Coverage-Based Grammar-Guided Genetic Programming Generation of Test Suites

Alfredo Ibias
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
aibias@ucm.es

Pablo Vazquez-Gomis
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
pavazq01@ucm.es

Miguel Benito-Parejo
DTRS research group
Universidad Complutense de Madrid
28040, Madrid, Spain
mibeni01@ucm.es

***Abstract*—Software testing is fundamental to ensure the reliability of software. To properly test software, it is critical to generate test suites with high fault finding ability. We propose a new method to generate such test suites: a coverage-based grammar-guide genetic programming algorithm. This evolutionary computation based method allows us to generate test suites that conform with respect to a specification of the system under test using the coverage of such test suites as a guide. We considered scenarios for both black-box testing and white-box testing, depending on the different criteria we work with at each situation. Our experiments show that our proposed method outperforms other baseline methods, both in performance and execution time.**

***Index Terms*—Genetic Programming, Coverage, Software Testing**

I. Introduction

Testing software is a fundamental step on every software development process. The goal is to improve the quality of the software searching for faults in it, using as few resources as possible. However, due to its criticality, testing can cost more than 50% of the development budget [30]. Therefore, good, cheap and effective methods for testing software are fundamental for the software product cycle. One of the main techniques in the software testing field [1], [30] is to try to find faults through the execution of input sequences and comparing the outputs obtained with the expected ones. The combination of the input sequence and the expected outputs is a test suite, and the goal of the method is to use the test suites with higher fault finding ability. In order to generate such test suites, the most widely known approach is *mutation testing* [22], [32], which uses *mutants* (i.e. modified versions) of the System Under Test (SUT) (or more usually, its specification) to generate such test suites. However, one of the main issues with this method is its computational cost. More feasible approaches focus on finding good test suites with respect to a chosen criteria, reducing the computational and resources costs but diminishing at the same time the effectiveness of the method. In this paper we present one of such approaches.

This work has been supported by the Spanish MINECO/FEDER project FAME (RTI2018-093608-B-C31); the Region of Madrid project FORTE-CM (S2018/TCS-4314) co-funded by EIE Funds of the European Union; the Region of Madrid - Complutense University of Madrid (grant number PR65/19-22452); and the Santander – Complutense University of Madrid (grant number CT63/19-CT64/19).

Another problem of software testing is the so called *Oracle problem* [2], [25]. This problem focuses on how to decide that a SUT is correct or not given the obtained outputs, and for extension, on how well a test suite detects a fault in the program. For this task a lot of approaches had been tried, from classical deterministic solutions [7], [11] to more evolved ones, like those based on genetic algorithms [3], [4], [31]. In this paper we use Finite State Machines (FSMs) to represent the specification of the SUT, and we used a coverage-based criterion to decide how good the test suites are detecting faults. We also use the mutation score to compare our approach to others from the literature. Mutation score is a measure from mutation testing that calculates the percentage of mutants killed by a test suite. We say that a test kills a mutant when the mutation has been discovered when executing the test.

Evolutionary computation algorithms are a well known family of algorithms and meta-heuristics that focus on the evolution of a bunch of individual solutions to obtain an approximately optimal solution. This family ranges from the Genetic Algorithms [34] based on the evolution of the genomes to the Ant Colony Optimisation algorithm [9] based on the organisation of an ant colony, passing by the Particle Swarm Optimisation algorithm [5], [23] based on the development of a flock of birds. Genetic algorithms [34] are an approximation method to find good or nearly good solutions to computationally exponential problems. They focus on generating random solutions (called *individuals*) and improving them through the mixture and mutation of the best generated ones. To decide which individual is better they use a previously chosen criteria (called *fitness function*) that should guide the *evolution*. Genetic programming [24] is an extension of genetic algorithms that manage to deal with structured types, being able to find solutions to a wider range of problems. Specifically, genetic programming used to work with tree-like structures, which can be totally free or bounded to a grammar [27]. In our work we use a genetic programming algorithm whose individuals are bounded to a grammar that will ensure that they will conform to valid test suites of the SUT.

In this paper we present a Grammar-Guided Genetic Programming algorithm to generate test suites. This algorithm uses a coverage-based measure as a guide, and therefore obtains test suites that have a high coverage of the FSM that

models the SUT. The goal of this algorithm is to generate test suites with a high fault detection ability, and it uses coverage-based fitness functions to that end. Specifically, we use measures based on t-way coverage, which measures how many groups of t consecutive elements of the SUT are covered by the test suite. These coverage-based fitness functions are prepared for two scenarios: a white-box scenario where we have information about the internal structure of the SUT (like states) and a black-box scenario where we do not have such information and we can only observe input and outputs. We performed experiments to compare this approach to more traditional ones and to different variations of itself. Specifically, we compared our algorithm with a grammar-guided genetic programming algorithm that uses as fitness function the mutation score of the individuals. In these experiments we obtained that our solutions generated using coverage-based fitness functions were quite effective, as they took less time to be computed, while not losing a lot of (or even having better) finding faults potential. We found that the coverage criteria based on 1-way transition coverage is the preferable choice when generating test suites if we are in a black-box scenario. For a white-box scenario, the 2-way state coverage is preferred.

