Passive Testing of Timed Systems[⋆]

César Andrés, Mercedes G. Merayo, and Manuel Núñez

Dept. Sistemas Informáticos y Computación
Universidad Complutense de Madrid, 28040 Madrid, Spain
{c.andres,mgmerayo}@fdi.ucm.es, mn@sip.ucm.es

Abstract. This paper presents a methodology to perform passive testing based on invariants for systems that present temporal restrictions. Invariants represent the most relevant expected properties of the implementation under test. Intuitively, an invariant expresses the fact that each time the implementation under test performs a given sequence of actions, then it must exhibit a behavior in a lapse of time reflected in the invariant. In particular, the algorithm presented in this paper are fully implemented.

1 Introduction

Testing consists in checking the conformance of an implementation by performing experiments on it. In order to perform this task, several techniques, algorithms, and semantic frameworks have been introduced in the literature. The application of formal testing techniques to check the correctness of a system requires to identify the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behavior. In this line, the time consumed by each operation should be considered critical in a real-time system.

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. 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. 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 other types of testing techniques such as *passive testing*. In

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

Cha et al. (Eds.): ATVA 2008, LNCS 5311, pp. 418–427, 2008.

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. Passive testing has very large domains of application. For instance, it can be used as a monitoring technique to detect and report errors (this is the use that we consider in this paper). Another area of application is in network management to detect configuration problems, fault identification, or resource provisioning. It can be also used to study the feasibility of new features as classes of services, network security, and congestion control. Usually, execution traces of the implementation are compared with the specification to detect faults in the implementation. In most of these works the specification has the form of a finite state machine (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 [1]. 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 [2]. It allows to specify wildcard characters in invariants and to include a set of outputs as termination of the invariant. In addition, a new kind of invariants was introduced: Obligation invariants.

In this paper we extend [2] in order to deal with timed restrictions. We will use a simple extension of the classical concept of *Finite State Machines* that allows a specifier to explicitly denote temporal requirements for each action of a system. Intuitively, transitions in timed finite state machines indicate that if the machine is in a state s and receives and input i then after t time units it will produce and output o and it will change its state to s'. Next, we informally introduce the formalism to express temporal conditions in the invariants: Timed invariants. We distinguish between timed restrictions related to each action in the trace represented in the invariant and the one corresponding to the whole trace. For example we could represent properties as *"Each time that a user applies "a" and observes "y" the amount of time the system spends to perform the action is between 3 and 5 time units, if after performing some operations the user applies "b" then he observes "z" in 2 time units and the performance of all these actions does not exceed 10 time units"*.

In our approach, we perform two types of property verification: One on the specification and another one on the traces generated by the implementation. Due to the fact that we assume that the timed invariants can be supplied by the tester, the first step must be to check that the invariant is in fact correct with respect to the specification. An extension of the algorithm proposed in [2] to check this correctness is provided, taking into account the timed conditions that appear in the timed invariants. The next step is to check whether the trace produced by the IUT respects timed 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. Due to the last of space we could not include this part of our research in this paper. A longer version of this paper, including all the previously mentioned algorithms, can be found at [3].

The rest of the paper is organized as follows. In Section 2 we present the notation we apply along the paper. We also introduce our timed extension of the classical finite state machine model. In Section 3 notions of timed invariant and passive testing are presented, as well as the algorithms to check the correctness of invariants with respect to the specification. Finally, Section 4 presents the conclusions of the paper and some lines for future work.

2 Preliminaries

First we introduce notation regarding the definition of time intervals. In this paper we consider that these intervals are contained in $\mathbb{R}_+$, that is, they contain real values greater than or equal to zero.

