Supporting the extraction of timed properties for passive testing by using probabilistic user models

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

Abstract

Testing is one of the most widely used techniques to increase the quality and reliability of complex software systems. In this paper we extend our previous work on passive testing with invariants to incorporate (probabilistic) knowledge obtained from users of the system under test. In order to apply our technique, we need to obtain a set of invariants compiling the relevant properties of the system under test, and this is a time-intensive task. We present a novel approach to extract invariants from a specification, based on the idea that an invariant is better than another one if it can be checked more times in a given log. We present a formal approach where probabilistic user models are incorporated.

***Keywords*-Passive testing; Formal methods; Probabilistic formal models;**

I. Introduction

Techniques and methodologies to increase the quality of software are becoming increasingly important due to the growing complexity of current software systems. Among these techniques, testing [1], [2] is the most widely used in industrial environments. Unfortunately, testing is still a mainly manual activity, prone to errors that can mask real errors of the tested system. Traditionally, formal methods and testing have been seen as rivals. Thus, there was very little interaction between the two communities. Even though first steps to integrate both fields were given in the decade of 1990s (see, for example, [3]–[5]), in recent years, there has been a new emphasis on seeing these approaches as complementary [6]–[10]. Moreover, formal testing is not only dealing with *functional* properties but it is increasingly taking into account characteristics such as time, probabilities, distributed interfaces, etc (see, for example, [11]–[16] among very recent work on *non-classical* applications of formal testing).

Testing is usually based on the activities of a tester that stimulates the system under test (SUT) and checks the correction of the answers provided by SUT against a certain oracle (if we are working within a formal approach, the oracle is usually provided by the specification itself). However, in some situations this activity becomes difficult and even impossible to perform. For example, this is the case if the tester is not provided with a direct interface to interact with the SUT. Another conflictive scenario appears when the SUT is built from components that are running in their environment and cannot be shutdown or interrupted for a long period of time. In these situations, there is a particular interest in using *passive testing*. The main difference between active and passive testing is that in active testing testers can interact, by providing inputs, with the SUT and observe the obtained result while, in passive testing, testers analyze the behavior of the system, without interacting with it, and try to detect anomalous behaviors. Therefore, passive testing usually consists in recording the trace produced by the SUT and trying to find a *fault* by comparing this trace with the specification [17]–[21]. A new methodology to perform passive testing was presented in [22], [23]. The main novelty is that a set of *invariants* is used to represent the most relevant expected properties of the specification. An invariant can be seen as a restriction over the traces allowed to the SUT. We have recently extended this work to consider the possibility of adding time constraints as properties that traces extracted from the SUT must hold [24], [25]. It is worth to point out that our passive testing approach is somehow related to *runtime verification* [26] since they share similar objectives and

Research supported by the Spanish MEC project WEST/FAST (TIN2006-15578-C02-01) and by the UCM-BSCH programme to fund research groups (GR58/08 - group number 910606).

procedures.

We can consider different methods to obtain a set of invariants. The first one is that testers propose a set of representative invariants. Then, the correctness of these invariants, with respect to the specification has to be checked. The main drawback of this approach is that we still have to rely on the manual work of the tester. If we do not have a complete formal specification, another option is to assume that the set of invariants provided by the testers are correct by definition. In this paper we consider an alternative approach: We automatically derive invariants from the specification. First, we consider an adaptation of the algorithm presented in [22]. The problem with this first attempt is that the number of invariants that we extract from the specification is huge and we do not have a criteria to decide which are *the best ones*. In this paper, we propose an alternative algorithm to use knowledge extracted from *standard* users of the system, so that invariants can be sorted according to their *quality* to fit the behavior of these users.

In our approach we consider that we have some information that does not concern properties of the SUT itself. Following [27] we will work with a probabilistic model of the *typical user* that will interact with the SUT. This model defines the probability that the external environment (e.g., a human user, another system, an interface, etc) takes each available choice at each time. Intuitively, if A is the finite set of *ways to interact* with the SUT and, according to the user model, the probability that the user interacts with the SUT by following any behavior belonging to A is p, then if we test all behaviors given in A and the SUT passes all of them, then the probability that the user interacts with the SUT and finds an error is at most $1-p$ (we say *at most* because non tested behaviors could be correct indeed). That is, after applying a finite set of invariants that reflects these behaviors (out of an infinite set of required tests) the proportion of the system that has been checked is, in *probabilistic terms*, greater than 0. The problem that remains is that we do not have a sensible method to construct a user model (in [27] we assumed that this was an *input* of the development process). In order to construct probabilistic user models, in this paper we take advantage of methods inherited from data mining to adequately process information concerning users of the SUT.

Summarizing, in this paper we introduce a formal framework for testing timed systems following a passive testing methodology. The timed properties to be checked are given by mean of invariants that represent sequences of actions with additional time information about when these actions are performed. In addition, we present a methodology, based on data mining, to generate user models to extract the most useful invariants from the specification.

The rest of the paper is structured as follows. In Section II we review our passive testing framework. Next, in Section III, we present the proposed data mining process which will be used in the generation of user models. In Section IV we present the algorithm to extract invariants from the specification based on the information of a user model. Finally, in Section V, we present the conclusions and some lines for future work.

II. Timed Framework: Systems and Invariants

In this section we review our framework [24], [25]. We adapt the well known formalism of Finite State Machines (FSMs) to model our specifications.[1] The main difference with respect to usual FSMs consists in the addition of *time* to indicate the lapse between offering an input and receiving an output. We use positive real numbers as time domain for representing the time units that the output actions take to be executed.

Definition 1: A *Timed Finite State Machine (TFSM)* is a tuple $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$, where $\mathcal{S}$ is a finite set of states, $s_0 \in S$ is the initial state, $\mathcal{I}$ and $\mathcal{O}$, with $\mathcal{I} \cap \mathcal{O} = \varnothing$, are the finite sets of *input* and *output* actions, respectively, and $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{O} \times \mathbf{R}_+ \times \mathcal{S}$ is the set of transitions. Along this paper we will use $s \xrightarrow{i/o}_t s'$ as a shorthand to represent the transition $(s, i, o, t, s') \in \mathcal{T}$.