In this paper, Section II introduces the basic concepts over which our algorithm is developed. Section III introduces our coverage-based grammar-guided genetic programming algorithm for generating test suites. Later, Section IV presents our experiments evaluating our algorithm. In Section V we discuss some aspects of our algorithm and our experiments. Section VI evaluates the threats to the validity of our experiments. Finally, Section VII contains the conclusions of our work and lines for future work.

II. Theoretical Background

We model systems as *Finite State Machines* (FSMs). In order to define an FSM, we first introduce some notation.

Given set A, A^* denotes the set of finite sequences of elements of A; A^+ denotes the set of non-empty finite sequences of elements of A; and $\epsilon \in A^*$ denotes the empty sequence. We let $|A|$ denote the size of set A. Given a sequence $\sigma \in A^*$, $|\sigma|$ denotes its length. Given a sequence $\sigma \in A^*$ and $a \in A$, we have that σa denotes the sequence σ followed by a and $a\sigma$ denotes the sequence σ preceded by a.

Throughout this paper we let $\mathcal{I}$ be the set of input actions and $\mathcal{O}$ be the set of output actions. It is important to differentiate between input actions and *inputs* of the system. An input of a system will be a non-empty sequence of input actions, that is, an element of $\mathcal{I}^+$ (similarly for outputs and output actions).

An FSM is a (finite) labelled transition system in which every transitions is labelled with an *input/output pair* (a pair containing an input action and an output action). We use this formalism to define specifications.

Definition 1: A *Finite State Machine* (FSM) is represented by a tuple $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ in which Q is a finite set of states, $q_{in} \in Q$ is the initial state, $\mathcal{I}$ is a finite set of input actions, $\mathcal{O}$ is a finite set of output actions, and $T \subseteq Q \times (\mathcal{I} \times \mathcal{O}) \times Q$ is the transition relation. The meaning of a *transition* $(q, (i, o), q') \in T$, also denoted by $(q, i/o, q')$, is that if M receives input action i when in state q then it can move to state q' and produce output action o.

We say that M is *deterministic* if for all $q \in Q$ and $i \in \mathcal{I}$ there exists at most one pair $(q', o) \in Q \times \mathcal{O}$ such that $(q, i/o, q') \in T$.

We assume that FSMs are deterministic. This simplifies our scenario so we can use genetic algorithms, without losing applicability. However, our algorithm can be used on non-deterministic FSMs with some adaptations to the specification. An FSM can be represented by a diagram in which nodes represent states and transitions are represented by arcs between the nodes. In our case, all states are final as long as they are reachable from the initial state.

We will assume the *minimal test hypothesis* [21]: the SUT can be modelled as an (unknown) object described in the same formalism as the specification (here, an FSM).

We say that a mutant $M' = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T')$ of an FSM $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ is another FSM such that T and T' only differ in one transition in the form of $(q, (i, o), q') \in T$ and $(q, (i, o), q'') \in T'$ with $q' \neq q''$.

Our main goal while testing is to decide whether the behaviour of an SUT conforms to the specification of the system that we would like to build. In order to detect differences between specifications and SUTs, we need to compare their behaviours, and the main notion to define such behaviours is given by the concept of a *trace*.

Definition 2: Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an FSM, $\sigma = (i_1, o_1) \dots (i_k, o_k) \in (\mathcal{I} \times \mathcal{O})^*$ be a sequence of pairs and $q \in Q$ be a state. We say that M can perform σ from q if there exist states $q_1 \dots q_k \in Q$ such that for all $1 \leq j \leq k$ we have $(q_{j-1}, i_j/o_j, q_j) \in T$, where $q_0 = q$. If $q = q_{in}$ then we say that σ is a *trace* of M. We denote by $\texttt{traces}(M)$ the set of traces of M. Note that $\epsilon \in \texttt{traces}(M)$ for every FSM M.

Next we define the notion of test. As previously explained, a test is a sequence of (input action, output action) pairs. A test suite will be a set of tests.

Definition 3: Let $M = (Q, q_{in}, \mathcal{I}, \mathcal{O}, T)$ be an FSM. We say that $t = (i_1, o_1) \dots (i_k, o_k) \in (\mathcal{I} \times \mathcal{O})^+$ is a *test* for M if $t \in \texttt{traces}(M)$. The *length* of t is the length of the sequence, that is, $|t| = k$. In addition, the sequence of input actions of t is $\lambda = i_1 \dots i_k$ and the sequence of output actions of t is $\mu = o_1 \dots o_k$. We will sometimes use the notation $t = (\lambda, \mu) \in (\mathcal{I}^+ \times \mathcal{O}^+)$. We write $(i, o) \in t$ to denote that the pair (i, o) appears in the test t; $(i, o) \in_n t$ denotes that the pair (i, o) appears n times in the test t.

