Formal correctness of a passive testing approach for timed systems*

César Andrés, Mercedes G. Merayo, Manuel Núñez
Deptartamento 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 extend our previous work on passive testing of timed systems to establish a formal criterion to determine correctness of an implementation under test. In our framework, an invariant expresses the fact that if 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 a previous paper we gave an algorithm to establish the correctness of an invariant with respect to a specification. In this paper we continue the work by providing an algorithm to check the correctness of a log, recorded form the implementation under test, with respect to an invariant. We show the soundness of our method by relating it to an implementation relation. In addition to the theoretical framework we have developed a tool, called PASTE*, that facilitates the automation of our passive testing approach.*

1 Introduction

The complexity of current systems, the large number of persons working on them, and the number of different modules that interact with each other, make it difficult to decide the correctness of these systems. *Testing techniques* allow us to provide a grade of confidence in the correctness of a system. Testing techniques can use formal methods [16, 6, 22, 12] in order to semi-automatically perform some tasks through the use of tools [26] that help to increase the confidence on the correctness of the systems under test. Formal testing originally targeted the functional behavior of systems, such as determining whether the tested system, on the one hand, performs certain actions and, on the other hand, does not perform some unexpected ones. The application of formal testing techniques to check the correctness of a system requires identifying the *critical* aspects of the system, that is, those aspects that will make the difference between correct and incorrect behaviors. 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. Thus, during the last decade formal testing techniques also deal with non-functional properties. In this paper we focus on systems that contain temporal restrictions, being already several proposals for timed testing (e.g. [17, 8, 14, 23, 9, 19, 18, 11]).

In testing, there is usually a distinction between two approaches: *Passive* and *Active*. The main difference between them is whether a tester can interact with the implementation under test (`IUT`). If the tester can interact with the `IUT` we are in the active testing paradigm. On the contrary, if the tester simply monitors the behavior of the `IUT`, then we are in the passive testing paradigm. Actually, it is very frequent that the tester is unable to interact with the `IUT`, or that the internal non determinism of the system makes difficult to interact with it. In particular, such interaction can be difficult in the case of large systems working 24/7 since this interaction might produce a wrong behavior of the system. An example of these systems is the one that we present along this paper, which we name `SSadmin`. This system allows students to check their marks, to change their personal profile and, at the beginning of the academic year, to make the registration of the subjects to be taken. Testers are not allowed to introduce their own set of tests because this could damage the database structure. So, they cannot perform active testing. Therefore, they use passive testing techniques by interacting with *logs* recorded from `SSadmin`. These logs contain the historic interaction between the students and the system. Thus, they can be compared with information extracted from the specification to detect faults in the implementation.

There are several papers on passive testing, but the number diminishes when we restrict ourselves to formal approaches. First, we would like to mention the approach studied in [7]. There, a set of properties, called *invariants*, are extracted from the specification and checked on

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

978-0-7695-3671-2/09 $25.00 © 2009 IEEE
DOI 10.1109/ICSTW.2009.34

the traces observed from the implementation to test their correctness. One of the drawbacks of this work is the limitation on the grammar used to express invariants. A new formalism that overcomes this restriction for expressing invariants was presented in [3, 5]. In particular, it allows to specify wild-card characters in invariants and to include a set of outputs as termination of the invariant.

Even though work on passive testing has been carried on for several years, it can be dated back at least to [4], work on passive testing of timed systems is very recent [1, 2]. We propose an extension of the previous work on formal passive testing so that temporal properties can be also taken into account. For example, we may be interested in checking whether a system, more properly, the logs observed in a system, fulfill the following property:

> *Each time that a user applies* a *and observes* y *the amount of time the system spends to perform the action* y *is between* 3 *and* 5 *time units; if after performing some operations the user applied* b *then he must observe* z *before* 2 *time units and the performance of all these actions must not exceed* 10 *time units.*

The approach to passively test timed systems presented in this paper was initiated in [1]. In that paper we gave the syntax for invariants and an algorithm to check the appropriateness of an invariant with respect to a specification. In this paper we introduce an algorithm to check the correctness of a log extracted from the IUT with respect to a time invariant. We show that this process is *sound* in the sense that the if log does not match a *correct*, from a certain specification, invariant then the IUT does not conform to this specification. Such a formal *soundness* of passive testing is ignored in most passive testing approaches. Obviously, our procedure cannot be *complete* since passive testing cannot ensure that a faulty IUT will reveal a fault in the observed log. We also present our tool PASTE. This tool implements all the algorithms presented in this framework. We also include in the core of PASTE an algorithm to classify time invariants with respect to their power to detect errors.

The rest of the paper is structured as follows. In Section 2 we introduce our formal framework to specify timed systems and in Section 3 the one to describe time invariants. These sections review previous work reported in [1]. In addition, we include a new case study to evaluate our approach and provide an alternative characterization of the notion of correctness introduced in [1]. Section 4 presents the material related to the correctness of the approach: A mismatch of a log with a correct invariant implies a faulty IUT. In Section 5 we present some features of our tool PASTE. We conclude this paper in Section 6 where we give our conclusions and some lines for future work.

2 Preliminaries

In this section we present our formal framework [1]. First, we provide some notation about intervals and time.

Definition 1 We call any value $t \in \mathbf{R}_+$ a *fixed time value*. For all $t \in \mathbf{R}_+$ we have both $t < \infty$ and $t + \infty = \infty$.