We say that M is *deterministic* if for all state s and input i there exists at most one transition labeled by i departing from s. We say that M is *input-enabled* if for all state $s \in \mathcal{S}$ and $i \in \mathcal{I}$ there exists a transition $s \xrightarrow{i/o}_t s'$. $\square$

A transition belonging to $\mathcal{T}$ is a tuple (s, i, o, t, s') where $s, s' \in \mathcal{S}$ are the initial and final states of the transition, $i \in \mathcal{I}$ and $o \in \mathcal{O}$ are the input and output actions, respectively, and $t \in \mathbf{R}_+$ denotes the time that the transition needs to be completed. In this paper we consider that all defined TFSMs are input-enabled and deterministic.

We will use a simple running example to illustrate most of the theoretical concepts introduced in this paper.

Example 1: We adapt the model of a basic ATM presented by Matthias Stallmann (see http://people.engr.ncsu.edu/efg/210/s99/Notes/fsm/) to our formalism. In this system, each transition is labeled with an input coming from the user (such as insert_card) or from a central database (such as PIN_OK versus bad_PIN) and the result observed in

[1]We have chosen a timed extension of FSMs, instead of other models such as timed automata [28], because we wanted to reuse previous work on passive testing which is mainly performed in the context of a FSM model. However, the methodology can be easily adapted to cope with other models that represent timed systems as long as they use a notion of action transition to represent the change of state.

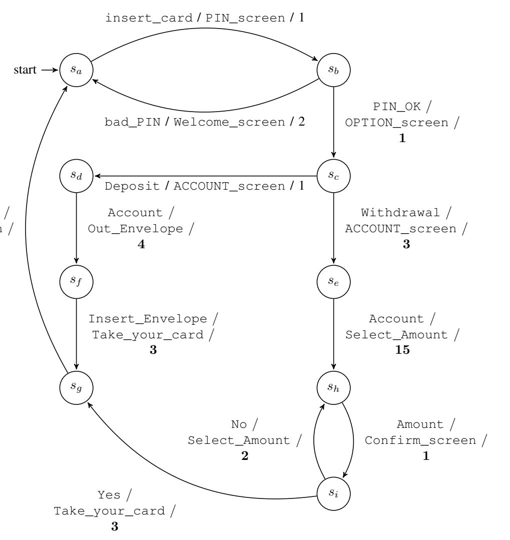

insert_card / PIN_screen / 1
start
s_a
s_b
bad_PIN / Welcome_screen / 2
PIN_OK / OPTION_screen / 1
s_d
s_c
Deposit / ACCOUNT_screen / 1
Withdraw_card / Welcome_screen / 2.5
Account / Out_Envelope / 4
Withdrawal / ACCOUNT_screen / 3
s_f
s_e
Insert_Envelope / Take_your_card / 3
Account / Select_Amount / 15
s_g
s_h
No / Select_Amount / 2
Amount / Confirm_screen / 1
s_i
Yes / Take_your_card / 3

Figure 1. Example of TFSM.

the output screen. In the real system, each transition may involve a number of complex steps which are not shown in the diagram since our TFSM is being used as a system design tool, rather than as a detailed description of the ATM's operation.

In Figure 1 we show the TFSM associated with the ATM system. In order to not overload the figure, we do not draw the complete set of transitions. We assume that for all states and all possible inputs of the ATM, if there does not exist a transition from a state and labelled with an specific input, then there exists a self-loop transition for this state labeled by this input producing as output the last screen that the system had shown, and the amount of time associated to the transition would be 2 time units. Let us note that our notion of input-enabled implies that the machine will be able to accept any input once is placed in a state, that is, is not performing a transition.

Continuing with the missing transitions, we consider a new input Cancel. For all state s, except the initial state, we include a transition $s \xrightarrow{\texttt{Cancel}/\texttt{Cancel\_screen}}_1 s_g$. This set of transitions allows the user to cancel the operation at any time, withdrawing her credit card. Regarding the initial state, we add the transition $s_a \xrightarrow{\texttt{Cancel}/\texttt{Welcome\_screen}}_1 s_a$ □

Next we introduce the notion of *trace*. A trace represents a sequence of actions that the system may perform from its initial state. As usual, a *log* is a sequence of actions representing the historical evolution of a system, that is, a timed trace of the system.

Definition 2: Let $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be a TFSM. We say that $\omega = \langle i_1/o_1/t_1, \ldots, i_n/o_n/t_n \rangle$ is a *timed trace*, or simply *trace*, of M if there is a sequence of transitions such that

$$s_1 \xrightarrow{i_1/o_1}_{t_1} s_2 \xrightarrow{i_2/o_2}_{t_2} s_3 \ldots s_{n-1} \xrightarrow{i_n/o_n}_{t_n} s_n$$

We denote by length of a trace the number of pairs that are contained in the trace.

We denote the empty trace by $\langle\rangle$ and by $\texttt{tr}(M)$ the set of all traces of M.

If $\omega = \langle i_1/o_1/t_1, \ldots, i_n/o_n/t_n \rangle$ is a timed trace of M, then the sequence $\langle i_1/o_1, \ldots, i_n/o_n \rangle$ is a *non-timed trace* of M. We denote by $\texttt{nttr}(M)$ the set of all non-timed traces of M, and by $\texttt{nttr}_\texttt{n}(M)$ the set of all traces with length less than or equal to n. □

Example 2: Let us consider the ATM specification of the previous example. Next, we present some traces of M. We have a trace with length 1 such as

$$\langle \texttt{insert\_card}/\texttt{PIN\_screen}/1 \rangle$$

This timed trace represents the existence of the transition $s_a \xrightarrow{\texttt{insert\_card}/\texttt{PIN\_screen}}_1 s_b$. We have traces with length 2 such as

$$\langle \texttt{Cancel} \;/\; \texttt{Welcome\_screen} \;/\; 1, \\ \texttt{insert\_card} \;/\; \texttt{PIN\_screen} \;/\; 1 \rangle$$

that is obtained from the sequence of transitions $s_a \xrightarrow{\texttt{Cancel}/\texttt{Welcome\_screen}}_1 s_a \xrightarrow{\texttt{insert\_card}/\texttt{PIN\_screen}}_1 s_b$. □

Invariants allow us to express properties that must be fulfilled by the SUT. For example, we can express that the time the system takes to perform a transition always belongs to a specific interval. In our framework, an invariant expresses the fact that each time the SUT performs a given sequence of actions, it must exhibit a behavior reflected in the invariant. In order to express invariants in a concise way, we can take advantage of the wild-card characters ? and $\star$. The wild-card ? represents any value in the sets of inputs and outputs, while $\star$ represents a sequence of input/output pairs.

