section 5 we give a test derivation algorithm and we prove that the derived test suite is sound and complete. In Section 6 we review previous works on testing timed systems. Finally, in Section 7 we present our conclusions.

2 Extending Stream X-machines with timeouts

In this section we introduce our extension of the classical stream X-machine model in order to deal with systems that can evolve by raising *timeouts*. Intuitively, if after a given amount of time and depending on the current state, we do not receive any 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 of a system can be properly specified. We use a discrete time domain to model *timeouts*. In particular, we will sometimes enumerate the elements of `Time` simply as $0, 1, 2$ and so on. During the rest of the paper we will use the following notation.

Definition 1 A tuple of elements $(a_1, a_2 \ldots, a_n)$ will be denoted by $\bar{a}$. We say that $\hat{a} = [a_1, a_2)$ is an interval if $a_1, a_2 \in \mathbb{N}$ and $a_1 \leq a_2$. A tuple of intervals $(\hat{p_1}, \ldots, \hat{p_n})$ will be denoted by $\breve{p}$. Let $\bar{t} = (t_1, \ldots, t_n)$ 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 $t_j \in \hat{q_j}$;
- $\pi_i(\bar{t})$, for all $1 \leq i \leq n$, denotes the value t_i.

□

Definition 2 A *Timeout stream X-machine*, in short `TOSXM`, 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 from $M \times I$ to $O \times M$ called 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 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 state of the transition, and $\phi \in \Phi$ is the processing function associated with the transition. □

Intuitively, we can think of a TOSXM as a state transition diagram where arcs are labelled by processing functions. Each function receives an input and current memory values and produces an output modifying the memory.

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 action, and the state to which the machine evolves if no input is received in time. 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 ∞. Next we present some definitions regarding stream X-machines that will be used throughout the paper.

Definition 3 A TOSXM $X = (I, O, S, M, \Phi, F, s_{in}, m_{in})$ 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 $(s, \phi) \in dom(F)$. □

A TOSXM is deterministic if given a state s, for any input value and any memory value there is only one processing function that can be applied. In turn, a TOSXM is *completely specified* if for all $s \in S, m \in M$, and $i \in I$ there is always a possible transition. If a deterministic TOSXM is completely specified then there is only one transition for any $s \in S, m \in M$, and $i \in I$. Next we introduce a partial function that establishes the relation between a pair (memory values, input sequence) and a pair (output sequence, updated memory values) produced by the application of a sequence of processing functions.

Definition 4 Let $X = (I, O, S, M, \Phi, F, s_{in}, m_{in})$ be a TOSXM. Given a sequence $\bar{\phi} \in \Phi^*$, we consider $\| \bar{\phi} \|$: $M \times I^* \rightarrow O^* \times M$ to be a partial function inductively defined in the following way:

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

$$\| \bar{\phi}\phi \| = \left\{ ((m, \bar{i}i), (\bar{o}o, m')) \;\middle|\; \begin{array}{l} \exists\, m'' \in M : \\ ((m, \bar{i}), (\bar{o}, m'')) \in \| \bar{\phi} \| \\ \wedge\, ((m'', i), (o, m')) \in \phi \end{array} \right\}$$

□

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 TOSXM each computation from the initial state to any other state is completely determined by the input sequence and the initial memory value. The following notion allow us to compose several transitions possibly preceded by timeouts.

Definition 5 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TOSXM. A tuple $(s_0, \phi, s, \hat{t})$ is a *step* of X if there exist $k \geq 0$ states $s_1, \ldots, s_k \in S$, such that for all $1 \leq j \leq k$ we have $TO(s_{j-1}) = (s_j, t_j)$, $F(s_k, \phi) = s$ and $\hat{t} = \left[\sum_{j=1}^{k} t_j, \sum_{j=1}^{k} t_j + \pi_2(TO(s_k))\right)$.

