SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2011; **00**:1–43
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Formal Passive Testing of Timed Systems: Theory and Tools[†]

César Andrés, Mercedes G. Merayo, and Manuel Núñez[*]

Departamento Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Spain

SUMMARY

This paper presents a methodology to perform passive testing of timed systems. In passive testing the tester does not interact with the implementation under test. On the contrary, execution traces are observed without interfering with the behavior of the system. *Invariants* are used to 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, it must exhibit a behavior in a lapse of time reflected in the invariant. There are two types of invariants: *consequent* and *observational*. The paper gives two algorithms to decide the correctness of proposed invariants with respect to a given specification and algorithms to check the correctness of a log, recorded from the implementation under test, with respect to an invariant. The soundness of this methodology is shown by relating it to an implementation relation.
In addition to the theoretical framework, a tool called PASTE has been developed. This tool helps in the automation of the passive testing approach since it implements all the algorithms presented in this paper. PASTE takes advantage of mutation testing techniques in order to evaluate the goodness of an invariant according to its capability to detect errors in logs generated from mutants. An empirical study where PASTE was used to analyze a non-trivial system is also reported. Copyright © 2011 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Formal testing; Passive testing; Timed systems; Conformance testing; Tools for testing

1. INTRODUCTION

The complexity of current systems, the large number of people working on them, and the number of different modules that interact with each other, make it difficult to evaluate their correctness. *Testing techniques* [51, 2] allow their users to provide a degree of confidence on the correctness of the systems. These techniques can be combined with the use of formal methods [40, 26, 29, 60, 25, 59] in order to semi-automatically perform some tasks involved in testing [70].

The application of formal testing techniques to check the correctness of a system requires to identify its *critical* aspects, that is, those characteristics 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, formal testing techniques started to study other issues such as time and probabilistic information. This paper considers systems where time plays a fundamental role. It is worth mentioning that there are several proposals for formal testing of timed systems [44, 17, 28, 65, 21, 47, 46, 24, 71, 73, 27].

In testing, there is usually a distinction between two approaches: *passive* and *active*. The main difference between them is that in active testing a tester can interact with the implementation under

[*]e-mail: c.andres@fdi.ucm.es, mgmerayo@fdi.ucm.es, mn@sip.ucm.es
[†]Research partially supported by the Spanish MEC project TESIS TIN2009-14312-C02-01 and the Santander-UCM Programme to fund research groups GR35/10-A-910606.
[*]Correspondence to: César Andrés, Departamento Sistemas Informáticos y Computación, Facultad de Informática, Universidad Complutense de Madrid, 28040 Madrid, Spain. c.andres@fdi.ucm.es

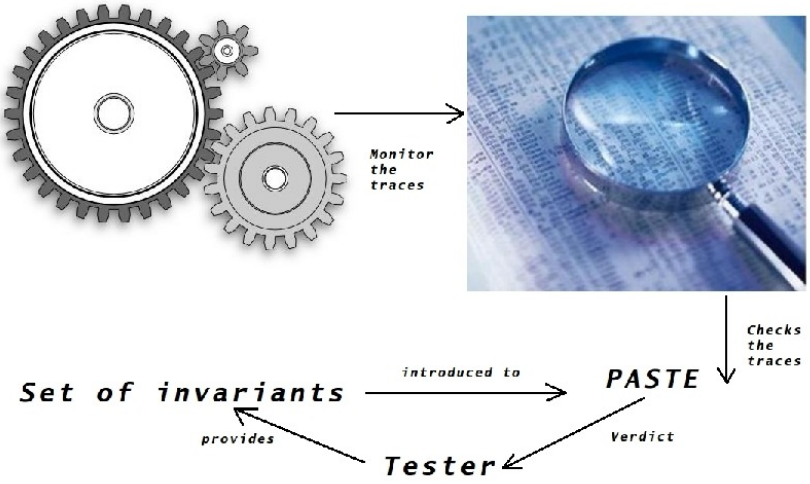


Figure 1. Graphical presentation of the framework.

test (in short, IUT), while in passive testing the tester simply monitors the behavior of the IUT. Even though most work on formal testing considers the active approach, it is very frequent that the tester is unable to interact with the IUT. 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.

Even though work on passive testing has been carried on for several years, it can be dated back at least to the 1970s [10], formal passive testing of timed systems did not receive enough attention until very recently. An initial work [4] introduced the syntax for so-called *consequent invariants* and an algorithm to check the correctness of these invariants with respect to a specification. Afterwards, algorithms to check the correctness of logs recorded from the IUT were presented [5]. This work also proved that the process is *sound* in the sense that, given a specification, if a log extracted from an IUT does not match a correct invariant, then the IUT does not conform to the specification. In addition, a tool called PASTE was developed. The main goal of this tool is to support the theoretical framework. In particular, this tool implements all the algorithms presented in these papers. The framework was extended with a novel approach that makes use of mutation testing techniques as a way to provide a classification of invariants according to their power to find errors [3]. The process works as follows. First, mutants are generated from a specification by applying different mutation operators. Then, these mutants generate logs. If the user has to decide between two correct invariants, then the one that finds more errors in the logs produced by mutants will be chosen. Since the number of correct invariants for a specification is potentially infinite, it is very important to have a method to select among a big number of invariants those which theoretically are more capable to detect errors in faulty IUTs. The approach is graphically depicted in Figure 1.

It is interesting pointing out some differences and similarities of this passive testing approach and runtime verification [42], the discipline of computer science that deals with the study, development, and application of verification techniques to check whether a run of a system under scrutiny satisfies or violates a given correctness property. Both approaches have the same goal but they work with different techniques and formalisms. In this paper, a set of invariants formally expresses a set of properties that have to be checked. If an error is detected when a log extracted from the IUT is checked against an invariant, then it can be claimed that the IUT does not conform to the specification. Therefore, before checking the correctness of the collected traces with respect to the specification, it is necessary to ensure that invariants are correct with respect to the specification. In contrast, a complete specification is rarely available in runtime verification techniques. On a different line, if the tester is using an invariant expressing an interesting property of the system and suddenly access to the system is granted, so that he can become an *active tester*, it is straightforward to transform the invariant into test cases. In order to overcome this difficulty, there are approaches to combine runtime analysis with test case generation [8]. The idea is that for each considered input sequence, a *property generator* constructs a set of properties that must hold when the

DOI: 10.1002/stvr

implementation under test is executed with these inputs. Afterwards, the tester must check that all these properties are satisfied.

This paper represents a revised, enhanced, and extended version of previous work on passive testing of timed systems [4, 5] and of the mutation testing techniques introduced in PASTE [3]. In addition to put under a common notation the previous work, this paper introduces a novel type of invariants: *observational* invariants. These invariants were motivated by the experience while working with complex case studies where the original notion, that is, consequent invariant, was inadequate to appropriately assess the correctness of some features of the studied systems. Consequent invariants can be used to express properties such as

> *each time that a user asks for login and access is granted in less than ten time units, if after performing some operations the user asks for disconnection, then he is disconnected, and this operation is performed in less than twenty time units.*

Observational invariants allow users to express properties about actions that were performed between two differentiated events. For example, it may be necessary to check that the logs extracted from the `IUT` fulfill a property such as

> *a logged user that has been connected to the system at most twenty time units can only check his profile but cannot change it.*

The difference between these two types of properties is that the pattern to define consequent invariants is "if something happens then something must happen" while observational invariants express properties such as "if something happens and after a while something else happens, then all the actions in between must fulfill a certain property."

In addition to present the syntax and examples of the new type of invariant, the paper also provides algorithms to check the correctness of observational invariants with respect to a specification, as well as algorithms to check the correctness of the logs recorded from an `IUT` with respect to observational invariants. The soundness of the methods, for both types of invariants, is shown by relating them to an implementation relation. Finally, PASTE has been extended with the new type of invariants and their associated algorithms.

The second main contribution of this paper is a complete case study where a non-trivial system, called `SSadmin`, is studied. This system allows students to check their marks, to modify their personal profile and, at the beginning of the academic year, to register 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, passive testing techniques must be used to study the *logs* recorded from `SSadmin`.

An additional contribution of this paper is to provide a related work section that reviews most of the work on formal passive testing with a special emphasis on proposals based on invariants. This section concludes by pointing out the relation between the passive testing approach presented in this paper and *runtime verification*. Finally, a new formal translation procedure from timed invariants into Extended Finite State Machines is also provided. This translation allows users of the methodology to rely on a formal semantics for invariants based on a well established formalism.

The rest of the paper is structured as follows. Section 2 reviews the main proposals to perform formal passive testing and some work on runtime verification. Section 3 introduces the formal framework to specify timed systems. Section 4 describes consequent and observational invariants. Section 5 contains the material related to the correctness of the approach: a mismatch of a log with a correct invariant implies a faulty `IUT`. This section also gives algorithms to check the correctness of logs with respect to invariants. Section 6 presents some features of PASTE and an empirical study of the `SSadmin` system. Section 7 presents the conclusions. Finally, the appendix of the paper provides a translation from timed invariants into Extended Finite State Machines.

DOI: 10.1002/stvr

2. RELATED WORK

This section briefly reviews previous work on passive testing. Therefore, this section can be considered as a small survey on the field. First, general monitoring and passive testing techniques, with a focus on formal approaches, are enumerated and briefly described. Afterwards, the focus goes to proposals to perform passive testing based on invariants. Finally, some relevant work in the area of runtime verification is reviewed.

2.1. Monitoring techniques

An initial focus was on developing expert systems capable of diagnosing faults and taking corrective actions on likely faulty scenarios [76]. The major difficulty here is that the experience of human experts is generally required to develop these expert systems. Each system or subsystem must be handled separately, in an ad-hoc fashion, and in the case of new developed systems this method may pose problems.

A later goal was to look for unifying principles in fault detection and identification [72]. Actually, many network problems that occur due to intrusion and security violations can be addressed by using passive testing. This is clear from the observation that unwanted intrusions matter only if they are successful in changing the input/output behavior of the system under attack. The authors develop classes of fault detection mechanisms that broadly apply across a variety of communication systems. This work focuses on a group of very simple *observers*, capable of detecting almost all possible faults in the system under observation, excluding deadlock and livelock situations. An algorithm for constructing these observers and a fast real-time fault detection mechanism used by each observer was given. Since observers run in parallel and independently, one immediate benefit of this approach is *graceful degradation*: one failed observer will not cause the collapse of the fault management system.

Other work concentrated on providing an algorithm to trace the values of variables and determine the current state of the system [41]. The authors presented two efficient implementations of their approach. The first implementation narrowed down the range of each variable as much as possible whenever additional information could be derived from a transition. A set of range operations was introduced and several examples were given to illustrate that usage. In the second implementation, the constraints derived from a transition path are recorded and the executability of the path is verified by solving these constraints as a system of linear equations/inequalities. These algorithms can deal with commonly encountered operations on variable values associated with state transitions and also provide efficient variable value determination for the protocol data portion fault detection.

Passive testing has been used for network fault management [37]. In this line, faults are detected in a network protocol system by passively observing its input/output behaviors without interrupting the normal network operations. The authors introduce methods for passive fault detection in deterministic and nondeterministic Finite State Machines (in short, `FSMs`). This work takes into account that it is important for communication networks to detect faults "in-process", that is, while the network is in its normal operation. The authors apply their techniques to the management of a signaling network operating under Signaling System 7 and report on experimental results, which show the feasibility of applying passive testing to practical systems. This work has been very influential and its underlying ideas have been applied to other `FSM`-based systems [75, 77, 69] and were extended to systems specified as Extended Finite State Machines (in short, `EFSMs`) [67, 35, 1, 36, 68, 13] and to systems specified as Communicating Finite State Machines (in short, `CFSM`) [48, 49, 50].

Another line of work is to define a general formal model for passive conformance testing, where `FSMs` are used to model the *protocol control portion*, and design and implement fault detection algorithms for both deterministic and nondeterministic systems [54]. The framework was applied to detect faults at run-time for the Signaling System 7 protocol.

A systematic study of passive testing of the data portion of protocols was also carried out [35]. Variables contain important information concerning the behavior of protocol systems, in particular, they determine the system states and their external behaviors. The authors presented two algorithms

DOI: 10.1002/stvr

using an Event-driven EFSM. First, an effective passive testing algorithm for EFSMs was proposed. Second, an algorithm based on variable determination with the constraints on variables was presented. This algorithm allows users to trace the values of variables as well as the system state. However, not all transfer errors can be detected. To overcome this limitation, a new approach based on backward tracing was proposed [1]. This algorithm is strongly inspired by previous work [35], but the trace is processed backwards in order to further narrow down the possible configurations for the beginning of the trace and to continue the exploration in the past of the trace with the help of the specification. This new algorithm was applied to the Simple Connection Protocol (in short, SCP) that allows to connect two entities after a negotiation of the quality of service required for the connection.

An algorithm, inspired on previous work in *active* testing [39, 38], where heuristics are used to achieve high coverage of transitions in CFSMs was also developed [64]. Mutation testing techniques were considered since they have a good performance for a range of particular types of errors. This approach defines mutation functions with special properties such that only mutants with single faults need to be considered for test generation. As a case study, the authors modeled the predicate *absence fault type* and presented and analyzed the test generation algorithm. The well-known Needham-Schroeder on mutual authentication protocol [52, 43] was used to illustrate their formal model and testing algorithms.

A passive testing algorithm has been used to analyze the TCP protocol [74]. Experimental results show that the protocol had a high transition coverage compared to other testing experiments. Detailed analysis of the experiments is presented and shows a possible way of combining passive testing and active testing.

2.2. *Passive Testing based on Invariants*

This section reviews previous approaches to perform passive testing based on the concept of *invariant*, that is, properties that must be fulfilled by any log observed in the IUT. The approach for passive testing of timed systems presented in this paper builds on top of the approaches presented in this section, more specifically, on previous work also considering invariants [7, 12].

The classical approach to perform passive testing consists in recording the trace produced by the implementation under test and trying to find a fault by comparing this trace with the specification. A novel approach [16] was supported by the following idea: a set of *invariants* represents 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, it must exhibit a behavior reflected in the invariant. The authors use the SCP to assess their theoretical framework. This first approach was able to partially evaluate the data flow, but not in a very satisfactory way. At least two drawbacks can be identified. First, invariants were automatically extracted from the specification. Even though this fact allows users of the methodology to partially automatize the testing process, the number of derived invariants is so big that in order to put the approach into practice a manual processing to select *relevant* invariants is needed. Second, the grammar used to express invariants was very limited, so that important properties could not be specified as invariants.

Later work presented a step forward in the use of invariants for passive testing [7]. The authors proposed that invariants can be, initially, supplied by the expert/tester. Therefore, the first step consists in checking that invariants are in fact correct with respect to the specification. An algorithm to check this correctness was provided. The complexity, in the worst case, of the algorithm was linear with respect to the number of transitions of the specification. Once a set of (correct) invariants was available, the second step consisted in checking whether the trace produced by the IUT matched the invariants. In order to do so, a simple adaptation of the classical algorithms for pattern matching on strings [14, 32] was implemented. This work was extended [12] to study a new type of invariants (obligation), to present a tool that implements the approach, and to give a complete case study on the Wireless Application Protocol. It is worth pointing out that this protocol represents a typical example where active testing cannot be applied since, in general, there is no direct access to the

Prepared using stvrauth.cls
DOI: 10.1002/stvr

interfaces between the different layers. Thus, testers cannot control how internal communications were established.

The work reported on this paper builds on top of previous work on passive testing of timed systems. First, the notion of consequent timed invariant and the algorithm for checking the correctness of this kind of invariants with respect to the specification was introduced [4]. The correctness of this methodology was formally proved [5] by showing that if an invariant detects a fault in a log, then the IUT that has produced this log does not conform to the underlying specification. A novel methodology to classify invariants by using mutation techniques with respect to the number of detected faults was later defined [3]. This framework was extended to consider systems where time information is given in terms of probability distribution functions [6]. The PASTE tool implements all these algorithms and methodologies. This tool can be downloaded from https://simba.fdi.ucm.es/paste. Formerly, PASTE was developed in Java but the code has been recently *translated* into a C++ library in order to improve its performance. This tool has been used in different scenarios, not only in academic studies. In particular, it has been included as a module to perform formal passive testing in Osmius, an Open Source monitoring tool (see http://www.osmius.com/en/product). More details about the PASTE tool are given in Section 6.

2.3. Runtime Verification

This section explains the relation between the passive testing approach presented in this paper and *runtime verification* [42]. The main difference between runtime verification and (formal) passive testing is given by the theoretical tools underlying the application of the techniques. Runtime verification techniques usually require an implementation under test, an observer of the program which collects the executions of the target system, and a set of requirements, that is, causal relations among actions and temporal constraints on their performance, which are often written in some *Linear Temporal Logic* (in short, LTL) [56]. Runtime verification techniques can be used for checking online and offline stored traces, or for solving some problems involved in concurrency, such us data races [62, 9] and deadlock detection [18]. In particular, the use of temporal logic for runtime verification has been investigated during the last years for reactive systems [58, 23, 31, 63, 8, 20].

Temporal logics have served as a model for tools such as MaC [30, 61, 31] and PathExplorer [23], where a three-valued logic is used, the commercial tool Temporal Rover [19, 15] that supports a fixed future and past line LTL, and EAGLE [11], which is based on recursive parameterized rule definitions over three primitive temporal operators. There are some approaches that do not use temporal logics to represent requirements. Some of them use mathematical predicates to specify properties [57], or implement algorithms addressing specific problems such as the Eraser tool [62] that dynamically detects data races.

Dealing with timed systems, some studies focus on introducing logics to represent time, such as the Metric Temporal Logic [33] and LTL extended with real time constructs embodied by a freeze quantifier together with atomic clock constraints LTL_t [34].

Note that all these languages are, in general, more expressive than the invariants considered in this paper. The main problem with such expressive languages is that it is far from easy for a (passive or active) tester used to define relations between applied inputs and observed outputs, to write properties to be checked against the IUT as a temporal logic formula. On the contrary, the syntax of invariants is very similar to the usual sequences of inputs and outputs used in model-based testing. In conclusion, runtime verification techniques can achieve what this passive testing approach based on invariants can achieve but the theoretical framework would be unnecessarily involved and it would be more difficult for *classical* active testers to become passive testers.

DOI: 10.1002/stvr

3. PRELIMINARIES

This section introduces the formalism to specify timed systems. First, the paper gives notation regarding the definition of time intervals: intervals are used to represent time information and therefore contain real values greater than or equal to zero.

Definition 1
Any value $t \in \mathbf{R}_+$ is a *constant time value*. For all $t \in \mathbf{R}_+$, both $t < \infty$, $t + \infty = \infty$, and $\infty - t = \infty$. The real interval $\hat{p} = [p_1, p_2]$ is a *time interval* if $p_1 \in \mathbf{R}_+$, $p_2 \in \mathbf{R}_+ \cup \{\infty\}$, and $p_1 < p_2$. The set of all time intervals is denoted by $\mathcal{I}_{\mathbb{R}_+}$.

Let $\hat{p} = [p_1, p_2]$ and $\hat{q} = [q_1, q_2]$ be two time intervals and t be a constant time value. The following functions can be defined: the addition of time intervals is $[p_1, p_2] + [q_1, q_2] = [p_1 + q_1, p_2 + q_2]$, the subset relationship of time intervals is $[p_1, p_2] \subseteq [q_1, q_2] = (q_1 \leq p_1 \wedge q_2 \geq p_2)$, and the `Update` function is:

$$\texttt{Update}([p_1, p_2], [q_1, q_2]) = \begin{cases} [p_1, p_2] & \text{if } [q_1, q_2] = [0, 0] \\ [\min(p_1, q_1), \max(p_2, q_2)] & \text{if } [q_1, q_2] \neq [0, 0] \end{cases}$$

□

Time intervals will be used to express time constraints, associated with the performance of actions, in the definition of invariants. The idea is that if a time interval $[p_1, p_2] \in \mathcal{I}_{\mathbb{R}_+}$ is associated with a task, then it is expected that this task takes at least p_1 time units and at most p_2 time units to be performed. Intervals like $[0, p_2]$, $[p_1, \infty]$, or $[0, \infty]$ denote the absence of a temporal lower/upper bound and the absence of any bound, respectively. Note that there is an abuse of notation in $[p_1, \infty]$ and $[0, \infty]$ since these intervals represent, in fact, the intervals $[p_1, \infty)$ and $[0, \infty)$.

The formalism used to represent specifications of systems is given in the next definition. The framework is based on an adaptation of the well-known *finite state machine* formalism where constant time is added 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. The set of all `TFSM`s will be denoted by `SETTFSM`.