Definition 3: We say that $\hat{p} = [p_1, p_2]$ is a *time interval* if $p_1 \in \mathbf{R}_+$, $p_2 \in \mathbf{R}_+ \cup \{\infty\}$, and $p_1 \le p_2$. We assume that for all $t \in \mathbf{R}_+$ we have $t < \infty$ and $t + \infty = \infty$. We consider that $\mathcal{IR}$ denotes the set of time intervals. Let $\hat{p} = [p_1, p_2]$ and $\hat{q} = [q_1, q_2]$ be time intervals. We overload the function sum of intervals as $[p_1, p_2] + [q_1, q_2] = [p_1 + q_1, p_2 + q_2]$.

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

1) φ is defined according to the following EBNF:

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

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

2) φ is *correct* with respect to M (defined in [24]).

We define the *length* of an invariant as the number of time intervals that belongs to it minus one. □

An invariant represents both timed and untimed properties of the specification. Untimed properties are represented by the sequences of input/output pairs that conform the invariant while timed properties are given by the intervals that indicate when these actions must be performed. The previous EBNF expresses that an invariant is either a sequence of symbols where each component but the last one is either an expression $a/z/\hat{p}$, with a being an input action or the wild-card character ?, z being an output action or the wild-card character ?, and $\hat{p}$ being a time interval, or an expression $\star/\hat{p}$. There are two restrictions to this rule. First, an invariant cannot contain two consecutive expressions $\star/\hat{p}_1$ and $\star/\hat{p}_2$. In the case that such situation was needed to represent a property, the tester could simulate it by means of the expression $\star/(\hat{p}_1+\hat{p}_2)$. The second restriction is that an invariant cannot present a component of the form $\star/\hat{p}$ followed by an expression beginning with the wild-card character ?, that is, the input of the next component must be a *real* input action $i \in \mathcal{I}$. In fact, $\star$ represents any sequence of inputs/outputs pairs such that the input is not equal to i. The last component, corresponding to the expression $i \mapsto O/\hat{p} \triangleright \hat{q}$ is an input action followed by a set of output actions and two timed restrictions, denoted by means of two intervals $\hat{p}$ and $\hat{q}$. The first one is associated to the last expression of the sequence. The last is related to the total time associated to the whole sequence of input/output pairs in the invariant. For example, the meaning of an invariant as $i/o/\hat{p}, \star/\hat{p}_\star, i' \mapsto O/\hat{p'} \triangleright \hat{q}$ is that if we observe the pair i/o in a time belonging to the interval $\hat{p}$, then the first occurrence of the input symbol i' must be followed by an output belonging to the set O after a lapse of time belonging to the interval $\hat{p'}$. The interval $\hat{q}$ makes reference to the total time that the system must spend to perform the whole trace.

Let us note that time conditions established in invariants are expressed by using time intervals, while time restrictions in `TFSMs` are represented by using real values. The reason for this *mismatch* is that the same input/output pair can label different transitions of the specification with different time values. Thus, an interval is used to express than all these values are correct. Another reason to use time intervals is that the artifacts measuring time while testing/observing a system might not be as precise as desirable. In this case, an apparent wrong behavior due to bad timing can be in fact correct since it may happen that the *clocks* are not working properly. A longer explanation on the use of time intervals to deal with imprecisions can be found in [29].

Next we give some examples in order to illustrate how invariants work considering the ATM specification previously presented.

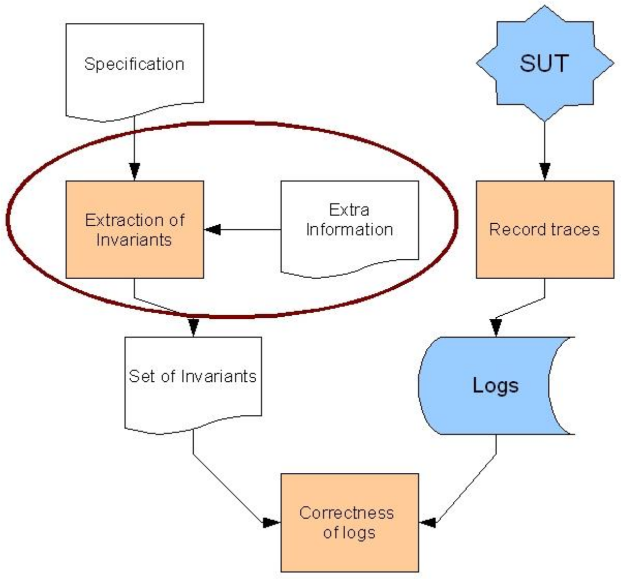


Figure 2. Schema to generate an invariant suite from a specification.

Example 3: Let us consider the ATM specification presented in Figure 1. For example:

$$\texttt{Cancel} \mapsto \{\texttt{Welcome\_screen}, \texttt{Cancel\_screen}\}[0,1] \triangleright [0,1]$$

reflects the fact that each time the input action `Cancel` is observed, the `Welcome_screen` or `Cancel_screen` is showed, and this task takes at most 1 time unit. We can express more complex properties such as:

$$\texttt{insert\_card}/\texttt{PIN\_screen}/[0,2], \texttt{PIN\_OK} \mapsto \{\texttt{OPTION\_screen}\}[0,2] \triangleright [0,5]$$

In this case, the invariant expresses that if the actions `insert_card`/`PIN_screen` are observed in an amount of time belonging to the interval [0,2] and then pin is ok, represented by the input action `PIN_OK`, then it must be followed by the `OPTION_screen`. The lapse between the input `PIN_OK` and the output `OPTION_screen` must be less than or equal to 2. Moreover, the whole sequence of operations must be performed in at most 5 time units.