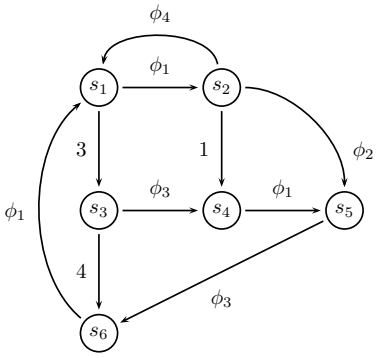
We say that $(\hat{t}_1/\phi_1, \ldots, \hat{t}_r/\phi_r)$ is an *evolution* of X if there exist r steps of X $(s_{in}, \phi_1, s_1, \hat{t}_1), \ldots, (s_{r-1}, \phi_r, s_r, \hat{t}_r)$. We denote the set of evolutions of X by $\texttt{Evol}(X)$. □

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 processing functions 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 implementation relation, 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 processing function sequences that TOSXMs can perform. This information is encoded into the intervals.

We will sometimes refer to the tuple $(\hat{t}_1/\phi_1, \ldots, \hat{t}_r/\phi_r) \in \texttt{Evol}(X)$ as $(\breve{t}, \bar{\phi})$, where $\breve{t} = (\hat{t}_1, \ldots, \hat{t}_r)$ and $\bar{\phi} = (\phi_1, \ldots, \phi_r)$.

Example 1 Let us consider the machine depicted in Figure 1 in which the initial state is s_1. Next, we give some of the steps that the machine can generate. For example, $(s_1, \phi_1, s_2, [0, 3))$, represents a transition where no timeouts precede it. The processing function ϕ_1 can be performed before 3 time units pass (this is indicated by the interval $[0, 3)$), if an input belonging to its domain is received. The second example, $(s_1, \phi_3, s_4, [3, 7)))$ is built from the timeout associated to the state s_1 and the transition outgoing from s_3. The step represents that if the machine is at state s_1 and after 3 time units no input is received then the timeout associated with that state will be triggered and the state will change to s_3. After this, the machine can accept an input that belongs to the domain of the processing function ϕ_3 before 4 time units pass, that is, the timeout assigned to the state s_3. Therefore, during the time interval $[3, 7)$ if the machine receives an appropriate input it will emit an output and the state will change to s_4. Similarly, we can obtain the step $(s_1, \phi_1, s_1, [7, \infty))$, using the timeouts corresponding to s_1 and s_3 and the transition outgoing from s_6.

Now, we present an example of evolution built from two steps: $([7, \infty)/\phi_1, [3, 7)/\phi_3)$. □



$I = \{a, b\}$ $\quad S = \{s_1, \ldots, s_6\}$
$O = \{x, y, z\}$ $\quad \Phi = \{\phi_1, \phi_2, \phi_3, \phi_4\}$
$M = \{1, 0\}$ $\quad s_{in} = s_1\ m_{in} = 0$

$\phi_1(0, a) = (z, 1)$	$\phi_3(0, a) = (x, 1)$
$\phi_1(0, b) = (y, 1)$	$\phi_3(0, b) = (y, 0)$
$\phi_1(1, a) = (x, 0)$	$\phi_3(1, a) = (z, 0)$
$\phi_1(1, b) = (x, 1)$	$\phi_3(1, b) = (x, 0)$
$\phi_2(1, a) = (z, 0)$	$\phi_4(0, b) = (y, 1)$
$\phi_2(1, b) = (z, 0)$	$\phi_4(0, a) = (x, 1)$

F and TO are graphically represented.

Figure 1. Example of Timeout Stream X-machine.

Definition 6 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TOSXM and $e = (\hat{t}_1/\phi_1, \ldots, \hat{t}_r/\phi_r)$ be an *evolution* of X. We say that the tuple $(t_1/\phi_1, \ldots, t_r/\phi_r)$ is an *instanced evolution of* e if for all $1 \leq j \leq r$ we have $t_j \in \hat{t}_j$.