We say that $\hat{p} = [p_1, p_2]$ is a *time interval* if $p_1 \in \mathbf{R}_+$, $p_2 \in \mathbf{R}_+ \cup \{\infty\}$, and $p_1 \leq p_2$. We consider that $\mathcal{IR}$ denotes the set of time intervals. We consider the following functions:

- $+ : \mathcal{IR} \times \mathcal{IR} \rightarrow \mathcal{IR}$. If $\hat{p} = [p_1, p_2]$ and $\hat{q} = [q_1, q_2]$ are time intervals then $\hat{p} + \hat{q} = [p_1 + q_1, p_2 + q_2]$.
- $\odot : \mathcal{IR} \times \mathbf{R}_+ \rightarrow \{true, false\}$. If $\hat{p} = [p_1, p_2]$ is a time interval and $t \in \mathbf{R}_+$ then $\hat{p} \odot t = (t \leq p_2)$.

□

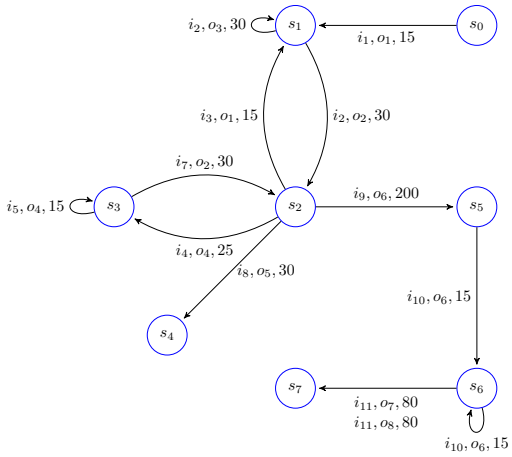
In our framework we consider an adaptation of the well known finite state machines model where we add time to transitions. The time value associated with each transition represents the amount of time that this transition needs to be performed.

Definition 2 A *Timed Finite State Machine*, in the following TFSM, is a tuple $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ where $\mathcal{S}$ is a finite set of states, $\mathcal{I}$ is the set of input actions, $\mathcal{O}$ is the set of output actions, $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{O} \times \mathbf{R}_+ \times \mathcal{S}$ is the set of transitions, and $s_0 \in \mathcal{S}$ is the initial state.

We say that M is *input-enabled* if for all state $s \in \mathcal{S}$ and input $i \in \mathcal{I}$, there exist $s' \in \mathcal{S}$, $o \in \mathcal{O}$, and $t \in \mathbf{R}_+$ such that $s \xrightarrow{i,o}_t s' \in \mathcal{T}$. We say that M is *observable* if it does not have two transitions such as $s \xrightarrow{i,o}_{t_1} s_1$ and $s \xrightarrow{i,o}_{t_2} s_2$. □

A transition belonging to $\mathcal{T}$ is a tuple (s, i, o, t, s') where $s, s' \in \mathcal{S}$ 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 $t \in \mathbf{R}_+$ denotes the time that the transition needs to be completed. Along this paper we will use $s \xrightarrow{i,o}_t s'$ as a shorthand to represent the transition (s, i, o, t, s').

Regarding the notion of input-enabled, let us comment the differences with respect to the classical notion. What we mean by input-enabled is that the machine will be able to accept any input only when it is willing to accept inputs. This means that if a machine is performing a transition $s \xrightarrow{i,o}_t s'$, then during the t time units that the transition needs to be completed we assume that the machine is not accepting inputs. Informally, we can consider that the inputs received in this lapse can be stored to be processed once the transition is performed. In order to formalize the previous intuition, we could use an adaption to our formalism of [15] where a similar concept is used in the context of



i_1= `connect`
i_2= `login`
i_3= `disconnection`
i_4= `profile`
i_5= `data`
i_6= `cancel`
i_7= `save`
i_8= `marks`
i_9= `register`
i_{10}= `data_subject`
i_{11}= `save_registration`
i_{12}= `return_option`
o_1= `welcome_screen`
o_2= `option_screen`
o_3= `error_user`
o_4= `profile_screen`
o_5= `marks_screen`
o_6= `register_screen`
o_7= `confirm_screen`
o_8= `no_confirmation_screen`
o_9= `not_spected`

Figure 1. Specification of `SSadmin` using a `TFSM` model.

input-output labeled transition systems. Let us note that our notion of observability allows some degree of nondeterminism. For example we allow transitions such as $s \xrightarrow{i,o_1}_{t_1} s_1$ and $s \xrightarrow{i,o_2}_{t_1} s_2$, as far as $o_1 \neq o_2$. In this paper we consider that all defined `TFSMs` are input-enabled and observable.

Next, we present the specification of `SSadmin` modeled by a `TFSM`. As we mentioned in the introduction such a system is used by students to check their marks, their student information profile and, at the beginning of the academic year, to register their subjects. The system presented in this paper is a simplification (in particular, we do not use data in our formalism) of the one used in our University. In Figure 1 we give a graphical representation of the specification: Nodes represent states, s_0 being the initial state, and edges represent transitions, in $\mathcal{T}_0 \subset \mathcal{T}$. For example, the loop transition from s_1 to s_1 represents $(s_1, i_2, o_3, 30, s_1)$. Let us note that not all transitions are drawn in the figure since this would overload the graph. There are two sets of transitions, $\mathcal{T}_1$ and $\mathcal{T}_2$, that also take part in set of transitions $\mathcal{T}$ of M. Thus, $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{T}_1 \cup \mathcal{T}_2$ where

$$T_1 = \{(s, i_{12}, o_2, 10, s_2) | s \in \{s_3, s_4, s_5, s_6, s_7\}\}$$

$$\mathcal{T}_2 = \left\{ (s, i, o_9, 5, s) \;\middle|\; \begin{array}{l} \nexists s' \in \mathcal{S}, o \in \mathcal{O}, t \in \mathbf{R}_+ : \\ (s, i, o, t, s') \in \mathcal{T}_0 \cup \mathcal{T}_1 \end{array} \right\}$$

The first set of additional transitions, $\mathcal{T}_1$, allows the students, when they are connected, to return to the `option_screen`. The second set, $\mathcal{T}_2$, ensures that the specification is input-enabled by adding *bogus* transitions.

