Multi-objective Genetic Algorithms: Construction and Recombination of Passive Testing Properties

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

***Abstract*—Passive testing uses the historical interaction files of the systems in order to decide their correctness. In our testing framework, we represent the most expected properties to check by using invariants. Normally, a tester introduces the set of invariants to perform this task, but in many cases the size of this set is too small. This paper presents a way to provide a bigger set of representative invariants to the testers. This novel approach is based on applying genetic algorithms in a small set of invariants, in order to generate a representative bigger invariant suite.**

***Index Terms*—Passive Testing, Genetic Algorithms**

I. 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 decide their correctness. *Testing techniques* [1], [2] allow us to provide a degree of confidence in the correctness of a system. These techniques can be combined with the use of formal methods [3], [4], [5], [6], [7], [8] in order to semi-automatically perform some tasks involved in testing through the use of tools [9]. 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, during the last years formal testing techniques also dealt with non-functional properties. In this paper we focus on systems that contain temporal restrictions, being already several proposals for timed testing (e.g. [10], [11], [12], [13]).

In testing, we can distinguish between two approaches: *Passive* and *Active*. The main difference between them is whether a tester can interact with the System Under Test (SUT), or not. If the tester can interact with the SUT, then we are in the active testing paradigm. On the contrary, if the tester simply monitors the behavior, then we are in the passive testing paradigm. Actually, it is very frequent that the tester is unable to interact with the SUT. 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, or for example, let us consider the task of testing a bank account manager. A possible test suite to check the correctness of this system might be:

1) *Create a new account.*
2) *Change the ID of a user of the system.*
3) *Insert* $\$1.000.000$ *in the account XXX.YYY.ZZZ.*
4) *Delete a user.*

As we observe, it is very risky to perform these actions in systems that are working in this way. So, the testers would use passive testing techniques in order to test it.

In our formal passive testing framework [14], [15], we consider the following properties. First of all, we are provided with a formal *specification* of the system. In this specification all the correct behaviours of the system are represented. We also consider that we are provided with a set of formal requirements, that we will call *invariants*. The invariants represent the most expected properties to be checked in the SUT. In these invariants a tester is able to represent not only functional, but also non functional requirements.

Usually we consider different ways to obtain a set of invariants. In our classical approach we present that this set is suggested by the tester. In this case, before using the invariants to check the correctness of the system, we have to check the correctness of them with respect to the specification. It make not sense to check the correctness of a system with some possible wrong properties. Within this first approach we detected the following two problems: The set of invariants might be erroneus and the size of the set of invariants was too small. Thus, actually the computational resources which we are provided allow us to check huge set of invariants at the same time without decreasing the perfomance, and normally the testers "from scratch" do not generate a set of invariants so big. In this way in [16] we show how to use the specification of the system to provide the tester with a set of invariants. We presented an algorithm that automatically derives a set of invariants. Unfortunately, the set of derived invariants was so big, sometimes infinite, and it was non-representative. With non-representative we mean that we were unable to establish any criterion to compare invariants. For example to decide the set of 4-better invariants. To solve this handicap, in [17] we assume the hypothesis of having knowledge about the system concerning the past-interactions of the users with the SUT. This information was collected in

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

a database. We applied data mining techniques in order to generate a probabilistic model [18] to describe the users of the system. We adapted the algorithm of [16] to use this new information. The underline idea was to guide the invariant generation in order to check the most frequent user sequence actions presented in the probabilistic model.

However, we consider that these are not the unique and optimum possible ways to provide a tester with a set of invariants. Within the last approach we generated the properties basing on the most common input sequences performed by the users of this SUT. But, let us consider that we have produced several implementations, and they are placed in different locations, and we want to generate a representative invariant suite to test all of them. Usually the users that are interacting with a SUT do not interact with another SUT. Regarding our previous approach we would generate a user model for each system, but we were not able to generate a unique invariant suite, sharing the information presented in each user model. In this paper we present a novel methodology based on *genetic algorithms* to cross the sets of invariants that we have for each SUT, in order to generate a *powerful* set of representative invariants with respect to any user model. The use of genetic algorithms applied to formal testing tasks is not new [19], [20], [21], [22], so we consider that it is a feasible way to work with them.