We denote by $\texttt{InsEvol}(X)$ the set of instanced evolutions of X. □

Instanced evolutions will be used to consider the concrete time when actions are performed. We will sometimes refer to the tuple $(t_1/\phi_1, \ldots, t_r/\phi_r) \in \texttt{InsEvol}(X)$ as $(\bar{t}, \bar{\phi})$, where $\bar{t} = (t_1, \ldots, t_r)$ and $\bar{\phi} = (\phi_1, \ldots, \phi_r)$.

Example 2 If we consider the evolution $([7, \infty)/\phi_1, [3, 7)/\phi_3)$ showed in the previous example we have that the tuples $(8/\phi_1, 5/\phi_3)$ and $(12/\phi_1, 3/\phi_3)$ are two of its instanced evolutions. □

A TOSXM gives rise to a relation between the input sequences applied to the machine and the output sequences that it produces. 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, for dealing with the specified timeouts, to extend this relation to take into account the times when the machine receives an interaction from the environment. Thus, we introduce a new notion of correspondence between input sequences, time sequences and output sequences.

Definition 7 Let $X = (I, O, S, M, \Phi, F, s_{in}, m_{in})$ be a deterministic TOSXM. The *function computed* by X $f_X : I^* \times \texttt{Time}^* \rightarrow O^*$ is defined as follows:

$$f_X = \left\{ (\bar{i}, \bar{o}, \bar{t}) \;\middle|\; \begin{array}{l} \exists\, m \in M, \bar{\phi} \in \Phi^* : \\ (\bar{t}, \bar{\phi}) \in \texttt{InsEvol}(X) \wedge \\ ((m_{in}, \bar{i}), (\bar{o}, m)) \in \| \bar{\phi} \| \end{array} \right\}$$

□

Let us note that if the machine X is deterministic and completely specified then f_X is total.

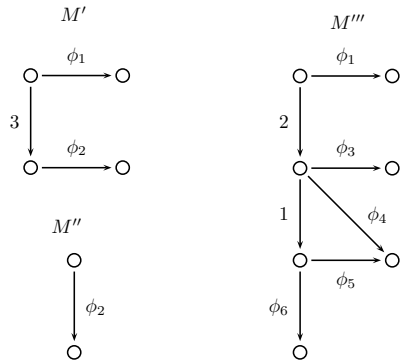
3 A notion of conformance

In order to properly define how to test an implementation against a specification it is necessary to state what it means for an implementation to conform to a specification. It is usual when testing from a stream X-machine to assume that the implementation under test (IUT) behaves like an unknown stream X-machine. As usual we will assume that both, implementation and specification, are given by completely specified and deterministic TOSXMs, but the result can be extended to deal with non-deterministic machines by adapting [13] to the framework developed in this paper.

Next, we introduce the implementation relation $\texttt{conf}_f$. 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 implementation 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. This is illustrated in the following example.

Example 3 Let us consider the schematic machines depicted in Figure 2. These diagrams represent simplified TOSXMs. 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 will have that M' is not conforming to M''. If an input is received before 3 units of time pass, the function



$I = \{a\}$
$O = \{x, y, z\}$
$M = \{1, 0\}$

$\phi_1(1, a) = (y, 0)$	$\phi_1(0, a) = (y, 0)$
$\phi_2(1, a) = (x, 0)$	$\phi_2(0, a) = (z, 0)$
$\phi_3(1, a) = (y, 0)$	$\phi_4(0, a) = (y, 1)$
$\phi_5(1, a) = (x, 1)$	$\phi_6(0, a) = (z, 0)$

Figure 2. Examples of conformance.