Finally if we are interested in checking that after the output `Take_your_card` if the `Withdraw_card` is observed then the `Welcome_screen` will be emitted, we can express it by using the invariant:

$$\begin{array}{c} ?/\texttt{Take\_your\_card}/[0,3], \\ \texttt{Withdraw\_card} \mapsto \{\texttt{Welcome\_screen}\}[0,2.5] \\ \rhd [0.5,8] \end{array}$$

□

The correctness of our passive testing methodology for timed systems was established in [25]. We simply recall the main result.

Theorem 1: Let M and M' be two TFSMs and φ be a correct time invariant with respect to M. Let e be a log recorded from M'. If the invariant φ does not match e, then M' does not conform to M. □

III. Probabilistic user models

As we explained in the introduction, there are several ways to obtain a set of invariants. In this paper we propose that invariants can be automatically derived from the specification. The main problem with this approach is that the set of correct invariants is huge, and potentially infinite. Thus, we need a methodology to select *good* invariants among the set of candidates. In Figure 2 we graphically present the most significant components of our approach. This scheme builds on top of our previous proposals for passively testing timed systems [24], [25] and the use of probabilistic models in testing [27]. The main contribution of this paper is the definition of an algorithm to derive invariants from a specification by considering "extra information". This information corresponds to the *user model* extracted from the interactions of real users with the system. In order to obtain this additional information we apply *data mining techniques*.

A. Data mining to produce user models

Data mining is becoming an increasingly important tool to transform data into information. This technique is commonly used in a wide range of profiling practices, such as marketing, surveillance, fraud detection, and scientific discovery. Roughly speaking, data mining is the process of extracting hidden patterns from data sets. In Figure 3 we represent the process that we use in order to perform the extraction of the most relevant information related to the interaction of users with the SUT. It follows the scheme presented in [30]. First, we consider that we are provided with a set of data recorded from the interaction between different users and the SUT. Let us note that a bigger number of records in the database reflects that there exist more *kinds* of users represented in it. Next, a selection of data is made and preprocessed in order to check that there does not exist any incongruence, that is, the set of data represents traces of users that have been

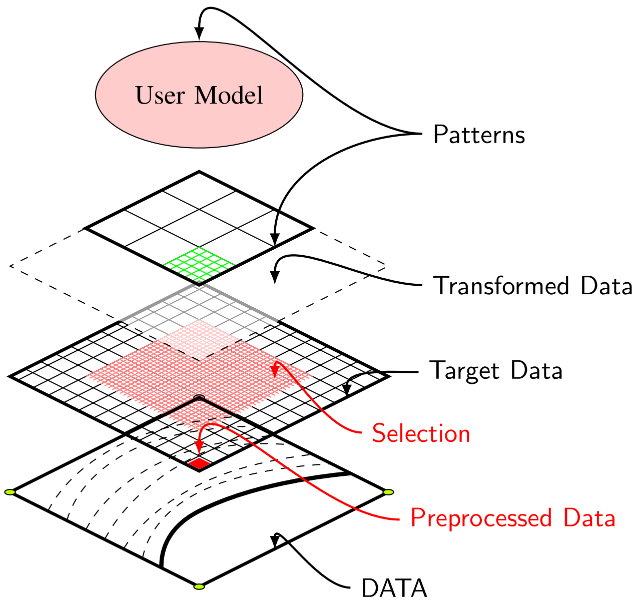


Figure 3. The data mining process.

observed during their interaction with the SUT. We will focus on extracting sets of *relevant behaviors* from this set of data, that is, those sequences of interactions of the users with the system that appear more frequently. In order to determine these behaviors we use the usual techniques for obtaining frequent patterns from a database (see, for example, [31], [32]). The task of discovering the frequency of each behaviour in the database is performed by means of the applications of any of the existing tools. In our approach we propose [33]. Once we have collected the information corresponding to the usual behaviours and their frequency, we can *convert* this information into a user model that we can use in our algorithm.

Next, let us describe the extraction of a probabilistic model to represent how users decide to interact with the SUT, using the processed data that we have obtained with the data mining process. Due to space limitations, we simply describe the intuitive idea to deal with the construction of the model by using our running example. Let us suppose that we have recorded several interactions from different users with the ATM. In our simplified system, we only focus on two ideas. The first idea is that users go to the ATM to interact with it (99%). The second idea is that users go to the ATM to obtain money (80%), to deposit money (19%), and to do nothing with the ATM

(1%). This last situation happens if, for example, a user presses a sequence of inputs without inserting her credit card.

This first idea can be simulated in the small automaton presented in Figure 4, up. The transitions are labeled by the input/output pair performed by the user/ATM and the probability associated with the transition. Let us note that the loop transition in s_a (it means, $\mathcal{I}\backslash$ $\{$insert_card$\}$ / Welcome_screen , 0.01) is a shorthand for representing a set of transitions $s_a \xrightarrow{i_j/o_j}_{p_j} s_a$ where for all $i_j \neq$ insert_card such that $\sum p_j = 0.01$.

Our approach can express more complex behaviors. For example we can express that the probability to stop (press Cancel) after selecting the option to extract money is 15%; the probability to stop after pressing the option of deposit money would be 5%. This knowledge, extracted from the database containing the *dialogues* between users and the ATM, can be expressed in our probabilistic model as presented in Figure 4, down. Next, we formally define user models which are the specific kind of probabilistic model that we are going to use in the extraction of invariants from the specification.