Definition 1. *We say that* $\hat{a} = [a_1, a_2]$ *is a time interval if* $a_1 \in \mathbb{R}_+$, $a_2 \in \mathbb{R}_+ \cup \{\infty\}$, *and* $a_1 \leq a_2$. *We assume that for all* $t \in \mathbb{R}_+$ *we have* $t < \infty$ *and* $t + \infty = \infty$. *We consider that* $\mathcal{I}_{\mathbb{R}_+}$ *denotes the set of time intervals. Let* $\hat{a} = [a_1, a_2]$ *and* $\hat{b} = [b_1, b_2]$ *be time intervals. We consider the following functions:*

- $\oplus : \mathcal{I}_{\mathbb{R}_+} \times \mathbb{R}_+ \rightarrow \mathcal{I}_{\mathbb{R}_+}$ *defined as* $\oplus(\hat{a}, t) = [a_1 + t, a_2 + t]$.
- $\boxminus : \mathcal{I}_{\mathbb{R}_+} \times \mathcal{I}_{\mathbb{R}_+} \rightarrow \mathcal{I}_{\mathbb{R}_+}$ *defined as* $\boxminus(\hat{a}, \hat{b}) = [\min(a_1, b_1), \max(a_2, b_2)]$ *where* min *and* max *denote the minimum and maximum value respectively.*
- $+ : \mathcal{I}_{\mathbb{R}_+} \times \mathcal{I}_{\mathbb{R}_+} \rightarrow \mathcal{I}_{\mathbb{R}_+}$ *defined as* $[a_1, a_2] + [b_1, b_2] = [a_1 + b_1, a_2 + b_2]$. □

Time intervals will be used to express time constraints associated with the performance of actions. The idea is that if we associate a time interval $[t_1, t_2] \in \mathcal{I}_{\mathbb{R}_+}$ with a task we indicate that this task should take at least t_1 time units and at most t_2 time units to be performed. Intervals like $[0, t]$, $[t, \infty]$, or $[0, \infty]$ denote the absence of a temporal lower/upper bound and the absence of any bound, respectively. Let us note that in the case of $[t, \infty]$ we are abusing the notation since this interval represents, in fact, the interval $[t, \infty)$.

Next we introduce our timed extension of the classical finite state machine model. 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.

Definition 2. A *Timed Finite State Machine*, in the following TFSM, 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, t) 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

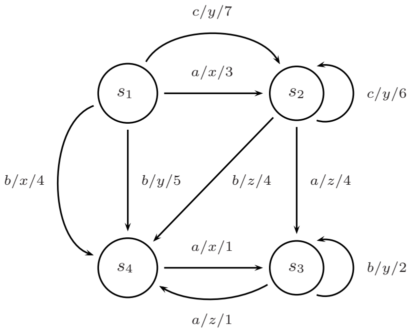

Fig. 1. Example of TFSM

actions, respectively, and $t \in \mathbb{R}_+$ denotes the time that the transition needs to be completed. We say that M is *input-enabled* if for all state $s \in S$ and input $i \in I$, there exist $s' \in S$, $o \in O$, and $t \in \mathbb{R}_+$ such that $(s, s', i, o, t) \in Tr$. □

Intuitively, a transition (s, s', i, o, t) indicates that if the machine is in state s and receives the input i then, after t time units, the machine emits the output o and moves to s'. We denote this transition by $s \xrightarrow{i/o,t} s'$. In Figure 1 we give a graphical representation of a TFSM where s_1 is the initial state. In this paper we assume that all the machines are input enabled. Next, we introduce the notion of *trace* of a TFSM. As usual, a trace is a sequence of input/output pairs. In addition, we have to record the time that the trace needs to be performed.

Definition 3. Let $M = (S, I, O, Tr, s_{in})$ be a TFSM. We say that a tuple such as $(s, s', (i_1/o_1, \ldots, i_r/o_r), t)$ is a *timed trace*, or simply *trace*, of M if there exist $(s, s_1, i_1, o_1, t_1) \ldots (s_{r-1}, s', i_r, o_r, t_r) \in Tr$ such that $t = \sum t_i$. We will sometimes denote a trace $(s, s', (i_1/o_1, \ldots, i_r/o_r), t)$ by (s, σ, s'), where $\sigma = ((i_1/o_1, \ldots, i_r/o_r), t)$. □

3 Timed Invariants

In this section we introduce the notion of timed invariant. These invariants allow us to express temporal properties that must be fulfilled by the implementation. For example, we can express that the time the system takes to perform a transition always belongs to a specific interval. Thus, timed invariants are used to express the temporal restrictions of a trace. In our formalism we assume that timed invariants are given by the tester and are derived from the original requirements. Alternatively, we could consider that invariants are extracted from the specification. In fact, we can do this easily by adapting the method given in [1] to our timed framework. However, this leads to a huge set of invariants, where not all of them are relevant. In our approach we need to check that the timed invariants proposed by the tester are correct with respect to the specification. Once we have a collection of correct timed invariants, we will have to check if these invariants are satisfied by the traces produced by the implementation. In the extended version of the paper [3] we provide an algorithm to check the correctness of the log, recorded from the implementation, with respect to an invariant.