ϕ_1 will be performed by M'. However, after the same delay M'' will perform ϕ_2. The outputs produced by the machines will be different for any memory value. We can say the same regarding the conformance of M'' with respect to M'. The function ϕ_2 is allowed by M' only in the case that the input has been received after 3 time units. So, under our conformance framework, M'' does not conform to M'. On the contrary, this is not the case when considering the conformance of M' with respect to M'''. The outputs produced by M' when it receives the input a are equal to the ones accepted by M''' at any time. So, M' conforms to M'''. □

Definition 8 Let S and I be two TOSXMs. We say that I *conforms* to S, denoted by I $\mathtt{conf}_f$ S, if $f_I = f_S$. □

4 Definition and application of tests

In our setting, tests represent sequences of inputs applied to an IUT. Once an output is received, it is necessary to check that whether it belongs to the set of expected ones or not. In addition to checking the functional behavior of the IUT, tests have also to detect whether wrong timed behaviors appear. 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 IUT by offering input actions after a specific delay. Let us note that we cannot observe when the IUT takes a timeout. However, it is still possible to check the IUT behavior after different delays.

Definition 9 A *test* is a tuple $T = (S, I, O, Tr, s_0, S_I, S_O, S_F, S_P, W)$ where S is the set of states, I and O are disjoint sets of input and output actions, 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 states $s \in S_I$ 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 states $s \in S_O$ 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 $(s, a, s') \in Tr$.

Finally, the function $W : S_I \longrightarrow \mathtt{Time}$ associates delays with input 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 $\overline{t} \in \mathtt{Time}^r$. 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 3 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.

Definition 10 Let I be a TOSXM, T be a valid test, s^T be a state of T, $\sigma = (\overline{i}, \overline{o})$ where $\overline{i} = (i_1, \ldots, i_r)$ and $\overline{o} =$

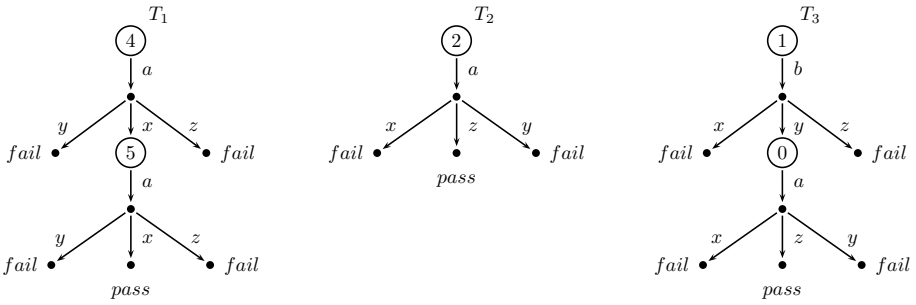


Figure 3. Examples of Tests.

$(o_1, \ldots, o_r)$, 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 $(\overline{t}, \sigma) \in f_I$. We say that

- I *passes* the test suite $\mathcal{T}$, denoted by $\texttt{pass}(I, \mathcal{T})$, if for all test $T \in \mathcal{T}$ there do not exist σ, s^T, $\overline{t}$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\overline{t}} s^T$ and $s^T \in S_F$.

□

5 Test derivation

In this section we present an algorithm to derive tests from specifications. As usual, 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 function will be used in the forthcoming derivation algorithm.

Definition 11 Let $X = (I, O, S, M, \Phi, F, TO, s_{in}, m_{in})$ be a TOSXM. The function $\texttt{afterTO}(s, t)$ computes the state that will be reached in X if we start in the state s and t time units pass without receiving an input.

