Passive Testing of Stochastic Timed Systems*

César Andrés, Mercedes G. Merayo, Manuel Núñez
Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid
E-28040 Madrid. Spain.
{c.andres,mgmerayo}@fdi.ucm.es, mn@sip.ucm.es

Abstract

In this paper we introduce a formal methodology to perform passive testing, based on invariants, for systems where the passing of time is represented in probabilistic terms by means of probability distributions functions. In our approach, invariants express the fact that each time the implementation under test performs a given sequence of actions, then it must exhibit a behavior according to the probability distribution functions reflected in the invariant. We present algorithms to decide the correctness of the proposed invariants with respect to a given specification. Once we know that an invariant is correct, we check whether the execution traces observed from the implementation respect the invariant. In addition to the theoretical framework we have developed a tool, called PASTE*, that helps in the automation of our passive testing approach. We have used the tool to obtain experimental results from the application of our methodology.*

1 Introduction

Formal testing techniques [4, 13, 21] allow to test the correctness of a system with respect to a specification by performing experiments on it. After the initial consolidation stage, formal testing techniques started to deal with *non-functional* properties. In this line, the time consumed by each operation should be considered critical in a real-time system. The testing community has shown a growing interest in extending these frameworks so that not only functional properties but also quantitative ones could be tested. Thus, during the last years there have been several proposals for timed testing (e.g. [14, 9, 23, 7, 19, 20, 10, 16, 17, 8, 22]). In these papers, with the only exception of the model initially introduced in [19], time is considered to be *deterministic*, that is, time requirements follow the form "after/before t time units..." In fact, in most of the cases time is introduced by means of clocks following [1]. Even though the inclusion of time allows to give a more precise description of the system to be implemented, there are frequent situations, most notably when studying biological systems, that cannot be accurately described by using this notion of deterministic time. For example, we may desire to specify a system where a message is expected to be received with probability $\frac{1}{2}$ in the interval $(0, 1]$, with probability $\frac{1}{4}$ in $(1, 2]$, and so on.

Most testing approaches consist in the generation of a set of tests that are applied to the implementation in order to check its correctness with respect to a specification. Thus, testing is based on the ability of a tester to stimulate the implementation under test (IUT) and check the correction of the answers provided by the implementation. However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the IUT or the implementation is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. In addition, the activity of testing could be specially difficult if the tester must check temporal restrictions. In these situations, the instruments of measurement could be not so precise as required or the results could be distorted due to mistakes during the observation (see [15] where small measuring errors while testing timed systems are considered). As a result, undiscovered faults may result in failures at runtime, where the system may perform untested traces. In these situations, there is a particular interest in using *passive testing* techniques. In passive testing the tester does not need to interact with the IUT. On the contrary, execution traces are observed and analyzed without interfering with the behavior of the IUT. Usually, execution traces of the implementation are compared with the specification to detect faults in the implementation ([12, 18, 24, 25]). In most of these works the specification has the form of a finite state machine

*Research partially supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01).

(FSM) and the studies consist in verifying that the executed trace is accepted by the FSM specification. A drawback of these first approaches is the low performance of the proposed algorithms (in terms of complexity in the worst case) if non-deterministic specifications are considered. A new approach was proposed in [6]. There, a set of properties called *invariants* were extracted from the specification and checked on the traces observed from the implementation to test their correctness. One of the drawbacks of this work was the limitation on the grammar used to express invariants. A new formalism that overcomes this restriction for expressing invariants was presented in [3]. It allows to specify wild-card characters in invariants and to include a set of outputs as termination of the invariant. In [2] a temporal extension of [3] was introduced in order to deal with timed restrictions, where time is considered to be *deterministic*. A simple extension of the classical concept of FSM was used which allows a specifier to explicitly denote temporal requirements for each action of a system.

In this paper we present a *testing methodology* based on passive testing, where the time consumed by the system while it performs its tasks is given by *probability distribution functions*. In order to deal with *stochastic time* we consider a suitable extension of the FSM model: *Stochastic Timed Finite State Machines*. Instead of having expressions such as "the action o takes t units of time to be performed" we will have expressions such as "with probability p the action o will be performed before t units of time." Thus, the interpretation of a transition $s \xrightarrow{i/o}_F s'$ is "if the machine is in state s and receives an input i, it will produce the output o and it will change its state to s' before an amount of time t with probability $F(t)$, where F is the probability distribution function associated with the transition."

Invariants will be used to express properties that are fulfilled in all the parts of the IUT: If we observe a trace from the IUT that does not match a *correct* invariant, then we conclude that the IUT is faulty. Our invariants will express both causality relations among inputs and outputs, and relations between the time values observed in the trace and certain probability distribution functions appearing in invariants. Invariants can represent properties such as "Each time that the system produces i/o if the system receives the input i' then the paired output must belong to a given set O." In addition, the invariant will have two probability distribution functions, associated to the previously mentioned transitions, so that all the time values appearing in the trace attached to the performance of each pair could be generated by the corresponding probability distribution function. Thus, we have to perform two types of property verification concerning invariants: One on the specification and another one on the traces generated by the implementation. Since invariants can be supplied by the tester, the first step is to check that these invariants are in fact correct with respect to the specification. The next step is to check whether the trace produced by the IUT fulfills invariants. In this case, we propose an algorithm that is an adaption of the classical algorithms for string matching. It works, in the worst case, in time $O(m \cdot n)$ where m and n are the length of the trace and the invariant, respectively. Let us remark that we cannot achieve complexities as good as the ones in classical algorithms because we have to find all the occurrences of the pattern.