In order to express traces in a concise way, we will use the wildcard characters ? and $\star$. The wildcard ? 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* TFSM*. We say that the sequence I is a (simple) timed invariant for M if the following two conditions hold:*

1. *I is defined according to the following EBNF:*

$$\begin{array}{l} I ::= a/z/\hat{p}, I \mid \star/\hat{p}, I' \mid i \mapsto O/\hat{p} \rhd \hat{t} \\ I' ::= i/z/\hat{p}, I \mid i \mapsto O/\hat{p} \rhd \hat{t} \end{array}$$

   In this expression we consider $\hat{p}, \hat{t} \in \mathcal{I}_{\mathbb{R}_+}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.
2. *I is correct with respect to M.*
3. *We denote the set of simple timed invariants by* SimpleTInv*.* □

Let us remark that time conditions established in invariants are given by intervals. However, machines in our formalism present time information expressed as fix amounts of time. This fact is due to consider that it can be admissible that the execution of a task sometimes lasts more than expected: If most of the times the task is performed on time, a small number of delays can be tolerated. Moreover, another reason for the tester to allow imprecisions is that the artifacts measuring time while testing a system might not be as precise as desirable. In this case, an apparent wrong behavior due to bad timing can be in fact correct since it may happen that the *watches* are not working properly. A longer explanation on the use of time intervals to deal with imprecisions can be found in [4].

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 $a/z/\hat{p}$, with a being an input action or the wildcard character ?, z being an output action or the wildcard character ?, and $\hat{p}$ being a timed interval, or an expression $\star/\hat{p}$. There are two restrictions to this rule. First, an invariant cannot contain two consecutive expressions $\star/\hat{p}_1$ and $\star/\hat{p}_2$. In the case that such a situation was needed to represent a property, the tester could simulate it by means of the expression $*, (\hat{p}_1 + \hat{p}_2)$. The second restriction is that an invariant cannot present a component of the form $\star/\hat{p}$ followed by an expression beginning with the wildcard character ?, that is, the input of the next component must be a *real* input action $i \in \mathcal{I}$. In fact, $\star$ represents any sequence of inputs/outputs pairs such that the input is not equal to i.

The last component, corresponding to the expression $i \mapsto O/\hat{p} \rhd \hat{t}$ is an input action followed by a set of output actions and two timed restrictions, denoted by means of two intervals $\hat{p}$ and $\hat{t}$. The first one is associated to the last expression of the sequence. The last one is related to the sum of time values associated to all input/output pairs performed before. For example, the meaning of an invariant as $i/o/\hat{p}, \star/\hat{p_\star}, i' \mapsto O/\hat{p'} \rhd \hat{t}$ is that if we observe the transition i/o in a time belonging to the interval $\hat{p}$, then the first occurrence of the input symbol i' after a lapse of time belonging to the interval $\hat{p_\star}$, must be followed by an output belonging to the set O. The interval $\hat{t}$ makes reference to the total time that the system must spend to perform the whole trace. This notion of invariant allows us to express several properties of the system under study. Next we introduce some examples in order to present how invariants work.

Example 1. The simplest invariant we can define with our framework for expressing a property of the system follows the scheme $i \mapsto \{o\}/[2,3] \rhd [2,3]$. The idea is that each occurrence of the input i is followed by the output o and this transition is performed between 2 and 3 time units.