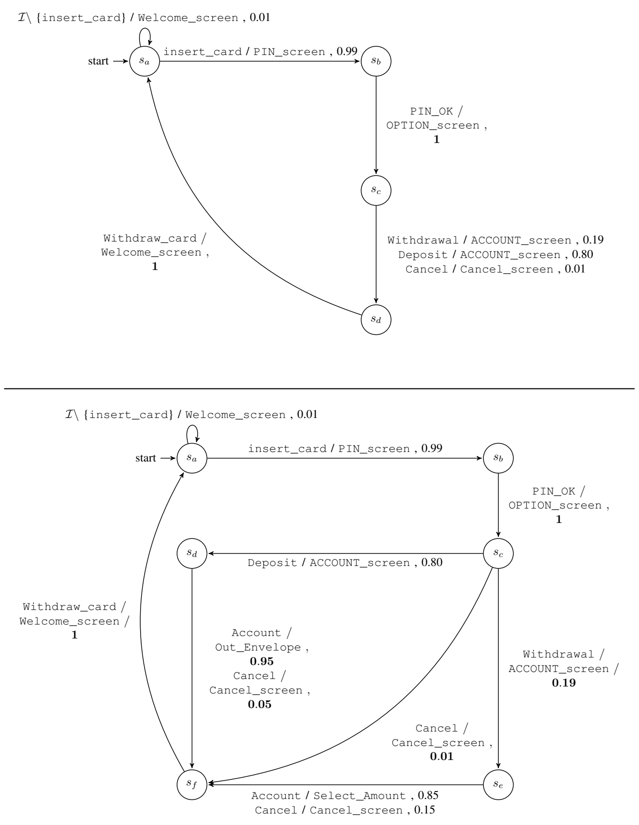
B. User model approach

As we already sketched, a user model includes the probability that a user chooses each input in each situation. We will use a particular case of *Probabilistic Machine* (PM) to represent user models. Let us note that the interaction of a user with a SUT ends whenever the user decides to stop it. Consequently, models denoting users must represent this event as well. Therefore, given a PM representing a user model, we will require that the sum of the probabilities of all inputs associated to a state is lower than or equal to 1. The remainder up to 1 represents the probability of stopping the interaction at this state.

Example 4: In our ATM System we consider that a user stops the interaction when she returns to the Welcome_screen. Let us note that there exists several situations from which a user can finish her interaction. In particular, she can do it from all the states of the machine. □

Definition 4: A *user model* is represented by a tuple $U = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, F, T)$ where $\mathcal{S}$ is the finite set of states and $s_0 \in \mathcal{S}$ is the initial state. $\mathcal{I}$ and $\mathcal{O}$, with $\mathcal{I} \cap \mathcal{O} = \varnothing$, are the finite sets of *input* and *output* actions, respectively. $F = \{f_s | s \in S\}$ is a set of probabilistic functions such that for all state $s \in \mathcal{S}$, $f_s : \mathcal{I} \mapsto [0, 1]$ fulfills that $\sum_{i \in \mathcal{I}} f_s(i) \leq 1$. $T \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{O} \times \mathcal{S}$ is the set of transitions.

We say that U is *deterministically observable* if there do not exist in T two transitions $(s, i, o_1, s_1), (s, i, o_2, s_2) \in T$ with $o_1 = o_2$ and $s_1 \neq s_2$. □



$\mathcal{I}\backslash$ {insert_card} / Welcome_screen , 0.01
start
s_a
insert_card / PIN_screen , 0.99
s_b
PIN_OK / OPTION_screen , 1
s_c
Withdrawal / ACCOUNT_screen , 0.19
Deposit / ACCOUNT_screen , 0.80
Cancel / Cancel_screen , 0.01
Withdraw_card / Welcome_screen , 1
s_d
$\mathcal{I}\backslash$ {insert_card} / Welcome_screen , 0.01
start
s_a
insert_card / PIN_screen , 0.99
s_b
PIN_OK / OPTION_screen , 1
s_d
Deposit / ACCOUNT_screen , 0.80
s_c
Withdraw_card / Welcome_screen / 1
Account / Out_Envelope , 0.95
Cancel / Cancel_screen , 0.05
Withdrawal / ACCOUNT_screen / 0.19
Cancel / Cancel_screen , 0.01
s_f
s_e
Account / Select_Amount , 0.85
Cancel / Cancel_screen , 0.15

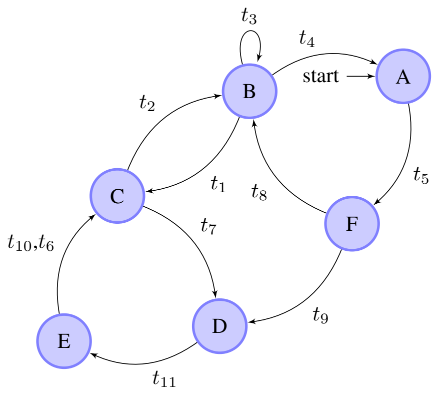
Figure 4. Extraction of a probabilistic model using data mining techniques.

In this paper, we assume that user models are deterministically observable. In fact, user models constructed by following the technique described in Section III-A are deterministically observable. We adopt $s \xrightarrow{i/o}_p s'$ as a shorthand to express $(s, i, o, s') \in T \wedge p = f_s(i)$.

Example 5: Next we are going to build a basic user model from the already studied ATM system. In Figure 5 we represent a subset of the transitions of this model. We only consider three transitions: t_5, t_8, and t_9. Let us remember that due to the fact that this model is deterministic, and input-enabled, the number of transitions is equal to 72, since we have 6 states and 12 inputs.

The initial state of any user (with respect to this model) is the state A. The probability that a user goes to the next step, that is, she inserts her credit card and goes to F in the system is given by f_A. In this case, this probability is equal to 0.95. The remainder up to 1, that is, 0.05 is the probability to stop in state A.

When the user is at the state F the probability associ-



$$t_5 = A \xrightarrow{\texttt{insert\_card/PIN\_screen}}_{0.95} F$$

$$t_8 = F \xrightarrow{\texttt{PIN\_OK/OPTION\_screen}}_{0.80} B$$

$$t_9 = F \xrightarrow{\texttt{bad\_PIN/Welcome\_screen}}_{0.10} D$$

Figure 5. Example of user model.

ated to insert a PIN_OK is represented by $f_F(\texttt{PIN\_OK})$. In this case, we obtain 0.80.

The probability that the user applies a bad_PIN is 0.10. Thus, the probability that the user cancels the operation at state F is 0.10. $\square$

Next, we identify the probabilistic traces of user models. These traces will be used in order to provide a coverage degree of a possible user behavior with respect to another.

Definition 5: Let $\mathcal{I}$ be a set of input actions and $\mathcal{O}$ be a set of output actions. A *probabilistic trace* is a sequence $\langle(i_1/o_1, p_1), (i_2/o_2, p_2), \ldots, (i_n, o_n, p_n)\rangle$, with $n \geq 0$, such that for all $0 \leq k \leq n$ we have $i_k \in \mathcal{I}$, $o_k \in \mathcal{O}$, and $p_k \in [0, 1]$.