A machine M is *observable* if there do not exist two different transitions (s, i, o, t_1, s_1) and (s, i, o, t_2, s_2) belonging to $\mathcal{T}$. □

Given a transition (s, i, o, t, s') belonging to $\mathcal{T}$, s and s' are the initial and final states of the transition, i and o are the input and output actions, and t is the time that the transition needs to be completed. Along this paper, $s \xrightarrow{i/o}_t s'$ will be a shorthand for $(s, i, o, t, s') \in \mathcal{T}$.

All the machines considered in this paper are observable. Note that the notion of observability makes possible to have some degree of nondeterminism. For example, a machine can have two transitions $s \xrightarrow{i/o_1}_{t_1} s_1$ and $s \xrightarrow{i/o_2}_{t_2} s_2$, as far as $o_1 \neq o_2$.

Example 1
Figure 2 presents a running example of `TFSM`. Its initial state is s_1. Each transition is labeled with the input that the machine receives, the output that it produces, and the amount of time that the system needs to produce the output since the reception of the input.

For example, the transition $s_1 \xrightarrow{i_2/o_1}_3 s_2$ means that if the machine is at state s_1 and it receives the input i_2, then in 3 time units it will produce the output o_1 and will move to state s_2. □

The next definition introduces the concepts of *trace* and *log*. A trace represents a finite sequence of actions that the system may perform from any of its states. This notion differs from the usual one where the sequence is always performed from the initial state. A log represents the historical evolution of a system.

DOI: 10.1002/stvr

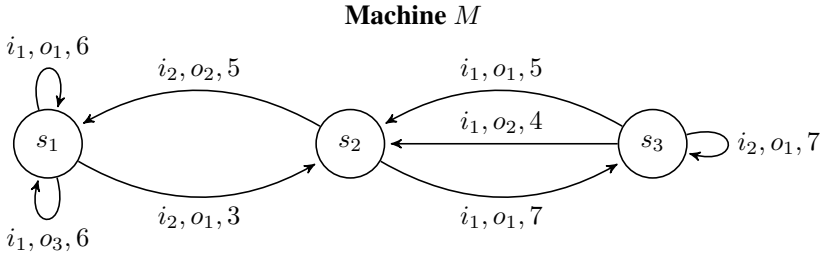


Figure 2. Example of a TFSM.

Definition 3
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM, $i_1, \ldots, i_r \in \mathcal{I}$, $o_1, \ldots, o_r \in \mathcal{O}$, and $t_1, \ldots, t_r \in \mathbf{R}_+$. A sequence $e = \langle i_1/o_1/t_1, \ldots, i_r, /o_r/t_r \rangle$ is a *trace* of M if there exist a sequence of transitions $s_1 \xrightarrow{i_1/o_1}_{t_1} s_2, s_2 \xrightarrow{i_2/o_2}_{t_2} s_3, \ldots, s_r \xrightarrow{i_r/o_r}_{t_r} s_{r+1}$ in $\mathcal{T}$. $\mathtt{Traces}(M)$ denotes the set of all traces of M. A *log* from M is a sequence belonging to $\mathtt{Traces}(M)$.

The function $\mathtt{Len}_{\mathcal{I}/\mathcal{O}} : (\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star \longrightarrow \mathbf{N}$ is such that for all $e = \langle i_1/o_1/t_1, \ldots, i_r, /o_r/t_r \rangle$, $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e) = r$. Note that $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\langle\rangle) = 0$.

The function $\mathtt{TT}_M : \mathtt{Traces}(M) \longrightarrow \mathbf{R}_+$ is such that for all $e = \langle i_1/o_1/t_1, \ldots, i_r, /o_r/t_r \rangle$, $\mathtt{TT}_M(e) = \sum_{j=1}^{r} t_j$. □

Note that traces and logs are indeed simply *traces*. Nevertheless, two different names are used to denote the same concept in order to distinguish between objects that, even though they look similar, have different nature. In this paper, the name *trace* refers to the finite sequences that the specification of a system can perform while the term *log* refers to finite sequences observed from the IUT. Usually, $e_1, e_2, \ldots$ will denote traces while $l_1, l_2, \ldots$ will denote logs.

Example 2
Consider the TFSM M presented in Figure 2. For instance, both $e_1 = \langle i_1/o_1/6, i_2/o_1/3 \rangle$, starting at state s_1, and $e_2 = \langle i_1/o_1/5, i_2/o_2/5 \rangle$, starting at state s_3, are traces of M. In addition, $\mathtt{TT}_M(e_1) = 9$, $\mathtt{TT}_M(e_2) = 10$, $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e_1) = 2$, and $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e_2) = 2$. □

4. TIMED INVARIANTS

This section introduces the notion of *timed invariant*. Timed invariants are used to represent the properties that must be checked against the logs extracted from the IUT. First, after receiving a set of timed invariants and before checking them against the logs, they must be checked against the specification; otherwise, the tester might be using an invariant that violates the requirements expressed by the specification. Another possibility would be to consider that invariants are correct *by definition*. In this case the specification could be completely ignored.

This paper uses two different types of timed invariants: Timed *consequent invariants* and timed *observational invariants*. The first type is used to check that an event is performed within certain time-bounds after a given trace of events has been observed. The second type is used to check that a given sequence of events is always performed between two given events within certain time-bounds.

4.1. Timed consequent invariant

Timed consequent invariants [4] are a natural extension with time of a previous notion [7, 12].

DOI: 10.1002/stvr

Definition 4
Let $\mathcal{I}, \mathcal{O}$ be two sets of input and output actions, respectively. A sequence ϕ is a *consequent invariant* if 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 the previous expression, $\hat{p}, \hat{q} \in \mathcal{I}_{\mathbb{R}_+}$, $i \in \mathcal{I}$, $a \in \mathcal{I} \cup \{?\}$, $z \in \mathcal{O} \cup \{?\}$, and $O \subseteq \mathcal{O}$. The set of timed consequent invariants for the sets $\mathcal{I}$ and $\mathcal{O}$ is denoted by $\Phi_{\mathcal{I}/\mathcal{O}}$. During the rest of the paper, a generic timed consequent invariant will be represented by $\alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$, where $\alpha_1, \ldots, \alpha_n \in ((\mathcal{I} \cup \{?\} \times \mathcal{O} \cup \{?\}) \cup \{\star\}) \times \mathcal{I}_{\mathbb{R}_+}$, $i_f \in \mathcal{I}$, $O \subseteq \mathcal{O}$, and $\hat{p}_f$ and $\hat{q}_f \in \mathcal{I}_{\mathbb{R}_+}$. □

Intuitively, the previous EBNF expresses that a timed consequent invariant is a sequence of symbols. Each component, but the last one, is either an expression $a/z/\hat{p}$, with a being an input or the ? wildcard character, z being an output or the ? wildcard character, and $\hat{p}$ being a timed interval, or an expression $\star/\hat{p}$. The special symbol ? represents any input or output. Therefore, for all $a \in \mathcal{I} \cup \mathcal{O}$, $a =?$ holds. In addition, the special symbol $\star$, whose occurrences are always followed by an input i, represents any sequence in $(\mathcal{I} \setminus \{i\} \times \mathcal{O} \times \mathbf{R}_+)^\star$.

This EBNF imposes two restrictions. First, an invariant cannot contain two consecutive components $\star/\hat{p}_1$ and $\star/\hat{p}_2$. The second restriction is that an invariant cannot present a component of the form $\star/\hat{p}$ followed by a wildcard character ?, that is, the input of the next component must be an input action $i \in \mathcal{I}$. The last component of the invariant, corresponding to the expression $i \mapsto O/\hat{p} \rhd \hat{q}$, is an input action followed by a set of output actions and two time restrictions. The first time interval, that is, $\hat{p}$, is associated with the last input/output pair of the sequence. The second time interval, that is, $\hat{q}$, concerns the sum of the time values associated with all the input/output pairs appearing in the sequence.

Note that time conditions established in invariants are given by intervals. However, machines present time information expressed as constant amounts of time. The use of intervals allows testers to consider that different executions of the same task can take different amounts of time to be completed. Another reason for the tester to use intervals, even if the system always takes the same time to perform a certain task, is to consider that the artifacts measuring time are not as precise as desirable. In this case, an apparently wrong behavior due to bad timing can be in fact correct since it may happen that the *clocks* are not working properly. A longer explanation on the use of time intervals to deal with imprecisions can be found in [45].

The next examples show the intended meaning of consequent timed invariants. In the appendix of the paper a formal semantics of consequent invariants, by translating them into `EFSMs`, is given.

Example 3
The following invariant expresses that "after performing i_1, either o_1 or o_2 will be observed within a time belonging to $[2, 8]$":

$$\phi_1 = i_1 \mapsto \{o_1, o_2\}[2, 8] \rhd [2, 8]$$

More complex properties can be specified. The following invariant means that "each time that the input i_1 is followed by the output o_1 in a time belonging to $[6, 7]$ and a sequence of inputs /outputs that does not contain the input i_2[†] is performed in a time belonging to $[6, 12]$, then when the input i_2 is observed, it must be followed by the output o_1, in a time belonging to $[1, 7]$. Additionally, the sum of all time values, from i_1 to o_1, must belong to $[5, 30]$":

$$\phi_2 = i_1/o_1/[6, 7], \star/[6, 12], i_2 \mapsto \{o_1\}[1, 7] \rhd [5, 30]$$

□

Since invariants can be defined by a tester, it must be ensured that they are correct with respect to the specification. Next, the most relevant aspects of the algorithm (shown in Figure 3) to decide

[†]Note that $\star$ matches any sequence of actions not containing the next input symbol appearing in the invariant, that is, i_2 in this case.

DOI: 10.1002/stvr

whether an invariant is correct with respect to a specification are presented. The algorithm has two parts. The first one, corresponding to the first loop of the algorithm, determines the set of states that can be reached in the specification after *matching* the first n elements, that is $\alpha_1, \ldots, \alpha_n$. The second part, after the first loop, is used to check the restrictions included in the invariant. The algorithm verifies that for all the states computed in the previous step, if the last input of the invariant can be performed, then the possible outputs belong to the set of outputs appearing in the set of outputs O. In addition, the algorithm also checks if the transitions traversed in the specification have associated a time value that belongs to the corresponding time interval of the invariant and that the sum of all the time values verifies the last time constraint.

Before starting with the explanation of the algorithm, the following definition introduces some auxiliary functions to deal with timed consequent invariants. The $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$ function computes the length of a timed consequent invariant. The $\mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)$ function returns a timed consequent invariant removing the first element of ϕ. The $\mathtt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi)$ function returns the number of occurrences of the *wildcard* $\star$ in ϕ. Finally, the $\mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\phi)$ function returns the next input of the timed consequent invariant ϕ.

Definition 5
Let $\mathcal{I}, \mathcal{O}$ be two sets of input and output actions, respectively. The function $\mathtt{Len}_{\mathcal{I}/\mathcal{O}} : \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ is such that for all $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$, $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = n$. Note that $\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f) = 0$.

The function $\mathtt{Tail}_{\mathcal{I}/\mathcal{O}} : \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \Phi_{\mathcal{I}/\mathcal{O}}$ is such that for all $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$:

$$\mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi) = \begin{cases} i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = 0 \\ \alpha_2, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \end{cases}$$

The function $\mathtt{Nstars}_{\mathcal{I}/\mathcal{O}} : \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ returns the number of occurrences of the *wildcard* $\star$ in a consequent timed invariant, that is, for all $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$,

$$\mathtt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi) = \begin{cases} 0 & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = 0 \\ 1 + \mathtt{Nstars}_{\mathcal{I}/\mathcal{O}}(\mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)) & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \wedge \alpha_1 = \star/\hat{p} \\ \mathtt{Nstars}_{\mathcal{I}/\mathcal{O}}(\mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)) & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \wedge \alpha_1 \neq \star/\hat{p} \end{cases}$$

The function $\mathtt{NInp}_{\mathcal{I}/\mathcal{O}} : \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathcal{I}$ is such that for all $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$:

$$\mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\phi) = \begin{cases} i_f & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = 0 \\ i_1 & \text{if } \alpha_1 = i_1/o_1/t_1 \wedge \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \\ \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)) & \text{if } \alpha_1 = \star/t_1 \wedge \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \end{cases}$$

□

The next definition introduces some auxiliary functions to deal with transitions and time intervals. The $\mathtt{AfterCond}_M(S, I, O, \hat{p})$ function computes the set of transitions outgoing from a state in S, labeled by an input in I and an output in O, and such that the associated time value belongs to the interval $\hat{p}$. The $\mathtt{SetStatesI}_M(i)$ function computes the set of goal states of all transitions of M that are labeled with the input i.

Definition 6
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. The function $\mathtt{AfterCond}_M : \wp(\mathcal{S}) \times \wp(\mathcal{I} \cup \{?\}) \times \wp(\mathcal{O} \cup \{?\}) \times \mathcal{I}_{\mathbb{R}_+} \longrightarrow \wp(\mathcal{T})$ is such that for all $S_{aux} \subseteq \mathcal{S}$, $I_{aux} \subseteq \mathcal{I} \cup \{?\}$, $O_{aux} \subseteq \mathcal{O} \cup \{?\}$, and $\hat{p} \in \mathcal{I}_{\mathbb{R}_+}$:

$$\mathtt{AfterCond}_M(S_{aux}, I_{aux}, O_{aux}, \hat{p}) \quad = \quad \left\{ tr \;\middle|\; \begin{array}{l} \exists s \in S_{aux}, s' \in \mathcal{S}, i \in \mathcal{I}, a \in I_{aux}, o \in \mathcal{O}, \\ z \in O_{aux}, t \in \mathbf{R}_+ : i = a \ \wedge \ o = z \ \wedge \ t \in \hat{p} \wedge \\ tr = (s, i, o, t, s') \in \mathcal{T} \end{array} \right\}$$

DOI: 10.1002/stvr

Algorithm `Correctness_Consequent_Specification`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) : \texttt{SETTFSM}, \phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f : \Phi_{\mathcal{I}/\mathcal{O}}$
Result: `Bool`

Initialization of variables

$\bar{x}$: array of $\mathcal{I}_{\mathbb{R}_+}$; $\bar{x} \leftarrow []$; $\mathcal{S}_c \leftarrow \mathcal{S}$;$j \leftarrow 1$;
($\bar{x}$ is an array of size $|\mathcal{S}|$ and $[0,0]$ is the initial value of all positions and $\mathcal{S}_c$ is the set of current states to be evaluated *)*

Main loop

while $j \leq \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) \wedge \mathcal{S}_c \neq \emptyset$; **do**
  $\mathcal{S}_{aux} \leftarrow \emptyset$;
  $\bar{y} \leftarrow []$;
  ($\mathcal{S}_{aux}$ computes the next set of states and $\bar{y}$ is an array with $|\mathcal{S}|$ components, used to update time information. $[0,0]$ is the initial value stored in all the positions of the array *)*
  if $\alpha_j = \star/\hat{p}_j$ **then**
    while $\mathcal{S}_c \neq \emptyset$ **do**
      `Choose` $s_a \in S_c$; $S_c \leftarrow \mathcal{S}_c \setminus \{s_a\}$;
      $(\bar{z}, S_{index}) \leftarrow \texttt{AfterInT}(M, s_a, \texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\alpha_j, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f), \hat{p}_j)$;
      while $S_{index} \neq \emptyset$ **do**
        `Choose` $s_b \in S_{index}$; $S_{index} \leftarrow \mathcal{S}_{index} \setminus \{s_b\}$;
        $\bar{y}[b] \leftarrow \texttt{Update}(\bar{z}[b], \bar{y}[b])$; $\mathcal{S}_{aux} \leftarrow \mathcal{S}_{aux} \cup \{s_b\}$;
      end
    end
  else
    ($\alpha_j = a_j/z_j/\hat{p}_j$ *)*
    $\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\mathcal{S}_c, \{a_j\}, \{z_j\}, \hat{p}_j)$;
    while $\mathcal{T}_{aux} \neq \emptyset$ **do**
      `Choose` $(s_a, i, o, t, s_b) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s_a, i, o, t, s_b)\}$;
      $\bar{y}[b] \leftarrow \texttt{Update}(\bar{x}[a] + [t,t], \bar{y}[b])$; $\mathcal{S}_{aux} \leftarrow \mathcal{S}_{aux} \cup \{s_b\}$;
    end
  end
  $\bar{x} \leftarrow \bar{y}$; $\mathcal{S}_c \leftarrow \mathcal{S}_{aux}$; $j \leftarrow j + 1$;
end
$\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\mathcal{S}_c, \{i_f\}, \mathcal{O}, [0, \infty])$;
$error \leftarrow (\mathcal{T}_{aux} = \emptyset)$;
($\mathcal{T}_{aux}$ contains the set of transitions that can produce errors *)*
while $\mathcal{T}_{aux} \neq \emptyset \wedge \neg error$ **do**
  `Choose` $(s_a, i_f, o, t, s) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s_a, i_f, o, t, s)\}$;
  $error \leftarrow (o \notin O) \vee (t \notin \hat{p}_f) \vee (\bar{x}[a] + [t,t] \not\subseteq \hat{q}_f)$;
end
return$(\neg error)$;

Figure 3. Correctness of a timed consequent invariant with respect to a specification.

DOI: 10.1002/stvr

Algorithm `AfterInT`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) : \texttt{SETTFSM}, s : \mathcal{S}, i : \mathcal{I}, [p_1, p_2] : \mathcal{I}_{\mathbb{R}_+}$
Result: array of $\mathcal{I}_{\mathbb{R}_+} \times \wp(\mathcal{S})$

Initialization of variables

$\bar{x}$: array of $\mathcal{I}_{\mathbb{R}_+}$; $\bar{x} \leftarrow []$;
($\bar{x}$ is an array of size $|\mathcal{S}|$ and $[0,0]$ is the initial value of all positions*)*
$Nodes \leftarrow \{(s, \{s\}, [0,0])\}$;
$S_c \leftarrow \emptyset$;
$S_{terminal} \leftarrow \texttt{SetStatesI}_M(i)$;
($Nodes$ is a set of all nodes of the breadth first search algorithm, S_c is the set of reached states, and $S_{terminal}$ with $S_c \subseteq S_{terminal}$ is the set of possible reached states*)*

Main loop

while $Nodes \neq \emptyset$ **do**
  `Choose` $(s_a, S_{visited}, [q_1, q_2]) \in Nodes$; $Nodes \leftarrow Nodes \setminus \{(s_a, S_{visited}, [q_1, q_2])\}$;
  if $s_a \in S_{terminal} \wedge [q_1, q_2] \subseteq [p_1, p_2]$ **then**
    $S_c \leftarrow S_c \cup \{s_a\}$;
    $\bar{x}[a] \leftarrow \texttt{Update}([q_1, q_2], \bar{x}[a])$;
  end
  $\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\{s_a\}, \mathcal{I} \setminus \{i\}, \mathcal{O}, [0, \infty])$;
  while $\mathcal{T}_{aux} \neq \emptyset$ **do**
    `Choose` $(s_a, i', o, t, s_b) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s_a, i', o, t, s_b)\}$;
    if $p_2 = \infty$ **then**
      if $q_2 \neq \infty \wedge s_b \in S_{visited}$ **then**
        (A loop is detected *)*
        $Nodes \leftarrow Nodes \cup \{(s_b, S_{visited}, [q_1 + t, \infty])\}$;
      end
      if $s_b \notin S_{visited}$ **then**
        $Nodes \leftarrow Nodes \cup \{(s_b, S_{visited} \cup \{s_b\}, [q_1 + t, q_2 + t])\}$;
      end
      if $q_2 = \infty \wedge q_1 < p_1$ **then**
        (A loop ($q_2 = \infty$) is executed a finite number of times until the lower bound is exceeded *)*
        $Nodes \leftarrow Nodes \cup \{(s_b, S_{visited} \cup \{s_b\}, [q_1 + t, q_2])\}$;
      end
    else
      if $q_2 + t \leq p_2$ **then**
        $Nodes \leftarrow Nodes \cup \{(s_b, S_{visited} \cup \{s_b\}, [q_1 + t, q_2 + t])\}$;
      end
    end
  end
end
return$(\bar{x}, S_c)$;

Figure 4. Function to compute the set of reached states and the amount of time to reach each of these states.