A *test suite* for M is a set of tests for M. Given a test suite $\mathcal{T} = \{t_1, \dots, t_n\}$, the *length* of the test suite is the sum of the lengths of its tests, that is, $|\mathcal{T}| = \sum_{i=1,\dots,n} |t_i|$.

Let $t = (\lambda, \mu)$ be a test for M. We say that the application of t to an FSM M' fails if there exists μ' such that $(\lambda, \mu') \in \texttt{traces}(M')$ and $\mu \neq \mu'$. Similarly, let $\mathcal{T}$ be a test suite for M. We say that the application of $\mathcal{T}$ to an FSM M' fails if there exists $t \in \mathcal{T}$ such that the application of t to M' fails.

The notion of coverage is quite simple in our context: it represent how much of the FSM a test suite traverses. We define three different coverage criteria based on the t-way coverage definition:

- t-way transition coverage: the percentage of sets of t consecutive transition labels that the test suite traverses. We define transition label as a pair input/output of the FSM (that is, a pair input/output that corresponds to the execution of a transition).
- t-way state coverage: the percentage of sets of t consecutive states that the test suite visits. We define state as a state of the FSM.
- t-way action coverage: the percentage of sets of t consecutive actions that the test suite executes. We define action as an input of the FSM.

We differentiate between these 3 types of coverage as they are the most widely used in the literature [26], [33]. Each coverage type focus on a different element of the FSM and therefore it is mandatory to compare between the three to see which one yields better results. In order to use state coverage we need to be aware of the internal state of the FSM, therefore it can only be considered within a white-box scenario. However, action coverage and transition coverage only needs observable information and can also be used in a black-box scenario. Finally, for a notion of coverage based in transitions (as defined in Def. 1), we require a white-box scenario as well, since the structure of the FSM is also needed. This last case will be studied in detail in section IV.

The t-way coverage groups the transition labels/states/actions of the FSM in sets with exactly t elements. These sets contains t consecutive elements, that is, there exists a sequence of transitions of the FSM such that the transition labels/states/actions involved in it are exactly the ones in the set. For example, for 1-way state coverage, the states of the FSM are group in sets with only one state, and therefore there are as many sets as states. In the case of 2-way state coverage however, the states of the FSM are grouped in sets of pairs of states. Specifically each set contains two consecutive states, that is, two states that are connected by a transition. Then, in this case there are as many sets as transitions, but not as many as combinations of two states, because if two states are not connected trough a transition, then they are not consecutive.

With the t-way sets of transition labels/states/actions of the FSM, we can check the number of them that appear (without repetitions) in a test suite. The percentage of these sets that appear in the test suite will be its t-way transition/state/action coverage score. This score represents how much coverage of the SUT this test suite will provide.

Formally, we define the t-way transition/state/action coverage score as follows.

Definition 4: Given an FSM M, a grouping G (with $|G|$ elements) of its transition labels/states/actions in sets of t consecutive elements, and a test suite that traverses s of such sets (without repetitions), the t-way transition/state/action coverage score is $\frac{s}{|G|}$.

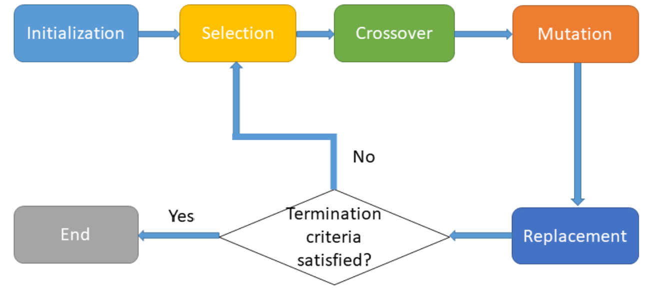

Fig. 1. GA flowchart

In our work, t ranges from 1 to 3. Along this paper we will call *G set* to the grouping of the transition labels/states/actions in sets of t consecutive elements of an FSM.

Genetic programming is a meta-heuristic that is often used to obtain good enough solutions to complicated optimisation problems. They are non-deterministic algorithms that consider multiple possible solutions or *individuals* at a time, and combine their information in order to obtain different solutions each iteration. Since the objective is to improve the final solution, a *fitness function* evaluates each individual to prioritise those possible solutions with a better score. This method is derived from the classical genetic algorithm, with some adaptations to work with tree-like structures that we use in our work. Usually, a genetic algorithm is divided in 5 steps structured as follows:

- The *initialisation* step generates the initial *population*, acting as a seed for the whole process. Such an initialisation is usually random, in order not to bias the behaviour of the algorithm.
- The *selection* step is focused on obtaining the most suited individuals to perform the following steps, and achieving a better solution in next generations.
- The *crossover* step consists of pairing the individuals obtained in the selection step, and exchanging parts of the structure within each couple.
- The *mutation* step considers each individual after the crossover step, and with a small probability performs slight variations or mutations. This process, although might seem counter-intuitive, it tends to avoid obtaining local maximum solutions, by possibly substituting the negative-impact elements of the individual for new ones.
- The *replacement* step, finally takes the current population and its offspring, and decides which individuals amongst them conform the following generation.