Let U be a user model and $\sigma = \langle(i_1/o_1, p_1), (i_2/o_2, p_2), \ldots, (i_n/o_n, p_n)\rangle$ be a probabilistic trace. We say that σ is a trace of U, denoted by $\sigma \in \texttt{ptr}(U)$, if there is a sequence of transitions of U

$$s_0 \xrightarrow{i_1/o_1} s_1 \xrightarrow{i_2/o_2} s_2 \cdots \xrightarrow{i_n/o_n} s_n$$

and for all $1 \leq k \leq n$ we have $p_k = f_{s_{k-1}}(i_k)$. We define the *stopping probability of* U *after* σ as

$$\mathtt{s}(U, \sigma) = 1 - \sum\{f_{s_n}(i) \mid i \in \mathcal{I}\}$$

We denote the *complete probability* of σ as

$$\texttt{prob}(U, \sigma) = \left(\prod_{1 \leq j \leq n} p_j\right) \cdot \mathtt{s}(U, \sigma)$$

$\square$

The next definition characterizes the traces we can observe if a user model and a TFSM (representing a SUT) interact with each other. These traces are the result of the inputs chosen by the user model according to its probabilities.

Definition 6: Let M be a deterministic TFSM, U be a user model, and σ be a probabilistic trace $\langle(i_1/o_1, p_1), (i_2/o_2, p_2), \ldots, (i_n/o_n, p_n)\rangle$. We say that σ is a *probabilistic trace of the parallel composition* of M and U if $\sigma \in \texttt{ptr}(U)$ and $\langle i_1/o_1, i_2/o_2, \ldots, i_n/o_n\rangle \in \texttt{nttr}(M)$. We denote by $\texttt{ptr}(M \parallel U)$ the set containing the probabilistic traces of the parallel composition of M and U. $\square$

Let us note that if we generate invariants that reflect only a finite set of input sequences, then we can calculate the proportion of covered interactions between the user and the SUT. In terms of *number of interactions*, this proportion is 0 because we are considering a finite number of cases out of an infinite set of possibilities. However, the *probability* that the user interacts with the SUT and performs one of the traces considered in a given finite set of cases is not 0 in general. Thus, we may use this value as a *coverage measure* of a finite set of invariants.

Definition 7: An input trace is a sequence of input actions $\alpha = \langle i_1, i_2, \ldots, i_n\rangle$, with $n \geq 0$. If α' is a subtrace of α then we denote it by $\alpha' \leq \alpha$.

Let M be a deterministic TFSM and U be a user model. We say that an input trace $\alpha = \langle i_1, i_2, \ldots, i_n\rangle$ is an *input trace of the parallel composition* of M and U if there exists a probabilistic trace $\sigma = \langle(i_1/o_1, p_1), (i_2/o_2, p_2), \ldots, (i_n/o_n, p_n)\rangle \in \texttt{ptr}(M \parallel U)$. In this case, we say that σ is the *output completion* of α under U in M, and we denote it by $\texttt{comp}(U, M, \alpha) = \sigma$. We denote by $\texttt{itr}(M \parallel U)$ the set containing all the input traces of the parallel composition of M and U.

The *behavior* of a sequence of inputs is the set containing the sequence itself and all of its prefixes. Formally, $\texttt{tu}(\alpha) = \{\alpha' \mid \alpha' \leq \alpha\}$.

The *prefix-closed coverage* of a set of input sequences Σ is defined as:

$$\mathtt{c}_{M \parallel U}(\Sigma) = \sum_{\alpha \in \bigcup_{\alpha' \in \Sigma} \texttt{tu}(\alpha')} \texttt{prob}(U, \texttt{comp}(U, M, \alpha))$$

$\square$

Let us note that, due to its definition, $\langle i_1, i_2, \ldots, i_n\rangle$ can simulate the specific following behaviors of a user: the user decides to stop at its initial state without offering any input; the user chooses to execute input i_1, receives an output and then stops; the user chooses to execute input i_1, receives an output, then chooses to execute i_2, receives an output and then stops; and so on. Thus, we define the *coverage* of that invariant as the sum of all that independent behaviors. If we have a set of invariants then its coverage is equal to the addition of the coverage of all behaviors.

```
in : $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, Tr)$ is the specification.
     $U = (\mathcal{S}', s_0', \mathcal{I}, \mathcal{O}, F, T)$ is the user model.
     $n$ length of the invariant.
out: var_queue queue of pairs $\langle$Value,invariant$\rangle$.
     // var_queue is a queue that
     // does not have twice the same invariant.

var_set_nttr= $\mathtt{nttr_n}(M)$;
// We select the set of non-timed traces from the
// specification with length less than or equal to n.
while (var_set_nttr$\neq \emptyset$) do
    $\omega$ = moreRepr(var_set_nttr, $M, U$);
    var_set_nttr=var_set_nttr$\setminus\{\omega\}$;
    // We choose a non-timed trace.
    $\varphi$ = generateT($M, \omega$);
    $\alpha$ = seq_inputs($\omega$);
    var_queue.insert($\mathtt{c}_{M\|U}(\{\alpha\}), \varphi$);
end
return var_queue;
```

Figure 6. Algorithm for generating invariants from a TFSM using a user model.

However, we must take into account that if two behaviors share a common prefix then the probability of this prefix must be accounted only once.

Let us suppose that M is the TFSM that denotes our specification and Σ is a set of input sequences. Intuitively, $\mathtt{c}_{M\|U}(\Sigma)$ denotes the *proportion* of behaviors of M that we check if we apply all the input sequences in Σ to the SUT. This proportion is measured in terms of its *probabilistic weight* for the user, that is, in terms of the probability the user has to choose any sequence belonging to Σ *if* the SUT produced the outputs required by M, that is, if the SUT worked correctly for those sequences.

IV. Relating user model and extraction of invariants

Once we have a user model we need to use it in the task of generating invariants from the specification with respect to a determinate criterion. This paradigm allows us to build an appropriate strategy where rules are not the main source of knowledge, but cases are. So, they compute solutions by recovering the most significant previous cases. We assume the premise that the world is *regular*. Therefore, if we have a big number of user behaviors in the database, we are able to consider that we have an acceptable percentage of the amount of usual interactions with the system. Let us note that the idea of being provided with a knowledge base which contains a representation of the expertise in the domain, that is our database, is not new; it is a usual assumption considered in expert systems.