Next, we describe a usual interaction between a student and `SSadmin`. We divide the behavior of the system in three different stages. The first one corresponds to the connection. Initially, the student wants to `connect` to the system. She connects to the web page and the system shows the `welcome_screen`. Let us note that the specification is described from the point of view of the system, and this example is presented from a student point of view because it is more intuitive. Thus, inputs of the student are output of the system and viceversa. When the student sees the `welcome_screen` she can login into the system. If an erroneous `login` is introduced then the system shows the `error_user`. If the student introduces a correct `login`, then the system will show the `option_screen`. If the student wants to disconnect, then she only has to press `disconnection` and `SSadmin` returns to the `welcome_screen`. The time values associated with these transitions are 15 and 30 time units, respectively. The bigger amount of time denotes that the system is interacting with the database searching the student login.

In the second stage we include the more used operations of the students within `SSadmin`. These are the task of checking their marks, and the one for accessing and modifying their personal profile. When a student is connected with `SSadmin`, if she introduces `profile` then the system will show `profile_screen`. In that screen the student is able to change some personal information, such as email and telephone number. Each data that she wants to change is introduced by the `data` action and the system continues showing `profile_screen`. When the student introduces all the data she can both `save` them or `cancel` the operation. Both actions make `SSadmin` to show the `option_screen`. The second operation of this stage is for checking the marks. The student can see them by introducing the `marks` action and the system will show `marks_screen`. To return to the `option_screen` the student introduces the `cancel` action. The time values associated with these transitions reflects if the values are extracted from the disk, or if the values are in temporal mem-

ory (for example, when the student is changing her profile data and the dates are not saved yet).

The last stage corresponds to the *register* feature. This feature is available only at the beginning of an academic course. This is one of the most important and critical parts of SSadmin. A student who wants to register a set of subjects has firstly to connect to the system. After that, she can introduce register and the system will show the register_screen. The time associated with this transition is 200. This amount of time is big because the system has to search and check all possible subjects where the student can register. Then, the student can introduce the data_subject into the system in order to choose the correct subjects. When she wants to finish she introduces save_registration and the system takes 80 time units in order to show if the registration was correct (confirm_screen) or there was an error (no_confirmation_screen).

Next we introduce the notion of *trace*. A trace represents a sequence of actions that the system may perform from its origin state. A *timed evolution* is a simplification of the representation of the trace, focusing only in the sequence of input/outputs/times values of the trace. We distinguish between *non-timed* evolution and time evolution. All these notions will be used in the next section to define *implementation relations*. As usual, a *log* is a sequence of actions representing the historical evolution of a system, that is, a timed evolution of the system.

Definition 3 Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. A *trace* of M belongs to $\mathcal{S} \times (\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star \times \mathcal{S}$. We say that $(s, (i_1/o_1/t_1, \ldots, i_n/o_n/t_n), s_n)$ is a trace of M if there exist $s_1, \ldots, s_n$ such that we have the following transitions: $s \xrightarrow{i_1,o_1}_{t_1} s_1$,..., $s_{n-1} \xrightarrow{i_n,o_n}_{t_n} s_n$. We denote by $\texttt{Traces}(M)$ the set of all traces of M.

A *non-timed evolution* belongs to $(\mathcal{I} \times \mathcal{O})^\star$. We say that $(i_1/o_1, \ldots, i_n/o_n)$ is a non-timed evolution of M if there exist s, s' such that $(s, (i_1/o_1/t_1, \ldots, i_n/o_n/t_n), s') \in \texttt{Traces}(M)$. We denote by $\texttt{NTEvol}(M)$ the set of non-timed evolutions of M.

A *timed evolution* belongs to $(\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star$. We say that $(i_1/o_1/t_1, \ldots, i_n/o_n/t_n)$ is a timed evolution of M if there exist $s_1, \ldots, s_n$ such that $(s, (i_1/o_1/t_1, \ldots, i_n/o_n/t_n), s') \in \texttt{Traces}(M)$. We denote by $\texttt{TEvol}(M)$ the set of timed evolutions of M.

We define the function *total time* $\texttt{TT} : \texttt{TEvol} \to \mathbf{R}_+$ as the sum of all time values appearing in a timed evolution. Formally, if $e = (i_1/o_1/t_1, \ldots, i_n/o_n/t_n)$ then $\texttt{TT}(e) = \sum_{j=1}^{n} t_j$.

We define the *non-timed equality* relation between timed evolutions, denoted by $=_{nt}$, as follows. Let $e = (i_1/o_1/t_1, \ldots, i_n/o_n/t_n)$ and $e' = (i'_1/o'_1/t'_1, \ldots, i'_n/o'_n/t'_n)$ belong to $\texttt{TEvol}(M)$. We write $e =_{nt} e'$ iff for all $1 \leq j \leq n$ we have that $(i_j = i'_j \wedge o_j = o'_j)$.

A *log* from M is a finite sequence belonging to $\texttt{TEvol}(M)$. The set of all logs is denoted by Log. □

3 Time Invariants

In this section we introduce the notion of time invariant. Time invariants are used in our approach to represent the properties that we would like to check against the logs extracted from the IUT. The notion of time invariant being *correct* means that if the time invariant detects a mismatch, then the implementation that has generated this log is incorrect with respect to the specification. Firstly, after generating a set of time invariants and before checking them against the log, they must be checked against the specification; otherwise, we might have that an invariant which indicates an erroneous that might not violate the requirements expressed in the specification. The algorithm to decide the correctness of a time invariant with respect to a specification appears in [1]. Another possibility is to consider that invariants are correct *by definition*. In this case we can ignore the specification since a mismatch will automatically imply that a fault was detected.