The idea of genetic algorithms is to iterate the process to make the population evolve and produce a better solution. Therefore, it requires a termination criteria, which usually considers a bound on the number of iterations.

It is important to note that the loop for a genetic algorithm only considers the selection, crossover, mutation and replacement steps, as only one initialisation is required. A general flowchart of genetic programming is shown in Fig. 1.

For our genetic programming algorithm we use a grammar to guide the generation of the test suites. A grammar is a set of symbols and rules that restrict the generation capabilities of the algorithm in order to ensure the correctness of the generated individuals with respect to a chosen criteria (in our case, that they are valid test suites for the SUT).

III. Coverage-based Grammar-guided Genetic Programming algorithm

In this section, we describe the specific elements of our genetic algorithm, and the structure of the steps we use, to present a global idea of our contribution in a more detailed manner.

Our proposed algorithm is a genetic programming algorithm that works with a population of individuals of the same size. Each individual is a test suite consisting of n input/outputs pairs. However, a grammar-guided genetic programming works with tree-like structures (i.e. data structures that can be abstracted as trees) that conform to a grammar. Thus, we have to adapt our test suites to be abstractly represented as grammar-guided trees. We do such a transformation defining each node of the tree as an input/output pair (i.e. each node contains an input and an output) with a grammatical symbol, then each succession of nodes would be a succession of input/output pairs, which corresponds to a test. Finally, in order to combine the tests of the test suite into a tree, we join them with a *dummy node* that would be the root of the tree. This way, each test would be a branch of the tree. It is important to note that we work with a constrained size, which all trees share, in order not to obtain huge trees that compromise the score of the results, as a bigger size in a test suite would induce a better score. Along the rest of the paper we would use input/output pair and node interchangeably, and we would do the same for test and branch, and for test suite and tree. Finally, remark that a subtree of these trees would be a section of a test.

Since we decided to use a grammar-guided genetic programming approach, we need to generate a grammar that allows to generate test suites that conforms to the FSM. For this grammar, we define the following components [17]:

- A start non-terminal symbol S that starts the grammar.
- A non-terminal symbol TS that introduces each test of the test suite.
- A non-terminal symbol A for each state, where $A \in \mathbb{A}$ is the state number.
- A terminal symbol $'a/b'$ for each input/output pair present on the FSM, where a is the input and b is the output.
- A terminal symbol $'null'$ to represent the end of a test.
- A production rule $S \longrightarrow TS$ to generate the initial test.
- A production rule $TS \longrightarrow TS + TS$ to introduce a new test.
- A production rule $TS \longrightarrow\ 0$ to start each test in the FSM initial state.
- A production rule $A \longrightarrow\ 'a/b' + A$ for each transition from the left hand side state A to the right hand side state A with input/output pair (a, b).
- A production rule $A \longrightarrow' null'$ for each state A to a terminal to represent the end of the test.

With this components, we will produce the nodes of the trees that represent test suites.

In the *initialisation* step, we generate random test suites, avoiding duplicated tests. The idea is to generate an initial population as diverse as possible, in order to have a wider spectrum to explore with the following steps.

To evaluate and compare the individuals of our population, we need a fitness function. As the main contribution of our work, we propose several coverage-based fitness functions, considering the notion of coverage defined in Def. 4. These fitness functions are built over coverage criteria to determine which test suites have a higher fault finding capability. We developed 12 fitness functions using t-way coverage criterion. In particular, we considered transition, state, action t-way coverage, and an extension of transition t-way coverage that includes the initial and final states of each transition (i.e. it considered the actual transitions). We instantiated these criteria with $t \in \{1, 2, 3\}$. We will later on compare the results that we obtain with each of these fitness functions.

In the selection step, we aim at an exploration goal, rather than an exploitation one. In that sense, we consider that our selection method should take into account the whole population, in order not to lose any genetic diversity. Therefore, our selection method simply matches pairs of individuals for the following steps.