In Figure 6 we present our algorithm to derive invariants, using the approach sketched in Figures 2 and 3. The next auxiliary functions are used in the algorithm.

Definition 8: Let $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be a TFSM, $\omega = \langle i_1/o_1, \dots, i_n/o_n\rangle$ be a non-timed trace of M. The function $\mathtt{seq\_inputs}(\omega)$ computes the sequence of inputs associated with the non-timed trace ω.

$$\mathtt{seq\_inputs}(\omega) = \langle i_1, \dots, i_n\rangle$$

This function can be generalized to deal with sets of traces. Let T be a set of non-timed traces of M. The function $\mathtt{setseq\_inputs}(T)$ computes the set of sequences of inputs associated with T.

$$\mathtt{setseq\_inputs}(T) = \{\mathtt{seq\_inputs}(\omega) | \omega \in T\}$$

The function moreRepr receives a set of non-timed traces T and a user model U and computes the non-timed trace that has the most representative value, that is $\mathtt{moreRepr}(T, M, U) = \omega$ if $\omega \in T$ and for all $\omega' \in T$, $\omega' \neq \omega$, we have $\mathtt{c}_{M\|U}(\omega) \geq \mathtt{c}_{M\|U}(\omega')$.

The function $\mathtt{outputs}(\omega, M)$ computes the set of all possible outputs associated with the last input of the non-timed trace ω, that is a non-timed trace of M.

$$\mathtt{outputs}(\omega, M) = \left\{o \;\middle|\; \langle i_1/o_1, \dots, i_n/o\rangle \in \mathtt{nttr}(M)\; \right\}$$

The functions $\mathtt{mT}(\omega, j, O)$ and $\mathtt{MT}(\omega, j, O)$ compute the minimum, respectively maximum, time value associated to the pair i_j/o_j in all the timed-traces of M corresponding to the trace ω, except the last output that must belong to O.

$$\mathtt{min}\left\{t_j \;\middle|\; \langle i_1/o_1/t_1, \dots, i_n/o/t_n\rangle \in \mathtt{tr}(M) \;\wedge\; o \in O\right\}$$

$$\mathtt{max}\left\{t_j \;\middle|\; \langle i_1/o_1/t_1, \dots, i_n/o/t_n\rangle \in \mathtt{tr}(M) \;\wedge\; o \in O\right\}$$

The function $\mathtt{generateT}(M, \omega)$ computes a set of correct invariants with respect to M corresponding to the sequence ω.

$$\left\{\varphi \;\middle|\; \begin{array}{c} \varphi = i_1/o_1/\hat{p}_1, \dots, i_n \mapsto O/\hat{p}_n \rhd \hat{q} \;\wedge \\ O = \mathtt{outputs}(\omega, M) \;\wedge \\ \forall 1 \leq k \leq n : \hat{p}_k = [\mathtt{mT}(\omega, k, O), \mathtt{MT}(\omega, k, O)] \;\wedge \\ \hat{q} = \sum \hat{p}_k \end{array}\right\}$$

□

The algorithm has three inputs parameters. The first one is the specification of the system, the next one is the formal user model extracted by using data mining techniques, and the last one is the maximum *length* of the invariants that we would like to extract. The algorithm returns a queue of

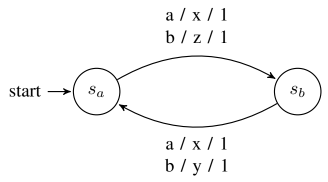

$$\begin{tikzcd}\text{start} \arrow[r] & s_a \arrow[r, bend left, "a / x / 1 \\ b / z / 1"] & s_b \arrow[l, bend left, "a / x / 1 \\ b / y / 1"]\end{tikzcd}$$

Figure 7. Example of `TFSM` **.**

items ⟨Value,invariant⟩, ordered with respect to Value, and without repetitive invariants. Let us remark that we limit the maximum length of the invariants because in [34] we showed, with empirical results, that longer invariants do not imply bigger error detection power. Next, given the returned queue we are able to obtain its *global coverage*.

Definition 9: Let `var_queue` be the queue given by the application of the algorithm in Figure 6. We define its *global coverage* associated as

$$\sum \texttt{var\_queue}.\text{Value}$$

□

Let us note that we cannot claim that the global coverage obtained in the algorithm is the same as the coverage of the non-timed traces that we use to calculate the invariants. For example, let M be the `TFSM` depicted in Figure 7. On the one hand, we have that the sum of all components Value of `var_queue`, that is $\sum \texttt{var\_queue}.\text{Value}$, is equal to $\mathtt{c}_{M\|U}(\langle a\rangle) + 2 * \mathtt{c}_{M\|U}(\langle a, b\rangle)$. On the other hand the coverage of the non-timed traces is $\mathtt{c}_{M\|U}(\{\langle a\rangle, \langle a, b\rangle\})$ and according to [27], we have that it is equivalent to $\mathtt{c}_{M\|U}(\{\langle a, b\rangle\})$. Thus, we have that they do not have the same value.

Finally, let us comment that the algorithm returns always a finite queue of invariants. In fact, we are considering input-enabled machines. So, given $n \in \mathbf{N}$, we have that $|\mathtt{nttr_n}(M)| \leq |\mathcal{I}| * |\mathcal{O}| * n$. In the algorithm, due to the fact that we are limiting the computation of the traces to those of length less than or equal to n, and this set is finite, the while-loop is computed $|\mathtt{nttr_n}(M)|$ times. So, it will return at most $|\mathtt{nttr_n}(M)|$ invariants (we exclude the repetitive invariants).

V. Conclusions and Future Work

In this paper we have extended our passive testing framework with an algorithm to generate invariants from the specification of a timed system. Our invariants can be seen as rules that reflect both temporal and nontemporal aspects of the specification that must be hold by all traces produced by the system under test. As usual, the set of all possible traces of a `SUT` is infinite and we want to generate a representative set of invariants which reflect the most often interactions between users and systems. We present a methodology, based on data mining, that takes as input a database containing interactions between users and the system, in order to obtain the most frequent interaction sequences. We use this extra information to construct a user model and guide our algorithm to extract a set of invariants.