Definition 4 Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. We say that the sequence ϕ is a *time invariant*, or simply invariant, for M if the following two conditions hold:

1. ϕ is defined according to the following EBNF:

$$\phi ::= a/z/\hat{p}, \phi \mid \star/\hat{p}, \phi' \mid i \mapsto O/\hat{p} \rhd \hat{q}$$
$$\phi' ::= i/z/\hat{p}, \phi \mid i \mapsto O/\hat{p} \rhd \hat{q}$$

   In this expression we consider $\hat{p}, \hat{q} \in \mathcal{IR}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$.

2. ϕ is *correct* with respect to M as defined in [1].

The set of invariants for M is denoted by Φ_M, where we will drop the subindex if it can be deduced from the context.

Let $e = (i_1/o_1/t_1, \ldots, i_r/o_r/t_r) \in \texttt{TEvol}(M)$ and $\phi = (\xi_1/\hat{p}_1, \ldots, \xi_n/\hat{p}_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f) \in \Phi_M$. We say that e *matches* ϕ if after applying Algorithm 1 we have that $n = j$ (the loop has visited the first n positions of the invariant) and $m < r$ (the trace contains at least one element so that its *head* can be compared with the last part of the invariant: $i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$). □

We can use the previous definition of matching to give an alternative characterization of the notion of correctness introduced [1]. The proof is easy but cumbersome since both notions are based on the same idea: Traverse the trace and detect a subsequence that contradicts what is stated by the invariant.

Lemma 1 Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM and $\phi \in \Phi_M$. We have that ϕ is correct with respect to M according to [1] iff for all $e = (i_1/o_1/t_1, \ldots, i_r/o_r/t_r) \in \texttt{TEvol}(M)$ matching ϕ we have that $i_e = i_f$ implies

$$o_e \in O \wedge t_e \in \hat{p}_f \wedge \texttt{TT}(i_{m'}/o_{m'}/t_{m'}, \ldots, i_m/o_m/t_m) \in \hat{q}_f$$

where $(i_e/o_e/t_e)$ is the m-*th* element of e, having the variables m and m' the values returned by Algorithm 1. □

Example 1 We consider the specification of SSadmin described in Section 2. A simple invariant can denote the property that after login, an user has to disconnect from the system. We represent this as the invariant

$$\begin{aligned}\phi_1 = \ & \texttt{login/option\_screen}/[20,40], \star/[0,\infty], \\ & \texttt{disconnection} \mapsto \\ & \quad \{\texttt{welcome\_screen}\}/[10,20] \rhd [35,\infty]\end{aligned}$$

ϕ_1 means that if we see in the log the input login followed by option_screen and an amount of time included in [20,40], and we observe any sequence of input/output/time without the input disconnection, then if we observe disconnection we have to see the output welcome_screen in an amount of time belonging to $[10, 20]$. In addition, the sum of all time values observed from the login until the welcome_screen output belongs to $[35, \infty]$.

Next we define another invariant for the first stage:

$$\begin{aligned}\phi_2 = \ & \texttt{login} \mapsto \\ & \quad \{\texttt{option\_screen}, \texttt{error\_user}\}/[10,40] \\ & \quad \rhd [10,40]\end{aligned}$$

In this invariant, the final set of outputs contains two different outputs. Intuitively, this invariant expresses that after observing any occurrence of login in the log then we have to observe option_screen or error_user. Moreover, the amount of time associated with this input/output must belong to the time interval $[10, 40]$.

From the second stage of SSadmin, the following invariant, focusing in the profile options, can be considered

$$\begin{aligned}\phi_3 = \ & \texttt{data/profile\_screen}/[10,20], \\ & \texttt{save} \mapsto \\ & \quad \{\texttt{option\_screen}\}/[20,40] \rhd [35,50]\end{aligned}$$

ϕ_3 denotes that after inserting the last change into the profile_screen, and after saving the current state of the system, the option_screen and the time associated with these operations are included in the intervals $[10, 20]$ and $[20, 40]$ respectively. In addition, the total amount of time to perform this activity must belong to the interval $[35, 50]$. Another invariant for the second stage can focus on the marks part. The invariant can represent that when the

//Input Data:
//$\phi = (\xi_1/\hat{p}_1, \ldots, \xi_n/\hat{p}_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f)$
//$e = (i_1/o_1/t_1, \ldots, i_r/o_r/t_r)$
$m = 1$;
$m' = 1$;
$j = 1$;
while $j \leq n \wedge (m' + n) < r \wedge m < r$ **do**
  if $j == 1$ **then**
    $m' = m' + 1$
  end
  $(i_e/o_e/t_e) = e[m]$//m-*th* element of e;
  if $\xi_j == (i/o)$ **then**
    if $i == i_e \wedge o == o_e \wedge t_e \in \hat{p}_j$ **then**
      $m = m + 1$; $j = j + 1$;
    else
      $j = 1$; $m = m'$;
    end
  end
  if $\xi_j == (?/o) \wedge t_e \in \hat{p}_j$ **then**
    if $o == o_e$ **then**
      $m = m + 1$; $j = j + 1$;
    else
      $j = 1$; $m = m'$;
    end
  end
  if $\xi_j == (i/?) \wedge t_e \in \hat{p}_j$ **then**
    if $i == i_e$ **then**
      $m = m + 1$; $j = j + 1$;
    else
      $j = 1$; $m = m'$;
    end
  end
  if $\xi_j == (?/?) \wedge t_e \in \hat{p}_j$ **then**
    $m = m + 1$; $j = j + 1$;
  end
  if $\xi_j == \star$ **then**
    $m'' = m$; $t'' = 0$;
    $(i', -) = (\xi_{j+1})$;
    while $i' \neq i_e \wedge \hat{p}_j \odot t'' \wedge m'' \leq r$ **do**
      $m'' = m'' + 1$; $t'' = t'' + t_e$;
      $(i_e/o_e/t_e) = e[m'']$;
    end
    if $t'' \in \hat{p}_j$ **then**
      $m = m''$; $j = j + 1$;
    else
      $j = 1$; $m = m'$;
    end
  end
end