DOI: 10.1002/stvr

The function $\texttt{SetStatesI}_M : \mathcal{I} \longrightarrow \wp(\mathcal{S})$ is such that for all $i \in \mathcal{I}$:

$$\texttt{SetStatesI}_M(i) = \left\{ s' \;\middle|\; \exists s, s' \in \mathcal{S}, o \in \mathcal{O}, t \in \mathbf{R}_+ : (s, i, o, t, s') \in \mathcal{T} \right\}$$

□

Initially, the algorithm obtains the set of states from which a transition labeled by the first input/output pair of the invariant can be performed and such that the associated time value belongs to the interval indicated in the invariant. Then, the algorithm computes the set of states that can be reached from this initial set of states after performing the transitions. The algorithm repeats this process until it traverses all the components α_j of the invariant ϕ, with $1 \leq j \leq \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$, taking into account the set of states reached in the previous step. Note that there is a distinction between input/output pairs, possibly including the ? wildcard character, and occurrences of the $\star$ wildcard character. In the latter case, the `AfterInT` auxiliary function (shown in Figure 4) computes the reached states and the amount of time needed to reach each of these states (represented by an interval). This function does not consider the input/output actions labeling the traversed transitions, as long as the corresponding input does not appear in the sequence.

The next step of the algorithm make use of the set of transitions that can be executed after matching the invariant. If this set is empty then the invariant is not correct. The idea is that a tester should not use an invariant if the sequence of input/output actions cannot be performed by the specification in the intervals appearing in the invariant.

If the set is not empty then the algorithm computes the set of transitions from these states labeled with the input i_f. If this set of transitions is not empty then it means that at least there exists a trace of the specification that is completed matched by the invariant. The final step of the algorithm checks that the outputs produced by the transitions outgoing from these states and labeled with the input i_f belong to the set of outputs O that appears in the invariant. In addition, the time value associated with all of these outputs must belong to the time interval $\hat{p}_f$. Finally, that the time associated with the performance of the whole trace is correct. In order to do it, all the time values associated with the transitions traversed in the specification are recorded during the previous phases of the algorithm. Each position of the array $\bar{x}$ contains an interval with bounds the minimal/maximal time values that are needed to reach the corresponding states after the whole invariant is traversed. For all the states having an interval recorded, the algorithm checks if this interval is contained in the interval appearing in the last position of the invariant, that is, in $\hat{q}_f$.

Let ϕ be a timed consequent invariant and $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a `TFSM`. In the worst case, the complexity of the algorithm to decide the correctness of ϕ with respect to M is $\mathcal{O}(|\mathcal{S}| \cdot |\mathcal{T}|^{|\mathcal{S}|} \cdot k + |\mathcal{T}| \cdot (s - k))$, where $k = \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi)$ and $s = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$. This worst case is computed by taking into account the following facts. In the first part of the algorithm there is a loop depending on $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$. If the token under evaluation contains the wildcard $\star$ then the `AfterInT` function is called $|\mathcal{S}|$ times. This function computes a breadth first search algorithm, being its complexity in $\mathcal{O}(|\mathcal{T}|^{|\mathcal{S}|})$. But, if the token under evaluation does not contain the wildcard $\star$, then it perfoms a loop depending on the set of transitions. Finally, the second part of the algorithm that can be translated into a for-loop being its complexity in $\mathcal{O}(\mathcal{T})$, and can be omitted.

Definition 7
Let ϕ be a timed consequent invariant and M be a `TFSM`. The invariant ϕ is *correct* with respect to M if the algorithm $\texttt{Correctness\_Consequent\_Specification}(M, \phi)$ returns `true`. □

Example 4
Consider the `TFSM` M presented in Figure 2 and the consequent invariants $\phi_1 = i_1 \mapsto \{o_1, o_2\}[2, 8] \rhd [2, 8]$ and $\phi_2 = i_1/o_1/[6, 7], \star/[6, 12], i_2 \mapsto \{o_1\}[1, 7] \rhd [5, 30]$ presented in Example 3.

The ϕ_1 invariant is not correct with respect to M because the algorithm $\texttt{Correctness\_Consequent\_Specification}(M, \phi_1)$ returns `false`. The first loop of the algorithm is not performed because $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi_1) = 0$. In the beginning of the second part of the algorithm, the set of states is $\mathcal{S}_c = \{s_1, s_2, s_3\}$. The idea is to select those transitions of M outgoing from states $s \in \mathcal{S}_c$ and such that their associated input is i_1. These transitions are: $s_1 \xrightarrow{i_1/o_1}_6 s_1$,

DOI: 10.1002/stvr

$\phi_2 =$	$\mathbf{i_1/o_1/[6,7]}$,	$\mathbf{\star/[6,12], i_2}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_1}_6 \mathbf{s_1}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_3}_6 \mathbf{s_1}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_1}_6 s_1, s_1 \xrightarrow{i_1/o_1}_6 \mathbf{s_1}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_3}_6 s_1, s_1 \xrightarrow{i_1/o_3}_6 \mathbf{s_1}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_1}_6 s_1, s_1 \xrightarrow{i_1/o_3}_6 \mathbf{s_1}$
	$s_1 \xrightarrow{i_1/o_1}_6 s_1$,	$s_1 \xrightarrow{i_1/o_3}_6 s_1, s_1 \xrightarrow{i_1/o_1}_6 \mathbf{s_1}$
	$s_2 \xrightarrow{i_1/o_1}_7 s_3$,	$s_3 \xrightarrow{i_1/o_1}_5 s_2, s_2 \xrightarrow{i_1/o_1}_7 \mathbf{s_3}$
	$s_2 \xrightarrow{i_1/o_1}_7 s_3$,	$s_3 \xrightarrow{i_1/o_2}_4 s_2, s_2 \xrightarrow{i_1/o_1}_7 \mathbf{s_3}$

Figure 5. Set of traces of M matched by ϕ_2.

$s_1 \xrightarrow{i_1/o_3}_6 s_1$, $s_3 \xrightarrow{i_1/o_1}_5 s_2$, $s_3 \xrightarrow{i_1/o_2}_4 s_2$, and $s_2 \xrightarrow{i_1/o_1}_7 s_3$. Due to the fact that there exists a transition $s \xrightarrow{i_1/o}_t s'$ in this set such that $s \in \mathcal{S}_c$ and $o \notin \{o_1, o_2\}$, that is, $s_1 \xrightarrow{i_1/o_3}_6 s_1$, the variable error changes its value to `true` and the algorithm returns `false`.

The ϕ_2 invariant is correct with respect to M because the algorithm `Correctness_Consequent_Specification`(M, ϕ_2) returns `true`. The first loop checks the initial part of the invariant: $i_1/o_1[6,7], \star/[6,12]$. Once the loop finishes, S_c contains the reached states while $\bar{x}$ contains time information regarding the amount of time that the system would spend if the initial sequence of the invariant would be performed without taking into account the initial state. Specifically, $S_c = \{s_1, s_3\}$ because the considered traces are the ones induced by the sequences of transitions presented in Figure 5. In this table, the first row (in boldface) separates the parts of the invariant that are used to match the traces, and the remaining rows show the traces that are matched.

In addition, $\bar{x}[1] = [12, 18]$ and $\bar{x}[3] = [18, 19]$ (the values 1 and 3 correspond to the states s_1 and s_3, respectively). The values of $\bar{x}$ are computed by adding the time values from the previous traces and considering the minimum and maximum values of them as the bounds of the interval.

The second phase of the algorithm checks the conditions presented in the invariant, that is $i_2 \mapsto \{o_1\}[1,7] \rhd [5,30]$. If there exists a transition from a state of $\mathcal{S}_c$ starting with i_2 and producing an error. The candidate transitions are $s_1 \xrightarrow{i_2/o_1}_3 s_2$ and $s_3 \xrightarrow{i_2/o_1}_7 s_3$. The functional restriction of ϕ_2 holds because $o_1 \in \{o_1\}$ in both transitions. In addition, the temporal restrictions hold on the one hand $3 \in [1,7]$ and $7 \in [1,7]$ and on the other hand, concerning the complete trace, $\bar{x}[1] + [3,3]$ and $\bar{x}[3] + [7,7]$ belong to $[5,30]$. □

The *match* predicate relates traces of a specification and timed consequent invariants. A trace *matches* a timed consequent invariant if it is correct with respect to both its *functional* and its *temporal* behaviour. In order to deal with traces the function $\mathtt{Tail}_M(e)$ is introduced. This function removes the first element of the trace e of M.

Definition 8
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a `TFSM`. The function $\mathtt{Tail}_M : \mathtt{Traces}(M) \longrightarrow \mathtt{Traces}(M)$ is such that for all $e = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle \in \mathtt{Traces}(M)$:

$$\mathtt{Tail}_M(e) = \begin{cases} \langle\rangle & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e) \leq 1 \\ \langle i_2/o_2/t_2, \ldots, i_r/o_r/t_r \rangle & \text{if } \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e) > 1 \end{cases}$$

□

The $\mathtt{MatchC}_M(e, \phi)$ function computes whether the trace e of M matches the timed consequent invariant ϕ. This definition uses the $\mathtt{MatchC}^{\star}_M(e, i, \hat{q}, t)$ auxiliary predicate to deal with occurrences of the $\star$ wildcard character in the invariant with an associated time $\hat{p}$ being the next input of the $wildcard$ the action i.

DOI: 10.1002/stvr

Definition 9
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. The function $\mathtt{MatchC}^{\star}_{M} : \mathtt{Traces}(M) \times \mathcal{I} \times \mathcal{I}_{\mathbb{R}_+} \times \mathbf{R}_+ \longrightarrow \mathtt{Traces}(M)$ is such that for all $e = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle \in \mathtt{Traces}(M)$, $i \in \mathcal{I}$, $\hat{q} = [q_1, q_2] \in \mathcal{I}_{\mathbb{R}_+}$, and $t \in \mathbf{R}_+$:

$$\mathtt{MatchC}^{\star}_{M}(e, i, \hat{q}, t) = \begin{cases} \langle\rangle & \text{if } (t > q_2) \vee \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e) = 0 \vee (i_1 = i \wedge t \notin \hat{q}) \\ e & \text{if } i_1 = i \wedge t \in \hat{q} \\ \mathtt{MatchC}^{\star}_{M}(\mathtt{Tail}_{M}(e), i, \hat{q}, t + t_1) & \text{if } t \leq q_2 \wedge i_1 \neq i \end{cases}$$

The function $\mathtt{MatchC}_M : \mathtt{Traces}(M) \times \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathtt{Bool}$ is such that for all $e_1 = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle \in \mathtt{Traces}(M)$ and $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$, $\mathtt{MatchC}_M(e_1, \phi)$ is equal to:

$$\begin{cases} \mathtt{false} & \text{if } (u - 1 < s - k) \vee ((u > 1) \wedge (s = 0)) \\ i_1 = i_f & \text{if } u = 1 \wedge s = 0 \\ \mathtt{MatchC}_M(e_2, \mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)) & \text{if } r > 1 \wedge s > 0 \wedge \alpha_1 = \star/\hat{q} \\ i_1 = a \wedge o_1 = z \wedge t_1 \in \hat{q} \wedge & \\ \mathtt{MatchC}_M(\mathtt{Tail}_M(e_1), \mathtt{Tail}_{\mathcal{I}/\mathcal{O}}(\phi)) & \text{if } r > 1 \wedge s > 0 \wedge \alpha_1 = a/z/\hat{q} \end{cases}$$

where $e_2 = \mathtt{MatchC}^{\star}_{M}(e_1, \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\phi), \hat{q}, 0)$, $k = \mathtt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi)$, $s = \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$, and $u = \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e_1)$.

Let M be a TFSM, ϕ be a timed consequent invariant, and e be a trace of M. The sequence e *matches* ϕ if $\mathtt{MatchC}_M(e, \phi)$ returns true. □

The concept of matching can be used to give a characterization of the notion of correctness introduced by the algorithm given in Figure 3.

Lemma 1
Let M be a TFSM and $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$ be a timed consequent invariant. The invariant ϕ is correct with respect to M if there exits $e_1 \in \mathtt{Traces}(M)$ such that e_1 matches ϕ and for all traces $e_2 \in \mathtt{Traces}(M)$ such that e_2 matches ϕ, the following conditions hold: $o_{\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e_2)} \in O$, $t_{\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(e_2)} \in \hat{p}_f$, and $\mathtt{TT}_M(e_2) \in \hat{q}_f$. □

The proof of the previous result is straightforward since it is enough to take into account that the first loop of the algorithm, presented in Figure 3, computes the *matches* function introduced in Definition 9. In particular, the algorithm computes the set of traces of the specification matching the invariants and stores the reached states in $\mathcal{S}_c$. If there does not exist a matching sequence (first condition of Lemma 1), then the invariant is not matched, and it produces an error. This situation is represented in the second part of the algorithm with the assignments $error \leftarrow (\mathcal{T}_{aux} = \emptyset)$. After that, the second condition of Lemma 1 is checked for each matched trace.

4.2. Timed observational invariants

Even though timed consequent invariants allow testers to represent a wide range of properties, some classes of properties cannot be expressed with them. As already commented in Section 2, the original untimed framework provided two types of invariants: (simple) invariants and obligation invariants [12]. While simple invariants allow testers to check properties taking into account what was observed to ensure that something will happen in the future, obligation invariants allow testers to check properties concerning events that already were observed.

The previously presented timed consequent invariant framework represents a temporal adaptation of (simple) invariants. With respect to the temporal adaptation of obligation invariants, the idea is still to follow the pattern "if one observes something in the future, then something has happened in the past". For example, if a disconnection message is observed, then it is necessary to check that the user previously logged into the system. In this paper, this pattern is slightly modified so that *timed observational invariants* can be used to express properties such as "if one observes a certain output in the future and a certain input was observed in the past, then it is necessary to check that some

DOI: 10.1002/stvr

properties hold between these two actions". For example, if a user logged into the system and after 20 time units he receives the disconnection screen, then it is necessary to check that the user both introduced the correct password and that he later pressed the disconnection button.

Definition 10
Let $\mathcal{I}, \mathcal{O}$ be two sets of input and output actions, respectively. The function $\texttt{CInv} : \mathcal{I} \times \mathcal{O} \longrightarrow (\mathcal{O} \cup \{?\}) \times \mathcal{I}_{\mathbb{R}_+} \times (((\mathcal{I} \cup \{?\} \times \mathcal{O} \cup \{?\}) \cup \{\star\}) \times \mathcal{I}_{\mathbb{R}_+})^* \times \mathcal{I} \cup \{?\}$ is such that

$$\texttt{CInv}(\mathcal{I}/\mathcal{O}) = \left\{ \delta \;\middle|\; \begin{array}{l} \exists \alpha_1, \ldots, \alpha_n, \in (((\mathcal{I} \cup \{?\} \times \mathcal{O} \cup \{?\}) \cup \{\star\}) \times \mathcal{I}_{\mathbb{R}_+}), a \in \mathcal{I} \cup \{?\}, \\ z \in \mathcal{O} \cup \{?\}, \hat{m} \in \mathcal{I}_{\mathbb{R}_+} : \delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \wedge \\ \forall j : 1 \leq j < n : (\alpha_j = \star/\hat{p}) \implies (\alpha_{j+1} \neq \star/\hat{q}) \end{array} \right\}$$

An element of $\texttt{CInv}(\mathcal{I}/\mathcal{O})$ will be called a *pattern trace*. The set of all pattern traces will be denoted by $\texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$. During the rest of the paper, a generic pattern trace will be presented by $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a$, where $z \in \mathcal{O} \cup \{?\}$, $\hat{m} \in \mathcal{I}_{\mathbb{R}_+}$, $\alpha_1, \ldots, \alpha_n \in (((\mathcal{I} \cup \{?\} \times \mathcal{O} \cup \{?\}) \cup \{\star\}) \times \mathcal{I}_{\mathbb{R}_+})$ and $a \in \mathcal{I} \cup \{?\}$.

The sequence μ is called a *timed observational invariant*, or simply an observational invariant, if μ is defined according to the following EBNF:

$$\mu ::= i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q}$$

In this expression $\hat{p}, \hat{q} \in \mathcal{I}_{\mathbb{R}_+}$, $i \in \mathcal{I}$, $\beta \subseteq \texttt{CInv}(\mathcal{I}/\mathcal{O})$, and $o \in \mathcal{O}$. The set of timed observational invariants for $\mathcal{I}/\mathcal{O}$ is denoted by $\Psi_{\mathcal{I}/\mathcal{O}}$. □

Intuitively, the previous EBNF expresses that a timed observational invariant starts with an input followed by a set of *pattern traces* and finishes with an output and two time intervals. A pattern trace is a sequence of symbols where each of its components but the first and the last ones is either an expression $a/z/\hat{p}$, with a being either an input or the wildcard ?, z being either an output or the wildcard ?, and $\hat{p}$ being an interval, or an expression $\star/\hat{p}$. The first component of a pattern trace is an expression $z/\hat{p}$, where z is either an output or the wildcard ?, while the last component of the invariant is either an input or the wildcard ?. The occurrence of two consecutive wildcard characters $\star$ is not allowed.

Example 5
The invariant μ_1 means "whenever the input i_2 is observed and in a time belonging to $[10, 12]$ the output o_3 is observed, then it must be checked that the system has performed the output action o_2 in a time belonging to $[4, 6]$, then the system received the input i_1, and emitted the output o_3 in a time belonging to $[5, 7]$".

$$\mu_1 = i_2 \rightarrow \{\langle o_2/[4,6], i_1\rangle\} \leftarrow o_3, [5,7]/[10,12]$$

A complex timed observational invariant can represent more than one behavior in it. For example, the invariant

$$\mu_2 = i_1 \rightarrow \left\{ \begin{array}{c} \langle o_2/[3,16], i_1\rangle, \\ \langle ?/[1,7], \star/[0,\infty], i_3\rangle \end{array} \right\} \leftarrow o_3, [5,7]/[15,17]$$

indicates that "whenever i_1 is observed and in the future o_3 is observed in a time belonging to $[15, 17]$, then it must be checked that the behavior of the system between these two actions conformed to one of the two considered pattern traces. The first one represents that after observing the input i_1, the output o_2 will be observed in a time belonging to $[3, 16]$, followed by the input i_1 and the output o_3 in a time belonging to $[5, 7]$. The second one represents the fact that after observing the input i_1 followed by a (possibly empty) sequence of input/output pairs without occurrences of the input i_3, and observing the input i_3, the output o_3 will be observed in a time belonging to $[5, 7]$". □

Since observational invariants can be defined by a tester, similar to consequent invariants, it must be checked that they are correct with respect to the specification. The general scheme of the algorithm that checks the correctness of an observational invariant with respect to a specification

DOI: 10.1002/stvr

Algorithm `Correctness_Observational_Specification`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) : \texttt{SETTFSM}, \mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q} : \Psi_{\mathcal{I}/\mathcal{O}}$
Result: `Bool`
`Struct` $\mathcal{A}$ $\{current\_state : \mathcal{S}$
$ptt : \wp(\texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \times \mathbf{R}_+)$
$ttime : \mathbf{R}_+\}$;

Initialization of variables

$error \leftarrow$ `false`;
$matched \leftarrow$ `false`;
$Nodes \leftarrow$ `InitialNodes`(M, μ);

Main loop

while $Nodes \neq \emptyset \wedge \neg error$ **do**
`Choose` $node \in Nodes$; $Nodes \leftarrow Nodes \setminus \{node\}$;
$(error, matched_{aux}) \leftarrow$ `CheckNode`$(M, \mu, node)$;
if $\neg error$ **then**
$Nodes \leftarrow Nodes \cup$ `NewNodes`$(M, \mu, node)$;
end
$matched \leftarrow matched \vee matched_{aux}$;
end
$error \leftarrow error \vee \neg matched$;
return$(\neg error)$;

Figure 6. Correctness of a timed observational invariant with respect to a specification.

appears in Figure 6. This algorithm makes use of the functions `InitialNodes`, `CheckNode`, and `NewNodes` defined in Figures 7, 8 and 9 respectively. Next, some additional notation has to be defined for dealing with pattern traces and timed observational invariants. The $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta)$ function computes the length of a given pattern trace. The $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\mu)$ function returns the addition of the lengths of the pattern traces associated with a timed observational invariants. The $\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)$ function reduces a give pattern trace into another pattern trace. The $\texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\delta)$ function returns the number of occurrences of the $wildcard$ $\star$ in a pattern trace, while the $\texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\mu)$ function returns the addition of the number of occurrences of the $wildcard$ $\star$ in the pattern traces associated with a timed observational invariants. The $\texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\delta)$ function computes the next input associated with a pattern trace.