The rest of the paper is structured as follow. In Section II our formal framework is presented. In Section III we present the use of genetic algorithms in order to provide a set of representative invariants. Finally, in Section IV we present the conclusions and future work.

II. Formal framework

In this section we introduce our framework to specify and (passively) test timed systems. We extend the well-known Finite State Machines model by adding time information concerning the amount of time that the system needs to perform transitions.

Definition 1: A *Timed Finite State Machine (TFSM)* is a tuple $(\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$, 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 $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{I} \times \mathcal{O} \times \mathbb{R}_+ \times \mathcal{S}$ is the set of transitions.

A TFSM $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ is *deterministic* if for all state s and input i there exists at most one transition (s, i, o, t, 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, i, o, t, s'). □

Let us consider the TFSM depicted in Figure 1 and the transition $(s_1, a, x, 3, s_2)$. Intuitively, if the machine is at state s_1 and it receives the input a, then it will produce the output x after 3 time units. We usually write $s \xrightarrow{i/o}_t s'$ as a shorthand of $(s, i, o, t, s') \in \mathcal{T}$. As usual in formal testing approaches, we assume that both specifications and implementations can be represented by the same formalism. In our case, the formalism is the set of deterministic input-enabled TFSMs.

Next we introduce the notion of *trace* of a system. A trace captures the behaviour of an implementation. Traces collect the outputs obtained after sending some inputs to the implementation and the amount of time between each input/output pair. The classical notion of trace, which does not include any time information, is called a *non-timed trace*.

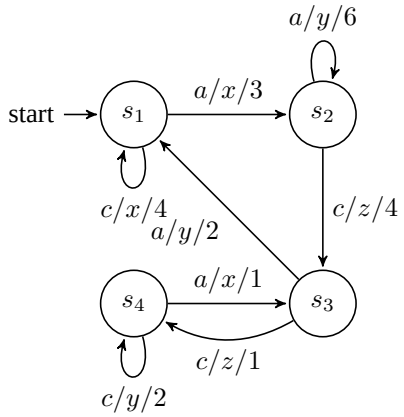

Figure 1. Example of TFSM.

Definition 2: Let $M = (\mathcal{S}, s_0, \mathcal{I}, \mathcal{O}, \mathcal{T})$ be a TFSM. We say that $\langle i_1/o_1/t_1, i_2/o_2/t_2, \ldots, i_n/o_n/t_n \rangle$ is a *timed trace*, or simply trace, of M if there is a sequence of transitions such that

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

If $\langle i_1/o_1/t_1, \ldots, i_n/o_n/t_n \rangle$ 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. □

For example, if we consider the machine M depicted in Figure 1, we have that $\langle c/x/4, c/x/4, a/x/3, c/z/4, c/z/1, a/x/1 \rangle$ is a timed trace of M. If we remove time information, we obtain *non-timed* traces. For instance, $\langle c/x, c/x, a/x, c/z, c/z, a/x \rangle$ is the non-timed trace associated with the previous timed trace.

Our passive testing approach is similar to other methodologies since the basic idea consists in recording traces from the SUT to detect unexpected behaviours. The main novelty is that a set of *invariants* is used to represent the most relevant properties of the specification. Intuitively, an invariant expresses the fact that each time the SUT performs a given sequence of actions, then it must exhibit a behaviour reflected in the invariant. In order to express traces in a concise way, we will use the wild-card characters ? and ⋆. The wild-card ? represents any value in the sets of inputs or outputs, while ⋆ represents a sequence of input/output pairs.

Definition 3: We say that $\hat{p} = [p_1, p_2]$ is a *time interval* if $p_1 \in \mathbb{R}_+$, $p_2 \in \mathbb{R}_+ \cup \{\infty\}$, and $p_1 \leq p_2$. We assume that for all $t \in \mathbb{R}_+$ we have $t < \infty$ and $t + \infty = \infty$. We consider that $\mathcal{IR}$ denotes the set of time intervals.