For the grammar-guided crossover, we implemented a variation on the standard crossover, where we select a random node from each parent such that both nodes have the same grammatical symbol. Then, we exchange the selected subtrees while maintaining the grammatical correctness of the whole tree. This means that the resulting individuals still represent a valid test suite for the given SUT. However, in order to keep the length of the test suite (note that such length is the sum of the length of each test), we will have to modify the resulting trees accordingly. We have two different scenarios: first, we have to extend test suites that provide a longer subtree while receiving a shorter one; and second we have to bound test suites that receive a longer subtree than the one they provide. This is necessary to control the length of a test suite, avoiding an incremental increase of the size of the individuals. The reason to keep the length invariant is to have a reasonable comparison between the solutions, as a bigger test suite would produce inherently a better score.

We perform the extension of a subtree by randomly generating a grammatically valid continuation of the subtree. That means that we expand from the leaf of the subtree generating new nodes (i.e. input/output pairs) until reaching the adequate length. In the case that we are unable to extend such subtree up to the desired length, we generate a new random test to substitute the remaining nodes. For the bound on larger subtrees, we simply eliminate its final nodes, in order for the test suite to match the required length. Due to this variation of the standard crossover, in most cases we add a

little extra genetic information apart from the one belonging to the parents.

Our grammar-guided mutation step consists in randomly replacing tests of the test suite for newly generated ones. The idea of this mutation, is to incorporate different tests that have not been considered before for the test suite. With this procedure, we add new genetic information (previously unseen) to the individual and avoid reaching local optimum. We do not consider a mutation on the test suite as such, but on each test in the test suite. Since each test is represented by a branch of the tree, we remove such test by deleting the branch, and we include the new test by adding a new branch to the *dummy root node*. It is important to note that a test is either fully removed, or is not modified, as we do not interfere with partial branches of the tree.

Finally, the last step of our loop combines the best elements of both the parent and the offspring populations, to prepare the final individuals (either for following iterations, or the end of the process). We divided this step in several phases, starting by obtaining the average score for the offspring population. With it, we automatically consider the individuals that improve such average value to be in the final population. Next, for the remaining individuals, which have a worse score than the average, there is a probability for them to be maintained. To finish, in order to complete the size of the population, that should remain invariant, we randomly select among the best individuals from the parent generation.

The termination criteria that we consider is to perform 100 iterations. However, if during the execution we find that the last 20% of iterations do not improve the score, we terminate the process. We added such extra condition as we want to avoid an excessive amount of iterations and computing power that will not yield a better result.

IV. Experiments

The experiments we performed aimed to compare our algorithm with another test suite generation algorithm based on mutation score. For these experiments we took as experiment subjects a set of 100 randomly generated `FSM`s, each one with 100 states. In them, each state has between 5 and 50 outgoing transitions, each one with a label conformed by an input/output pair. These pairs are generated from both input and output alphabets, each one with 50 elements. This set of `FSM` is generated with the idea of stretching the capabilities of the proposed algorithm, being big enough `SUT`s and trying to be as representative of real-world applications as possible.

The idea of the experiments was to compare our proposed algorithm (*coverage algorithm*) to a genetic programming algorithm whose fitness function is the mutation score (*mutation score algorithm*). *Mutation score algorithm* uses the same grammar-guided genetic programming algorithm as *coverage algorithm*, the only real change is in the fitness function, and therefore in the obtained solutions. Specifically, it generates a new set of 10 random mutants each iteration to calculate the mutant score of the individuals of the population. In these experiments each individual of the population represents a test suite of fixed length 1000, and therefore the solutions will be test suites for the given `FSM`. Then, we compare the solution from both algorithms using mutation score, as it is a well established method to compare test suites [32].

The experiments were developed as follows: for each `FSM` we generated two test suites (one using *coverage algorithm* and other using *mutation score algorithm*). Then, we generated 100 mutants of the `FSM` and computed the mutation score of each test suite. We repeated this procedure for each of the 100 `FSM`s from the experiment subjects set, and we computed the average values for all the `FSM`s. Finally, we performed this whole experiment 10 times to obtain a final mean value that is less prone to the randomisation influence. We display these final averages on Table I, where:

- The *Success* columns indicate the percentage of times where *coverage algorithm* performed better than the *mutation score algorithm* (and vice-versa).
- The *Mutants Killed* columns indicate the percentage of mutants each algorithm was able to kill.
- The *Execution Time* columns indicate the time that each algorithm took to run.

The results of the experiments are displayed on Table I. There we can see that the three of t-way transition coverage, t-way state coverage and t-way action coverage beat mutation score to a extent, both in fault finding capability and in computation time. Moreover, we can observe an interesting phenomena: the mutation score of the different coverage notions is not uniform with respect to the value of t. This implies that a greater t does not imply that the resulting test suite will have a higher mutation score.

From these results arises the question of what would the situation be if we computed the t-way transition coverage using transitions instead of transition labels. We call this notion t-way extended transition coverage. It is obvious that the transitions have more information about the underlying `FSM` and its structure, so they could perform better, while not needing a lot of extra time as they follow a similar concept of coverage. However, the transitions can only be generated if we have an oracle (an `FSM` representing the `SUT`) or we are in a white-box testing scenario, which limits its applicability. We repeated the experiment using the transitions and obtained the results displayed on Table II. As we can observe there, the results are in the line of the ones from the other coverage types, without a huge difference in computation time. Therefore, we can conclude that the extra requirements are not worth it.