Definition 11
Let $\mathcal{I}$ and $\mathcal{O}$ be two sets of input and output actions respectively. The function $\texttt{Len}_{\mathcal{I}/\mathcal{O}} : \texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ is such that for all $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \in \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$, $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = n$. Note that this function is overloaded for pattern traces and returns the length of a pattern trace. The function $\texttt{Len}_{\mathcal{I}/\mathcal{O}} : \Psi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ is such that for all $\mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q} \in \Psi_{\mathcal{I}/\mathcal{O}}$:

$$\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\mu) = \sum_{\delta \in \beta} \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta)$$

The function $\texttt{Tail}_{\mathcal{I}/\mathcal{O}} : \texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \longrightarrow \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$ is such that for all $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \in \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$:

DOI: 10.1002/stvr

Algorithm `InitialNodes`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) : \texttt{SETTFSM}, \mu = i \to \beta \leftarrow o, \hat{p}/\hat{q} : \Psi_{\mathcal{I}/\mathcal{O}}$
Result: $\wp(\texttt{Struct}\ \mathcal{A})$

Initialization of variables

$Nodes \leftarrow \emptyset$;
$\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\mathcal{S}, \{i\}, O \setminus \{o\}, [0, \infty])$;

Main loop

while $\mathcal{T}_{aux} \neq \emptyset$ **do**
  `Choose` $(s, i, o_{aux}, t_{aux}, s_{aux}) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s, i, o_{aux}, t_{aux}, s_{aux})\}$;
  $node_{new}.current\_state \leftarrow s_{aux}$;
  $node_{new}.ptt \leftarrow \emptyset$;
  $node_{new}.ttime \leftarrow t_{aux}$;
  $\beta_{aux} \leftarrow \beta$;
  while $\beta_{aux} \neq \emptyset$ **do**
    `Choose` $\delta \in \beta_{aux}$; $\beta_{aux} \leftarrow \beta_{aux} \setminus \{\delta\}$;
    ($\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a$ *)*
    if $z = o_{aux} \wedge t_{aux} \in \hat{m}$ **then**
      $node_{new}.ptt \leftarrow node_{new}.ptt \cup \{(\delta, 0)\}$;
    end
  end
  $Nodes \leftarrow Nodes \cup \{node_{new}\}$;
end
return$(Nodes)$;

Figure 7. Function to compute the set of initial nodes.

$$\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta) = \begin{cases} z/\hat{m}, a & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = 0 \\ z/\hat{m}, \alpha_2, \ldots, \alpha_n, a & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) \geq 1 \end{cases}$$

The function $\texttt{Nstars}_{\mathcal{I}/\mathcal{O}} : \texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ is such that for all $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \in \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$:

$$\texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\delta) = \begin{cases} 0 & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = 0 \\ 1 + \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)) & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) > 0 \wedge \alpha_1 = \star/\hat{p} \\ \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)) & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) > 0 \wedge \alpha_1 \neq \star/\hat{p} \end{cases}$$

The function $\texttt{Nstars}_{\mathcal{I}/\mathcal{O}} : \Psi_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathbf{N}$ is such that for all $\mu = i \to \beta \leftarrow o, \hat{p}/\hat{q} \in \Psi_{\mathcal{I}/\mathcal{O}}$:

$$\texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\mu) = \sum_{\delta \in \beta} \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\delta)$$

The function $\texttt{NInp}_{\mathcal{I}/\mathcal{O}} : \texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \longrightarrow \mathcal{I}$ is such that for all $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \in \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$:

$$\texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\delta) = \begin{cases} a & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = 0 \\ i_1 & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) > 0 \wedge \alpha_1 = i_1/o_1/\hat{p}_1 \\ \texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)) & \text{if } \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) > 0 \wedge \alpha_1 = \star/\hat{p}_1 \end{cases}$$

□

DOI: 10.1002/stvr

Algorithm `CheckNode`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) : \texttt{SETTFSM}, \mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q} : \Psi_{\mathcal{I}/\mathcal{O}}, node : \texttt{Struct}\ \mathcal{A}$
Result: `Bool` $\times$ `Bool`

Initialization of variables

$s \leftarrow node.current\_state$;
$\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\{s\}, \mathcal{I}, \{o\}, [0, \max(0, q_2 - node.ttime)])$;
$error \leftarrow$ `false`;
$matched \leftarrow$ `false`;

Main loop

while $\mathcal{T}_{aux} \neq \emptyset \wedge \neg error$ **do**
  `Choose` $(s, i_{aux}, o, t_{aux}, s_{aux}) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s, i_{aux}, o, t_{aux}, s_{aux})\}$;
  if $(node.ttime + t_{aux}) \in \hat{q}$ **then**
    $\beta T \leftarrow node.ptt$;
    $checked \leftarrow$ `false`;
    while $\beta T \neq \emptyset \wedge \neg checked$ **do**
      `Choose` $(\delta, t') \in \beta T$; $\beta T \leftarrow \beta T \setminus (\delta, t')$;
      ($\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a$ *)*
      if $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = 0 \vee (\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) = 1 \wedge \alpha_1 = \star/\hat{m}' \wedge a = i_{aux} \wedge t' \in \hat{m}')$ **then**
        $matched \leftarrow$ `true`;
        $checked \leftarrow (a = i_{aux} \wedge t_{aux} \in \hat{p})$;
      end
    end
    $error \leftarrow \neg checked$;
  end
end
return$(error, matched)$;

Figure 8. Function to check the correctness of a node.

The `Correctness_Observational_Specification`(M, μ) algorithm is essentially a breadth first search. It begins at the root node and explores all the neighboring nodes. Then, for each of those adyacent nodes, it explores their unexplored neighbor, and so on, until it finds a mismatch between the invariant and the specification. An additional structure `Struct` $\mathcal{A}$ is defined to codify the nodes.

The initial phase of the algorithm calls the `InitialNodes` function. It computes the initial set of nodes. Next, the algorithm takes a node and examines it performing the `CheckNode` function. This function provides a verdict about the correctness of this node. Finally, if there is not error then using this node and performing `NewNodes` function, a new set of nodes is computed.

Let $\mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q}$ be a timed observational invariant and $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a `TFSM`. In the worst case, the complexity of the algorithm to decide the correctness of μ with respect to M is in $(|\mathcal{T}|^{|\mathcal{S}|} \cdot |\mathcal{T}| \cdot |\beta|)$. This worst case is computed by taking into account the following facts. The algorithm performs once the function `InitialNodes`. The complexity of this function is in $\mathcal{O}(|\mathcal{T}| \cdot |\beta|)$. Next, a loop that computes a breadth first search algorithm, beings its complexity in $\mathcal{O}(|\mathcal{T}|^{|\mathcal{S}|})$ is perfomed. The functions `CheckNode` and `NewNodes` are executed inside this loop.

DOI: 10.1002/stvr

Algorithm `NewNodes`
Data: $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$: `SETTFSM`, $\mu = i \to \beta \leftarrow o, \hat{p}/\hat{q} : \Psi_{\mathcal{I}/\mathcal{O}}, node$: `Struct` $\mathcal{A}$
Result: $\wp$(`Struct` $\mathcal{A}$)

Initialization of variables

$s \leftarrow node.current\_state$;
$\mathcal{T}_{aux} \leftarrow \texttt{AfterCond}_M(\{s\}, \mathcal{I}, O \setminus \{o\}, [0, q_2 - node.ttime])$;
$Nodes \leftarrow \emptyset$;

Main loop

while $\mathcal{T}_{aux} \neq \emptyset$ **do**
  `Choose` $(s, i_{aux}, o_{aux}, t_{aux}, s_{aux}) \in \mathcal{T}_{aux}$; $\mathcal{T}_{aux} \leftarrow \mathcal{T}_{aux} \setminus \{(s, i_{aux}, o_{aux}, t_{aux}, s_{aux})\}$;
  $node_{new}.current\_state \leftarrow s_{aux}$;
  $node_{new}.ptt \leftarrow \emptyset$;
  $node_{new}.ttime \leftarrow node.ttime + t_{aux}$;
  $\beta T \leftarrow node.ptt$; $all \leftarrow$ `true`; $\beta_a \leftarrow \emptyset$; $\beta_b \leftarrow \emptyset$;
  while $\beta T \neq \emptyset$ **do**
    `Choose` $(\delta, t') \in \beta T$; $\beta T \leftarrow \beta T \setminus (\delta, t')$; $\beta_a \leftarrow \beta_a \cup \{\delta\}$;
    ($\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a$ *)*
    if $\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) > 0$ **then**
      $cond_1 \leftarrow (\alpha_1 = a'/z'/[m_1', m_2'] \wedge a' = i_{aux} \wedge z' = o_{aux} \wedge t_{aux} \in [m_1', m_2'])$;
      $cond_2 \leftarrow (\alpha_1 = \star/[m_1', m_2'] \wedge \texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\delta) = i_{aux} \wedge t' \in [m_1', m_2'])$;
      $cond_3 \leftarrow (\alpha_1 = \star/[m_1', m_2'] \wedge \texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\delta) \neq i_{aux} \wedge (t_{aux} + t') \leq m_2')$;
      $all \leftarrow all \wedge (\alpha_1 = \star/[m_1', m_2']) \wedge m_2' = \infty$;
      if $cond_1 \vee cond_2$ **then**
        $node_{new}.ptt \leftarrow node_{new}.ptt \cup \{(\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta), 0)\}$;
        $\beta_b \leftarrow \beta_b \cup \{\texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)\}$;
      end
      if $cond_3$ **then**
        $node_{new}.ptt \leftarrow node_{new}.ptt \cup \{(\delta, t_{aux} + t')\}$;
        $\beta_b \leftarrow \beta_b \cup \{\delta\}$;
      end
    end
  end
  $cond_4 \leftarrow (q_2 = \infty)$;
  $cond_5 \leftarrow (node_{new}.ttime \geq q_1)$;
  if $(\neg cond_4) \vee (cond_4 \wedge ((\neg cond_5) \vee (cond_5 \wedge ((\beta_a \neq \beta_b) \vee (\beta_a = \beta_b \wedge \neg all)))))$ **then**
    $Nodes \leftarrow Nodes \cup \{node_{new}\}$;
  end
end
return$(Nodes)$;

Figure 9. Function to generate a set of nodes from another node.

DOI: 10.1002/stvr

The complexity of the function `CheckNode` is in $\mathcal{O}(|\mathcal{T}| \cdot |\beta|)$ and the complexity of the function `NewNodes` is also in $\mathcal{O}(|\mathcal{T}| \cdot |\beta|)$.

Definition 12
Let μ be a timed observational invariant and M be a `TFSM`. The invariant μ is *correct* with respect to M if the algorithm `Correctness_Observational_Specification`(M, μ) returns `true`. □

Example 6
Consider the `TFSM` M presented in Figure 2 and the consequent invariants

$$\mu_1 = i_2 \rightarrow \{\langle o_2/[4,6], i_1\rangle\} \leftarrow o_3, [5,7]/[10,12]$$

and

$$\mu_2 = i_1 \rightarrow \left\{ \begin{array}{c} \langle o_2/[3,16], i_1\rangle, \\ \langle ?/[1,7], \star/[0,\infty], i_3\rangle \end{array} \right\} \leftarrow o_3, [5,7]/[15,17]$$

introduced in Example 5. The invariant μ_1 is correct with respect to M while μ_2 is incorrect. Concerning μ_1, the algorithm initially computes the first node. It represents $s_2 \xrightarrow{\mathbf{i_2}/o_2}_5 s_1$. The next step of the algorithm generates, for the initial node, the next nodes taking into account that it can produce an error. In this example, only one node is generated. This node represents that there is a trace matched by the invariant: the one composed from the sequence of transitions $s_2 \xrightarrow{\mathbf{i_2}/o_2}_5 s_1, s_1 \xrightarrow{i_1/\mathbf{o_3}}_6 s_1$. The next step of this algorithm checks whether the trace is correct with respect to the pattern trace $o_2/[4,6], i_1$. Since this trace is checked by this pattern, the invariant μ_1 is correct.

Concerning μ_2, the set of traces that must check the invariant is: $s_3 \xrightarrow{i_1/o_2}_4 s_2, s_2 \xrightarrow{i_2/o_2}_5 s_1, s_1 \xrightarrow{i_1/o_3}_6 s_1$ and $s_3 \xrightarrow{i_1/o_1}_5 s_2, s_2 \xrightarrow{i_2/o_2}_5 s_1, s_1 \xrightarrow{i_1/o_3}_6 s_1$. There exists a trace in this set that does not check a pattern belonging to the invariant. The trace $s_3 \xrightarrow{i_1/o_2}_4 s_2, s_2 \xrightarrow{i_2/o_2}_5 s_1, s_1 \xrightarrow{i_1/o_3}_6 s_1$ does not match either $o_2/[3,16], i_1$ or $?[1,7], \star[0,\infty], i_3$. Therefore, μ_2 is incorrect with respect to M. □

As in the case of timed consequent invariants, a *match* relation will be used to provide a formal alternative characterization of the notion of correctness introduced in Definition 12. Intuitively, a trace *matches* an observational invariant if the initial input and the last output match the initial input and the last output represented in the invariant, respectively, and the addition of all time values belongs to the time interval presented in the invariant. Some additional notation for dealing with traces has to be defined. The $\texttt{SetTracesO}_M(i, \hat{p}, o)$ function computes the set of traces having as initial input i, such that output o appears only once at the end of the trace, and such that the sum of the time values appearing in the timed trace must belong to $\hat{p}$.

Definition 13
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a `TFSM`. The function $\texttt{SetTracesO}_M : \mathcal{I} \times \mathcal{I}_{\mathbb{R}_+} \times \mathcal{O} \longrightarrow \wp(\texttt{Traces}(M))$ is such that for all $i \in \mathcal{I}$, $o \in \mathcal{O}$, and $\hat{p} \in \mathcal{I}_{\mathbb{R}_+}$:

$$\texttt{SetTracesO}_M(i, \hat{p}, o) = \left\{ e \;\middle|\; \begin{array}{l} \exists i_2, \ldots, i_r \in \mathcal{I}, o_1, \ldots, o_{r-1} \in \mathcal{O}, t_1, \ldots, t_r \in \mathbf{R}_+, r \geq 2 : \\ e = \langle i/o_1/t_1, i_2/o_2/t_2, \ldots, i_r/o/t_r\rangle \in \texttt{Traces}(M) \wedge \\ \forall j : 1 \leq j < \texttt{Len}_{\mathcal{I}/\mathcal{O}}(e) : o_j \neq o \wedge \texttt{TT}_M(e) \in \hat{p} \end{array} \right\}$$

The function $\texttt{MatchO}_M : \texttt{Traces}(M) \times \Psi_{\mathcal{I}/\mathcal{O}} \longrightarrow \texttt{Bool}$ is such that for all $e \in \texttt{Traces}(M)$ and $\mu \in \Psi_{\mathcal{I}/\mathcal{O}}$:

$$\texttt{MatchO}_M(e, \mu) = \left\{ \begin{array}{ll} \texttt{true} & \text{if } e \in \texttt{SetTracesO}_M(i, \hat{q}, o) \\ \texttt{false} & \text{if } e \notin \texttt{SetTracesO}_M(i, \hat{q}, o) \end{array} \right.$$

Let M be a `TFSM`, μ be a timed observational invariant, and $e \in \texttt{Traces}(M)$. The trace e *matches* μ if $\texttt{MatchO}_M(e, \mu)$ returns `true`. □

DOI: 10.1002/stvr

The $\texttt{Check}_M(e, \delta)$ function computes whether the trace e of M checks the pattern trace δ. This definition uses the $\texttt{Check}^{\star}_M(e, i, \hat{p}, t)$ auxiliary predicate to deal with occurrences of the $\star$ wildcard character in a time $\hat{p}$ without performing the input i.

Definition 14
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. The function $\texttt{Check}^{\star}_M : \texttt{Traces}(M) \times \mathcal{I} \times \mathcal{I}_{\mathbb{R}_+} \times \mathbf{R}_+ \longrightarrow \texttt{Traces}(M)$ is such that for all $e = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle \in \texttt{Traces}(M)$, $i \in \mathcal{I}$, $\hat{p} \in \mathcal{I}_{\mathbb{R}_+}$, and $t \in \mathbf{R}_+$:

$$\texttt{Check}^{\star}_M(e, i, \hat{p}, t) = \begin{cases} \langle\rangle & \text{if } (t > p_2) \vee (\texttt{Len}_{\mathcal{I}/\mathcal{O}}(e) = 0) \vee \\ & \quad (i_1 = i \wedge t \notin \hat{p}) \\ e & \text{if } i_1 = i \wedge t \in \hat{p} \\ \texttt{Check}^{\star}_M(\texttt{Tail}_M(e), i, \hat{p}, t + t_1) & \text{if } t \leq p_2 \wedge i_1 \neq i \end{cases}$$

The function $\texttt{Check}_M : \texttt{Traces}(M) \times \texttt{PATTR}_{\mathcal{I}/\mathcal{O}} \longrightarrow \texttt{Bool}$ is such that for all $e_1 \in \texttt{Traces}(M)$ and $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a \in \texttt{PATTR}_{\mathcal{I}/\mathcal{O}}$, $\texttt{Check}_M(e_1, \delta)$ is equal to:

$$\begin{cases} \texttt{false} & \text{if } (u = 0) \vee (u > 1 \wedge s = 0) \vee (u = 1 \wedge s > 0) \\ i_1 = a & \text{if } u = 1 \wedge s = 0 \\ \texttt{Check}_M(e_2, \texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)) & \text{if } u > 1 \wedge s > 0 \wedge \alpha_1 = \star/\hat{p} \\ i_1 = a \wedge o_1 = z \wedge t_1 \in \hat{p} \wedge & \\ \texttt{Check}_M(\texttt{Tail}_M(e_1), \texttt{Tail}_{\mathcal{I}/\mathcal{O}}(\delta)) & \text{if } u > 1 \wedge s > 0 \wedge \alpha_1 = a/z/\hat{p} \end{cases}$$

where $e_2 = \texttt{Check}^{\star}_M(e_1, \texttt{NInp}_{\mathcal{I}/\mathcal{O}}(\delta), \hat{p}, 0)$, $s = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\delta)$, and $u = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(e_1)$.

A trace $e = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle$ of M *checks* the pattern trace $\delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a$ if $o_1 = z$, $t_1 \in \hat{m}$, and $\texttt{Check}_M(\langle i_2/o_2/t_2, \ldots, i_r/o_r/t_r \rangle, \delta)$ returns `true`. □

The previous definitions of matching and checking can be used to give an alternative characterization of the soundness of a timed observational invariant with respect to a specification. This notion is based on the idea that if a trace of the specification matches the invariant, then it must check a pattern trace belonging to β.

Lemma 2
Let M be a TFSM M and $\mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q}$ be a timed observational invariant. The invariant μ is *correct* with respect to M if the following conditions hold: there exists at least one trace $e \in \texttt{Traces}(M)$ that matches μ and for all trace $e \in \texttt{Traces}(M)$ that matches μ there exists $\delta \in \beta$ such that e checks δ. □

The proof of the previous result is easy and it is based on the following ideas. There are two conditions in Lemma 2 that must be checked to ensure the correctness of an invariant. The first one depends on the existence of a matched trace of the specification and the second condition of Lemma 2 checks that for all matched trace e there exists a pattern trace δ in the invariant such that e checks δ.

These two conditions are computed together in the algorithm presented in Figure 6. First, the algorithm computes the tree of matched traces. For each node, the algorithm checks that if the trace is matched, then there exists a pattern trace of the invariant that checks this trace. Finally, if the `Correctness_Observational_Specification` algorithm returns `true`, then the invariant is correct with respect to the specification.

5. CORRECTNESS OF LOGS AGAINST INVARIANTS

This section presents an implementation relation to formally define what a good implementation is with respect to a specification. The ultimate goal is to show the correctness of the passive testing approach presented in this paper. The section gives two algorithms: one that checks the conformance between logs and timed consequent invariants and another one that checks the conformance between

DOI: 10.1002/stvr

logs and timed observational invariants. A simple timed implementation relation is considered but other alternative relations [47] could be easily incorporated to the framework.

Definition 15
Let M_S and M_I be two TFSMs. M_I conf M_S denotes that $\texttt{Traces}(M_I) \subseteq \texttt{Traces}(M_S)$. □

5.1. Correctness of logs against timed consequent invariants

Essentially, a log is *incorrect* with respect to a timed consequent invariant if there exists a subsequence of the log that matches the invariant, that is, it is *coherent* until the last input of the invariant, but it does not fulfill the requirements expressed in its last part. Therefore, a log is *correct* with respect to an invariant if it does not violate any requirement expressed in the invariant.

An algorithm to establish the conformance between logs and timed consequent invariants is presented. The core of the algorithm is given in Figure 10. The algorithm traverses all the elements of the log and checks whether each subsequence of it matches the invariant. If this happens, then the restrictions of the invariant are checked.