As future work we plan to integrate (probabilistic) implementer models into our methodology to further increase the coverage of the selected invariants, more precisely, to reduce the size of the set of invariants while keeping the desired coverage.

References

[1] G. Myers, *The Art of Software Testing*, 2nd ed. John Wiley and Sons, 2004.

[2] P. Ammann and J. Offutt, *Introduction to Software Testing*. Cambridge University Press, 2008.

[3] B. Bosik and M. Uyar, "Finite state machine based formal methods in protocol conformance testing," *Computer Networks & ISDN Systems*, vol. 22, pp. 7–33, 1991.

[4] D. Lee and M. Yannakakis, "Principles and methods of testing finite state machines: A survey," *Proceedings of the IEEE*, vol. 84, no. 8, pp. 1090–1123, 1996.

[5] E. Brinksma and J. Tretmans, "Testing transition systems: An annotated bibliography," in *4th Summer School on Modeling and Verification of Parallel Processes, MOVEP'00, LNCS 2067*. Springer, 2001, pp. 187–195.

[6] M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds., *Model-based Testing of Reactive Systems, LNCS 3472*. Springer, 2005.

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

[8] R. Hierons, J. Bowen, and M. Harman, Eds., *Formal Methods and Testing, LNCS 4949*. Springer, 2008.

[9] J. Jacky, M. Veanes, C. Campbell, and W. Schulte, *Model-Based Software Testing and Analysis with C#*. Cambridge University Press, 2008.

[10] R. Hierons, K. Bogdanov, J. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Luettgen, A. Simons, S. Vilkomir, M. Woodward, and H. Zedan, "Using formal methods to support testing," *ACM Computing Surveys*, vol. 41, no. 2, 2009.

[11] M. Merayo, M. Núñez, and I. Rodríguez, "Formal testing from timed finite state machines," *Computer Networks*, vol. 52, no. 2, pp. 432–460, 2008.

[12] M. Uyar, S. Batth, Y. Wang, and M. Fecko, "Algorithms for modeling a class of single timing faults in communication protocols," *IEEE Transactions on Computers*, vol. 57, no. 2, pp. 274–288, 2008.

[13] R. Hierons and H. Ural, "The effect of the distributed test architecture on the power of testing," *The Computer Journal*, vol. 51, no. 4, pp. 497–510, 2008.

[14] J. Chen, R. Hierons, and H. Ural, "Testing in the distributed test architecture," in *Formal Methods and Testing, LNCS 4949*. Springer, 2008, pp. 157–183.

[15] M. Merayo, M. Núñez, and I. Rodríguez, "Extending EFSMs to specify and test timed systems with action durations and timeouts," *IEEE Transactions on Computers*, vol. 57, no. 6, pp. 835–848, 2008.

[16] R. Hierons, M. Merayo, and M. Núñez, "Testing from a stochastic timed system with a fault model," *Journal of Logic and Algebraic Programming*, vol. 78, no. 2, pp. 98–115, 2009.

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

[18] M. Tabourier and A. Cavalli, "Passive testing and application to the GSM-MAP protocol," *Information and Software Technology*, vol. 41, no. 11-12, pp. 813–821, 1999.

[19] R. E. Miller, D. Chen, D. Lee, and R. Hao, "Coping with nondeterminism in network protocol testing," in *17th Int. Conf. on Testing of Communicating Systems, TestCom'05, LNCS 3502*. Springer, 2005, pp. 129–145.

[20] 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*. Springer, 2007, pp. 335–350.

[21] A. Benharref, R. Dssouli, M. Serhani, and R. Glitho, "Efficient traces' collection mechanisms for passive testing of web services," *Information & Software Technology*, vol. 51, no. 2, pp. 362–374, 2009.

[22] A. Cavalli, C. Gervy, and S. Prokopenko, "New approaches for passive testing using an extended finite state machine specification," *Information and Software Technology*, vol. 45, no. 12, pp. 837–852, 2003.

[23] E. Bayse, A. Cavalli, M. Núñez, and F. Zaïdi, "A passive testing approach based on invariants: Application to the WAP," *Computer Networks*, vol. 48, no. 2, pp. 247–266, 2005.

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

[25] ——, "Formal correctness of a passive testing approach for timed systems," in *5th Workshop on Advances in Model Based Testing, A-MOST'09*. IEEE Computer Society Press, 2009, pp. 67–76.

[26] M. Leucker and C. Schallhart, "A brief account of runtime verification," *Journal of Logic and Algebraic Programming*, vol. 78, no. 5, pp. 293–303, 2009.

[27] C. Andrés, L. Llana, and I. Rodríguez, "Formally transforming user-model testing problems into implementer-model testing problems and viceversa," *Journal of Logic and Algebraic Programming*, vol. 78, no. 6, pp. 425–453, 2009.

[28] R. Alur and D. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, pp. 183–235, 1994.

[29] M. 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*. Springer, 2007, pp. 160–174.

[30] G. Piatetsky-Shapiro, "Data mining and knowledge discovery 1996 to 2005: overcoming the hype and moving from "university" to "business" and "analytics"," *Data Mining and Knowledge Discovery*, vol. 15, no. 1, pp. 99–105, 2007.

[31] R. Agrawal, T. Imieliński, and A. Swami, "Mining association rules between sets of items in large databases," in *19th ACM Int. Conf. on Management of Data, SIGMOD'93*. ACM Press, 1993, pp. 207–216.

[32] T. Mielikäinen, "Frequency-based views to pattern collections," *Discrete Applied Mathematics*, vol. 154, no. 7, pp. 1113–1139, 2006.

[33] E. Frank, M. Hall, G. Holmes, R. Kirkby, and B. Pfahringer, "Weka - a machine learning workbench for data mining," in *The Data Mining and Knowledge Discovery Handbook*, O. Maimon and L. Rokach, Eds. Springer, 2005, pp. 1305–1314.

[34] C. Andrés, M. Merayo, and C. Molinero, "Advantages of mutation in passive testing: An empirical study," in *4th Workshop on Mutation Analysis, Mutation'09*. IEEE Computer Society Press, 2009, pp. 230–239.