Algorithm 1: Matching e and ϕ.

student is in the option_screen, if she inserts the input interaction marks, then the marks_screen will appear:

$$\phi_4 = \ ?/\texttt{option\_screen}/[5,35], \ \texttt{marks} \mapsto \{\texttt{marks\_screen}\}/[20,40] \rhd [30,70]$$

□

4 Correctness of checking invariants against logs

In this section we first define an implementation relation to show the correctness of our approach. Next, we present an algorithm to check the conformance between logs and invariants. We start by introducing an implementation relation where time is not considered.

Definition 5 Let S and I be two TFSMs. We say that I *non-timely conforms* to S, denoted by $I\ \texttt{conf}_{nt}\ S$, if for all $e = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o_n) \in \texttt{NTEvol}(S)$, with $n \geq 1$, we have that

$$e' = (i_1/o_1, \ldots, i_{n-1}/o_{n-1}, i_n/o'_n) \in \texttt{NTEvol}(I) \Downarrow e' \in \texttt{NTEvol}(S)$$

□

The idea underlying the definition of non-timely conformance relation is that the implementation does not *invent* anything for those inputs that are *specified* in the formal model. Next we introduce our timed implementation relation. We will define only one implementation relation, but any of the timed relations introduced in [19] could be easily incorporated in our framework. The $\texttt{conf}_a$ relation (conforms *always*) considers that for all timed evolution e of the implementation, if e is a non-timed evolution of the specification S, then e is also a timed evolution of S. With this relation we express that the implementation mimics the timed behavior of the formal model. Let us remark that since we are considering observable machines, there is at most one evolution fulfilling the previous conditions.

Definition 6 Let S and I be two TFSMs. We write $I\ \texttt{conf}_a\ S$ iff $I\ \texttt{conf}_{nt}\ S$ and for all $e \in \texttt{NTEvol}(I) \cap \texttt{NTEvol}(S)$ we have that $e \in \texttt{TEvol}(I)$ implies $e \in \texttt{TEvol}(S)$. □

Next, we define the correctness of a log observed from a TFSM with respect to a invariant. Essentially, we will detect an error if there exists a subsequence of the log that does not match the invariant. This definition combines the notion of matching, introduced in Definition 4, and how to deal with the last part of the invariant, as used in Lemma 1.

Definition 7 Let $\phi = \xi_1/\hat{p}_1, \ldots, \xi_n/\hat{p}_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$ be a time invariant and e be a log from a TFSM. We say that e is *correct* with respect to ϕ, denoted by $e \approx_a \phi$, iff either e does not match ϕ (see Definition 3) or e matches o and $i_e = i_f$ implies

$$o_e \in O \wedge t_e \in \hat{p}_f \wedge \texttt{TT}(i_{m'}/o_{m'}/t_{m'}, \ldots, i_m/o_m/t_m) \in \hat{q}_f$$

where $(i_e/o_e/t_e) = e[m]$, having the variables m and m' the values returned by Algorithm 1. □

Theorem 1 Let S and I be two TFSMs and ϕ be a correct time invariant with respect to S. Let e be a log recorded from I. If the invariant ϕ does not match e then I does not conform to S.

Proof: Let $\phi = \xi_1/\hat{p_1}, \ldots, \xi_n/\hat{p_n}, i_f \mapsto O/\hat{p_f} \rhd \hat{q}_f$ be a correct invariant. Let us assume that $e \approx_a \phi$ does not hold and we will find a contradiction. First, if $e \approx_a \phi$ does not hold then, applying Definition 7, there exists a subsequence $e' = (i_1/o_1/t_1, \ldots, i_k/o_k/t_k)$ of e such that $i_k = i_f$ and $o_k \notin O$, or $t_k \notin \hat{p}_f$, or $\texttt{TT}(e') \notin \hat{q}_f$. Let us consider the three possible cases.

If $o_k \notin O$ then we have $e' \in \texttt{NTEvol}(I)$ but $e' \notin \texttt{NTEvol}(S)$. Thus, applying Definition 5, we have that $I\ \texttt{conf}_{nt}\ S$ does not hold, which automatically implies $I\ \texttt{conf}_a\ S$ does not hold.

If $t_k \notin \hat{p_n}$ then we have that $e' \in \texttt{TEvol}(I)$ but $e' \notin \texttt{TEvol}(S)$. Thus, applying Definition 6, we obtain that $I\ \texttt{conf}_a\ S$ does not hold.

If $\texttt{TT}(e') \notin \hat{q}_f$ then we also deduce $e' \notin \texttt{TEvol}(S)$ because time values of invariants are coherent with those of the specification. In this case, by applying again Definition 6, we obtain that $I\ \texttt{conf}_a\ S$ does not hold. □

We conclude this section by giving an alternative algorithm to establish the conformance between logs and invariants. The problem with Algorithm 1, and its derived notion introduced in Definition 7, is its inefficiency, since it repeats a lot of computations when applied to all the subsequences of a trace due to backtracking when a prefix of the invariant does not match the current subsequence of the log. Nevertheless, the algorithm is interesting from the theoretical point of view since it provides a compact and formal criterion to decide whether a trace matches an invariant. It is straightforward to prove that the new algorithm returns the same results as the former one.