Definition 16
Let ϕ be a timed consequent invariant and l be a log recorded from an IUT. The log l is *correct* with respect to ϕ if the algorithm $\texttt{Correctness\_Logs\_Consequent}(l, \phi)$ returns true. □

Let ϕ be a timed consequent invariant and l be a log recorded from an IUT. The complexity of the pattern matching strategy is in the worst case $\mathcal{O}(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)^2 \cdot k + \texttt{Len}_{\mathcal{I}/\mathcal{O}}(l) \cdot (s-k))$, where $k = \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi)$ and $s = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$. Note that even though *good* algorithms for pattern matching on strings perform in $\mathcal{O}(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(l))$ (after the *pre-processing* phase) this complexity cannot be achieved because all the occurrences of the pattern in the log must be checked. However, if the length of the invariant is *much smaller* than the length of the log and the number of stars is low, as it is usually the case, the complexity is almost linear with respect to the length of the log. The next result states the soundness of the approach.

Lemma 3
Let ϕ be a timed consequent invariant, M_I be a TFSM, and $l \in \texttt{Traces}(M_I)$ be a log of M_I. The log l *is correct* with respect to ϕ if for all $l' = \langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k \rangle$, with $1 \leq j \leq k \leq \texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)$, such that l' matches ϕ, $o_k \in O$, $t_k \in \hat{p}_f$, and $\texttt{TT}_M(l') \in \hat{q}_f$. □

The proof of this result is easy and it is based on the following ideas. A log l is correct with respect to a timed consequent invariant if each matching sublog l' of l respects some conditions about its structure. The Correctness_Logs_Consequent algorithm implements indeed these conditions (see Definition 16). The two while loops in Figure 10 compute the sublogs of the log by using the m and k indexes for the first and last elements, respectively. If the sublog matches the invariant then the last two lines of the loop check the conditions of Lemma 3.

The following example shows how the sublogs of a log are considered to check the correctness of the approach.

Example 7
Let $l = \langle i_1/o_1/3, i_1/o_2/4 \rangle$ be a log of a system and $\phi = i_1 \mapsto \{o_1\}[3,4] \rhd [3,4]$ be a timed consequent invariant. It is easy to check that $\langle i_1/o_1/3 \rangle$ matches ϕ but l is not correct with respect to ϕ since it contains the sublog $\langle i_1/o_2/4 \rangle$. □

Note that the subsequence of the matched log can be longer than the invariant because invariants can include the wildcard $\star$. The next result presents the relation among specification, implementation, logs, and timed consequent invariants. Consider a log recorded from an implementation and a correct timed consequent invariant with respect to a specification. If the timed consequent invariant detects an error in the log, then the implementation does not conform to the specification.

Theorem 1
Let M_S and M_I be two TFSMs and ϕ be a correct timed consequent invariant with respect to M_S.

DOI: 10.1002/stvr

Algorithm `Correctness_Logs_Consequent`
Data: $l = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r\rangle : (\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star$,
$\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f : \Phi_{\mathcal{I}/\mathcal{O}}$
Result: `Bool`

Initialization of variables

$error \leftarrow$ `false`; $m \leftarrow 1$;

Main loop

while $m \leq \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \ \wedge \neg\, error$ **do**
  $k \leftarrow m$; $tt \leftarrow 0$; $j \leftarrow 1$; $matching \leftarrow$ `true`;
  while $j \leq \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) \wedge k \leq \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \ \wedge \ matching$ **do**
    if $\alpha_j = a/z/\hat{p}$ **then**
      $matching \leftarrow (i_k = a \wedge o_k = z \wedge t_k \in \hat{p})$; $tt \leftarrow tt + t_k$; $k \leftarrow k + 1$;
    else
      (* $\alpha_j = \star/[p_1, p_2]$ *)
      $t_{partial} \leftarrow 0$;
      while
      $k < \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \wedge t_{partial} < p_2 \wedge i_k \neq \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\alpha_j, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f)$ **do**
        $t_{partial} \leftarrow t_{partial} + t_k$; $k \leftarrow k + 1$;
      end
      $tt \leftarrow tt + t_{partial}$;
      $matching \leftarrow (i_k = \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(\alpha_j, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f) \wedge t_{partial} \in [p_1, p_2])$;
    end
    $j \leftarrow j + 1$;
  end
  (* $j = \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) + 1$ *indicates that the invariant was completely traversed* *)
  if $matching \wedge j = (\mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) + 1) \wedge (i_k = i_f)$ **then**
    $error \leftarrow (o_k \notin O \vee t_k \notin \hat{p}_f \vee tt + t_k \notin \hat{q}_f)$;
  end
  $m \leftarrow m + 1$;
end
return$(\neg error)$;

Figure 10. Correctness of a log with respect to a timed consequent invariant.

Let l be a log recorded from M_I. If l is not correct with respect to ϕ, then M_I `conf` M_S does not hold.

Proof: If l is not correct with respect to $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$, then there exists a subsequence $l' = \langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k\rangle$ of l such that l' matches ϕ and either $o_k \notin O$ or $t_k \notin \hat{p}_f$ or $\mathtt{TT}_M(l') \notin \hat{q}_f$. If $o_k \notin O$, then $l' \in \mathtt{Traces}(M_I)$ but $l' \notin \mathtt{Traces}(M_S)$. Thus, M_I `conf` M_S does not hold. If $t_k \notin \hat{p_f}$, then $l' \in \mathtt{Traces}(M_I)$ but $l' \notin \mathtt{Traces}(M_S)$. Thus, M_I `conf` M_S does not hold. Finally, if $\mathtt{TT}_M(l') \notin \hat{q}_f$, then $l' \notin \mathtt{Traces}(M_S)$ because time values of invariants fit those of the specification. In this case, again, M_I `conf` M_S does not hold.

□

DOI: 10.1002/stvr

5.2. Correctness of logs against timed observational invariants

This section presents a method to establish the correctness of a log, collected from an IUT, with respect to a timed observational invariant. The main algorithm is given in Figure 11. The idea is to traverse the log and decide whether there exists a subsequence e matching the invariant. In this case, it must be established that at least one pattern trace belonging to β checks it. For this task, the Checked_PatternTrace function, given in Figure 12, is used. If there is not a pattern trace checked by e, then an error is produced.

Definition 17
Let μ be a timed observational invariant and l be a log of an IUT. The log l is *correct* with respect to μ if the algorithm $\texttt{Correctness\_Logs\_Observational}(l, \mu)$ returns true. □

Let $\mu = i \rightarrow \beta \leftarrow o, \hat{p}/\hat{q}$ be a timed observational invariant and l be a log. This matching strategy works in the worst case in $\mathcal{O}(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)^3 \cdot k + \texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)^2 \cdot (s-k))$, where $k = \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\mu)$ and $s = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\mu)$. As it was the case with timed consequent invariants, if the invariant is much shorter than the length of the log and the number of appearances of the wildcard $\star$ is low, as it is usually the case, then this complexity becomes almost $\mathcal{O}(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)^2)$. The following result states the soundness of the approach.

Lemma 4
Let μ be a timed observational invariant, M_I be an IUT, and $l \in \texttt{Traces}(M_I)$ be a log of M_I. The log l is correct with respect to μ if for all $l' = \langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k \rangle$, with $1 \leq j < k \leq \texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)$, such that l' matches μ there exists $\delta \in \beta$ such that l' checks δ. □

This result is immediate since the algorithm given in Figure 11 initially computes the set of sublogs that are matched. If this set is empty, then the log is correct; otherwise, for each matched (sub)log, the algorithm looks for a pattern trace checked by this sublog. This task is carried out by the Checked_PatternTrace function, introduced previously.

In a similar way to timed consequent invariants, it is possible to give a relation among invariants, implementation, logs, and specification. The idea is again that if there is a correct timed observational invariant with respect to a specification, then if this invariant detects an error in a log recorded from an implementation, then this implementation does not conform to the specification.

Theorem 2
Let M_S and M_I be two TFSMs and μ be a correct timed observational invariant with respect to M_S. Let l be a log recorded from M_I. If l is not correct with respect to μ, then M_I does not conform to M_S.

Proof: If l is not correct with respect to μ, then there exists a subsequence $l' = \langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k \rangle$, with $1 \leq j < k \leq \texttt{Len}_{\mathcal{I}/\mathcal{O}}(l)$, of l such that l' matches μ and there does not exist $\delta \in \beta$ such as e checks δ. If μ is correct with respect to M_S, then for all trace $e \in \texttt{Traces}(M_S)$ that matches μ there exists $\delta \in \beta$ such that e checks δ. Thus, $l' \in \texttt{Traces}(M_I)$ but $l' \notin \texttt{Traces}(M_S)$. Therefore, M_I conf M_S does not hold.

□

6. PASTE

This section presents a PASive TEsting tool, called PASTE, that allows users to work with the formal framework presented in this paper. The original core and the GUI were implemented in JAVA and initially were a stand-alone project. Later, it was decided to integrate this academic tool as a module of the monitoring software developed by the Spanish SME Peopleware. In order to improve the performance of the tool and the possibilities to integrate it with other existing tools, the core of PASTE was rewritten in C++ and the GUI was adapted accordingly. The tool can be downloaded from https://simba.fdi.ucm.es/paste.

DOI: 10.1002/stvr

Algorithm: `Correctness_Logs_Observational`
Data: $l = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r\rangle : (\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star, \mu = i \to \beta \leftarrow o, \hat{p}/\hat{q} : \Psi_{\mathcal{I}/\mathcal{O}}$
Result: `Bool`

Initialization of variables

$error \leftarrow$ `false`; $j \leftarrow 1$;

Main loop

while $j < \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \wedge \neg error$ **do**
  if $i_j = i$ **then**
    $tt \leftarrow t_j$; $k \leftarrow j + 1$;
    (tt stores the time of the matched sequence, j computes the initial index of the matched sequence, and k computes the final index of the matched sequence*)*
    while $k \leq \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \wedge (o_k \neq o) \wedge tt < q_2$ **do**
      $tt \leftarrow tt + t_k$;
      $k \leftarrow k + 1$;
    end
    if $o_k = o \wedge (tt + t_k) \in \hat{q}$ **then**
      $\beta_{aux} \leftarrow \beta$; $cmatching \leftarrow$ `false`;
      ($cmatching$ holds if $\langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k\rangle$ checks a pattern trace in β*)*
      while $\beta_{aux} \neq \emptyset \wedge \neg cmatching$ **do**
        `Choose` $\delta \in \beta_{aux}$; $\beta_{aux} \leftarrow \beta_{aux} \setminus \{\delta\}$;
        $cmatching \leftarrow t_k \in \hat{p} \wedge$ `Checked_PatternTrace`$(\langle i_j/o_j/t_j, \ldots, i_k/o_k/t_k\rangle, \delta)$
      end
      $error \leftarrow \neg cmatching$;
    end
  end
  $j \leftarrow j + 1$;
end
return$(\neg error)$;

Figure 11. Correctness of a log with respect to a timed observational invariant.

6.1. Functionalities

PASTE is a tool that allows users to automatize the passive testing methodology presented in this paper. The tool obtains the data from a database that contains a set of invariants, the logs to be checked, and, optionally, a specification represented as a `TFSM` model. First, the information in the database is transformed into the internal data format of the application. In particular, in this phase time information of the log is transformed so that it is expressed in the same time units as the specification and invariants.

Next, the algorithm that checks the correctness of the invariants with respect to the specification is applied. Invariants have to be checked against the specification before the logs are checked with respect to the invariants. If the specification is not provided, then the tool considers that the invariants are correct. Once a correct set of invariants is fixed, it is possible to check the correctness of the logs with respect to the invariants by executing the corresponding algorithms. If an error is detected, then PASTE notifies it to the tester.

DOI: 10.1002/stvr

Algorithm Checked_PatternTrace
Data: $l = \langle i_1/o_1/t_1, \ldots, i_r/o_r/t_r \rangle : (\mathcal{I} \times \mathcal{O} \times \mathbf{R}_+)^\star, \delta = z/\hat{m}, \alpha_1, \ldots, \alpha_n, a : \mathtt{PATTR}_{\mathcal{I}/\mathcal{O}}$
Result: Bool

Initialization of variables

$checked \leftarrow (o_1 = z) \wedge (t_1 \in \hat{m}) \wedge (i_r = a)$;
($checked$ denotes that the trace continues being checked with respect to the pattern trace*)*
$k \leftarrow 2$; $j \leftarrow 1$;

Main loop

while $k < \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \ \wedge \ checked \wedge j \leq \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\delta)$ **do**
  if $\alpha_j = a/z/\hat{p}$ **then**
    $checked \leftarrow (a = i_k \wedge z = o_k \wedge t_k \in \hat{p})$;
    $k \leftarrow k + 1$;
  else
    ($\alpha_j = \star/[p_1, p_2]$ *)*
    $t_{partial} \leftarrow 0$;
    while $k < \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l) \wedge t_{partial} < p_2 \wedge i_k \neq \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(z/\hat{m}, \alpha_j, \ldots, \alpha_n, , a)$ **do**
      $t_{partial} \leftarrow t_{partial} + t_k$;
      $k \leftarrow k + 1$;
    end
    $checked \leftarrow (i_k = \mathtt{NInp}_{\mathcal{I}/\mathcal{O}}(z/\hat{m}, \alpha_j, \ldots, \alpha_n, , a)) \wedge t_{partial} \in [p_1, p_2]$;
  end
  $j \leftarrow j + 1$;
end
(Make sure that the complete trace has been checked against the complete pattern trace *)*
$checked \leftarrow checked \wedge (k = \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(l)) \wedge (j = \mathtt{Len}_{\mathcal{I}/\mathcal{O}}(\delta) + 1)$;
return ($checked$);

Figure 12. Function to compute whether a log checks a pattern trace.

In addition to the theoretical framework, PASTE implements a module to provide a measure of how *good* a set of invariants is. In order to do it, a methodology based on mutation testing [55, 22, 66, 53] is used. In PASTE, the specification is *mutated* and for each *mutant* a log is recorded. These logs are checked against the set of available invariants in order to determine, based on the obtained results, their level of fault detection. If the evaluation of the log against an invariant finds an error, then the invariant *kills* the mutant that generated this log. The idea is that if an invariant finds many errors in the logs recorded from mutants, then the probability that it detects an error in a faulty IUT is higher. Note that only *first order mutants* are considered, that is, mutants obtained by the application of one mutation operator. PASTE provides three different mutation operators: *Changing the Goal State of a transition* (CGS), *Changing Output* (CO), and *Changing Time* (CT). The first one corresponds to create a mutant of a specification by changing the final state of a transition. The second mutation operator creates a mutant by changing the output associated to a transition. Finally, the last one modifies the time value associated with a transition.

Definition 18
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. The *changing goal state* mutant operator $\mathtt{CGS}_M : \mathcal{T} \times \mathcal{S} \longrightarrow$ SETTFSM, the *changing output* mutant operator $\mathtt{CO}_M : \mathcal{T} \times \mathcal{O} \longrightarrow$ SETTFSM, and the *changing time*

DOI: 10.1002/stvr

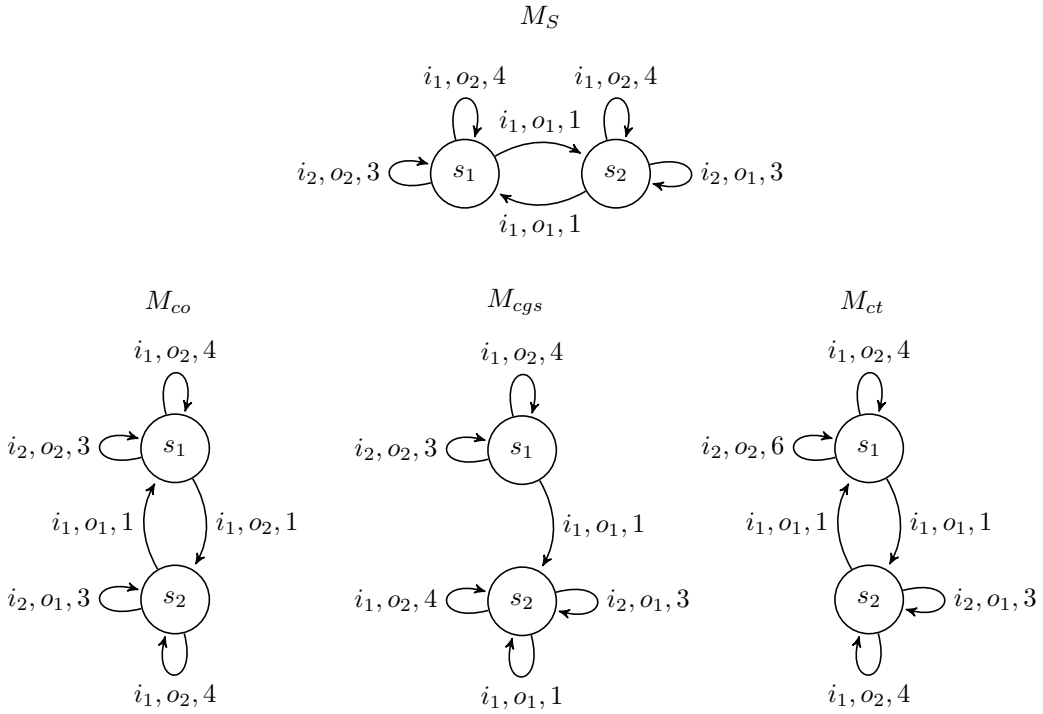


Figure 13. A specification M_S and three of its mutants.

value mutant operator $\texttt{CT}_M : \mathcal{T} \times \mathbf{R}_+ \longrightarrow \texttt{SETTFSM}$ are functions to generate mutants such that for all $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0) \in \texttt{SETTFSM}$, $tc = (s, i, o, t, s') \in \mathcal{T}$, $s_m \in \mathcal{S}, o_m \in \mathcal{O}, \alpha \in \mathbf{R}_+$:

$$\texttt{CGS}_M(tc, s_m) = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}_{\texttt{CGS}}, s_0), \text{where } \mathcal{T}_{\texttt{CGS}} = \{tr | tr \in \mathcal{T} \wedge\ tr \neq tc\} \cup \{(s, i, o, t, s_m)\}$$

$$\texttt{CO}_M(tc, o_m) = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}_{\texttt{CO}}, s_0), \text{where } \mathcal{T}_{\texttt{CO}} = \{tr | tr \in \mathcal{T} \wedge\ tr \neq tc\} \cup \{(s, i, o_m, t, s')\}$$

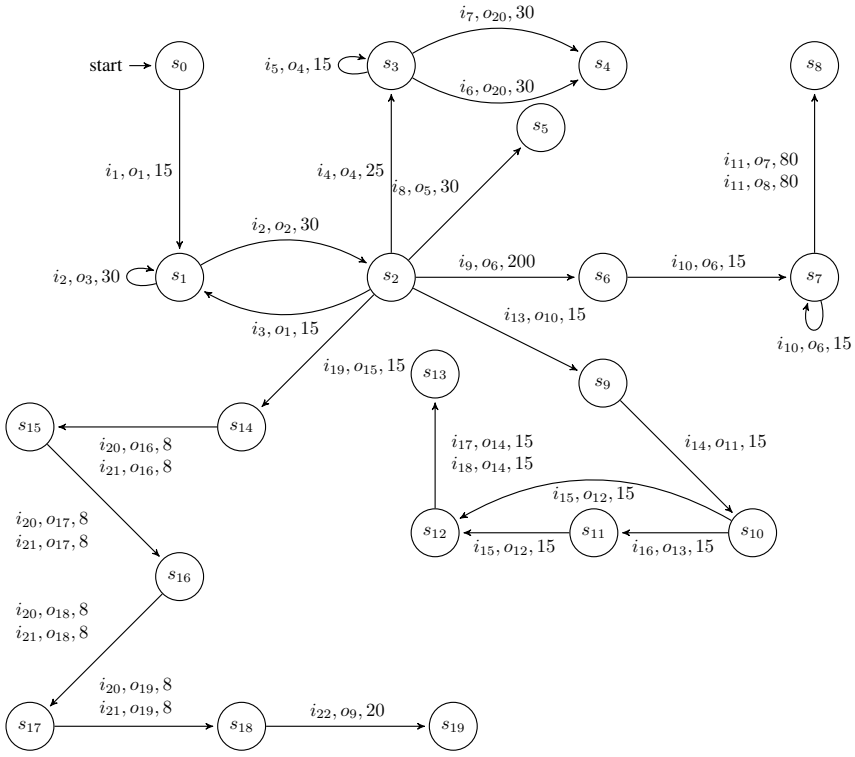
$$\texttt{CT}_M(tc, \alpha) = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}_{\texttt{CT}}, s_0), \text{where } \mathcal{T}_{\texttt{CT}} = \{tr | tr \in \mathcal{T} \wedge\ tr \neq tc\} \cup \{(s, i, o, t + \alpha \cdot t, s')\}$$