$\texttt{afterTO}(s, t) =$

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

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 4. 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 X, we denote this test suite by $\texttt{tests}(X)$. Essentially, our algorithm consists in traversing the specification X in all the possible ways. Next we explain how the algorithm works. The *pending situation* (s^X, m^X, s^T) keeps a triple denoting the state that could have been reached in the transversal of the specification and the corresponding value of memory that could appear in a state of the test whose definition (that is, the construction of its outgoing transitions) has not been yet completed. Specifically, it indicates that we did not complete the state s^T of the test and the situation in the specification is given by the state s^X and the memory value m^X. Let us consider the different steps of the algorithm. The triple (s^X, m^X, s^T) initially contains the initial situation of the specification (that is, the initial state and the initial value of the memory) and the initial state of the test. We have two possibilities (under the heading *cases*). The first possibility simply indicates that the state of the test under construction becomes a passing state (case 1 of the algorithm). If the second possibility is chosen then it has to be checked that there exists a delay t_d, a processing function ϕ, and an input i such that the specification can perform an output after applying the input i after the delay t_d for the current memory value (this is formalized in the side condition associated with the second case). If this is the case, we generate an input transition in the test labelled by i and having as delay t_d (steps 2.b–d of the algorithm). Then, the whole sets of outputs is considered to generate a new transition in the test for each of these outputs. If the output is not expected by the specification (step 2.e of the algorithm) then a transition leading to a failing state is created. This could be simulated by a single branch in the test, labelled by `else`, leading to a failing state. For the expected output (step 2.f–g of the algorithm) we create a transition with the corresponding output action. Finally, we update (s^X, m^X, s^T) (step 2.h of the algorithm), that is, the new

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

Initialization:

- $S' := \{s_{in}\}, Tran' := S_I := S_O := S_F := S_P := \emptyset$.
- $(s^X, m^X, s^T) := (s_{in}, m_{in}, s_{tin})$.

Cases: Choose one of the following two options until $(s^X, m^X, s^T) = null$.

1. Perform:
   (a) $S_P := S_P \cup \{s^T\}$.
   (b) $(s^X, m^X, s^T) := null$.
2. If $\exists\ t_d \in \texttt{Time}, \phi \in \Phi, i \in I$ such that $(s_M, \phi) \in dom(F)$ and $(m^X, i) \in dom(\phi)$ with $s_M = \texttt{afterTO}(s^X, t_d)$, then perform:
   (a) Choose $t_d \in \texttt{Time}$ and $i \in I$ fulfilling the previous conditions.
   (b) $s_M := \texttt{afterTO}(s^X, t_d)$; $W(s^T) := t_d$.
   (c) Consider a fresh state $s' \notin S'$ and let $S' := S' \cup \{s'\}$.
   (d) $S_I := S_I \cup \{s^T\}$; $S_O := S_O \cup \{s'\}$; $Tran' := Tran' \cup \{(s^T, i, s')\}$.
   (e) For all $o \neq \pi_1(\phi(m^X, i))$ do
      - Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
      - $S_F := S_F \cup \{s''\}$; $Tran' := Tran' \cup \{(s', o, s'')\}$.
   (f) Consider a fresh state $s'' \notin S'$ and let $S' := S' \cup \{s''\}$.
   (g) $Tran' := Tran' \cup \{(s', \pi_1(\phi(m^X, i)), s'')\}$.
   (h) $(s^X, m^X, s^T) := (F(s_M, \phi), \pi_2(\phi(m^X, i)), s'')$.

Figure 4. Derivation of test cases from a specification.

pending situation after traversing the transition corresponding to the processing function ϕ. Let us note that finite tests are constructed simply by considering a step where the second case is not applied.

Example 4 Next we show how our test generation algorithm works. In Figure 3 we present some tests derived from the specification presented in Figure 1. We suppose that the initial memory value of the TOSXM is 0.

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 a is chosen. A transition labelled by this input is generated in the test. Due to the timeout associated with the initial state, the machine would have changed the state to s_3. Then, the processing function ϕ_3 is considered in order to determine the expected output and the new memory value. The processing function ϕ_3 produces the output x when a is applied and the memory value is 0. Thus, a transition for the output x is created in the test (step 2.f–g of the algorithm). Moreover, two transitions leading to a fail state are created for the outputs y and z respectively (step 2.e of the algorithm). After this, a delay of 5 time units is established for this input state and the input a is selected again. Then, the corresponding transitions are created in the test for the accepted/forbidden outputs in the specification. Finally, the step 1 of the algorithm is applied in order to conclude the generation of this test. Only one pass state is created. The tests T_1 and T_2 consider the same input, a, in the first transition. The difference lies in the delays that have been considered for each of them, 4 and 2 time units, respectively. This fact makes that for the test T_2 the output z, produced by the function ϕ_1, leads to a pass state. □

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 asume a bound n on the number of states of the IUT, and we consider specification having at most m states, then we can adapt [14] to the current framework to conclude that we can restrict ourselves to tests having at most $m\dot{n} + 1$ inputs.