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

1) ϕ is defined according to the following EBNF:

$$\begin{aligned} \phi &::= \texttt{Body} \mapsto \texttt{Consequent} \\ \texttt{Body} &::= a/z/\hat{p}, \texttt{Body} \mid \star/\hat{p}, \texttt{Body}' \mid i \\ \texttt{Body}' &::= i/z/\hat{p}, \texttt{Body} \mid i \\ \texttt{Consequent} &::= O/\hat{p} \rhd \hat{t} \end{aligned}$$

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

2) ϕ is *correct* with respect to M (formally defined in [14]). □

Even though, our machines present time information expressed as fix amounts of time, time conditions established in invariants are given by intervals. This fact is due to consider that it can be admissible that the execution of a task sometimes lasts more time than expected: If most of the times the task is performed on time, a small number of delays can be tolerated. Concerning the notion of correctness, the idea is that an invariant is correct with respect to a machine M, if M cannot perform a timed trace that would contradict what the invariant expresses.

Intuitively, the previous EBNF expresses that an invariant is either a sequence of symbols where each component, but the last one, by means the Body, 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 $?$, and $\hat{p}$ being an 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, by means the Consequent, corresponding to the expression $i \mapsto O/\hat{p} \rhd \hat{t}$ is an input action followed by a set of output actions and two timed restrictions, denoted by means of two intervals $\hat{p}$ and $\hat{t}$. The former is associated to the last expression of the sequence. The latter is related to the sum of times values associated to all input/output pairs performed before. For example, the meaning of an invariant as $i/o/\hat{p}, \star/\hat{p_\star}, i' \mapsto O/\hat{p}' \rhd \hat{t}$ is that if we observe the transition i/o in a time belonging to the interval $\hat{p}$, then the first occurrence of the input symbol i' after a lapse of time belonging to the interval $\hat{p_\star}$, must be followed by an output belonging to the set O. The interval $\hat{t}$ makes reference to the total time that the system must spend to perform the whole trace. Next we introduce some examples in order to present how invariants work.

Example 1: Let us consider the TFSM presented in Figure 1. One of the most simple invariants that we can define within our framework follows the scheme $a \mapsto \{x, y\}/[1, 6] \rhd [1, 6]$. The idea is that each occurrence of the symbol a is followed by either the output symbol x or y and this transition is performed between 1 and 6 time units.

We can specify a more complex property by taking into account that we are interested in observing the output x after the input a when the pair c/x was previously observed. In addition, we include time intervals corresponding to the amount of time the system takes for each of the transitions and to the total time it spends in the whole trace. We could express this property by means of the invariant $c/x/[3, 5], \star/[0, 5], a \mapsto \{x\}/[2, 4] \rhd [4, 13]$. An observed trace will be correct with respect to this invariant if each time that we find a (sub)sequence starting with the c/x pair, performed in an amount of time belonging to the interval $[3, 5]$, if there is an occurrence of the input a before 5 time units pass, then it must be paired with the output symbol x and the lapse of time between a and c must belong to the interval $[2, 4]$. In addition, the whole sequence must take a time belonging to the interval $[4, 13]$. □

Next we elaborate on the notion of the user models. A user model denotes the probability that a user chooses each input in each situation. We will use a particular case of Probability Finite State Machine (PFSM) to represent user models, for technical details see [18]. Let us note that the interaction of a user with an implementation ends whenever the user decides to stop it. Consequently, models denoting users must represent this event as well. Given a PFSM 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. We will combine the use of user models and the non-timed trace of the specification to assess to a inputs sequence the probability to be performed.

III. Genetic algorithm

In this section we present the notion of a genetic algorithm, and how we can make use of this paradigm, such a feasible way, to generate a new set invariants. A genetic algorithm [23] is a search technique used in computing to find exact or approximate solutions to optimization and search problems. The *evolution* usually starts from a *population* of randomly generated individuals and happens in *generations*. Thus, in each generation of the algorithm, the fitness of every individual in the population is evaluated. Then multiple individuals are stochastically selected from the current population, and modified to form a new population. Next, the new population is then used in the next iteration of the algorithm. Commonly, the algorithm terminates when either a maximum number of generations has been produced, or a satisfactory fitness level has been reached for the population.

According to this scheme, next we present the structure of the rest of this section. In Section III-A is presented the initialization of the set of invariants. After that in Section III-B we present how we compare a set of invariants. Next, in Section III-C the reproduction process is described, and finally in Section III-D the termination of the algorithm is presented.