Mutants of M are generated after applying mutant operators to M. The set of all mutants of M is denoted by $\texttt{MUT}_M$. Let $\mathcal{M} \subseteq \texttt{MUT}_M$ be a set of mutants generated from M and consider that a set of logs is extracted from these mutants. $\texttt{MTrazes}_{\mathcal{M}}$ is a set of pairs (mutant, log), where several logs can be associated with the same mutant. A mutant $M_u \in \mathcal{M}$ is *killed* by the invariant ψ if there exist a pair $(M_u, l) \in \texttt{MTrazes}_{\mathcal{M}}$ such that l is incorrect with respect to ψ. $\texttt{Skm}_M(\psi, \texttt{MTrazes}_{\mathcal{M}}) \subseteq \mathcal{M}$ denotes the set of mutants killed by ψ. Similarly, $\texttt{Remove}_M(\psi, \texttt{MTrazes}_{\mathcal{M}}) \subseteq \texttt{MTrazes}_{\mathcal{M}}$ returns those pairs (mutant, log) belonging to $\texttt{MTrazes}_{\mathcal{M}}$ such that the mutant has not been killed by ψ. □

Example 8

Figure 13 presents different TFSMs to illustrate the mutation operators. Consider the machine M_{co}. In this case, the CO mutant operator has been applied. The transition $s_1 \xrightarrow{i_1/o_1}_1 s_2$ belonging to M_S has been replaced by $s_1 \xrightarrow{i_1/\mathbf{o_2}}_1 s_2$. In order to build another mutant, the CGS mutant operator can be applied, for example, to generate the machine M_{cgs}. In this case, the transition $s_2 \xrightarrow{i_1/o_1}_1 s_1$ has been replaced by the transition $s_2 \xrightarrow{i_1/o_1}_1 \mathbf{s_2}$. Finally, M_{ct} is obtained by applying the CT mutant operator to M_S. In this case, the operator replaces the time value associated with the transition $s_1 \xrightarrow{i_2/o_2}_3 s_1$ by 6. □

DOI: 10.1002/stvr



$$\mathcal{I} = \left\{ \begin{array}{lll} i_1 = \texttt{connect} & i_2 = \texttt{login} & i_3 = \texttt{disconnection} \\ i_4 = \texttt{profile} & i_5 = \texttt{data} & i_6 = \texttt{cancel} \\ i_7 = \texttt{save} & i_8 = \texttt{marks} & i_9 = \texttt{register} \\ i_{10} = \texttt{data\_subject} & i_{11} = \texttt{save\_registration} & i_{12} = \texttt{return\_option} \\ i_{13} = \texttt{send\_msg} & i_{14} = \texttt{write\_msg} & i_{15} = \texttt{send} \\ i_{16} = \texttt{attachF} & i_{17} = \texttt{save\_and\_send} & i_{18} = \texttt{cancel\_and\_exit} \\ i_{19} = \texttt{sel\_quest} & i_{20} = \texttt{sel\_true} & i_{21} = \texttt{sel\_false} \\ i_{22} = \texttt{send\_quest} & & \end{array} \right\}$$

$$\mathcal{O} = \left\{ \begin{array}{lll} o_1 = \texttt{welcome\_scr} & o_2 = \texttt{option\_scr} & o_3 = \texttt{error\_user} \\ o_4 = \texttt{profile\_scr} & o_5 = \texttt{marks\_scr} & o_6 = \texttt{register\_scr} \\ o_7 = \texttt{confirmation\_scr} & o_8 = \texttt{no\_confirmation\_scr} & o_9 = \texttt{quest\_sent} \\ o_{10} = \texttt{msg\_scr} & o_{11} = \texttt{msg\_written} & o_{12} = \texttt{msg\_sent} \\ o_{13} = \texttt{attached\_scr} & o_{14} = \texttt{ok\_msg} & o_{15} = \texttt{question\_1} \\ o_{16} = \texttt{question\_2} & o_{17} = \texttt{question\_3} & o_{18} = \texttt{question\_4} \\ o_{19} = \texttt{quest\_finished} & o_{20} = \texttt{updating\_scr} & \end{array} \right\}$$

Figure 14. Specification of `SSadmin` using a `TFSM` model.

6.2. The system `SSadmin`

This section presents the results obtained from the experiments that were performed to estimate the *quality* of a set of the invariants. Figure 14 presents the specification of the system: `SSadmin`. This system is used by students to check their marks, their student information profile, to send emails, to fill questionnaires and, at the beginning of the academic year, to register their subjects. This paper considers a simplified version of the system. Essentially, data is not used but the main features concerning the input/output behavior of the system are included.

The sets of input and output actions, $\mathcal{I}$ and $\mathcal{O}$, are given in Figure 14. The initial state, s_0, corresponds to the point in which the users connect to the system. The nodes represent the states of the model and the edges represent its transitions. $\mathcal{T}_0$ is the subset of depicted transitions. For the sake of clarity, not all the transitions are included in the figure since this would overload the graph. Specifically, the transitions corresponding to the functionality that allows users to return to `option_scr` by introducing the `return_option` input at some states have been removed. These

Prepared using stvrauth.cls
DOI: 10.1002/stvr

transitions are given by the set

$$\mathcal{T}_1 = \{s_i \xrightarrow{i_{12}/o_2}_{10} s_2 \mid s_i \in \{s_4, s_5, s_8, s_{13}, s_{19}\}\}$$

Thus, the set of transitions of `SSadmin` corresponds to the union of these two sets of transitions, that is, $\mathcal{T} = \mathcal{T}_0 \cup \mathcal{T}_1$.

Next, the standard interaction between a student and `SSadmin` is described. Note that the specification is described from the point of view of the system while this example is presented from the point of view of the student because it is more intuitive. Thus, inputs of the student are outputs of the system and vice versa. The behavior of the system has been divided in five different stages. The first one corresponds to the *connection* phase. When students `connect`, the system shows the `welcome_scr` message. At this point, students can log into the system. If an erroneous `login` is introduced, then the system returns the `error_user` message. If the student introduces the correct `login`, then the system will show `option_scr`. At this screen, the student can log out by answering `disconnection` and the system will return to `welcome_scr`. The time values associated with the processes of connection and disconnection are 30 and 15 time units, respectively. The difference between these amounts is due to the fact that during the login phase the system must access the database for checking the correctness of the provided information. When the student logs out, it is not necessary to interact with the database.

The second stage includes the most frequent operations performed by students: checking marks and accessing and modifying personal profiles. When a student is connected to `SSadmin`, if she introduces `profile`, then the system will show the `profile_scr`. Then, the student is able to change some personal information, such as her e-mail and telephone number. Each data that she might change is introduced by the `data` action and the system replies by showing the `profile_scr`. When the student updates her information, she can either `save` the changes or `cancel` the operation. Both actions lead to the `updating_scr` and when the student introduces `return_option` the system will show the `option_scr`. The second operation of this stage corresponds to checking the marks. The student can access them by using the `marks` action and the system will show `marks_scr`. In order to return to the `option_scr` the student must introduce the `return_option` action. The time values associated with these transitions reflect the difference between the values which are extracted from the disk and the values which are in temporal memory. For example, when the students are modifying their profile, the changes are not stored until the users save them. Thus, the access to this data is faster.

The third stage corresponds to the *register* feature. This feature is available only at the beginning of an academic course and allows the student to register their subjects. This is one of the most important and critical parts of `SSadmin`. After the student logs into the system, she must introduce `register` and the system will show the `register_scr`. The time associated with this transition is 200. This amount of time is due to the fact that the system has to search and filter the available subjects for the student. In order to choose the subjects, the student must introduce the `data_subject`. When she finishes the process and introduces `save_registration` the system will display either `confirmation_scr`, if the registration was correct, or `no_confirmation_scr`, if there was an error.

The fourth stage corresponds to the *internal mail* capability that allows a student to send messages to other students. These messages can include attachments. In order to send a message, the student must introduce `send_msg`. The system will show the `msg_scr`, where the student can write the text of the message. This action is performed by `write_msg` and the screen returned by the system is `msg_written`. After writing the message, the student can either send the message, by applying the input `send`, or attach a file, by performing the input `attachF`. The second option leads the student to the `attached_scr` where she can attach the file. After the student introduces the `send` input, the system will ask the student whether she confirms the sending of the message. The student confirms the sending, by applying the input `save_and_send`, or cancels it, by answering `cancel_and_exit`. In both cases, the system will show the `ok_msg` screen.

The last stage corresponds to the *questionnaire* feature. This feature allows `SSadmin` to obtain some feedback in order to improve future versions of the system based on user experiences. The

DOI: 10.1002/stvr

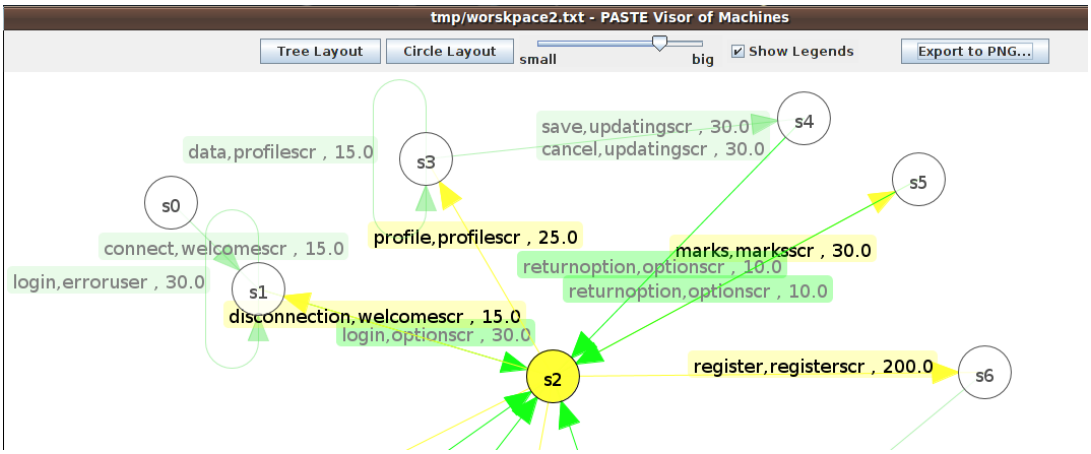


Figure 15. Specification of SSadmin in PASTE.

action sel_quest leads the student to the sequential presentation of the questionnaire screens: question_1, question_2, question_3, and question_4. The student will navigate them by answering either sel_true or sel_false. Finally, Figure 15 partially shows the SSadmin specification.

6.3. *A set of invariants for* SSadmin

The first invariant denotes the property that after login, if a user eventually disconnects from the system then the welcome_scr must be displayed.

$$Invar_1 = \begin{array}{c}\texttt{login/option\_scr}/[20,40],\\ \star/[0,\infty],\texttt{disconnection}\end{array} \mapsto \{\ \texttt{welcome\_scr}\ \}/[14.5,15.5] \rhd [35,\infty]$$

In addition to the functional behavior, the $Invar_1$ invariant establishes that the observation of login and the display of option_scr must belong to the interval $[20, 40]$. Similarly, the amount of time elapsed between the input disconnection and the output welcome_scr must be greater than 14.5 and less than or equal to 15.5. Finally, the sum of all the time values observed between login and welcome_scr must be greater than 35. The next invariant can be used to observe two different behaviors after the specified input:

$$Invar_2 = \texttt{login} \mapsto \left\{ \begin{array}{l}\texttt{option\_scr},\\ \texttt{error\_user}\end{array} \right\}/[29.5,30.5] \rhd [29.5,30.5]$$

Intuitively, this invariant expresses that after observing an occurrence of login in the log it is necessary to observe either option_scr or error_user. Moreover, the amount of time associated with these actions must belong to the time interval $[29.5, 30.5]$. The next invariant focuses on the profile option:

$$Invar_3 = \texttt{data/profile\_scr}/[10,20],\texttt{save} \mapsto \{\texttt{updating\_scr}\}/[29.5,30.5] \rhd [40,50]$$

This invariant denotes that after inserting the last update of the data in the system and save all the changes, the updating_scr must be displayed. In addition, the total amount of time to perform this process must belong to the interval $[40, 50]$. Note that the delay tolerated by the last timed restriction can be different from the sum of all time intervals presented in the invariant. For example, $[10, 20] + [29.5, 30.5] \neq [40, 50]$.

Finally, the invariant $Invar_4$ describes the fact that if a student is at the option_scr and she inserts the marks input, then the marks_scr must appear before a certain amount of time passes:

DOI: 10.1002/stvr

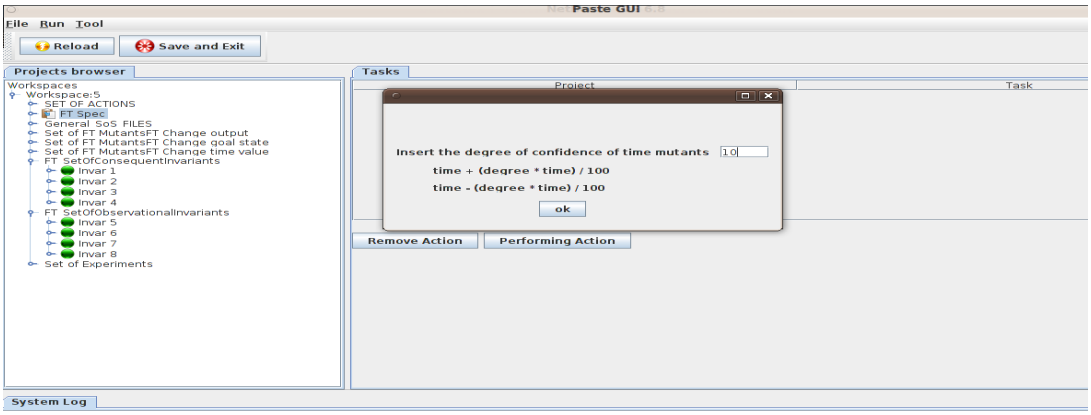


Figure 16. Correctness of invariants and generation of mutants for `SSadmin` in PASTE.

$$\mathit{Invar}_4 = ?/\texttt{option\_scr}/[5,35], \texttt{marks} \mapsto \{\texttt{marks\_scr}\}/[29.5,30.5] \rhd [39,70]$$

Next, some observational invariants are introduced. The next one expresses that whenever a user `connect` to `SSadmin`, if she observes the `option_scr` in a lapse of time belonging to $[20,80]$, she previously introduced the `login`. It could happen either that the login was correct and the `option_scr` was displayed or that the user introduced an erroneous login and she had to try another `login` before the `option_scr` was displayed.

$$\omega_{5a} = \langle \texttt{welcome\_scr}/[10,20], \texttt{login} \rangle$$

$$\omega_{5b} = \langle \texttt{welcome\_scr}/[10,20], \texttt{login}/\texttt{error\_user}/[25,35], \texttt{login} \rangle$$

$$\mathit{Invar}_5 = \texttt{connect} \rightarrow \{\omega_{5a}, \omega_{5b}\} \leftarrow \texttt{option\_scr}, [29.5,30.5]/[20,80]$$

Note that the correctness of this invariant depends on the time interval associated with the performance of the whole sequence that is, $[20,80]$. The next invariant represents some possible actions, in a time belonging to $[15,100]$, that a student can perform when she sends a message, that is, the sequence of actions observed between the introduction of `write_msg` by the student and the moment when the system shows the `msg_sent` screen. The student can either `write_msg` and `send` it or `write_msg` the message and `attachF` to this message before to `send` it.

$$\omega_{6a} = \langle \texttt{msg\_written}/[14.5,15.5], \texttt{send} \rangle$$

$$\omega_{6b} = \langle \texttt{msg\_written}/[14.5,15.5], \texttt{attachF}/\texttt{attached\_scr}[14.5,15.5], \texttt{send} \rangle$$

$$\mathit{Invar}_6 = \texttt{write\_msg} \rightarrow \{\omega_{6a}, \omega_{6b}\} \leftarrow \texttt{msg\_sent}, [14.5,15.5]/[15,100]$$

The observational invariant Invar_7 can be used to check the questionnaire option. If a student `sel_quest` to fulfill it and after a non empty sequence of actions in a time belonging to $[30,160]$ the `quest_sent` option appears, then the student has started the questionnaire.

$$\omega_7 = \langle \texttt{question\_1}/[14.5,15.5], \star/[20,35], \texttt{send\_quest} \rangle$$

$$\mathit{Invar}_7 = \texttt{sel\_quest} \rightarrow \{\omega_7\} \leftarrow \texttt{quest\_sent}, [19.5,20.5]/[30,160]$$

DOI: 10.1002/stvr

Computation time (in cycles of CPU) to check the correctness of the invariants with respect to the specification.

	$Invar_1$	$Invar_2$	$Invar_3$	$Invar_4$	$Invar_5$	$Invar_6$	$Invar_7$	$Invar_8$
Correctness Alg.	$0c.$	$10000c.$	$0c.$	$0c.$	$0c.$	$0c.$	$113000c.$	$114000c.$

Computation time (in cycles of CPU) to check the correctness of the traces with respect to each invariant.

$Invar/Length$	$100tokens$	$1000tokens$	$10000tokens$	$100000tokens$	$1000000tokens$
$Invar_1$	$600c.$	$4800c.$	$11600c.$	$92500c.$	$915500c.$
$Invar_2$	$700c.$	$1200c.$	$3200c.$	$16500c.$	$83100c.$
$Invar_3$	$700c.$	$1400c.$	$4700c.$	$19900c.$	$133500c.$
$Invar_4$	$400c.$	$1000c.$	$5900c.$	$26200c.$	$189600c.$
$Invar_5$	$0c.$	$100c.$	$2900c.$	$25600c.$	$218300c.$
$Invar_6$	$0c.$	$600c.$	$6500c.$	$29700c.$	$299300c.$
$Invar_7$	$0c.$	$700c.$	$2400c.$	$36400c.$	$359700c.$
$Invar_8$	$0c.$	$1500c.$	$8300c.$	$74900c.$	$744800c.$

Figure 17. Computation time for correctness algorithms in PASTE.

Instead of using the wildcard $\star$ character in $Invar_7$, it is possible to describe all the possible question/answer situations. The $Invar_8$ invariant, similar to the $Invar_7$ invariant, focuses on checking the questionnaire option. The main difference with respect to $Invar_7$ is that in $Invar_8$ all the possibilities are explicitly described, that is, the 16 different possibilities of answering `true/false` to each of the four questions are included. Even though $Invar_7$ and $Invar_8$ have similar behavior, the empirical results will show that their power of error detection is different.

$$\omega_{8a} = \begin{array}{c}\langle \texttt{question\_1}/[14.5,15.5], \texttt{sel\_true}/\texttt{question\_2}[7.5,8.5],\\ \texttt{sel\_true}/\texttt{question\_3}[7.5,8.5], \texttt{sel\_true}/\texttt{question\_4}[7.5,8.5],\\ \texttt{sel\_true}/\texttt{quest\_finished}[7.5,8.5], \texttt{send\_quest}\rangle\end{array}$$

$$\ldots$$

$$\omega_{8p} = \begin{array}{c}\langle \texttt{question\_1}/[14.5,15.5], \texttt{sel\_false}/\texttt{question\_2}[7.5,8.5],\\ \texttt{sel\_false}/\texttt{question\_3}[7.5,8.5], \texttt{sel\_false}/\texttt{question\_4}[7.5,8.5],\\ \texttt{sel\_false}/\texttt{quest\_finished}[7.5,8.5], \texttt{send\_quest}\rangle\end{array}$$

$$Invar_8 = \texttt{sel\_quest} \rightarrow \{\omega_{8a}, \ldots, \omega_{8p}\} \leftarrow \texttt{quest\_sent}, [19.5,20.5]/[30,160]$$

6.4. Estimation of power fault detection

Once a set of invariants is defined, it is necessary to check the correctness of this suite with respect to the specification. Benchmark results are presented in Figure 17. Note that the values presented in the figure represent the average of performing the experiments 100 times and the time value represent the number of clock ticks elapsed since the program was launched. In these experiments, there are exactly 1000000 clock cycles per second. As an additional remark, the time for computing the correctness of the observational invariants depends on the number of pattern traces that they have. In this case, $Invar_7$ and $Invar_8$ have a similar number and , therefore, similar performance is observed. Moreover, the time for checking the correctness of logs with respect to consequent invariants strongly depends on the appearance of the $wildcard$ $\star$, as in the case of $Invar_1$, while

Prepared using stvrauth.cls
DOI: 10.1002/stvr

# Mutants	$Invar_1$			$Invar_2$			$Invar_3$			$Invar_4$		
	#	-	%	#	-	%	#	-	%	#	-	%
CO	19	-	2.77%	36	-	5.26%	19	-	2.77%	19	-	2.77%
CGS	0	-	0%	0	-	0%	0	-	0%	0	-	0%
CT	2	-	2.77%	4	-	5.55%	2	-	2.77%	2	-	2.77%

# Mutants	$Invar_5$			$Invar_6$			$Invar_7$			$Invar_8$		
	#	-	%	#	-	%	#	-	%	#	-	%
CO	19	-	2.77%	38	-	5.55%	25	-	3.65%	169	-	24.7%
CGS	16	-	2.33%	9	-	1.31%	130	-	19%	136	-	19.88%
CT	2	-	2.77%	8	-	11.11%	4	-	5.55%	20	-	27.77%

Figure 18. Mutants killed by each of the invariants.

in the case of observational invariants, it depends on the length of the invariant, being $Invar_8$ the longest one.