The next result relates, for a specification S and an implementation I, implementation relation and application of test suites.

Theorem 1 Let S and I be two TOSXMs. We have that:

- I $\mathtt{conf}_f$ S iff I passes $\mathtt{tests}(S)$.

Proof:

First, let us show that I passes $\mathtt{tests}(S)$ implies I $\mathtt{conf}_f$ S. We will use the contrapositive, that is, we will suppose that $I \mathtt{conf}_f S$ does not hold and we will prove that I does not pass $\mathtt{tests}(S)$. Let us consider that I $\mathtt{conf}_f$ S does not hold. Then, we automatically infer that there exist $\bar{i} = (i_1, \ldots, i_r)$ and $\bar{t} = (t_1, \ldots, t_r)$ such that $f_I(\bar{i}, \bar{t}) \neq f_S(\bar{i}, \bar{t})$. Thus, the sequences of outputs produced by I and S as response to the sequence of inputs $\bar{i}$ applied after the delays established in the sequence $\bar{t}$, are different. We assume that the difference between them lies in the last output, that is, $f_I(\bar{i}, \bar{t}) = (o_1, \ldots, o_r)$ and $f_S(\bar{i}, \bar{t}) = (o_1, \ldots, o'_r)$ with $o_r \neq o'_r$. We will show that there exists a test $T = (S, I, O, Tr, s, S_I, S_O, S_F, S_P, W) \in \mathtt{tests}(S)$ 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 $u^T \in S_F$ where $\sigma = (i_1/o_1, \ldots, i_r/o_r)$ and $\sigma' = (i_1/o_1, \ldots, i_r/o'_r)$. 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 4. The algorithm will be resolved in the following way:

1. for $1 \leq j \leq r$ do
   - Apply case 2 for the input action i_j selecting $t_d := t_j$.

   endfor
2. Apply the first case for the last element (s^X, m^X, s^T).

Let us remark that step 1.(a) corresponds to continue the construction of the test in the state reached by the transition labelled by o_{j-1} (in the case of $j = 1$ we mean the initial state of the test).

Since $(\sigma', \bar{t}) \notin f_S$, the last application of the second case for the output o'_r must be necessarily associated to the step 2.(e). So, the previous algorithm generates a test T such that $T \stackrel{\sigma'}{\Longrightarrow}_{\bar{t}} u^T$, with $u^T \in S_F$. Since $(\sigma', \bar{t}) \in f_I$ 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 $\mathtt{tests}(S)$.

Let us show now that I $\mathtt{conf}_f$ S implies I passes $\mathtt{tests}(S)$. Again by contrapositive, we will assume that I does not pass $\mathtt{tests}(S)$ and we will conclude that I $\mathtt{conf}_f$ S does not hold.

Let us consider that $\mathtt{pass}(I, \mathtt{tests}(S))$ does not hold. This means that there exists a test $T \in \mathtt{tests}(S)$ and some $\sigma = (i_1/o_1, \ldots, i_r/o_r)$, $s^T \in S_F$, and $\bar{t}$ such that $I \parallel T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$. Then, there exists $\bar{t}$ such that $T \stackrel{\sigma}{\Longrightarrow}_{\bar{t}} s^T$. 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. Let us note that our algorithm allows to create a fail state only as the result of the application of the second case, when the output is not allowed by the specification. Due to the fact that we only consider deterministic and completely specified machines, there exists an output action o'_r, such that $f_S(\bar{i}, \bar{t}) = (o_1, \ldots, o'_r)$ where $\bar{i} = (i_1, \ldots, i_r)$. Given the fact that $f_I(\bar{i}, \bar{t}) = (o_1, \ldots, o_r)$, we have that $f_S(\bar{i}, \bar{t}) \neq f_I(\bar{i}, \bar{t})$. Thus, we conclude I $\mathtt{conf}_f$ S does not hold.