We have developed a tool, called PASTE, to automate our methodology. In particular, the tool includes the algorithms to check the correctness of invariants with respect to the specification and to decide whether the trace observed from the implementation fulfills the invariants. Moreover, the tool facilitates the task of choosing *good* invariants by automatically performing experiments based on mutation techniques.

This paper makes the following contributions. It advances the state of the art on testing systems presenting stochastic time, so that complements the *active* testing approach given in [19]. It provides, as far as we know, the first formal approach to perform passive testing of the important class of systems where time is represented in stochastic terms. Finally, it reports on a tool to automate passive testing activities of timed systems, in particular, of stochastic-timed systems.

The rest of the paper is organized as follows. In Section 2 we present the notation we apply along the paper. In Section 3 we introduce our stochastic-temporal extension of the classical finite state machine model. In Section 4 the notion of invariant for systems where timed restrictions are given in stochastic terms is presented, as well as the algorithms to check the correctness of invariants with respect to the specification and the conformance of the execution trace with respect to the invariants. In Section 5 we comment the tool developed to implement our methodology. Finally, Section 6 presents the conclusions of the paper and some lines for future work.

2 Preliminaries

Along this paper we use probability distribution functions to model the time output actions take to be executed. Thus, we need to introduce some basic concepts. We will consider that the sample space, that is, the domain of the probability distribution functions, is $\mathbf{R}_+$.

Definition 1 A *probability distribution function* is a function $F : \mathbf{R}_+ \longrightarrow [0, 1]$ having the following properties:

- $\lim_{t \to +\infty} F(t) = 1$.
- F is monotonically increasing, that is, for all t_1 and $t_2 \in \mathbf{R}_+$ such that $t_1 \leq t_2$ we have $F(t_1) \leq F(t_2)$.

- F is right-continuous, that is, for all $t \in \mathbf{R}_+$ we have:

$$\lim_{t' \to t^+} F(t') = F(t).$$

We denote the set of probability distribution functions by $\mathcal{F}$ (F, F_1, F_2 to range over $\mathcal{F}$). Let F_1 and F_2 be two probability distribution functions. We write $F_1 = F_2$ if for all $t \in \mathbf{R}_+$ we have $F_1(t) = F_2(t)$. We will call *sample* to any multiset of positive real numbers. We denote the set of multisets in $\mathbf{R}_+$ by $\wp(\mathbf{R}_+)$.

Let F be a probability distribution function and J be a sample. We denote the *confidence* of F in J by $\gamma(F, J)$. □

In our setting, samples will be associated with time values that implementations need to perform sequences of actions. We have that $\gamma(F, J)$ takes values in the interval $[0, 1]$. Intuitively, bigger values of $\gamma(F, J)$ indicate that the observed sample J is more likely to be produced by the probability distributed function F. That is, γ decides how *similar* is the probability distribution function generated by J and the one corresponding to F are.

Next, we introduce one of the standard ways to measure the confidence degree that a function F has on a sample. In order to do so, we will present a methodology to perform *hypothesis contrasts*. The underlying idea is that a sample will be *rejected* if the probability of observing that sample from a *natural* sample extracted from F is low. In practice, we will check whether the probability to observe a *discrepancy* lower than or equal to the one we have observed is low enough. We will present *Pearson's χ^2 contrast*.

Definition 2 The *Pearson's χ^2 contrast* can be applied both to continuous and discrete probability distribution functions. Once we have collected a sample of size n we perform the following steps:

- We split the sample into k classes which cover all the possible range of values. We denote by o_i the *observed frequency* at class i (i.e. the number of elements belonging to the class i).

- We calculate the probability p_i of each class, according to the proposed probability distribution function. We denote by e_i the *expected frequency*, which is given by the equation $e_i = n \cdot p_i$.

- We calculate the *discrepancy* between observed frequencies and expected frequencies as $X^2 = \sum_{i=1}^{k} \frac{(o_i - e_i)^2}{e_i}$. When the model is correct, this discrepancy is approximately distributed as the distribution χ^2.

- We estimate the number of freedom degrees of χ^2 as $k - r - 1$. In this case, r is the number of parameters of the model which have been estimated by maximal likelihood over the sample to estimate the values of p_i (i.e. $r = 0$ if the model completely specifies the values of p_i before the samples are observed).

- We will *accept* that the sample follows the proposed random variable if the probability to obtain a discrepancy greater or equal to the discrepancy observed is high enough, that is, if $X^2 < \chi^2_{(k-r-1),\alpha}$ for some α low enough. Actually, as such margin to accept the sample decreases as α decreases, we can obtain a measure of the validity of the sample as $\texttt{max}\{\alpha \mid X^2 < \chi^2_{(k-r-1),\alpha}\}$.

□

Example 1 Let us illustrate the previous definitions of probability distribution function, sample, confidence and Pearson χ^2 contrast in the following example. Let us suppose we have a dice, having its six sides the same probability. For example, the probability of obtaining 1 or less is $\frac{1}{6}$, and the probability of obtaining a number less than or equal to 4 is $\frac{4}{6}$.