A. Initialization

In this section we focus how to get a initial population of invariants. Normally, this initial set contains several hundreds or thousands of possible solutions. Traditionally, this population is generating randomly, but in our approach we make use of the algorithms to extract a set of representative invariants by taking into account the user model that we have, in order to generate a representative seed of the genetic algorithm. The idea is to short a set of invariant suites, with "the same" grade of representativity.

Definition 4: Let S be the specification of a system, and let $I_1, \ldots, I_n$ be a set of implementations of this system. Let $DB_1, \ldots, DB_n$ be a set of n databases where are recorded the interactions of users with respect to $I_1, \ldots, I_n$ respectively. We will define the initial population with a degree δ as:

$$\Phi = \bigcup_{i=1}^{n} \texttt{generate}(U_i, S, \delta)$$

where U_i corresponds to the user model generated from database DB_i, and `generate` is the algorithm of invariant extraction by using a user model, presented in [17]. □

B. Selection

In this section we present how we make a selection of invariants of the existing population. During each successive generation, a proportion of the existing population is selected to breed a new generation. Individual solutions are selected through a *fitness-based* process, where fitter solutions are typically more likely to be selected. In our approach we have two different ways to make the selection of the candidates from the initial state:

1) The first way is to generate a set randomly, thus it helps keep the diversity of the population large, preventing premature convergence on poor solutions.
2) The second way is focused on a tournament selection. In [24] we presented a heuristic approach to compare different invariants suites. This approach is based in mutation testing techniques, and the invariants are classified with respect to the probability to detect an erroneous behaviour in different environments.

Definition 5: Let S be the specification of a system, and let $I_1, \ldots, I_n$ be a set of implementations of this system. Let $DB_1, \ldots, DB_n$ be a set of n databases where are recorded the interactions of users with respect to $I_1, \ldots, I_n$ respectively. Let $\mathcal{U} = \{U_1, \ldots, U_n\}$ be the set of user models extracted from these databases respectively, and let Φ be an invariant suite. We define the `fitness` function as:

$$\texttt{fitness}_{\mathcal{U}\|S}(\Phi) = \sum_{U \in \mathcal{U}, \phi \in \Phi} \texttt{prob}_{U\|S}(\phi)$$

where $\texttt{prob}_{U\|S}(\phi)$ returns the degree of representativity to perform all the actions presented in ϕ, with respect to the parallel composition of the user model with the specification. This degree of representative is a positive real value. □

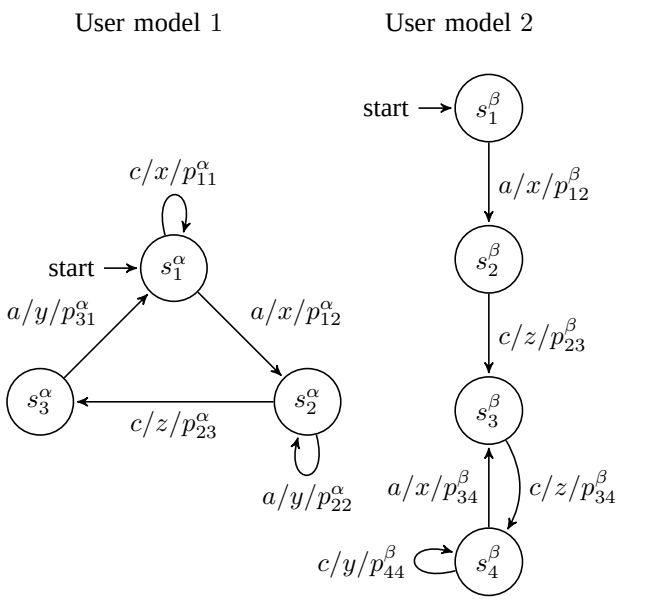

Figure 2. Example of two user models.

C. Reproduction

The next step is to generate a second generation population of solutions from those selected through the genetic operator of crossover. In some heuristics, they divide the population into a biggest new population, in order to increase their number. This approach cannot be used in our paradigm, due to the set of invariants have always to be correct with respect to the specification, and if we divide an invariant, we may obtain a set of invariants that might be correct or not.