First, we explain the main features of the new algorithm that to establish the conformance of a log obtained from the IUT with respect to an invariant. We present the core of the algorithm in Figure 2. We also use the auxiliary function treated presented in Figure 3. The algorithm traverses all the elements of the log, comparing each of them with the first component of the invariant. If the current element of the sequence matches the input/output pair presented in

input : $e = (i_1/o_1/t_1, \ldots, i_r/o_r/t_r) :: \texttt{Log}$,
$\phi = \{\xi_1/\hat{p_1}, \ldots, \xi_n/\hat{p_n}, i_f \mapsto O/\hat{p_f} \rhd \hat{q}_f\} :: \Phi$
// where for all $1 \leq l \leq n$ *we have that* $\hat{p}_l \in \mathcal{IR}$,
// and either $\xi_k = i_k/o_k$, *with* $i_k \in \mathcal{I} \cup \{?\}$
// and $o_k \in \mathcal{O} \cup \{?\}$, *or* $\xi_k = \star$;
// $i_f \in \mathcal{I}$, $O \subseteq \mathcal{O}$, *and* $\hat{p_f}, \hat{q}_f \in \mathcal{IR}$.
output: `Bool`

Struct $\mathcal{A}$ $\{t_t :: \mathbf{R}_+;\ t_e :: \mathcal{IR};\ t_a :: \mathbf{R}_+;$
$wild :: \texttt{Bool};\ \phi_{aux} :: \Phi;\}$
$b :: Stack[\mathcal{A}]$;
$b_{aux} :: Stack[\mathcal{A}]$;
$token :: \mathcal{A}$;
$error \leftarrow \texttt{false}$; $j \leftarrow 1$;

while $(j \neq \texttt{length}(e) \ \wedge \neg error)$ **do**
  // we access the j-position of the log
  $(i/o/t) \leftarrow e[j]$;
  $j \leftarrow j + 1$;
  $token.t_t \leftarrow 0$;
  $token.t_e \leftarrow [0, 0]$;
  $token.t_a \leftarrow 0$;
  $token.wild \leftarrow \texttt{false}$;
  $token.\phi_{aux} \leftarrow \phi$;
  $aux \leftarrow \texttt{treated}((i\ /\ o\ /\ t), token, error)$;
  // check if current position holds with
  // the first component of the invariant
  if $(aux \neq null)$ **then**
    $\texttt{push}(b_{aux}, aux)$;
  while $\neg(\texttt{isEmpty}(b))$ **do**
    $token \leftarrow \texttt{top}(b)$;
    $aux \leftarrow \texttt{treated}((i\ /\ o\ /\ t), token, error)$;
    if $(aux \neq null)$ **then**
      $\texttt{push}(b_{aux}, aux)$;
  $b \leftarrow b_{aux}$;
return$(\neg error)$;

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

input : $(i\ /\ o\ /\ t)$, $token$::$\mathcal{A}$, &$error :: \texttt{Bool}$.
output: $\mathcal{A}$.
switch $(\texttt{head}(token.\phi_{aux}))$ **do**
  `Case` : $(i_m\ /\ o_m/\ \hat{p_m})$
  if $(i = i_m)$ **then**
    if $[(token.wild \wedge token.t_a \in token.t_e) \vee \neg\, token.wild] \wedge o = o_m \wedge t \in \hat{p}_m$ **then**
      $token.t_t \leftarrow token.t_t + t$;
      $token.t_e \leftarrow [0, 0]$; $token.t_a \leftarrow 0$;
      $token.wild \leftarrow \texttt{false}$;
      $token.\phi_{aux} \leftarrow \texttt{tail}(token.\phi_{aux})$;
      $\texttt{return}(token)$;
    else
      $\texttt{return}(null)$;
  else
    $token.t_a \leftarrow token.t_a + t$;
    if $(token.wild \wedge (t_e \odot (token.t_a)))$ **then**
      $token.t_t \leftarrow token.t_t + t$;
      $token.\phi_{aux} \leftarrow \texttt{tail}(token.\phi_{aux})$;
      $\texttt{return}(token)$;
    else
      $\texttt{return}(null)$;
  `Case` : $(i_f \mapsto O/\hat{p_f} \rhd \hat{q}_f)$
  if $(i = i_f) \wedge ((\neg token.wild) \vee (token.wild \wedge (t_e \odot (token.t_a + t))))$ **then**
    $token.t_t \leftarrow token.t_t + t$;
    if $((o \in O) \wedge (t \in \hat{p_f}) \wedge (token.t_t \in \hat{q}_f))$ **then**
      $\texttt{return}(null)$;
    else
      $error \leftarrow \texttt{true}$; $\texttt{return}(null)$;
  else
    $\texttt{return}(null)$;
  `Case` : $(\star_m, \hat{p}_m)$
    $token.t_t \leftarrow token.t_t + t$; $token.t_e \leftarrow \hat{p}_m$;
    $token.t_a \leftarrow t$; $token.wild \leftarrow \texttt{true}$;
    $token.\phi_{aux} \leftarrow \texttt{tail}(token.\phi_{aux})$;
    $\texttt{return}(token)$;

Figure 3. `treated` **correct function.**

the invariant, then the algorithm checks if the associated time value falls in the interval marked in the invariant. If this holds, then the part of the invariant that has not been checked and the time registered in the current position of the sequence are stored in a stack. In this way, we will have a buffer with all the *pending* situations that must be checked when the algorithm reaches the next position of the log. Thus, for each step of the algorithm we will push a new element in the stack, if the new position reached in the sequence fulfills the requirements of the invariant. In addition, we will check all the pending situations in the stack against the new element of the log. If it does not hold, then the element is removed from the stack. On the contrary, if it holds, then the pending situation is updated with the remaining part of the invariant and the time of the element in the sequence. Let us remark that the fact that the algorithm finds no match of the recorded log with the invariant when we are checking the first n elements of the invariant does not indicate that the sequence does not fulfill the invariant. In that case, we have not found the preconditions established by it. It is only when we reach the last component of the invariant for each of the pending situations, when a verdict

can be emitted. If we find an error then the algorithm stops; otherwise, it continues reviewing the rest of the log and the elements remaining in the stack.