Next, the results obtained from the application of the mutation methodology for estimating a measure of the effectiveness of the invariants detecting errors are reported. In order to do this, the different mutation operators were applied to the specification of `SSadmin`. First, the `CO` operator was applied to all the transitions, modifying the associated output with each output available in the specification. Next, the `CGS` operator was used, resulting in the modification of the final state of the transitions of the specification. An exhaustive set of mutants was generated. However, this is not possible when the `CT` operator is applied. In this case, the amount of possible mutants is astronomic. Thus, it is necessary to establish a finite number of changes for each fixed time value. The right hand side of Figure 16 presents the screen displayed when this functionality is selected by the user. Consider the transition (s, i, o, t, s'). If the `CT` mutant operator is applied to this transition with two deviations α and $-\alpha$. Two different mutants are obtained: one of them presents the mutated transition $(s, i, o, t + \alpha \cdot t, s')$ and the other one the transition $(s, i, o, t - \alpha \cdot t, s')$.

Once the generation of mutants has finished, a list of all mutants is compiled. The mutants are sorted by the mutation operator used to generate them. In PASTE, for each mutant, it is possible to access the mutated transition, the new transition generated for this mutant, and the logs collected from the mutant.

For each mutant a log consisting of 10.000 interactions was collected. The size of the logs was limited due to the following two considerations. First, it was observed that if the invariants did not find an error *soon* then they were not able to find errors *later*. Second, the full set of collected logs requires $1.8GB$, what can be considered a reasonable size.

Figure 18 presents the data corresponding to the logs obtained from the application of the `CO`, `CGS`, and `CT` operators. Regarding the `CO` mutation operator, timed consequent invariants and timed observational invariants do not present materially different behaviors concerning the number of mutants killed. There is an exception: $Invar_8$. This fact is due to the number of pattern traces considered in this invariant. Note that occurrences of the wildcard $\star$ in pattern traces reduces the number of killed mutants. This situation happens in the case of $Invar_6$ and $Invar_7$. With respect to the length of consequent invariants, it is concluded that the shorter the length of the invariant, the higher the number of killed mutants. Note that the length of the $Invar_1$ invariant is different from the length of invariants $Invar_3$ and $Invar_4$, but they detect the same number of mutants. The reason is that $Invar_1$ contains an occurrence of the wildcard $\star$ together with the time interval $[0, \infty]$ and this pair matches any possible non-empty sequence of actions.

Concerning the results obtained from the logs recorded from the mutants generated by applications of the mutation operator `CGS`, it is worth pointing out that timed consequent invariants are not able to kill any mutant. The reason is that the application of the `CGS` operator does not

DOI: 10.1002/stvr

	20	40	60	80	120	140	4000	5000	6000
$Invar_1$	8	8	9	10	12	14	21	21	21
$Invar_2$	35	37	37	37	40	40	40	40	40
$Invar_3$	5	6	6	8	9	9	21	21	21
$Invar_4$	9	12	15	19	19	20	21	21	21
$Invar_5$	37	37	37	37	37	37	37	37	37
$Invar_6$	23	33	39	43	45	47	55	55	55
$Invar_7$	37	50	57	64	75	80	158	159	159
$Invar_8$	94	145	185	210	230	240	320	325	325

Figure 19. Mutants killed by each invariant according to the length of the logs.

affect the functional behavior of the system, that is, the mutation only changes the final state of a transition, not the input/output/time that labels it. Regarding timed observational invariants, all of them are able to kill mutants: $Invar_8$ and $Invar_7$ kill almost the same number of mutants. It has been observed that the longer the pattern traces associated with the observational invariant are, the higher their capacity to detect errors in logs.

Next, the results obtained when considering the logs derived from the application of the mutation operator `CT` are analyzed. The proportion of killed mutants is less than the one obtained with the other mutation operators. As it was mentioned before, the number of `CT` mutants is less than the number of the other mutants, specifically, it is equal to $2 \cdot |\mathcal{T}|$. However, this relation changes when the percentage of killed mutants is considered. The results obtained from all the invariants with respect to the `CT` mutation operator are better than the ones obtained for the `CO` mutation operator. Furthermore, the percentage is, in general, a little better than that for the `CGS` mutation operator.

Finally, the table presented in Figure 19 shows the results obtained by taking into account the invariants and the length of the logs. It can be concluded that timed observational invariants need logs of at least 5.000 interactions to be effective, while timed consequent invariants produce some useful results already with logs of size 100.

6.5. Metrics and heuristics

This section introduces a metric to compare sets of invariants. The core of the method is a heuristic to obtain the set of the n *most representative* invariants out of a (possibly huge) set of invariants. Essentially, an invariant suite is better than another one if the former kills more mutants than the latter. Even though the underlying idea is very simple and it can be indeed used as a first approach to compare invariants suite, a direct implementation of this approach cannot be effectively realized since it would require to generate the powerset of the initial set of invariants, with the consequent exponential explosion. Therefore, it is necessary to look for *approximate* solutions that can be computed in polynomial time. The main idea of the heuristic is to implement a greedy algorithm that in each iteration includes in the partial solution the *best* invariant still available. Even though this approach does not guarantee that the *best* set of invariants will be returned, the experiments show that the obtained suites are close enough to the optimal solution. The heuristic consists of the following steps.

1. Generate from the specification m mutants. From each mutant a set of logs is collected.
2. Check the correctness of each log with respect to each invariant.
3. Compute the number of mutants killed by each invariant.
4. Obtain the invariant ψ that kills more mutants and add it to the solution.
5. Delete ψ from the invariant suite. Delete the mutants killed by ψ from the set of mutants.
6. Jump to step 3 until the solution has n elements.

This heuristic is a greedy solution. Note that invariants have to be sorted after each iteration since their goodness depend on their performance against the set of mutants. Also note that since this set is reduced after each iteration, the initial sorting of the invariants cannot be used along the

DOI: 10.1002/stvr

execution of the algorithm. The main consequence of sorting each interaction is that this algorithm works in the worst case in $O(\texttt{NConsq} \cdot \mathcal{C}_1 + \texttt{NObs} \cdot \mathcal{C}_2 + n \cdot (\texttt{NConsq} + \texttt{NObs}))$, where NConsq and NObs denote respectively the number of consequent and observational invariants in the suite. Additional $\mathcal{C}_1$ and $\mathcal{C}_2$ denote respectively the complexity of checking a consequent and observational invariant, that is $\mathcal{C}_1 = |\mathcal{S}| \cdot |\mathcal{T}|^{|\mathcal{S}|} \cdot k + |\mathcal{T}| \cdot (s - k)$, where $k = \texttt{Nstars}_{\mathcal{I}/\mathcal{O}}(\phi)$ and $s = \texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi)$, and $\mathcal{C}_2 = |\mathcal{T}|^{|\mathcal{S}|} \cdot |\mathcal{T}| \cdot |\beta|$. Finally, n is the number of invariants that will be selected.

Definition 19
Let $M = (\mathcal{S}, \mathcal{I}, \mathcal{O}, \mathcal{T}, s_0)$ be a TFSM. The function $\texttt{Best}_M : \wp(\Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}}) \times \wp(\texttt{MTrazes}_M) \longrightarrow \Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}}$ is such that given a set of invariants and a set of pairs (mutant,log) returns one of the invariants that kills more mutants. Given $M \in \texttt{SETTFSM}$, $K \subseteq (\Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}})$ and $LM \subseteq \texttt{MTrazes}_M$, this function is defined as:

$$\texttt{Best}_M(K, LM) = \psi \;\big|\; \psi \in K \wedge \forall \rho \in K, \rho \neq \psi : |\texttt{Skm}_M(\psi, LM)| \geq |\texttt{Skm}_M(\rho, LM)|$$

The function $\texttt{MRepr}_M : \mathbf{N} \times \wp(\Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}}) \times \wp(\texttt{MTrazes}_M) \longrightarrow \wp(\Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}})$ returns a set of *representative* invariants with respect to a set of pairs (mutant,log). For any $M \in \texttt{SETTFSM}$, $n \in \mathbf{N}$, $K \subseteq (\Psi_{\mathcal{I}/\mathcal{O}} \cup \Phi_{\mathcal{I}/\mathcal{O}})$ and $LM \subseteq \texttt{MTrazes}_M$, $\texttt{MRepr}_M(n, K, LM)$ is defined as:

$$\begin{cases} \emptyset & \text{if } n = 0 \\ \{\psi'\} & \text{if } n = 1 \\ \{\psi'\} \cup \texttt{MRepr}_M(n-1, K \setminus \{\psi'\}, \texttt{Remove}_M(\psi', LM)) & \text{if } n > 1 \end{cases}$$

where $\psi' = \texttt{Best}_M(K, LM)$.

□

Note that $\texttt{MRepr}_M$ is non-deterministic: if several invariants kill the same number of mutants it non-deterministically chooses one of them. Also note that the definition of $\texttt{MRepr}_M$ is consistent since once $\texttt{Best}_M$ selects one invariant, this invariant is used in all possible branches of the definition. The following abstract example informally shows how the heuristic works.

Example 9
Let M be a specification and $K = \{\psi_1, \ldots, \psi_8\}$ be a set of eight correct invariants with respect to M. Let LM be a set of (mutant, log) pairs produced from 14 mutants of M. These mutants are denoted by $M_1, \ldots, M_{14}$. The next table shows the relation between the set of invariants and the mutants that they kill. A token in the position $[i, j]$ of the matrix represents that the invariant ψ_i killed the mutant M_j, that is, found an error in a log generated by the mutant.

	M_1	M_2	M_3	M_4	M_5	M_6	M_7	M_8	M_9	M_{10}	M_{11}	M_{12}	M_{13}	M_{14}
ψ_1	•	•	•	•	•	•	•	•	•	•				
ψ_2	•	•	•			•	•	•		•	•			•
ψ_3				•	•			•	•			•	•	
ψ_4		•						•	•	•	•	•	•	
ψ_5					•			•		•	•	•		
ψ_6			•						•	•	•		•	
ψ_7			•						•	•		•	•	
ψ_8	•							•	•	•	•	•		

If the heuristic is applied to select the n most representative invariants, the obtained results are:

DOI: 10.1002/stvr

$$
\begin{aligned}
\texttt{MRepr}_M(1, K, LM) &= \{\psi_1\} \\
\texttt{MRepr}_M(2, K, LM) &= \{\psi_1, \psi_4\} \\
\texttt{MRepr}_M(3, K, LM) &= \{\psi_1, \psi_4, \psi_2\} \\
\texttt{MRepr}_M(4, K, LM) &= \{\psi_1, \psi_4, \psi_2, \psi_3\} \\
\texttt{MRepr}_M(5, K, LM) &= \{\psi_1, \psi_4, \psi_2, \psi_3, \psi_5\} \\
\texttt{MRepr}_M(6, K, LM) &= \{\psi_1, \psi_4, \psi_2, \psi_3, \psi_5, \psi_6\} \\
\texttt{MRepr}_M(7, K, LM) &= \{\psi_1, \psi_4, \psi_2, \psi_3, \psi_5, \psi_6, \psi_7\} \\
\texttt{MRepr}_M(8, K, LM) &= \{\psi_1, \psi_4, \psi_2, \psi_3, \psi_5, \psi_6, \psi_7, \psi_8\}
\end{aligned}
$$

Note that the solutions provided by the heuristic are good, but sometimes there exists a better option. For example, the *real* set of the two more representative invariants is not $\{\psi_1, \psi_4\}$ but $\{\psi_2, \psi_3\}$. □

The heuristic has been applied to the running system SSadmin presented in this paper. Consider the set of 8 invariants previously introduced in Subsection 6.3: $\mathit{INVAR} = \{\mathit{Invar}_1, \ldots, \mathit{Invar}_8\}$. A total of 1.440 mutants were generated from the SSadmin specification and a log of length 10.000 was produced for each mutant. Let $LM = ((e_1, M_1), \ldots, (e_{1.440}, M_{1.440}))$ be the set of considered (mutant, log) pairs. The heuristic provided the following results:

$$
\begin{aligned}
\texttt{MRepr}_{\texttt{SSadmin}}(1, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(2, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(3, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(4, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2, \mathit{Invar}_5\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(5, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2, \mathit{Invar}_5, \mathit{Invar}_4\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(6, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2, \mathit{Invar}_5, \mathit{Invar}_4, \mathit{Invar}_3\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(7, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2, \mathit{Invar}_5, \mathit{Invar}_4, \mathit{Invar}_3, \mathit{Invar}_1\} \\
\texttt{MRepr}_{\texttt{SSadmin}}(8, \mathit{INVAR}, LM) &= \{\mathit{Invar}_8, \mathit{Invar}_6, \mathit{Invar}_2, \mathit{Invar}_5, \mathit{Invar}_4, \mathit{Invar}_3, \mathit{Invar}_1, \mathit{Invar}_7\}
\end{aligned}
$$

The method shows that the best invariant is $Invar_8$. Note that according to Figure 18, the two invariants that kill more mutants are $Invar_7$ and $Invar_8$, but when $\texttt{MRepr}(2, \mathit{INVAR}, LM)$ is computed, the result does not correspond to this pair of invariants. This situation means that many of the mutants killed by $Invar_7$ are also killed by $Invar_8$. Thus, if only two invariants have to be selected to check the correctness of the system, using $Invar_7$ and $Invar_8$ is less effective than using $Invar_6$ and $Invar_8$.

7. CONCLUSIONS

This paper presents a revised, enhanced, and extended version of previous work on passive testing of timed systems [4, 5]. The formal model to represent systems is a timed extension of the classical finite state machines model. Timed invariants are used to find errors on logs extracted from the IUT. This paper introduced a novel type of invariant that allows testers to study interesting properties that could not be represented with the original notion. In addition to present the syntax of invariants, the paper also provides algorithms to check the correctness of invariants with respect to a specification and algorithms to check the correctness of the logs recorded from an IUT with respect to invariants. The soundness of the approach is shown by relating it to an implementation relation.

This paper also reports on the PASTE tool, a tool that can be used to put in practice the theoretical results. In particular, this tool implements all the algorithms presented in this paper. The main task of the tool is to automate the process of checking the correctness of invariants with respect to a specification and determining whether logs extracted from an IUT are correct with respect to a given invariant.

Taking as initial step previous ideas [3], the paper provides an approach that uses mutation testing techniques as a way to classify invariants according to their power to find errors. In this methodology, specifications are mutated by using three mutation operators. Then, different logs, that

DOI: 10.1002/stvr

simulate real faults, are extracted from the mutants. These logs are used to evaluate the capabilities of the invariants proposed by the tester to find errors and estimate their effectiveness. The reported experiments show that timed observational invariants are able to detect more errors than timed consequent invariants. However, there are some errors that can be found only by timed consequent invariants. Therefore, the best approach is to use a set of invariants that keeps an appropriate balance between consequent and observational invariants.

Even though this paper presents a complete framework, so that there is not much room for extensions, it is possible to continue the research on formal passive testing of timed systems. Specifically, a first line of work is to evaluate invariants in a completely different environment. There is initial work on the application of the techniques presented in this paper to detect *malicious* patterns that can indicate threats to the integrity of the system. Since PASTE is a very light-weight tool, so that it does not overload the system where it is running, it would be interesting to use it as a *personal* anti-virus since the user can specify the patterns that he is looking for.

Acknowledgements

The authors would like to thank the anonymous reviewers of this paper for the careful reading and the useful suggestions to improve the quality of the paper.

References

1. B. Alcalde, A.R. Cavalli, D. Chen, D. Khuu, and D. Lee. Network protocol system passive testing for fault management: A backward checking approach. In *FORTE 2004, LNCS 3235*, pages 150–166. Springer, 2004.
2. P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
3. C. Andrés, M.G. Merayo, and C. Molinero. Advantages of mutation in passive testing: An empirical study. In *4th Workshop on Mutation Analysis, Mutation'09*, pages 230–239. IEEE Computer Society Press, 2009.
4. 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.
5. C. Andrés, M.G. Merayo, and M. Núñez. Formal correctness of a passive testing approach for timed systems. In *5th Workshop on Advances in Model Based Testing, A-MOST'09*, pages 67–76. IEEE Computer Society Press, 2009.
6. 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*, pages 71–80. IEEE Computer Society Press, 2009.
7. 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.
8. C. Artho, H. Barringer, A. Goldberg, K. Havelund, S. Khurshid, M. R. Lowry, C. S. Pasareanu, G. Rosu, K. Sen, W. Visser, and R. Washington. Combining test case generation and runtime verification. *Theoretical Computer Science*, 336(2-3):209–234, 2005.
9. C. Artho, K. Havelund, and A. Biere. High-level data races. *Software Testing, Verification and Reliability*, 13(4):207–227, 2003.
10. 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.
11. H. Barringer, A. Goldberg, K. Havelund, and K. Sen. Rule-based runtime verification. In *5th Verification, Model Checking, and Abstract Interpretation, VMCAI'05*, volume 2937 of *Lecture Notes in Computer Science*, pages 277–306, 2004.
12. 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.
13. A. Benharref, R. Dssouli, M.A. Serhani, A. En-Nouaary, and R. Glitho. New approach for EFSM-based passive testing of web services. In *Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581*, pages 13–27. Springer, 2007.
14. R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Communications of the ACM*, 20:762–772, 1977.
15. G.P. Brat, D. Drusinsky, D. Giannakopoulou, A. Goldberg, K. Havelund, M.R. Lowry, C.S. Pasareanu, A. Venet, W. Visser, and R. Washington. Experimental evaluation of verification and validation tools on martian rover software. *Formal Methods in System Design*, 25(2-3):167–198, 2004.
16. A. Cavalli, C. Gervy, and S. Prokopenko. New approaches for passive testing using an extended finite state machine specification. *Information and Software Technology*, 45(12):837–852, 2003.
17. 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.
18. S. Colin and L. Mariani. Run-time verification. In M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, editors, *Model-Based Testing of Reactive Systems, LNCS 3472*, chapter 19, pages 557–603. Springer, 2005.