Let us suppose we toss this dice three hundred times. We store the observed results in a sample denoted by ℓ. We have that ℓ is in $\wp(\{1, 2, 3, 4, 5, 6\})$. In order to represent the number of observed values associated which each side of the dice, let us suppose that in ℓ the observed frequencies are $o_1 = 43$, $o_2 = 49$, $o_3 = 56$, $o_4 = 45$, $o_5 = 66$, and $o_6 = 41$.

Now we show how can we decide the confidence of ℓ with respect to the probability distribution function considered. We will use for this task the *chi square goodness of fit test*. This test is particularly useful to determine how well a model fits observed data since it allows us to evaluate how *close* the observed values are to those which would be expected given the model considered.

We denote the expected frequency of value i by e_i. Since we expected that the dice is regular, we have $e_i = 50$ for all $1 \leq i \leq 6$.

The level of significance $\alpha \in [0, 1]$ allows us to let some discrepancies of the observed values with respect to the expected ones. We define the null hypothesis, denoted by H_0, and we must show that H_0 does not hold. The meaning of the null hypothesis in this example is "the dice is not regular", meaning that $F(x) \neq \frac{x}{6}$, for some $x \in \{1, \ldots, 6\}$.

$$H_0 = \sum_{i=1}^{6} \frac{(o_i - e_i)^2}{e_i} > \chi^2_{5,\alpha}$$

In other words, we are saying that with probability $1 - \alpha$, ℓ was obtained from F. In our case, we can accept that the dice is regular because H_0 does not hold since the left hand side of the inequality is equal to 8.96 while the right hand side is equal to 11.07. □

3 Stochastic Timed Finite State Machine

In this section we introduce our notion of finite state machines with stochastic time. The main difference with respect to usual FSMs consists in the addition of *time* to indicate the lapse between offering an input and receiving an output. As we have already indicated, we use probability distribution functions to model the (stochastic) time that output actions take to be executed.

Definition 3 A *Stochastic Timed Finite State Machine*, in short STFSM, is a tuple $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ where S is a finite set of states, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, Tr is the set of transitions, and s_{in} is the initial state.

A transition belonging to Tr is a tuple (s, s', i, o, F) where $s, s' \in S$ are the initial and final states of the transition, $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output actions, respectively, and $F \in \mathcal{F}$ denotes the time, in probabilistic terms, that the transition needs to be completed.

We say that M is *input-enabled* if for all state $s \in S$ and input $i \in \mathcal{I}$, there exist $s' \in S$, $o \in \mathcal{O}$, and $F \in \mathcal{F}$ such that $(s, s', i, o, F) \in Tr$. We say that M is *observable* if for all s, i, o there do not exist two different transitions $(s, s_1, i, o, F_1), (s, s_2, i, o, F_2) \in Tr$. We say that M has *regular stochastic information*, if there do not exist two different transitions (s_0, s_1, i, o, F_1) and (s_2, s_3, i, o, F_2) with $F_1 \neq F_2$. □

Intuitively, a transition (s, s', i, o, F) of a STFSM indicates that if the machine is in state s and receives the input i, then the machine emits the output o and moves to s' after a lapse less than or equal to t time units with probability $F(t)$. We usually denote such transition by $s \xrightarrow{i/o}_F s'$. Along the rest of this paper we assume that all the machines are input-enabled, observable and have regular stochastic information.

Example 2 Let us consider the STFSM depicted in Figure 1. We are modeling the timed behavior with the functions F_1, F_2, F_3 associated with each transition. In this example we show three possible, often used, functions. We consider that the values generated by F_1 are *uniformly distributed* in the interval $[0, 2]$. Uniform distributions allow us to keep compatibility with time intervals in (non-stochastic) timed models in the sense that the same *weight* is assigned to all the times in the interval. The function F_2 follows a Dirac distribution in 4. Dirac distributions concentrate the probability in a single point. Thus F_2 gives probability 1 to 4 and probability 0 to the rest of values. In timed terms, the idea is that the corresponding delay will be equal to 4 units of time. Dirac distributions allow us to simulate deterministic delays appearing in timed models. Finally, F_3 is *exponentially* distributed with parameter 3.

Let us consider the transition $(s_2, s_1, i_0, o_1, F_1)$. Intuitively, if the machine is in state s_2 and receives the input i_0 then it will produce the output o_1 after a time given by F_1 and will move to state s_3. The time associated with the transition is a value $0 \leq t \leq 2$, that can be drawn with the same probability. For example, we know that this delay will be less than 1 time unit with probability $\frac{1}{2}$, it will be less than 1.5 time units with probability $\frac{3}{4}$, and so on. Finally, once 2 time units have passed we know that the output o_1 has been performed (that is, we have probability 1). □

4 Stochastic Invariants

In this section we introduce the notion of *invariant* for systems where time conditions are given in stochastic terms. An invariant represents a relevant property that must be fulfilled by the IUT. Intuitively, an invariant expresses the fact that each time the IUT performs a given sequence of actions, then it must exhibit a behavior reflected in the invariant. These invariants should be supplied by the expert/tester. Thus, the first step is to check that the invariants are correct with respect to the specification and an algorithm is provided in order to establish it. After we have a collection of correct invariants we must check whether the traces produced by the IUT satisfy the properties expressed by them. Another approach consists in automatically extract invariants from the specification. In this case, we can adapt to our framework the algorithms given in [6]. The problem with this approach is that the number of possible invariants is huge. A third alternative is to assume that invariants are correct *by definition*. In this situation, a specification is not needed and the invariants can be considered as the *requirements* of the system to be implemented.