Example 2: Next, we will present why we cannot apply all the heuristic for reproducing the set of invariants. Let us consider the specification presented in Figure 1. We have that the invariant $\phi = a/x/[2.8, 3.1], c \mapsto \{z\}/[3.9, 4, 1] \rhd [6.8, 7.4]$ is a correct invariant for this specification. It means that always we observe the input a followed by the output x with a time value belonging to $[2.8, 3.1]$, then if we observe the input action c we have to observe the output z, follows with a time value that belong to $[3.9, 4, 1]$, and the global sum of time values, it means, from a to z belongs to $[6.8, 7.4]$.

Let $\phi_1 = a \mapsto \{x\}/[2.8, 3.1] \rhd [2.8, 3.1]$ be a reduction of ϕ. We have that ϕ_1 is not correct with respect to the specification. Let us note that this invariant contradicts what is represented in the specification. It means that always that we observe the input a we have to observe the output x, followed by a time value that belongs to the inteval $[2.8, 3.1]$. Taking into account the specification, we are able to produce the following transitions $s_2 \xrightarrow{a/y/6} s_2$ or $s_3 \xrightarrow{a/y/2} s_1$ or $s_4 \xrightarrow{a/x/y} s_1$ where this invariant is not correct. □

The idea of reproduction is being able to represent user behaviours that were not presented into the user models that we have. Let us consider the Figure 2. In this picture are represented two users models, for the specification of the

Figure 1. Let us consider the user model 1. From the first state, it means s_1^α, it simulates that users have three different possibilities. The first one is that they can perform the input c, with probability p_{11}^α returning the system the output z; the second one corresponds that they can perform the input a with probability p_{12}^α returning the system the output x, or the probability to stop in state s_1^α, that corresponds to $1-p_{11}^\alpha-p_{12}^\alpha$. Taking into account the user model presented with number 1, the probability associated to perform the input actions $\langle c, c, a\rangle$ is $p = p_{11}^\alpha \cdot p_{11}^\alpha \cdot p_{12}^\alpha \cdot (1 - (p_{23}^\alpha + p_{22}^\alpha))$. An invariant that checks this property is $\phi = c/x/[3.9, 4.1], c/x/[3.9, 4.1], a \mapsto \{x\}[2.9, 3.1] \rhd [10, 12]$, so the idea is that this invariant has associated p of degree of confidence. Let us note that previous invariant has a degree of representation of 0 with respect to the second user model, the philosophical ideal of this corresponds to never the sequence $\langle c, c, a\rangle$ has been performed in this second SUT.

Next, we present our crossover operation. The idea is to divide an invariant into its prefix, and combine them with other invariant by using the $\star$ operation. This wild-card represents any, possible empty, sequence of inputs and outputs. Intuitively, by taking into account the Figure 2, we want to represent that it might be a new user that starts as the users represented by the first model, and she finishes as the behaviour presented by the second model.

Definition 6: Let ϕ_1 and ϕ_2 be two invariants, and S be the specification of the system. We define the `cross` operation of ϕ_1 and ϕ_2 as:

$$\texttt{cross}(\phi_1, \phi_2, S) = \texttt{cross}'(\phi_1, \phi_2, S) \bigcup \texttt{cross}'(\phi_2, \phi_1, S)$$

$$\texttt{cross}'(\phi_1, \phi_2, S) = \bigcup_{\phi \in \texttt{prefix}(\phi_1)} \texttt{c}_S(\phi, \star/[0, \infty], \texttt{mult}(\phi_2))$$

where `prefix` will return the set of all prefix of an invariant, and `mult` will change the last time constrain of a invariant, by means, the one represented after the symbol $\rhd$ with $[0, \infty]$, and the function `c` returns $\phi' = (\phi, \star/[0, \infty], \texttt{mult}(\phi_2))$ if this invariant is correct with respect to S, otherwise it will return the empty set. □

Example 3: Let us consider the following two user sequences $\langle c, c, c\rangle$ and $\langle a, c, c\rangle$ performed from the user models presented in Figure 2. With the previous definition, we are able to represent a new invariant suite for checking the following behavours: $\langle c, a, c, c\rangle$, $\langle c, c, a, c, c\rangle$, $\langle c, c, c, a, c, c\rangle$, $\langle a, c, c, c\rangle$, $\langle a, c, c, c, c\rangle$, and $\langle a, c, c, c, c, c\rangle$ respectively.