Finally, we performed a statistical hypothesis test over all the results. The null hypothesis was that the coverage-based fitness functions and the mutation score fitness function gave similar results, that is, both produced test suites with similar mutation score. We applied a one-way ANOVA test[1] where we tested whether the values of both fitness functions were, on average, similar. Then, we computed the p-value for the experiments. Here, we observed an interesting situation: for

[1]Note that we could use the ANOVA test because we performed an homogeneity of variance check and it raised a positive result.

TABLE I
RESULTS OF THE COMPARISON WITH MUTATION SCORE.

Coverage Type	Success Coverage (Percentage)	Success Mutation Score (Percentage)	Mutants Killed Coverage (Percentage)	Mutants Killed Mutation Score (Percentage)	Execution Time Coverage (Seconds)	Execution Time Mutation Score (Seconds)
$1-way\ transition\ coverage$	0.5644	0.4356	0.3781	0.3660	31.0099	83.9273
$2-way\ transition\ coverage$	0.5585	0.4415	0.3738	0.3652	29.1904	84.5565
$3-way\ transition\ coverage$	0.5131	0.4869	0.3673	0.3660	9.8079	84.2539
$1-way\ state\ coverage$	0.4915	0.5085	0.3660	0.3666	6.8651	85.6319
$2-way\ state\ coverage$	0.6167	0.3833	0.3827	0.3633	31.7231	86.0107
$3-way\ state\ coverage$	0.5452	0.4548	0.3734	0.3636	31.2126	86.5856
$1-way\ action\ coverage$	0.4995	0.5005	0.3663	0.3670	6.8224	85.9126
$2-way\ action\ coverage$	0.5074	0.4926	0.3672	0.3654	31.4185	85.9683
$3-way\ action\ coverage$	0.4995	0.5005	0.3684	0.3667	10.0687	86.1290

TABLE II
RESULTS OF THE COMPARISON WITH MUTATION SCORE (USING TRANSITIONS).

Coverage Type	Success Coverage (Percentage)	Success Mutation Score (Percentage)	Mutants Killed Coverage (Percentage)	Mutants Killed Mutation Score (Percentage)	Execution Time Coverage (Seconds)	Execution Time Mutation Score (Seconds)
$1-way\ extended\ transition\ coverage$	0.6028	0.3972	0.3823	0.3663	31.2724	87.5435
$2-way\ extended\ transition\ coverage$	0.5458	0.4542	0.3722	0.3651	29.0016	86.3313
$3-way\ extended\ transition\ coverage$	0.4973	0.5027	0.3674	0.3656	10.1902	86.2759

1-way and 2-way transition and extended transition coverage, and for 2-way and 3-way state coverage the obtained p-values were lower than 0.05. However, for the other measures the p-values were higher than 0.05. Therefore, we can deny the null hypothesis for the experiments with the first coverage notions with a confidence higher than 0.95, and we have to accept the null hypothesis for the experiments with the second set of coverage notions. In order to double-check our results, we performed a t-test and obtained the same p-values.

We can conclude that some of our coverage-based fitness functions are better than a mutation score function, both regarding performance and computation time. However, there is something more that we have to discuss. More precisely, we detected that the computation time in some experiments was lower by a huge margin than for others. After a careful analysis, we concluded that this behaviour was produced by the different orders of magnitude between the different G sets. For example, the number of sets of 3-way transitions, actions and extended transitions are so huge, that with a test suite of length 1000 the variation on percentage between two test suites is almost null. Alternatively, we have that the number of sets of 1-way states and actions are very small (in fact, these numbers are 100 and 50 in our experimental subjects, respectively) and therefore with test suites of length 1000 is really easy to cover all the sets, obtaining a coverage of 100%. It is important to note that these two groups correspond to the ones that confirmed the null hypothesis. This lack of improvement (either because we reached the 100%, or the improvement is negligible) triggers the second condition of the termination criteria, stopping the execution before the 100 iterations are produced, and therefore reducing the execution time. This behaviour shows that not all the coverage notions that we propose in our work are useful for the average testing practices, because some will arise pointless results.

V. DISCUSSION

During the development of our algorithm there were some critical decisions we had to make and that can arise some questions from an experienced reader. Therefore, in this section we will address such decisions. We discuss in detail our decision of using only one kind of coverage measures and our decision of using a genetic algorithm with mutation score as fitness function as a baseline algorithm for comparing with our proposed algorithm. Finally, we also address the decision of which crossover to use for our genetic algorithm.