Due to the timed nature of our framework, we need to extend the notion of invariant to deal with temporal requirements. The invariants will allow to express properties such as "*After pressing the red button we receive a coke before* 4 *seconds with probability 0.95"*.

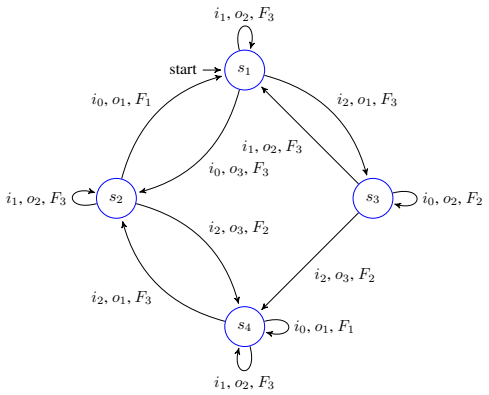
Next, we introduce the notion of invariant where stochastic-temporal behavior is taken into account.

In order to express invariants in a concise way, we will use the wild-card characters ? and $\star$. The wild-car ? represents any value in the sets I and O, while $\star$ represents a sequence of input/output pairs.

Definition 4 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a STFSM. We say that a sequence I is an *invariant* for M if the following two conditions hold:

1. I is defined according to the following EBNF:

$$I ::= a/z/F, I \mid \star, I' \mid i \mapsto \mathcal{M}$$
$$I' ::= i/z/F, I \mid i \mapsto \mathcal{M}$$



$$F_1(t) = \begin{cases} 0 & \text{if } t \leq 0 \\ \frac{t}{2} & \text{if } 0 < t < 2 \\ 1 & \text{if } t \geq 2 \end{cases}$$

$$F_2(t) = \begin{cases} 0 & \text{if } t < 4 \\ 1 & \text{if } t \geq 4 \end{cases}$$

$$F_3(t) = \begin{cases} 1 - e^{\frac{-t}{3}} & \text{if } t \geq 0 \\ 0 & \text{if } t < 0 \end{cases}$$

$$I = \{i_0, i_1, i_2\}\ O = \{o_1, o_2, o_3\}$$

Figure 1. Example of `STFSM`.

In this expression we consider $F \in \mathcal{F}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $\mathcal{M} \subseteq \mathcal{O} \times \mathcal{F}$.

2. I is *correct* with respect to M.

We denote the set of stochastic time invariants by `STInv`. □

Intuitively, the previous EBNF expresses that an invariant is either a sequence of symbols where each component, but the last one, is either an expression $a/z/F$, with a being an input action or the wild-card character ?, z being an output action or the wild-card character ?, and F being a probability distribution function, or an expression $\star$. There are two restrictions to this rule. First, an invariant cannot contain two consecutive expressions $\star$. The second restriction is that an invariant cannot present a component of the form $\star$ followed by an expression beginning with the wild-card character ?, that is, the input of the next component must be an input action $i \in \mathcal{I}$. In fact, $\star$ represents any sequence of input/output pairs such that the input is not equal to i, being i the next input appearing in the invariant.

The last component, corresponding to the expression $i \mapsto \mathcal{M}$, is an input action followed by a set of pairs $< o, F >$. Each pair represents a possible output that can be observed after the input i and the probability distribution function that draws the amount of time the system should spend for performing it.

Example 3 Let us consider the invariant (regarding the specification in Figure 1)

$$i_1/?/F_3, \star, i_0 \mapsto \{< o_1, F_1 >, < o_2, F_2 >, < o_3, F_3 >\}$$

This invariant reflects that each time we find in a trace a (sub)sequence starting with the input i_1 and paired with any output symbol, the first occurrence of the input i_0 should be paired with either the output o_1, the output o_2, or the output o_3. Only if we find the corresponding sequence, then we check the temporal behavior. The previous invariant requires that the amount of time the system spends to generate an output for the input i_1 must be given by F_3. Furthermore, always that we find i_0 and it is followed by o_1 or o_2 or o_3, the amount of time the system takes will happen with a probability given by F_1, F_2 and F_3 respectively.

We can refine the previous invariant if we consider only the cases where the pair i_1/o_2 was observed. The invariant for denoting this property is

$$i_1/o_2/F_3, \star, i_0 \mapsto \{< o_1, F_1 >, < o_2, F_2 >, < o_3, _3 >\}$$

□

4.1 Correctness of invariants with respect to specifications

Since we assume that invariants may be defined by a tester, we must ensure they are correct with respect to the specification. Next, we introduce an algorithm that allows us to establish this correctness and explain its most relevant aspects. First we introduce some auxiliary functions.