In a possible user model generated from the combination of these models, we will have that the probability of checking the sequence, $\langle c, c, c\rangle$ is $p_{11}^\alpha \cdot p_{11}^\alpha \cdot p_{11}^\alpha \cdot (1 - (p_{11}^\alpha + p_{12}^\alpha))$, and the probability of checking the sequence $\langle a, c, c, c, c, c\rangle$ is $p_{12}^\beta \cdot p_{23}^\beta \cdot p_{34}^\beta \cdot p_{11}^\alpha \cdot p_{11}^\alpha \cdot p_{11}^\alpha \cdot (1 - (p_{11}^\alpha + p_{12}^\alpha))$. □

Due to the following property of the real numbers, we suggest for our heuristic crossover function do not use the complete set of suffix of an invariant to generate a new one. Let us consider $0 \leq a \leq b \leq 1$ be two real values. We will have always that $a \cdot b \leq a$ and $a \cdot b \leq b$. We denote with this property that adding new values to check the correctness of the system, by means, adding new probability values to the chain of input sequences to generate the invariant, will reduce the degree of representativity of the new invariant with respect to its fathers. So, just to solve this problem, we suggest not to use the complete set of prefix of an invariant, but also the prefixes composed within γ values.

Definition 7: Let γ be a degree of representativity of the invariant. We redefine the function `cross`$'$ as:

$$\texttt{cross}'(\phi_1, \phi_2, S) = \bigcup_{\phi \in \texttt{prefix}_\gamma(\phi_1)} \texttt{c}_S(\phi, \star/[0, \infty], \texttt{mult}(\phi_2))$$

where $\texttt{prefix}_\gamma(\phi_1)$ represents the set of prefix with freedom grade γ. □

Example 4: Let us consider the previous users sequences, $\langle c, c, c\rangle$ and $\langle a, c, c\rangle$. If want to represent the behaviours of the invariant suite after the crossover with a degree of 2, then we are able to check the following behavours: $\langle c, a, c, c\rangle$, $\langle c, c, a, c, c\rangle$, $\langle c, c, \ldots, a, c, c\rangle$, $\langle a, c, c, c\rangle$, $\langle a, c, c, c, c\rangle$, $\langle a, c, \ldots, c, c, c, c\rangle$ and $\langle c, c, \ldots, c, c, c, c\rangle$ respectively. We remark that the probability to perform the token $\ldots$ is 1 always because it represents any (possible empty) sequence. □

For each new solution to be produced, a pair of parent solutions is selected for breeding from the pool selected previously. By producing a *child* solution using the above method of crossover, a new solution is created which typically shares many of the characteristics of its *parents*. New parents are selected for each new child, and the process continues until a new population of solutions of appropriate size is generated. These processes ultimately result in the next generation population of chromosomes that is different from the initial generation. Generally the average fitness will have increased by this procedure for the population, since only the best invariants from the first generation are selected for breeding, along with a small proportion of less fit solutions.

D. Termination

This process of generating a new set of invariants is repeated until a termination condition has been reached. Common terminating conditions in our approach are on the one hand, that the invariant suites that is found satisfies a minimum criteria provides by a tester. This criteria can be expressed for example by a fixed number of generations reached. Sometimes, in other environments, we may consider that the access to the user models are restrictive, it means, it has allocated a money budget to access to their information, so that, in this case, we may consider to accept a maximum access to this user models, until our amount of money have just finished. The last terminating condition corresponds to let the tester to check the invariant extraction process, and letting she to stop at any moment.

To finalize, next we present the basic steps of our genetic algorithm in pseudocode:

- Choose the initial population of invariants (with δ degree of representativity).
- Evaluate the fitness of each invariant in that population.

- Repeat on this generation until *termination* (different tester criteria):
  - Select the best-fit individuals for reproduction (by using the definition of `fitness` function).
  - Breed new individuals through crossover to give birth to offspring (using a degree γ of suffix provides by the tester).
  - Evaluate the individual fitness of new invariants.
  - Replace least-fit population with new invariants.

IV. Conclusions and future work

In this paper we have presented a novel approach to generate a set of invariants to tests systems. These invariants have been generated by using a genetic algorithm. The initial populations of this approach were obtained by applying the algorithm of invariants extraction with user models presented in [16], [17]. We showed how we can simulate new users that are not contained into these models, and to check these properties.