A. Coverage measures choice

In traditional coverage-based literature there are two kind of coverage metrics: the first one includes in the coverage metric all the elements a test traverses, and the second one only includes the last element (or set of elements) of each test. We decided to stick to the first kind of coverage metrics and forget about the second one because the characteristics of

the first kind would help better to the evolution of the genetic algorithm. We chose the first kind of coverage metrics because it is easier to obtain different scores for two test suites with the same amount of tests, while the second kind of coverage metrics would yield the same score for those two test suites (if there are no repetitions). For example, let us say we have two test suites with only one test: the test of the first test suite has 17 input/output pairs and the test of the second test suite has 5 input/output pairs. Let us say also consider the 2-way action coverage. Then, if we use the first kind of metrics the first test suite will cover 16 sets of actions and the second test suite will cover 4 sets of actions, while if we use the second kind of metrics both test suites cover only 1 set of actions. However, it is clear that we should prefer the first test suite over the second, as suggested by the first kind of coverage measure.

B. Baseline algorithm choice

Mutation score is a traditional and widely known measure used in mutation testing. It is typically used as a measure to compare different test suite generation algorithms, as it has been shown to be highly correlated with the fault finding ability of a test suite [32]. That's why we use it to compare between different coverage measures and to compare with the baseline algorithm. The choice we want to discuss here is why we used it as a fitness function of our baseline algorithm.

The reasoning behind this choice is that a genetic algorithm (in this case a grammar-guided genetic programming algorithm) whose evolution is guided by mutation score as fitness function will obtain test suites that obtain a high mutation score, and therefore, will obtain the best scores later when comparing with another algorithm using mutation score. However, such algorithm will take a lot of computation time due to the high computational cost of generating and using the bunch of mutants needed to obtain a mutation score. Therefore, this kind of algorithm should be a valid baseline with which to compare our proposed algorithm.

We could have used other state-of-the-art algorithms to which compare our algorithm, like genetic algorithms using fitness functions like Test Set Diameter [10], [17], or more classical algorithms like W [11] or Wp [7] methods. However, we considered that a genetic algorithm using mutation score as fitness function will perform better as a baseline measure. Anyway, the comparison with these other methods would be matter of future work, in order to ensure the suitability of our proposed algorithm.

C. Crossover choice

Concerning the crossover selected for our proposed grammar-guided genetic programming algorithm, our choice arises some concern due to its capability to include new (previously unseen) genetic information into the offspring. We chose this crossover because we wanted a crossover with two clear restrictions: first, the offspring had to be grammatically valid, and second the offspring had to keep the same length than its parents. Then, the spectrum of options that we had available was limited. In fact, we could only find two crossovers that conform to those restrictions.

The first crossover we considered was simpler but also harder to produce: it consisted in finding the same grammatical symbol with the same length to the leaf in both trees. This situation happens very rarely and therefore the amount of crossovers produced was less than optimal (even when trying to produce the crossover for each pair of trees). This crossover conformed to the required restrictions because the length was maintained through the interchange of subtrees with the same number of nodes, and the grammatical correctness was ensured because both subtrees started with the same grammatical symbol. However, it obtained worse results than our selected crossover.

The second crossover was our selected crossover. It is a bit more complex but at the same time easier to occur: it consisted in finding the same grammatical symbol at both trees, exchanging the subtrees that start at such symbols, and then solving the possible problems with the length of the offspring. It is in this last step where new genetic information was generated in order to extend the shorter tree into the desired length. This crossover conforms to the required restrictions because the grammatical correctness was ensured due to both subtrees starting with the same grammatical symbol, and the length being maintained by generating or bounding the offspring. This crossover does not have problems of incapability to be produced and therefore it obtains better results than other options.

VI. Threats to Validity

In this section we briefly discuss some of the possible threats to the validity of the results of our experiments. Concerning threats to *internal validity* (results validity), the main threat is associated with the possible faults in the developed experiments because they could lead to misleading results. In order to reduce the impact of this threat we tested our code with carefully constructed examples for which we could manually check the results. In addition, we repeated the experiments many times in order to get a mean so that the randomisation impact is reduced. Finally, there is the choice of baseline measure, that if poorly chosen can arise better results than the real ones. This concern is discussed in Section V and we consider it properly addressed.

The main threat to *external validity* (results generalisation), is the different possible systems to which we could apply our algorithm. Such a threat cannot be entirely addressed since the population of possible systems is unknown and it is not possible to sample from this (unknown) population. In order to diminish this risk, we perform our experiments over randomly generated FSMs prepared to stretch the capabilities of the algorithm.

Finally, we considered threats to *construct validity* (experiments *reality*), that is, whether our experiments reflect real-world situations or not. In our work, the main construct threat is what would happen if we used our algorithm with much more complex methods. In order to address this concern, we

have performed experiments with huge randomly generated `FSM`s, but there is still room for improvement and it will be matter of future work.