We can specify a more complex property by taking into account that we are interested in observing the output o after the input i only if the input i_0 was previously observed. In addition, we include intervals corresponding to the amount of time the system takes for each of the transitions and the total time it spends in the whole trace. We could express this property by means of the invariant $i_0/?/[1,4], \star/[2,5], i \mapsto \{o\}/[2,3] \rhd [2,12]$. An observed trace will be correct with respect to this invariant if each time that we find a (sub)sequence starting with the input i_0 and any output symbol which has been performed in an amount of time belonging to the interval $[1,4]$, then if there is an occurrence of the input symbol i before 5 time units pass then the input i must be paired with the output symbol o and the lapse between i and o must be in the interval $[2,3]$. In addition, the whole sequence must take a time belonging to the interval $[2,12]$. Let us remind that the notion of *correctness* that we just discussed concerns traces observed from the IUT and invariants. A different correctness concept, that we analyze in this paper, relates invariants and specifications.

We can refine the previous invariant if we consider only the cases where the pair i_0/o_0 was observed. The invariant for denoting this property is the following $i_0/o_0/[1,4], \star/[0,5], i \mapsto \{o\}/[2,3] \rhd [2,12]$. Let us remark that we could not deduce that we have found an error if the pair i_0/o_0 appears in the observed trace but the input i is not detected afterwards in the corresponding trace. In such a situation we cannot conclude that the implementation fails. Similarly, if we find the pair i_0/o_1 we cannot conclude anything since the premise of the invariant, that is, the whole sequence but the last pair was not found. Again, the situation is different when analyzing the correctness of an invariant with respect to a specification. For instance, if the specification cannot perform the trace induced by the invariant then we will consider that the invariant is not correct with respect to the specification.

Finally, an invariant such as $i \mapsto \{o_1, o_2\}/[1,4] \rhd [1,4]$ indicates that after input i we observe either the output o_1 or o_2 in a time belonging to [1,4]. □

Since we assume that invariants can be defined by a tester, we must ensure that they are correct with respect to the specification. Next we explain the most relevant aspects of our algorithm to decide whether an invariant is correct with respect to a specification. We separate the algorithm into three different parts.

```
in : $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$.
        $I = \{a_1/\hat{p_1}, \ldots, a_{n-1}/\hat{p_{n-1}}, i_n \mapsto O/\hat{p_n} \rhd \hat{p}\}$ where for all $1 \leq k \leq n-1$ we
have that $\hat{p}_k \in \mathcal{I}_{\mathbb{R}_+}$, and either $a_k = i_k/o_k$, with $i_k \in \mathcal{I} \cup \{?\}$ and $o_k \in \mathcal{O} \cup \{?\}$,
or $a_k = \star$; $i_n \in \mathcal{I}$, $O \subseteq \mathcal{O}$, and $\hat{p_n}, \hat{p} \in \mathcal{I}_{\mathbb{R}_+}$.
out: Bool.
$b$ :: array of $\mathcal{I}_{\mathbb{R}_+}[|S|]$ ;
// an array containing time intervals, having size $|S|$,
//and being $\perp$ the initial value of all positions
$I' = I$; $S' \leftarrow S$; $j \leftarrow 1$; $S'' \leftarrow \emptyset$;
while ($j < n$) do
    $b'$ :: array of $\mathcal{I}_{\mathbb{R}_+}[|S|]$;
    if (head($I'$) $= (\star/\hat{t})$) then
        while ($S' \neq \emptyset$) do
            Choose $s_\alpha \in S'$;
            $S' \leftarrow S' \setminus \{s_\alpha\}$;
            $S_{aux} \leftarrow$ afterInt($s_\alpha, \hat{t}, i_{j+1}$);
            while ($S_{aux} \neq \emptyset$) do
                Choose $(s_p, t) \in S_{aux}$;
                $S_{aux} \leftarrow S_{aux} \setminus \{(s_p, t)\}$;
                $S'' \leftarrow S'' \cup \{s_p\}$;
                if ($b'_p = \perp$) then
                    $b'_p \leftarrow \oplus(b_\alpha, t)$;
                else
                    $b'_p \leftarrow \boxminus(\oplus(b_\alpha, t), {b_p}')$;
    else
        while ($S' \neq \emptyset$) do
            Choose $s_a \in S'$;
            $S' \leftarrow S' \setminus \{s_a\}$;
            $Tr' \leftarrow$ afterCond($s_a, i_j, o_j, \hat{p}_j$);
            while ($Tr' \neq \emptyset$) do
                Choose $(s_a, s_b, i_j, o_j, t) \in Tr'$;
                $Tr' \leftarrow Tr' - \{(s_a, s_b, i_j, o_j, t)\}$;
                if ($b'_b = \perp$) then
                    $b'_b \leftarrow \oplus(b_a, t)$;
                else
                    $b'_b \leftarrow \boxminus(\oplus(b_a, t), {b_b}')$;
                $S'' \leftarrow S'' \cup \{s_b\}$;
    $I' =$ tail($I'$);
    $b \leftarrow b'$; $S' \leftarrow S''$; $S'' \leftarrow \emptyset$;  $j \leftarrow j + 1$;
```