DOI: 10.1002/stvr

19. D. Drusinsky. The temporal rover and the atg rover. In *7th Int. SPIN Workshop on Model Checking of Software, SPIN'00, LNCS 1885*, pages 323–330. Springer, 2000.
20. D. Drusinsky. *Modeling and Verification Using UML Statecharts: A Working Guide to Reactive System Design, Runtime Monitoring and Execution-based Model Checking*. Newnes, 2006.
21. 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.
22. 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.
23. K. Havelund and G. Roşu. An overview of the runtime verification tool Java PathExplorer. *Formal Methods in System Design*, 24(2):189–215, 2004.
24. 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.
25. 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.
26. R.M. Hierons, J.P. Bowen, and M. Harman, editors. *Formal Methods and Testing, LNCS 4949*. Springer, 2008.
27. R.M. Hierons, M.G. Merayo, and M. Núñez. Testing from a stochastic timed system with a fault model. *Journal of Logic and Algebraic Programming*, 78(2):98–115, 2009.
28. 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.
29. J. Jacky, M. Veanes, C. Campbell, and W. Schulte. *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.
30. M. Kim, S. Kannan, I. Lee, O. Sokolsky, and M. Viswanathan. Computational analysis of Run-time monitoring - fundamentals of Java-MaC. *Electronic Notes in Theoretical Computer Science*, 70(4), 2002.
31. M. Kim, M. Viswanathan, S. Kannan, I. Lee, and O. Sokolsky. Java-MaC: A run-time assurance approach for Java programs. *Formal Methods in System Design*, 24(2):129–155, 2004.
32. D.E. Knuth, J.H. Morris, and V.R. Pratt. Fast pattern matching in strings. *SIAM Journal on Computing*, 6(1):323–350, 1977.
33. R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4):255–299, 1990.
34. K.J. Kristoffersen, C. Pedersen, and H.R. Andersen. Runtime verification of timed LTL using disjunctive normalized equation systems. *Electronics Notes Theoretical Computer Science*, 89(2), 2003.
35. D. Lee, D. Chen, R. Hao, R. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In *10th IEEE Int. Conf. on Network Protocols, ICNP'02*, pages 122–131. IEEE Computer Society Press, 2002.
36. D. Lee, D. Chen, R. Hao, R.E. Miller, J. Wu, and X. Yin. Network protocol system monitoring: a formal approach with passive testing. *IEEE/ACM Transactions on Networking*, 14:424–437, 2006.
37. 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.
38. D. Lee, K.K. Sabnani, D.M. Kristol, and S. Paul. Conformance testing of protocols specified as communicating finite state machines - a guided random walk based approach. *IEEE Transactions on Communications*, 44(5):631–640, 1996.
39. D. Lee, K.K. Sabnani, D.M. Kristol, S. Paul, and M.Ü. Uyar. Conformance testing of protocols specified as communicating FSMs. In *12th Annual Joint Conf. of the IEEE Computer and Communications Societies. Networking: Foundation for the Future, INFOCOM'93*, pages 115–127. IEEE Computer Society Press, 1993.
40. D. Lee and M. Yannakakis. Principles and methods of testing finite state machines: A survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.
41. S. Lee and K. G. Shin. Probabilistic diagnosis of multiprocessor systems. *ACM Computer Surveys*, 26(1):121–139, 1994.
42. M. Leucker and C. Schallhart. A brief account of runtime verification. *Journal of Logic and Algebraic Programming*, 78(5):293–303, 2009.
43. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *2nd Int. Workshop on Tools and Algorithms for Construction and Analysis of Systems, TACAS'96, LNCS 1055*, pages 147–166. Springer, 1996.
44. 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.
45. 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.
46. 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.
47. M.G. Merayo, M. Núñez, and I. Rodríguez. Formal testing from timed finite state machines. *Computer Networks*, 52(2):432–460, 2008.
48. 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.
49. R.E. Miller and K.A. Arisha. On fault location in networks by passive testing. In *19th IEEE Int. Performance, Computing, and Communications Conference, IPCCC'00*, pages 281–287. IEEE Computer Society Press, 2000.
50. R.E. Miller and K.A. Arisha. Fault identification in networks by passive testing. In *34th Simulation Symposium, SS'01*, pages 277–284. IEEE Computer Society Press, 2001.

DOI: 10.1002/stvr

51. G.J. Myers. *The Art of Software Testing*. John Wiley and Sons, 2nd edition, 2004.
52. R. M. Needham and M. D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.
53. 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.
54. G. Noubir, K. Vijayananda, and H. J. Nussbaumer. Signature-based method for run-time fault detection in communication protocols. *Computer Communications*, 21(5):405–421, 1998.
55. 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.
56. A. Pnueli. The Temporal Logic of Programs. In *18th Annual Symposium on Foundations of Computer Science, SFCS'77*, pages 46–57. IEEE Computer Society Press, 1977.
57. O. Raz, P. Koopman, and M. Shaw. Semantic anomaly detection in online data sources. In *24th IEEE Int. Conf. on Software Engineering, ICSE'02*, pages 302–312. ACM Press, 2002.
58. D.J. Richardson, S.L. Aha, and T.O. O'Malley. Specification-based test oracles for reactive systems. In *14th Int. Conf. on Software Engineering, ICSE'92*, pages 105–118, 1992.
59. I. Rodríguez. A general testability theory. In *20th Int. Conf. on Concurrency Theory, CONCUR'09, LNCS 5710*, pages 572–586. Springer, 2009.
60. I. Rodríguez, M.G. Merayo, and M. Núñez. $\mathcal{HOTL}$: Hypotheses and observations testing logic. *Journal of Logic and Algebraic Programming*, 74(2):57–93, 2008.
61. U. Sammapun, R. Sharykin, M. DeLap, M. Kim, and S. Zdancewic. Formalizing Java-MaC. *Theoretical Computer Science*, 89(2):171–190, 2003.
62. S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T.E. Anderson. Eraser: A dynamic data race detector for multithreaded programs. *ACM Transactions on Computer Systems*, 14(4):391–411, 1997.
63. K. Schneider. *Verification of Reactive Systems: Formal Methods and Algorithms*. Springer, 2004.
64. G. Shu and D. Lee. Message confidentiality testing of security protocols - passive monitoring and active checking. In *18th Int. Conf. on Testing Communicating Systems, TestCom'06, LNCS 3964*, pages 357–372. Springer, 2006.
65. 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.
66. 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.
67. M. Tabourier and A. Cavalli. Passive testing and application to the GSM-MAP protocol. *Information and Software Technology*, 41(11-12):813–821, 1999.
68. H. Ural and Z. Xu. An EFSM-based passive fault detection approach. In *Joint 19th IFIP TC6/WG6.1 Int. Conf. on Testing of Software and Communicating Systems, TestCom'07, and 7th Int. Workshop on Formal Approaches to Software Testing, FATES'07, LNCS 4581*, pages 335–350. Springer, 2007.
69. H. Ural, Z. Xu, and F. Zhang. An improved approach to passive testing of FSM-based systems. In *2nd Int. Workshop on Automation of Software Test, AST'07*, page 6. IEEE Computer Society Press, 2007.
70. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan-Kaufmann, 2007.
71. M.Ü. Uyar, S.S. Batth, Y. Wang, and M.A. Fecko. Algorithms for modeling a class of single timing faults in communication protocols. *IEEE Transactions on Computers*, 57(2):274–288, 2008.
72. C. Wang and M. Schwartz. Fault detection with multiple observers. *IEEE/ACM Transactions on Networking*, 1(1):48–55, 1993.
73. Y. Wang, M.Ü. Uyar, S.S. Batth, and M.A. Fecko. Fault masking by multiple timing faults in timed EFSM models. *Computer Networks*, 53(5):596–612, 2009.
74. W. Wei, K. Suh, B. Wang, Y. Gu, J. Kurose, and D. Towsley. Passive online rogue access point detection using sequential hypothesis testing with TCP ACK-pairs. In *7th ACM SIGCOMM Internet Measurement Conference, IMC '07*, pages 365–378. ACM Press, 2007.
75. J. Wu, Y. Zhao, and X. Yin. From active to passive: Progress in testing of internet routing protocols. In *21st IFIP WG 6.1 Int. Conf. on Formal Techniques for Networked and Distributed Systems, FORTE'01*, pages 101–116. Kluwer Academic Publishers, 2001.
76. S.W. Yeh, C. Wu, H.D. Sheng, C.K. Hung, and R.C. Lee. Expert system based automatic network fault management system. In *13th Annual Int. Computer Software and Applications Conference, COMPSAC'89*, pages 767–774. IEEE Computer Society Press, 1989.
77. Y. Zhao, X. Yin, and J. Wu. Problems in the information dissemination of the internet routing. *Journal of Computer Science and Technology*, 18(2):139–152, 2003.

APPENDIX: TRANSLATION OF INVARIANTS TO AN ALTERNATIVE SEMANTIC MODEL

This section provides a method to transform timed consequent invariants into a specific class of Extended Finite State Machines (in short, EFSM). This formalism is an extension of the Finite State Machine formalism, where variables and conditions over these variables are introduced. The machines used in this paper need only two variables. These variables deal, respectively, with the time associated with occurrences of the $\star$ wildcard character and with the total time invested by the subsequences of the processed log. The only difference with respect to the original EFSM formalism is the inclusion of a specific *error state*. This translation provides a *formal semantics* of consequent invariants by translating them into an EFSM. Therefore, it helps to understand the meaning of these

Prepared using stvrauth.cls
DOI: 10.1002/stvr

invariants since it relies on a well-know formalism. The concepts related to invariants and traces/logs (e.g. correctness of a log with respect to a consequent invariant) are also adapted to this translation. The translation for observational invariants is not included because it follows a similar pattern: it essentially consists in repeating the translation process for each pattern trace.

Definition 20
An *Extended Finite State Machine* is a tuple $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in}, \bar{x}, s_e)$ 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*, s_{in} is the *initial state*, $\bar{x} \in \mathbf{R}_+^2$ is the pair of initial values of the *variables*, and s_e is the *error* state. The set of all EFSM will be denoted by SETEFSM.

A *transition* is a tuple (s, i, o, Q, Z, s') where $s, s' \in S$ are the initial and final state of the transition, $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output actions associated with the transition, $Q : \mathbf{R}_+^2 \longrightarrow \texttt{Bool}$ is a predicate on the set of variables and $Z : \mathbf{R}_+^2 \longrightarrow \mathbf{R}_+^2$ is a transformation over the variables.

A *configuration* in M is a pair $(s, \bar{x}_0)$, where $s \in S$ is the current state and $\bar{x}_0 \in \mathbf{R}_+^2$ is the tuple containing the current values of the variables. The initial configuration of M is $(s_{in}, \bar{x})$. □

Given a configuration $(s, \bar{x}_0)$, a transition (s, i, o, Q, Z, s') denotes that if the input i is received and $Q(\bar{x}_0)$ holds then the output o will be produced and the new configuration will be $(s', Z(\bar{x}_0))$.

As it was previously said, these machines have only two variables. During the rest of the appendix the variables considered in the machine will be denoted by x_1 and x_2. Next, the method for transforming a timed consequent invariant into an EFSM is given. Given an invariant ϕ, M_ϕ denotes the EFSM associated with this invariant.

Definition 21
The function $\texttt{Construct} : \texttt{SETEFSM} \times S \times \Phi_{\mathcal{I}/\mathcal{O}} \longrightarrow \texttt{SETEFSM}$ is such that for all $M = (S, \mathcal{I}, \mathcal{O}, Tr, s_{in}, \bar{x}) \in \texttt{SETEFSM}$, $s \in S$, and $\phi = \alpha_1, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f \in \Phi_{\mathcal{I}/\mathcal{O}}$:
If $(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = 0)$ **then return** $M_\phi = (S, \mathcal{I}, \mathcal{O}, Tr_\phi, s_{in}, \bar{x}, s_e)$ where

- $Tr_\phi = Tr \cup \bigcup_{i=1}^{5} Tr_i$ with
  $Tr_1 = \{(s, i_f, o, x_2 + t \in \hat{q}_f \wedge t \in \hat{p}_f, (0,0), s_{in}) | o \in O\}$
  $Tr_2 = \{(s, i_f, o, x_2 + t \notin \hat{q}_f \vee t \notin \hat{p}_f, \bar{x}, s_e) | o \in O\}$
  $Tr_3 = \{(s, i_f, o, \texttt{true}, \bar{x}, s_e) | o \notin O\}$
  $Tr_4 = \{(s, i, o, \texttt{true}, (0,0), s_{in}) | i \in \mathcal{I} \wedge i \neq i_f \wedge o \in \mathcal{O}\}$
  $Tr_5 = \{(s_e, i, o, \texttt{true}, \bar{x}, s_e | i \in \mathcal{I} \wedge o \in \mathcal{O}\}$

If $(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) = 1 \wedge \alpha_1 = \star/[p_1^\star, p_2^\star]$) **then return** $M_\phi = (S, \mathcal{I}, \mathcal{O}, Tr_\phi, s_{in}, \bar{x}, s_e)$ where

- $Tr_\phi = Tr \cup \bigcup_{i=1}^{7} Tr_i$ with
  $Tr_1 = \{(s, i, o, x_1 + t \leq p_2^\star, (x_1 := x_1 + t, x_2 := x_2 + t), s) | i \in \mathcal{I} \wedge i \neq i_f \wedge o \in \mathcal{O}\}$
  $Tr_2 = \{(s, i, o, x_1 + t > p_2^\star, (0,0), s_{in}) | i \in \mathcal{I} \wedge i \neq i_f \wedge o \in \mathcal{O}\}$
  $Tr_3 = \{(s, i_f, o, x_1 \geq p_1^\star \wedge x_2 + t \in \hat{q}_f \wedge t \in \hat{p}_f, (0,0), s_{in}) | o \in O\}$
  $Tr_4 = \{(s, i_f, o, x_1 \geq p_1^\star \wedge (x_2 + t \notin \hat{q}_f \vee t \notin \hat{p}_f), \bar{x}, s_e) | o \in O\}$
  $Tr_5 = \{(s, i_f, o, x_1 \geq p_1^\star, \bar{x}, s_e) | o \notin O\}$
  $Tr_6 = \{(s, i_f, o, x_1 < p_1^\star, (0,0), s_{in}) | o \in \mathcal{O}\}$
  $Tr_7 = \{(s_e, i, o, \texttt{true}, \bar{x}, s_e) | i \in \mathcal{I} \wedge o \in \mathcal{O}\}$

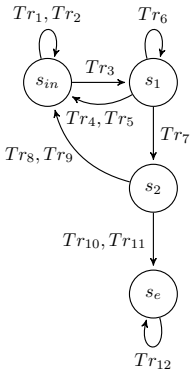
If $(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) > 0 \wedge \alpha_1 = a/z/\hat{p})$ **then return** $\texttt{Construct}(M_\phi, s_\alpha, (\alpha_2, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f))$ where $M_\phi = (S_\alpha, \mathcal{I}, \mathcal{O}, Tr_\phi, s_{in}, \bar{x}, s_e)$ with

- $S_\alpha = S \cup \{s_\alpha\}$ such that $s_\alpha \notin S$ is a fresh state
- $Tr_\phi = Tr \cup \bigcup_{i=1}^{3} Tr_i$ with
  $Tr_1 = \{(s, i, o, t \in \hat{p}, x_2 := x_2 + t, s_\alpha) | i \in \mathcal{I} \wedge a = i \wedge o \in \mathcal{O} \wedge z = o\}$
  $Tr_2 = \{(s, i, o, t \notin \hat{p}, (0,0), s_{in}) | i \in \mathcal{I} \wedge a = i \wedge o \in \mathcal{O} \wedge z = o\}$
  $Tr_3 = \{(s, i, o, \texttt{true}, (0,0), s_{in}) | i \in \mathcal{I} \wedge o \in \mathcal{O} \wedge (a \neq i \vee z \neq o)\}$

If $(\texttt{Len}_{\mathcal{I}/\mathcal{O}}(\phi) \geq 2 \wedge \alpha_1 = \star/[p_1, p_2] \wedge \alpha_2 = a/z/\hat{p})$ **then return** $\texttt{Construct}(M_\phi, s_\alpha, (\alpha_3, \ldots, \alpha_n, i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f))$ where $M_\phi = (S_\alpha, \mathcal{I}, \mathcal{O}, Tr_\phi, s_{in}, \bar{x}, s_e)$ with

DOI: 10.1002/stvr

$$\phi = i_1/?/[3,7], \star[20,79], i_1/o_2/[5,4], i_3 \mapsto \{o_7, o_8\} \rhd [3,14], [20,88]$$



$$\begin{array}{lcl}
Tr_1 & = & \{(s_{in}, i, o, \texttt{true}, (0,0), s_{in}) | i \in \mathcal{I} \setminus \{i_1\} \wedge o \in \mathcal{O}\} \\
Tr_2 & = & \{(s_{in}, i_1, o, t \notin [3,7], (0,0), s_{in}) | o \in \mathcal{O}\} \\
Tr_3 & = & \{(s_{in}, i_1, o, t \in [3,7], x_2 := x_2 + t, s_1) | o \in \mathcal{O}\} \\
Tr_4 & = & \{(s_1, i_1, o, x_1 < 20 \vee t \notin [5,4], (0,0), s_{in}) | o \in \mathcal{O}\} \\
Tr_5 & = & \{(s_1, i, o, x_1 + t > 79, (0,0), s_{in}) | i \in \mathcal{I} \setminus \{i_1\} \wedge o \in \mathcal{O}\} \\
Tr_6 & = & \{(s_1, i, o, x_1 + t \leq 79, x_1 := x_1 + t \wedge x_2 := x_2 + t, s_1) | i \in \mathcal{I} \setminus \{i_1\} \wedge o \in \mathcal{O}\} \\
Tr_7 & = & \{(s_1, i_1/o_2, t \in [5,4] \wedge x_1 \geq 20, x_1 := 0 \wedge x_2 := x_2 + t, s_2)\} \\
Tr_8 & = & \{(s_2, i, o, \texttt{true}, (0,0), s_{in}) | i \in \mathcal{I} \setminus \{i_3\} \wedge o \in \mathcal{O}\} \\
Tr_9 & = & \{(s_2, i_3, o, (x_2 + t) \in [20,88] \wedge t \in [3,14], (0,0), s_{in}) | o \in \{o_7, o_8\}\} \\
Tr_{10} & = & \{(s_2, i_3, o, \texttt{true}, \bar{x}, s_e) | o \in \mathcal{O} \setminus \{o_7, o_8\}\} \\
Tr_{11} & = & \{(s_2, i_3, o, x_2 + t \notin [20,88] \vee t \notin [3,14], \bar{x}, s_e) | o \in \mathcal{O}\} \\
Tr_{12} & = & \{(s_e, i, o, \texttt{true}, \bar{x}, s_e) | i \in \mathcal{I} \wedge o \in \mathcal{O}\}
\end{array}$$

Figure 20. Example of Extended Finite State Machine associated to an invariant.

- $S_\alpha = S \cup \{s_\alpha\}$ such that $s_\alpha \notin S$
- $Tr_\phi = Tr \cup \bigcup_{i=1}^{4} Tr_i$ with

$$\begin{array}{lcl}
Tr_1 & = & \{(s, i, o, x_1 + t \leq p_2, (x_1 := x_1 + t, x_2 := x_2 + t), s) | i \in \mathcal{I} \wedge i \neq a \wedge o \in \mathcal{O}\} \\
Tr_2 & = & \{(s, i, o, x_1 + t > p_2, (0,0), s_{in}) | i \in \mathcal{I} \wedge i \neq a \wedge o \in \mathcal{O}\} \\
Tr_3 & = & \{(s, i, o, \texttt{true}, (0,0), s_{in}) | i \in \mathcal{I} \wedge i = a \wedge o \in \mathcal{O}\} \\
Tr_4 & = & \{(s, i, o, x_1 \geq p_1 \wedge t \in \hat{p}, (x_1 := 0, x_2 := x_2 + t), s_\alpha) | i \in \mathcal{I} \wedge a = i \wedge o \in \mathcal{O} \wedge z = o\}
\end{array}$$

□

An informal explanation of the `Construct` function follows. Intuitively, the *spine* of the returned machine represents the trace reflected in the invariant. The rest of the transitions correspond to the set of *alternative* behaviors. The invariant is traversed and, for each of its components, transitions that reflect the expected and unexpected behaviors expressed in the invariant are generated. The initial call for the transformation of an invariant ϕ is $\texttt{Construct}(M_\phi, s_{in}, \phi)$, where $M_\phi = (\{s_{in}, s_e\}, \mathcal{I}, \mathcal{O}, \emptyset, s_{in}, (0,0), s_e)$. This machine initially has only two states, that is, the initial and the error states, while the set of transitions is empty. The process finishes when the last component of the invariant, that is, $i_f \mapsto O/\hat{p}_f \rhd \hat{q}_f$, is reached and processed. Depending on the component of the invariant that is being processed, the function generates a different set of transitions. The first and second options present the set of transitions produced for the last component of the invariant. Therefore, they do not include a recursive call. In addition to the set of transitions produced to deal with the behavior reflected in the invariant, a set of transitions for managing the possible errors are included. If the component corresponds to an expression $a/z/\hat{p}$, then a set of transitions is produced for dealing with the conditions related to the expected/unexpected input/output actions and the temporal restriction associated with them. If the component corresponds to a wildcard $\star$, then the function generates transitions that *control* the time associated with the sequences of input/output actions that do not contain the input of the next component of the invariant. Note that the input actions that can be matched with the wildcard $\star$ must be different from the input of the next component in the invariant. In this case, the function transforms not only the wildcard $\star$ component but also the next one.

The concepts related to correctness of consequent invariants with respect to traces/logs can be easily adapted to this translation. Given a specification M and a timed consequent invariant M_ϕ, the invariant ϕ is correct with respect to the specification M if the following two conditions hold:

- There does not exist a trace in $\texttt{Traces}(M)$ that when applied to M_ϕ reaches the error state and

DOI: 10.1002/stvr

- there is a transition labeled by $i_f/o_1, x_2 + t \in \hat{q}_f \wedge t \in \hat{p}_f, (0,0)$ with $o_1 \in O$ that is traversed at least once.

Note that this transition is triggered only when the last component of the invariant is found in the trace and the conditions represented in it are fulfilled. Intuitively, this means that there exists a trace that contains the pattern expressed in the invariant. Finally, a log is correct with respect to a timed consequent invariant M_ϕ, if when the log is applied to M_ϕ the last state reached is not the error state. Figure 20 shows an example of this translation.

DOI: 10.1002/stvr