Definition 5 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a `STFSM`, $s \in S$, $a \in \mathcal{I} \cup \{?\}$, and $z \in \mathcal{O} \cup \{?\}$. We define the set $\texttt{afterCond}(s, a, z)$ as the set of transitions belonging to Tr having as initial state s and labeled by a/z, that is,

$$\texttt{afterCond}(s, i, o) = \{(s, s', i, o, F) | (s, s', i, o, F) \in Tr\}$$

$$\texttt{afterCond}(s, ?, o) = \bigcup_{i \in \mathcal{I}} \texttt{afterCond}(s, i, o)$$

$$\texttt{afterCond}(s, i, ?) = \bigcup_{o \in \mathcal{O}} \texttt{afterCond}(s, i, o)$$

$$\texttt{afterCond}(s, ?, ?) = \bigcup_{i \in \mathcal{I}, o \in \mathcal{O}} \texttt{afterCond}(s, i, o)$$

We define the function $\texttt{afterInp}(s, i)$ as the function that computes the set of states that can be reached from state s without performing the input i:

$$\texttt{afterInp}(s,i) = \{s\} \cup$$

$$\left\{ s' \;\middle|\; \begin{array}{l} \exists s_1,\ldots,s_{n-1},i_1,\ldots,i_n,o_1,\ldots,o_n,F_1,\ldots,F_n : \\ s \xrightarrow{i_1/o_1}_{F_1} s_1 \xrightarrow{i_2/o_2}_{F_2} s_2 \ldots s_{n-1} \xrightarrow{i_n/o_n}_{F_n} s' \\ \\ \wedge\, i \notin \{i_1,\ldots,i_n\} \wedge n \geq 1 \end{array} \right\}$$

□

The algorithm to establish the correctness of an invariant with respect to a specification is given in Figure 2. Intuitively, the algorithm checks that the invariant is respected in all the possible paths of the specification. Initially, we need to consider all the states of the specification due to the fact that the invariant does not have to correspond with a trace beginning in the initial state. From each state in the specification the algorithm explores if there exists any transition that matches the first component of the invariant. The set of states reached by these transitions are collected and the same process is applied for the next components of the invariant, considering only the states gathered in the previous step. If during this process we obtain a empty set of states, it means that the invariant is useless for this specification and the algorithm stops. Otherwise, we must check the last component of the invariant, $i \mapsto \mathcal{M}$. We need to decide whether the transitions outgoing from the states reached in the first phase of the algorithm and labeled by the input i fulfill the requirements established in $\mathcal{M}$, that is, the output and the probability distribution function associated to these transitions belong to the set $\mathcal{M}$.

Lemma 1 Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a STFSM. The worst case of the algorithm given in Figure 2 checks the correctness of an invariant $I = i_1/o_1/F_1, \ldots, i_{n-1}/o_{n-1}/F_{n-1}, i_n \mapsto \mathcal{M}$ with respect to M:

- In time $\mathcal{O}(n \cdot |Tr|)$ and space $\mathcal{O}(|Tr|)$ if I does not present occurrences of $\star$.
- In time $\mathcal{O}(k \cdot |Tr|^2 + (n-k) \cdot |Tr|)$ and space $\mathcal{O}(|Tr|)$ if I presents k occurrences of $\star$ in I.

□

4.2 Conformance of traces with respect to invariants

In this section we explain how we can determine whether the execution traces obtained from the IUT satisfy the properties expressed by the invariants. As we commented previously, it is not relevant the state where the machine was placed when we started to observe the trace because invariants have to be fulfilled at any point of the IUT. In order to test the trace we perform a pattern matching strategy. We have implemented an adaption of the classical algorithms for pattern matching on strings, (i,e. [5, 11]).

in : $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$
$I = \{a_1, \ldots, a_{n-1}, i_n \mapsto \mathcal{M}\}$
// *either* $a_k = i_k/o_k/F_k$ *or* $a_k = \star$
out: Bool.
$I' \leftarrow I$; $S' \leftarrow S$; $j \leftarrow 1$; $S'' \leftarrow \varnothing$; $error \leftarrow$ `false`;
while $(j < n)$ **do**
  if $(\texttt{head}(I') = \star)$ **then**
    while $(S' \neq \varnothing)$ **do**
      Choose $s \in S'$;
      $S' \leftarrow S' \setminus \{s\}$;
      $S'' \leftarrow S'' \cup \texttt{afterInp}(s, i_{j+1})$;
  else
    while $(S' \neq \varnothing)$ **do**
      Choose $s_a \in S'$;
      $S' \leftarrow S' \setminus \{s_a\}$;
      $Tr' \leftarrow \texttt{afterCond}(s_a, i_j, o_j)$;
      while $(Tr' \neq \varnothing)$ **do**
        Choose $(s_a, s_b, i_j, o_j, F') \in Tr'$;
        $Tr' \leftarrow Tr' \setminus \{(s_a, s_b, i_j, o_j, F')\}$;
        if $F' = F_j$ **then**
          $S'' \leftarrow S'' \cup \{s_b\}$;
  $I' = \texttt{tail}(I')$;
  $j \leftarrow j + 1$; $S' \leftarrow S''$; $S'' \leftarrow \varnothing$;
if $(S' = \varnothing)$ **then**
  $error \leftarrow$ `true`;
while $(S' \neq \varnothing)$ **do**
  Choose $s_a \in S'$;
  $S' \leftarrow S' \setminus \{s_a\}$;
  $Tr' \leftarrow \texttt{afterCond}(s_a, i_n, ?)$;
  while $(Tr' \neq \varnothing)$ **do**
    Choose $(s_a, s_b, i_n, o, F') \in Tr'$;
    $Tr' \leftarrow Tr' \setminus \{(s_a, s_b, i_n, o, F')\}$;
    if $(< o, F' > \in \mathcal{M}))$ **then**
      $S'' \leftarrow S'' \cup \{s_b\}$;
    else
      $error \leftarrow$ `true`
if $(S'' = \varnothing)$ **then**
  $error \leftarrow$ `true`;
Return $(\neg error)$;

Figure 2. Correctness of an invariant with respect to a specification.