The function `treated` checks if an element of the sequence and a component of the invariant match. In this function, the treatment is different depending on the kind of component of the invariant being checked. The first one corresponds to elements of the form (input/output/time); the second one deals with the very last part of the invariant. Finally, the third one manages those elements that contain a $\star$ symbol. Let us remark that in the second case we are at the end of the invariant and we have to check all the restrictions imposed by it. It is the only place of the function where an error can be found.

Regarding the complexity of our pattern matching strategy, in the worst case we obtain $\mathcal{O}(r \cdot n + 1)$ ($n + 1$ is the length of the invariant and r is the length of the observed sequence). Let us remark that even though *good* algorithms for pattern matching on strings perform in $\mathcal{O}(r)$ (after the *pre-processing* phase) we cannot achieve this complexity because we must check all the occurrences of the pattern in the log. However, as we commented before, if we consider that the length of the invariant is *much smaller* than the length of the log, as it is usually the case, we have that this complexity is almost linear with respect to the length of the log.

5 PASTE

In this section we present our PASive TEsting tool, that we call PASTE. With this tool we show the feasibility of the formal framework presented in this paper. The kernel of the tool is described in Figure 4. The data is input in the tool by using a XML file. This module transforms formatted data files into the internal data format of the application. From the XML file, PASTE obtains the specification of the studied system, represented by a `TFSM` model, a set of invariants, and the logs to be checked.

The relationship `correct_spec?` between invariants and the specification is used to establish the correctness of invariants with respect to the specification. It is always the first relation that is performed in the tool. The reason is that tester can provide any set of invariants and if she wants to check a log with them, then these invariants have to be checked against the specification. In addition, if the specification is missing, then the tool considers that the provided invariants are correct and skips the correctness check of these invariants with respect to the specification. In any case, having a correct set of invariants, we can check the correctness of the logs with respect to them by calling the module `Correct_logs?` If an error is detected then PASTE notifies it.

In addition to the theoretical framework, PASTE implements a module with *mutation* techniques to provide a measure on how *good* a set of invariants is. Mutation is a technique for unit testing software (e.g. [21, 10, 24, 20, 13]) that, although powerful, is computationally expensive. The principal expense of mutation is that many variants of the specification, called *mutants*, must be repeatedly executed. In PASTE, the specification is *mutated* and for each mutant a log is recorded. Mutations are chosen in order to simulate *real faults*. The belief is that if we have an invariant that finds several errors in the logs recorded from mutants, then this invariant is more likely to find an error in a faulty `IUT`.

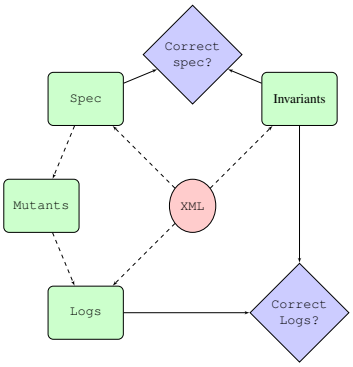


Figure 4. Adaptation of PASTE with mutants approach.

The mutation operators that we use in PASTE are changing the goal state of a transition, changing the output, and changing the time value associated with a transition. As usual, we only consider first order mutants but it is trivial to generate higher order mutants.

Let us note that we may generate a mutant that is functionally *equivalent* to the original specification. In order to establish the effectiveness of invariants, we will discard equivalent mutants. The notion of equivalence of a mutant with respect to the specification follows a criterium of *trace inclusion*. A mutant is *equivalent* to the original specification if for all possible sequences of inputs that can be performed by the mutant and the specification the outputs produced by the mutant are a subset of those considered by the specification. Intuitively, the mutant should not *invent* behaviors when provided with inputs specified in the specification. This pattern is borrowed from `ioco` [25].

Once the mutants are generated and checked that are not equivalent to the specification, PASTE applies the following loop to generate logs of a certain length from the mutant.

1. We use the variables $s_\delta \in \mathcal{S}$ and $i_\delta \in \mathcal{I}$. We start in the initial state, $s_\delta \leftarrow s_0$.
2. Then we calculate all possible inputs $I' \subseteq \mathcal{I}$ that can be applied in that state, that is, $I' = \{i | \exists s, s' \in \mathcal{S}, o \in$

$\mathcal{O}, t \in \mathbf{R}_+ : (s, i, o, t, s') \in \mathcal{T}\}$. In particular $I' = \mathcal{I}$ if we use input-enabled specification.

3. Randomly, we choose an input $i_\delta \in I'$.
4. Then, we calculate the set of transitions $T = \{(s_\delta, i_\delta, o, t, s') | \exists o \in \mathcal{O}, s' \in \mathcal{S}, t \in \mathbf{R}_+ (s_\delta, i_\delta, o, t, s') \in \mathcal{T}\}$.
5. Randomly, we choose a transition $s_\delta \xrightarrow{i_\delta, o}_t s'$ and PASTE simulates its performance, that is, it is included in the log.
6. We change the current state $s_\delta \leftarrow s'$ and we jump to step 2 as long as we want to increase the length of the sequence.