□

6 Related work

In terms of related work, our language is based on Stream X-machines, which have been extensively used by the formal testing community. This paper continues the work in [24] where a notion of stochastic time was added to the classical formalism. If we combine, following [25], this previous work and the work reported in this paper, we can obtain a temporal formalisms where we can express both timeouts and action durations. 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 [32]: Once the value of a clock variable exceeds the invariant, the system produces a prescribed output and enters a new state.

Due to the intrinsical difficulty behind testing timed systems, different approaches have been studied, falling into one or more of the following categories: (a) Only some behaviors, out of those that are relevant for the correctness of the implementation, are tested (see e.g. [11, 6]). 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*; (b) a *complete finite* test suite is derived from the specification, that is, if all tests in the finite suite are passed then the implementation is correct (see e.g. [7, 32]). Usually, the finiteness of this suite requires to introduce strong assumptions about the implementation, both to deal with functional requirements (e.g. to assume that the maximal number of states in the implementation is known) and timed 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; and (c) a complete *infinite* test suite is extracted from the specification (see e.g. [23, 29, 27, 4, 26]). In particular, a test derivation algorithm is defined in such a way that, for all implementation behaviors that must be tested before granting the correctness, a suitable test for checking this behavior is added by the algorithm after executing it some finite time. 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 these methods 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 the (c) 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).

7 Conclusions and future work

In this paper we have presented a formal testing framework for systems where timeouts are critical. The model introduced for specifying the systems is a suitable extension of the classical concept of stream X-machine. An implementation relation has been introduced for describing the notion of correctness of an implementation with respect to a specification. In addition, we have introduced a notion of test that can delay the execution of the implementation. Finally, we have also presented an algorithm to derive sound and complete test suites with respect to the implementation relation presented in this paper.

In terms of future work, we would like to take this work as a first step, together with [24], to define a testing theory for systems presenting stochastic time together with timeouts. In addition, we also consider to study different methods for reducing the size of the generated test suite.

References

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

[2] J. Barnard. COMX: A design methodology using communicating X–machines. *Information and Software Technology*, 40(5–6):271–280, 1998.

[3] 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.

[4] 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.

[5] 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.

[6] R. Cardell-Oliver. Conformance tests for real-time systems with timed automata specifications. *Formal Aspects of Computing*, 12(5):350–371, 2000.

[7] 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.

[8] T. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[9] 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.

[10] 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.

[11] 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.

[12] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys (in press)*, 2008.

[13] R. 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.

[14] R. Hierons, M. Merayo, and M. Núñez. Testing from a stochastic timed system with a fault model, 2008. Submitted.

[15] 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.

[16] M. Holcombe. X–machines as a basis for dynamic system specification. *Software Engineering Journal*, 3(2):69–76, 1988.

[17] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer, 1998.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[23] 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.

[24] M. 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.

[25] M. 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.

[26] M. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.

[27] M. Núñez and I. Rodríguez. Encoding `PAMR` into (timed) `EFSMs`. In *22nd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'02, LNCS 2529*, pages 1–16. Springer, 2002.

[28] M. Núñez and I. Rodríguez. Towards testing stochastic timed systems. In *23rd IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'03, LNCS 2767*, pages 335–350. Springer, 2003.

[29] J. Peleska and M. Siegel. Test automation of safety-critical reactive systems. *South African Computer Journal*, 19:53–77, 1997.

[30] 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.

[31] I. Rodríguez, M. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[32] J. Springintveld, F. Vaandrager, and P. 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.