Execution traces are essential in passive testing. Let us remember that, in our setting, testers cannot interact with the IUT. They are only provided with recorded traces, called *logs*, for performing testing. In a log we can observe several signals (input/output) and the time the system took to perform them. A log is a sequence

input : $s :: sequence$
$I = \{a_1, \ldots, a_{n-1}, i_n \mapsto \mathcal{M}\}$
$\lambda \in [0, 1]$
output: `Bool`.

times :: Set of $\wp(\mathbf{R}_+)$;
Fs :: Set of $\mathcal{I} \times \mathcal{O} \times \mathcal{F}$;
Struct $\mathcal{A}$ $\{wild :: \texttt{Bool}; I_{aux} :: \texttt{STInv}\}$
$b, b_{aux} :: Stack[\mathcal{A}]$; tok :: $\mathcal{A}$;
error $\leftarrow$ `false`;

for $(j \leftarrow 1; j \leq \texttt{length}(s) \wedge \neg\text{error}; j \leftarrow j + 1)$ **do**
  $(i, o, t) \leftarrow s[j]$;
  $\text{times}(i, o) \leftarrow \text{times}(i, o) \cup \{t\}$;
  tok.$wild \leftarrow$ `false`;
  tok.$I_{aux} \leftarrow I$;
  tok $\leftarrow \texttt{check}(s[j], \text{tok}, \text{error}, \text{Fs})$;
  if $(\text{tok} \neq null)$ **then**
    $\texttt{push}(b_{aux}, \text{tok})$;
  while $(\neg\texttt{isEmpty}(b))$ **do**
    tok $\leftarrow \texttt{check}(s[j], \texttt{top}(b), \text{error}, \text{Fs})$;
    if $(\text{tok} \neq null)$ **then**
      $\texttt{push}(b_{aux}, \text{tok})$;
  $b \leftarrow b_{aux}$;
while $(\text{Fs} \neq \varnothing \wedge \neg\text{error})$ **do**
  `Choose`$(i, o, F) \in \text{Fs}$;
  $\text{Fs} \leftarrow \text{Fs} \setminus \{(i, o, F)\}$;
  if $(\gamma(F, \text{times(i, o)}) < \lambda)$ **then**
    error $\leftarrow$ `true`;
return$(\neg\text{error})$;

Figure 3. Correctness of a log with respect to an invariant.

input : (i, o, t)
tok::$\mathcal{A}$
&error :: `Bool`
&Fs :: Set of $\mathcal{I} \times \mathcal{O} \times \mathcal{F}$
output: $\mathcal{A}$.
switch $(\texttt{head}(\text{tok}.I_{aux}))$ **do**
  `Case` : (i', o', F')
  $Fs \leftarrow Fs \cup \{(i', o', F')\}$;
  if $i = i'$ **then**
    if $o = o'$ **then**
      tok.wild $\leftarrow$ `false`;
      $\text{tok.I}_{\text{aux}} \leftarrow \texttt{tail}(\text{tok.I}_{\text{aux}})$;
      `return`(tok);
    else
      `return`$(null)$;
  else
    if tok.wild **then**
      `return`(tok);
    else
      `return`$(null)$;
  `Case` : $(i_n \mapsto \mathcal{M})$
  $Fs \leftarrow Fs \cup \{(i_n, o, F)\} | (o, F) \in \mathcal{M}\}$;
  if $i = i_n$ **then**
    if $(\exists F : (o, F) \in \mathcal{M})$ **then**
      `return`$(null)$;
    else
      error $\leftarrow$ `true`;
      `return`$(null)$;
  else
    `return`$(null)$;
  `Case` : $(\star)$
    tok.wild $\leftarrow$ `true`;
    $\text{tok.I}_{\text{aux}} \leftarrow \texttt{tail}(\text{tok.I}_{\text{aux}})$;
    `return`(tok);

Figure 4. The `check` function.

$i_1/o_1/t_1, i_2/o_2/t_2, i_3/o_3/t_3, \ldots, i_n/o_n/t_n$ where for all $1 \leq j \leq n$ we have $i_j \in I, o_j \in O$, and $t_j \in \mathbf{R}_+$.

Regarding stochastic-temporal requirements, we might require that any subsequence of an execution trace must have the same associated delay, that is, each input/output has an identical probability distribution function to the one appearing in the invariant. Although this is very reasonable, if we assume a passive testing framework then we cannot check whether the corresponding probability distribution is identical. This is so because we do not have the *implemented* function; we only have the amounts of time the system spent to perform the actions. In order to guarantee that these values *fit* the corresponding probability distribution function in the invariant we collect the execution times and compare them with the function using a *hypothesis contrast.*

In Figure 3 the algorithm that we use to establish the conformance of a log obtained from the IUT with respect to an invariant is presented. The algorithm can be divided in two different stages. The first one explores the correctness of the trace with respect to the input/output pairs appearing in the invariant. The second one is related to the stochastic-temporal restrictions.