Fig. 2. Correctness of an invariant with respect to a specification (1/3)

The first part of the algorithm (see Figure 2) is responsible for treating the *preface* of the invariant, that is, to determine the states that can be reached in the specification after the first $n-1$ input/output/time tuples have been traversed. The second phase (see Figure 3, left) is used to check that the last pair of the invariant is correct for the specification. In other words, to detect that for all the states computed in the previous step, if the last input of the invariant can be performed then the obtained output belongs to the set of outputs appearing in this last expression of the invariant. In addition we also check that these transitions are performed in the time interval appearing in the invariant. Finally, the third part of the algorithm (see Figure 3, right) verifies the last part of the invariant: The sequence is always performed in a time belonging to the corresponding interval. Next we introduce additional notation.

Definition 5. Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a `TFSM`, $s \in S$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $\hat{t} \in \mathcal{I}_{\mathbb{R}_+}$. We define the set $\mathtt{afterCond}(s, a, z, \hat{t})$ as the set of transitions belonging to Tr having as initial state s, as input a, as output z, and such that its time belongs to the interval $\hat{t}$.

$$\mathtt{afterCond}(s,i,o,\hat{t}) = \{(s,s',i,o,t) | \exists s' \in S, t \in \mathbb{R}_+ (s,s',i,o,t) \in Tr \wedge t \in \hat{t}\}$$

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

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

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

We define the function $\mathtt{afterInt}(s, \hat{t}, i)$ as the function that computes the set of pairs (s', t) of states $s' \in S$ that can be reached from state s after t time units, belonging t to the interval $\hat{t}$, and such that the input i is not performed. We will use an auxiliary function so that $\mathtt{afterIntAux}(s, \hat{t}, i) = \mathtt{afterIntAux}(s, \hat{t}, i, 0)$, being this function defined as follows:

$$\mathtt{afterIntAux}(s,\hat{t},i,tot) = \{(s,tot) | tot \in \hat{t}\} \\ \cup \\ \bigcup_{\substack{(s,s'',i',o,t) \in Tr \\ \hat{t} \odot (tot+t) \\ i \neq i'}} \mathtt{afterIntAux}(s'',\hat{t},i,tot+t)$$

where $\odot : \mathcal{I}_{\mathbb{R}_+} \times \mathbb{R}_+ \rightarrow \{true, false\}$ is defined as $[t_1, t_2] \odot t = (t \leq t_2)$. □

In the first phase of the algorithm we have to initially obtain the set of states that can perform the first input/output of the invariant. We compute the states that can be reached from that initial set after performing that transition and such that the time value associated with the transition falls within the range marked by the invariant. We iterate this process until we reach the last expression of the invariant. We consider two auxiliary functions: `head()` returns the first element of the invariant and `tail()` removes this first element from the invariant. Let us remark that we distinguish between input/output pairs, possibly including the wildcard ?, and occurrences of $\star$. In the latter case we will use the previously defined `afterInt`() function to compute the corresponding reached states.

```
$error \leftarrow$ false;
if $(S' = \emptyset)$ then
    $error \leftarrow$ true;
end
$b'$ :: array of $\mathcal{I}_{\mathbb{R}_+}[|S|]$;
while $(S' \neq \emptyset)$ do
    Choose $s_a \in S'$;
    $S' \leftarrow S' \setminus \{s_a\}$;
    $Tr' \leftarrow$ afterCond$(s_a, i_n, ?, [0, \infty])$;
    while $(Tr' \neq \emptyset)$ do
        Choose $(s_a, s_b, i_n, o, t) \in Tr'$;
        $Tr' \leftarrow Tr' \setminus \{(s_a, s_b, i_n, o, t)\}$;
        if $((o \in O) \wedge (t \in \hat{p_n}))$ then
            if $(b'_b = \perp)$ then
                $b'_b \leftarrow \oplus(b_a, t)$;
            else
                $b'_b \leftarrow \boxminus(\oplus(b_a, t), b'_b)$;
            $S'' \leftarrow S'' \cup \{s_b\}$;
        else
            $error \leftarrow$ true
```

```
if $(S'' = \emptyset)$ then
    $error \leftarrow$ true;
end
while $(S'' \neq \emptyset)$ do
    Choose $s_i \in S''$;
    $S'' \leftarrow S'' \setminus \{s_i\}$;
    if $(\neg(b'_i \subseteq \hat{p}))$ then
        $error \leftarrow$ true;
return $(\neg error)$;
```