Having a set of traces from the mutants and a set of invariants, we say that the invariant ϕ kills the mutant M if ϕ finds an error in a trace generated by M. The *goodness* measure of an invariant ϕ with respect to a set of mutants $\mathcal{M}$ is defined as

$$E(\phi, \mathcal{M}) = \frac{|\mathcal{M}_k|}{|\mathcal{M}|}$$

being $\mathcal{M}_k$ the set of mutants killed by ϕ.

Example 2 In Figure 5 we present the results of applying our mutation technique to the `SSadmin` system. The invariants used in this experiment, ϕ_1, ϕ_2, ϕ_3, and ϕ_4, were presented in Example 1. Intuitively, the idea is to determine which invariant kills more mutants. The number of generated mutants is equal to $2^{|\mathcal{S}|}$, being $\mathcal{S}$ the set of states of the specification. In this sample, the percentage of mutants generated applying the goal state mutant operator is 30%, the percentage by applying the output state operator is 30%, and 40% goes to mutants generated after applying the time mutant operator. From each mutant, we extract five traces of length $n \cdot |\mathcal{S}|$, with $n \in \{1, 5, 10, 15, 20\}$. As expected, the number of *killed* mutants increases with the length of the logs. The most effective invariant is ϕ_4, killing around 23% of the mutants when considering the longest logs. □

6 Conclusions and future work

In this paper we have presented a framework to perform passive testing in systems that contain temporal information. The underlying model to represent systems is a timed extension of the finite state machines model. We use time invariants, a timed extension of the notion introduced in [3, 5], to find error on logs extracted from the `IUT`. We provide two algorithms (one theoretical and another one more optimized) to decide whether a log matches an invariant; if a mismatch is found then we can conclude that the

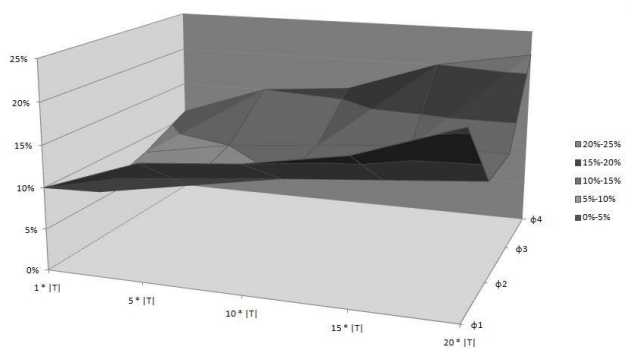


Figure 5. Invariants ϕ_1, ϕ_2, ϕ_3, ϕ_4 comparative by applying mutation techniques in `SSadmin` specification.

`IUT` is faulty. Finally, we have performed a small experiment on our running specification, the `SSadmin` system, to compare four invariants with respect to their power to find errors. As future work we plan to improve the capabilities of our framework by adding new classes of invariants.

References

[1] 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. An extended version is available at http://kimba.mat.ucm.es/manolo/papers/atva08-passive-extended.pdf.

[2] C. Andrés, M.G. Merayo, and M. Núñez. Passive testing of stochastic timed systems. In *2nd Int. Conf. on Software Testing, Verification, and Validation, ICST'09 (in press)*. IEEE Computer Society Press, 2009.

[3] J.A. Arnedo, A. Cavalli, and M. Núñez. Fast testing of critical properties through passive testing. In *15th Int. Conf. on Testing Communicating Systems, TestCom'03, LNCS 2644*, pages 295–310. Springer, 2003.

[4] J.M. Ayache, P. Azema, and M. Diaz. Observer: A concept for on-line detection of control errors in concurrent systems. In *9th Symposium on Fault-Tolerant Computing*, 1979.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] S.C.P.F. Fabbri, J.C. Maldonado, T. Sugeta, and P.C. Masiero. Mutation testing applied to validate specifications based on statecharts. In *10th IEEE Int. Symposium on Software Reliability Engineering, ISSRE'99*, pages 210–219. IEEE Computer Society Press, 1999.

[11] 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.

[12] R.M. Hierons, K. Bogdanov, J.P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A.J.H Simons, S. Vilkomir, M.R. Woodward, and H. Zedan. Using formal methods to support testing. *ACM Computing Surveys*, 41(2), 2009.

[13] R.M. Hierons and M.G. Merayo. Mutation testing from probabilistic finite state machines. In *3rd Workshop on Mutation Analysis, Mutation'07*, pages 141–150. IEEE Computer Society Press, 2007.

[14] 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.

[15] J. Huo and A. Petrenko. On testing partially specified IOTS through lossless queues. In *16th Int. Conf. on Testing Communicating Systems, TestCom'04, LNCS 2978*, pages 76–94. Springer, 2004.

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

[17] 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.

[18] 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.

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

[20] R. Nilsson, J. Offutt, and J. Mellin. Test case generation for mutation-based testing of timeliness. In *2nd Workshop on Model Based Testing, MBT'06*, pages 97–114. Electronic Notes in Theoretical Computer Science 164(4), 2006.

[21] J. Offutt. A practical system for mutation testing: Help for the common programmer. In *7th International Test Conference, ITC'94*, pages 824–830. IEEE Computer Society Press, 1994.

[22] 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.

[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] T. Sugeta, J.C. Maldonado, and W.E. Wong. Mutation testing applied to validate SDL specifications. In *16th IFIP Int. Conf. on Testing of Communicating Systems,TestCom'04, LNCS 2978*, pages 193–208. Springer, 2004.

[25] J. Tretmans. Testing concurrent systems: A formal approach. In *10th Int. Conf. on Concurrency Theory, CONCUR'99, LNCS 1664*, pages 46–65. Springer, 1999.

[26] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.