In the first phase we transverse the trace, looking for any incorrect behavior of the implementation with respect to the property expressed in the invariant without considering the probability distribution functions.We use an auxiliary function `check`, depicted in Figure 4, that checks whether an element of the trace and a component of the invariant match. The treatment depends on the kind of component of the invariant being checked: A triple $i/o/F$, a $\star$ symbol or $i \mapsto \mathcal{M}$ corresponding to the last component of the invari-

ant. In this phase, an error is detected only when we check if the log and the last component of the invariant match. Let us remark that the fact that the recorded log and the invariant do not fit when we are checking the first $n-1$ elements of the invariant, it does not indicate that the trace does not fulfill the invariant. In that case, we have not found the preconditions established by it. In the same way, we could not deduce that we have found an error if all the components of the invariant appear in the observed trace but the last one. In such a situation we cannot conclude that the implementation fails. Similarly, if we find the last component, we cannot conclude anything since the rest of the components of the invariant were not found. It is only when we reach the last component of the invariant when a verdict can be emitted. If we find the input i_n and the corresponding output o_n does not appear in any pair $< o, F >$ belonging to $\mathcal{M}$, then the trace log does not fulfill the property expressed by the invariant and the algorithm emits an error.

During the first stage time values registered in the log are stored. The next phase of the algorithm corresponds to verify if the execution time values collected for each input/output considered in the invariant *fit* the associated probability distribution function. This notion of *fitting* is given by the function γ given in Definition 1 and its specialization by using the contrast hypothesis given in Definition 2. The input parameter α denotes the minimum confidence level that we can let to the time values of the log trace.

5 PASTE: a PASsive TEsting tool

In addition to the theoretical framework we have developed a tool called PASTE that helps in the automation of our passive testing approaches. In order to use the tool, we suppose that the tester has a specification and a set of invariants written in XML and following the internal syntax of the tool (due to lack of space we do not elaborate either on the specific syntax or on the types of probability distribution functions included, by default, in the tool). The tool PASTE provides two major functionalities. The first one fully implements the framework presented in this paper. The first step performed by the tool consists in checking the correctness of the invariants with respect to the specification. Once this correctness is established, the tool has to be provided with one or more logs recorded from the IUT. The user can sort the invariants so that the order in which they are checked against the logs is fixed.

PASTE includes a second functionality that notably helps in the task of performing passive testing for big systems where logs have to be necessarily long and the number of invariants can be very big. In this situation, the task of sorting invariants to decide which ones have to be applied first is of vital importance but very difficult to be done manually. In order to overcome this problem PASTE implements a *mutation-based* approach. Intuitively, by constructing mutants from the specification and by applying the considered invariants, beforehand, to the logs generated by these mutants, we may classify invariants according to the number of mutants that they detect. As usual, it is expected that the more mutants an invariant kills, the more effective will be to detect errors in *real* implementations.

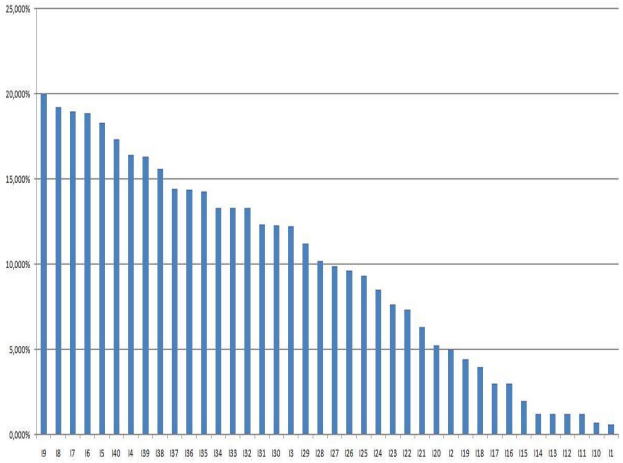


Figure 5. Effectiveness of individual invariants.

In order to generate mutants, we consider three types of mutations: Producing a wrong output in a transition, changing the final state of a transition, and changing one probability distribution function associated with the performance of different transitions. While the first two types are standard in mutation testing, the last one deserves some comments. Mutations consist in slightly changing, between -10% and 10%, one of the parameters of the corresponding function. In the case of discrete probability functions, we can change either one value having probability greater than zero or the probability associated with one value (this last change implies, obviously, that another probability has to be adjusted so that all the *weights* add up to 1). In the case of continuous functions, for example, in exponential or uniform distributions, we change one of the parameters appearing in the definition. Once we have a set of mutants, we generate random traces from these mutants to obtain a big sample where we are able to *test* the effectiveness of the proposed invariants.

Next we report on the performed experiments. First, we generated 500 mutants from a specification and observe the percentage of killed mutants after considering a specific set of 40 invariants. We do not remove *conforming* mutants, that is, mutants that even after inducing a fault are still equivalent to the original specification. This experiment has been performed 50 times with the same specification.