Fig. 3. Correctness of an invariant with respect to a specification (2/3) and (3/3)

The *input* of the second phase of the algorithm (see Figure 3, left) is the set of states that can be reached after the preface of the invariant is performed. In addition, we also record the time that it took to reach each of these states. If this set is empty then the invariant is not correct. The idea is that we should not use an invariant such that its sequence of input/output/interval cannot be performed in the specification. If this set is not empty, we will check that for all reached states if they can perform the last input of the invariant then the obtained output must belong to the set of outputs appearing in this last expression of the invariant. In addition, time values have to belong to the time interval of the invariant.

The third step of the algorithm (Figure 3, right) will be devoted to check that the time behavior of the whole invariant is correct with respect to the specification. In order to do this, in the previous stages we recorded all the time values associated with the performance of input/output pairs. We use the functions $\oplus$ and $\boxminus$ to operate with the recorded time values and construct an interval. Thus, in the position k of the array b we store an interval that has as bounds the minimal/ maximal times that are needed to reach the state k after performing the whole invariant. If a state is not reachable after the sequence associated with the invariant then $b[k] = \perp$. Next, we concentrate only in states of the specification that can be reached, that is, $b[k] \neq \perp$ and check that all those intervals are contained in the interval appearing at the very last position of the invariant.

Lemma 1. *Let $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in})$ be a* TFSM. *The worst case of the algorithm given in Figures 2 and 3 checks the correctness of the invariant $I = i_1/o_1/\hat{p}_1, \ldots, i_{n-1}/o_{n-1}/\hat{p}_{n-1}, i_n \mapsto O/\hat{p}_n \rhd \hat{t}$ 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 occurrences of* $\star$*, being* k *the number of* $\star$*'s in* I*.* □

4 Conclusions and Future Work

In this paper we have introduced a new methodology for passive testing systems that present timed constraints over the duration of the actions. We introduced an extension of the classical Finite State Machine model in order to deal with this kind of systems. This methodology extends the definition of an invariant, allowing to express properties regarding temporal conditions that the IUT must fulfill. We presented an algorithm which allows to establish the correctness of the proposed invariants with respect to a given specification. In a longer version of this paper [3] we also deal with the correctness of an observed trace with respect to an invariant.

Regarding future work, we plan to extend the family of invariants. In fact, we have already developed a timed version of *obligation invariants* [2]. The second line of future work consists in performing *real* experiments. The experience gained with the WAP protocol during the preparation of [2] makes this a good candidate to study time properties in our passive testing framework.

References

1. Cavalli, A., Gervy, C., Prokopenko, S.: New approaches for passive testing using an extended finite state machine specification. Journal of Information and Software Technology 45, 837–852 (2003)
2. Bayse, E., Cavalli, A., Núñez, M., Zaïdi, F.: A passive testing approach based on invariants: Application to the WAP. Computer Networks 48(2), 247–266 (2005)
3. Andrés, C., Merayo, M., Núñez, M.: Passive testing of timed systems (2008), `http://kimba.mat.ucm.es/~manolo/papers/atva08-passive-extended.pdf`
4. Merayo, M., Núñez, M., Rodríguez, I.: Formal testing of systems presenting soft and hard deadlines. In: Arbab, F., Sirjani, M. (eds.) FSEN 2007. LNCS, vol. 4767, pp. 160–174. Springer, Heidelberg (2007)