Genetic algorithms are an artificial intelligent technique that uses metaphors of mechanisms present in nature for organism to develop, adapt and reproduce, to try to adapt to the system in which they act and live. Its main operators are mutation, reproduction (genetic crossing) and selection of the fittest individuals. Genetic algorithms are a good method to use with black-box testing since if we do not have any information regarding the internal structure of the SUT, then the testing problem can be expressed as a search in the space of solutions, guided by a heuristic. In our case, the space of solutions is the set of all possible implementations that may have generated, located in different places, from the given specification. Genetic algorithms can adapt themselves to find this optimum, as long as the fitness function is correctly defined.

As future work, we would like to compare this approach with others existing in this field, in order to compare the set of invariants that we get. Next, we present the techniques that we would like to compare. The first one is Ant colony optimization which uses many ants to traverse the solution space and find locally productive areas. Another approach is adding an entropy with respect to an invariant in order to use a cross-entropy method to generate candidates solutions via a parameterized probability distribution. The last approach to investigate is the cultural algorithm, which consists of the population component almost identical to that of the genetic algorithm and, in addition, a knowledge component called the belief space.

References

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

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

[3] 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.

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

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

[6] I. Rodríguez, M. Merayo, and M. Núñez, "HOTL: Hypotheses and observations testing logic," *Journal of Logic and Algebraic Programming*, vol. 74, no. 2, pp. 57–93, 2008.

[7] 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.

[8] I. Rodríguez, "A general testability theory," in *20th Int. Conf. on Concurrency Theory, CONCUR'09, LNCS 5710*, pp. 572–586, Springer, 2009.

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

[10] J. Springintveld, F. Vaandrager, and P. D'Argenio, "Testing timed automata," *Theoretical Computer Science*, vol. 254, no. 1-2, pp. 225–257, 2001. Previously appeared as Technical Report CTIT-97-17, University of Twente, 1997.

[11] A. En-Nouaary, R. Dssouli, and F. Khendek, "Timed Wp-method: Testing real time systems," *IEEE Transactions on Software Engineering*, vol. 28, no. 11, pp. 1024–1039, 2002.

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

[13] Y. Wang, M. Uyar, S. Batth, and M. Fecko, "Fault masking by multiple timing faults in timed EFSM models," *Computer Networks*, vol. 53, no. 5, pp. 596–612, 2009.

[14] 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*, pp. 418–427, Springer, 2008.

[15] C. Andrés, M. 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*, pp. 67–76, IEEE Computer Society Press, 2009.

[16] C. Andrés, M. Merayo, and M. Núñez, "Using a mining frequency patterns model to automate passive testing of real-time systems," in *21st Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'09*, pp. 426–431, Knowledge Systems Institute, 2009.

[17] C. Andrés, M. Merayo, and M. Núñez, "Supporting the extraction of timed properties for passive testing by using probabilistic user models," in *9th Int. Conf. on Quality Software, QSIC'09*, pp. 145–154, IEEE Computer Society Press, 2009.

[18] 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.

[19] D. Fatiregun, M. Harman, and R. M. Hierons, "Evolving transformation sequences using genetic algorithms," in *4th IEEE Int. Workshop on Source Code Analysis and Manipulation, SCAM'04*, pp. 65–74, IEEE Computer Society Press, 2004.

[20] Q. Guo, R. M. Hierons, M. Harman, and K. Derderian, "Computing unique input/ouput sequences using genetic algorithms," in *3rd Int. Workshop on Formal Approaches to Software Testing, FATES'03, LNCS 2931*, pp. 169–184, Springer, 2004.

[21] D. Berndt and A. Watkins, "High volume software testing using genetic algorithms," in *38th Annual Hawaii Int. Conf. on System Sciences, HICSS'05*, p. 318b, IEEE Computer Society Press, 2005.

[22] K. Derderian, M. Merayo, R. Hierons, and M. Núñez, "Aiding test case generation in temporally constrained state based systems using genetic algorithms," in *10th Int. Conf. on Artificial Neural Networks, IWANN'09, LNCS 5517*, pp. 327–334, Springer, 2009.

[23] D. Goldberg, *Genetic Algorithms in Search, Optimisation and Machine Learning*. Addison-Wesley, 1989.

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