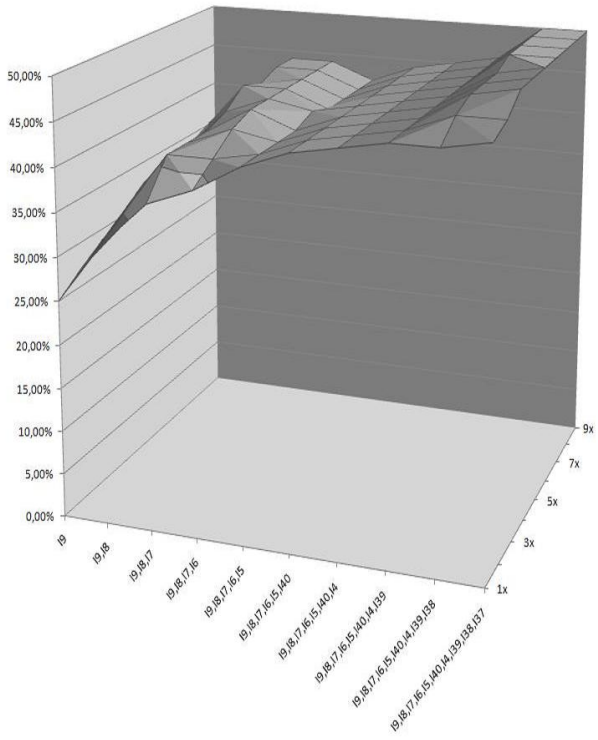


Figure 6. Comparative of invariants with respect to the length of observed logs.

In each simulation we use 200 mutants having an induced output-fault, 200 mutants having with an induced final state fault, and 100 mutants presenting an induced probability distribution function fault. We take different lengths of the log and generate ten logs for each considered length. Obviously, the longer the log is, the higher is the probability of detecting an error in a non-conforming mutant. The length of the logs is computed as a multiple of the total number of transitions appearing in the specification. For example, if the mutant has 25 transitions, then a trace $2x$ means that its length is 50. In these experiments we used traces of length ranging from $1x$ to $10x$.

Next, we selected a subset containing the ten invariants, among the 40 considered, providing the best results. We obtained the results given in Figure 5, where the numbers indicate the percentage of erroneous traces found by each invariant. We would like to stress that invariants are a very powerful tool to find errors among faulty implementations since some of them were able to kill almost 20% of the mutants. For the next stage of the experiment, we produced 25 new different mutants of each class, and extracted, for each length ranging from $1x$ to $9x$, 10 traces. In Figure 6 we observe that the coverage of finding an error (kill a mutant) with a big set of invariants in this concrete specification is closer to 50%, what we consider a good number for a passive testing approach.

6 Conclusions and Future Work

In this paper we have introduced a passive testing methodology for systems that present stochastic timed restrictions. Based on an extension of the classic FSM formalism, and a formal definition of the notion of invariant, we have introduced two algorithms to establish the correctness of an invariant with respect to a specification and to determine whether a log obtained from the IUT satisfy the properties expressed by mean of these invariants. In addition to the theoretical framework we have developed a tool called PASTE that allows the automation of our passive testing approach. In particular, the algorithms presented in the paper are fully implemented and we have some interesting results obtained from the experiments performed.

As future work we plan to improve the capability of our framework by adding new classes of invariants. In addition, we are currently studying the correlation between length of invariants, number of possible invariants for a certain length, and their effectiveness.

Acknowledgements

We would like to thank the anonymous reviewers of this paper for their valuable comments.

References

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

[2] C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of timed systems. In *6th Int. Symposium on Automated Technology for Verification and Analysis, ATVA'08, LNCS 5311*, pages 418–427. Springer, 2008.

[3] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi. A passive testing approach based on invariants: Application to the WAP. *Computer Networks*, 48(2):247–266, 2005.

[4] 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.

[5] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.

[6] A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45:837–852, 2003.

[7] 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.

[8] 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.

[9] 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.

[10] G.-D. Huang and F. Wang. Automatic test case generation with region-related coverage annotations for real-time systems. In *3rd Int. Symposium on Automated Technology for Verification and Analysis, ATVA'05, LNCS 3707*, pages 144–158. Springer, 2005.

[11] D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.

[12] D. Lee, A.N. Netravali, K.K. Sabnani, B. Sugla, and A. John. Passive testing and applications to network management. In *5th IEEE Int. Conf. on Network Protocols, ICNP'97*, pages 113–122. IEEE Computer Society Press, 1997.

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

[14] 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.

[15] M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing of systems presenting soft and hard deadlines. In *2nd IPM Int. Symposium on Fundamentals of Software Engineering, FSEN'07, LNCS 4767*, pages 160–174. Springer, 2007.

[16] 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.

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

[18] R.E. Miller. Passive testing of networks using a CFSM specification. In *IEEE Int. Performance Computing and Communications Conference*, pages 111–116. IEEE Computer Society Press, 1998.

[19] 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.

[20] M. Núñez and I. Rodríguez. Conformance testing relations for timed systems. In *5th Int. Workshop on Formal Approaches to Software Testing, FATES'05, LNCS 3997*, pages 103–117. Springer, 2006.

[21] I. Rodríguez, M.G. Merayo, and M. Núñez. HOTL: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.

[22] 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.

[23] 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.

[24] M. Tabourier and A. Cavalli. Passive testing and application to the GSM-MAP protocol. *Journal of Information and Software Technology*, 41:813–821, 1999.

[25] M. Tabourier, A. Cavalli, and M. Ionescu. A GSM-MAP protocol experiment using passive testing. In *World Congress on Formal Methods in the Development of Computing Systems, FM'99, LNCS 1708*, pages 915–934. Springer, 1999.