VII. Conclusions

Generating test suites with a high fault finding ability is fundamental for ensuring the quality of software. Moreover, performing this task in a quick and efficient manner is critical for software budgets. In this paper we have presented a grammar-guided Genetic Programming algorithm to generate such tests suites, based on coverage criteria fitness functions. We compared our proposal with another method that happens to be worst than ours. We also found that 1-way transition coverage is the preferable choice when generating test suites if we are in a black-box scenario, and 2-way state coverage if we are in a white-box scenario.

For future work, we would like to explore a wider range of coverage notions, specifically t-way coverage with $t > 3$. Evolving over this line of work, we would like to explore the significance of the relation between the size of the G set and the length of the test suites. We would like to deal with bigger numbers of mutants. Therefore, we would like to include recent work on producing and managing big sets of mutants [6], [8], [12], [13] into our framework. We would also like to explore the possibility of using another kind of evolutionary computation algorithms, instead of a genetic programming algorithm, like tree swarm optimisation [14], [15]. In another line of work, we would like to use our recent work on testing using Information Theory concepts [18]–[20] to implement genetic algorithms using the induced measures as fitness functions. Finally, we would like to extend our framework to deal with `FSM`s that can represent systems with distributed components [16], [28], [29].

References

[1] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2nd edition, 2017.

[2] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.

[3] M. Benito-Parejo, I. Medina-Bulo, M. G. Merayo, and M. Núñez. Using genetic algorithms to generate test suites for FSMs. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 741–752. Springer, 2019.

[4] M. Benito-Parejo and M. G. Merayo. An evolutionary algorithm for selection of test cases. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24535: 1–8. IEEE Computer Society, 2020.

[5] C. Blum and D. Merkle, editors. *Swarm Intelligence: Introduction and Applications*. Springer, 2008.

[6] P. C. Cañizares, A. Núñez, and M. G. Merayo. Mutomvo: Mutation testing framework for simulated cloud and HPC environments. *Journal of Systems and Software*, 143:187–207, 2018.

[7] T. S. Chow. Testing software design modeled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[8] P. Delgado-Pérez and I. Medina-Bulo. Search-based mutant selection for efficient test suite improvement: Evaluation and results. *Information and Software Technology*, 104:130–143, 2018.

[9] M. Dorigo, V. Maniezzo, and A. Colorni. The ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man and Cybernetics B*, 26(1):29–41, 1996.

[10] R. Feldt, S. M. Poulding, D. Clark, and S. Yoo. Test set diameter: Quantifying the diversity of sets of test cases. In *9th IEEE Int. Conf. on Software Testing, Verification and Validation, ICST'16*, pages 223–233. IEEE Computer Society, 2016.

[11] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test selection based on finite-state models. *IEEE Transactions on Software Engineering*, 17(6):591–603, 1991.

[12] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. A tool for domain-independent model mutation. *Science of Computer Programming*, 163:85–92, 2018.

[13] P. Gómez-Abajo, E. Guerra, J. de Lara, and M. G. Merayo. Wodel-Test: a model-based framework for language-independent mutation testing. *Software and Systems Modeling (in press)*, 2021.

[14] D. Griñán and A. Ibias. Generating tree inputs for testing using evolutionary computation techniques. In *22nd IEEE Congress on Evolutionary Computation, CEC'20*, pages E–24267: 1–8. IEEE Computer Society, 2020.

[15] D. Griñán, A. Ibias, and M. Núñez. Grammar-based tree swarm optimization. In *2019 IEEE Int. Conf. on Systems, Man and Cybernetics, SMC'19*, pages 76–81. IEEE Press, 2019.

[16] R. M. Hierons, M. G. Merayo, and M. Núñez. Bounded reordering in the distributed test architecture. *IEEE Transactions on Reliability*, 67(2):522–537, 2018.

[17] A. Ibias, D. Griñán, and M. Núñez. GPTSG: a Genetic Programming Test Suite Generator using Information Theory measures. In *15th Int. Work-Conf. on Artificial Neural Networks, IWANN'19, LNCS 11506*, pages 716–728. Springer, 2019.

[18] A. Ibias, R. M. Hierons, and M. Núñez. Using Squeeziness to test component-based systems defined as Finite State Machines. *Information & Software Technology*, 112:132–147, 2019.

[19] A. Ibias and M. Núñez. `SqSelect`: Automatic assessment of failed error propagation in state-based systems. *Expert Systems with Applications*, 174:114748, 2021.

[20] A. Ibias, M. Núñez, and R. M. Hierons. Using mutual information to test from Finite State Machines: Test suite selection. *Information & Software Technology*, 132:106498, 2021.

[21] ISO/IEC JTCI/SC21/WG7, ITU-T SG 10/Q.8. Information Retrieval, Transfer and Management for OSI; Framework: Formal Methods in Conformance Testing. Committee Draft CD 13245-1, ITU-T proposed recommendation Z.500. ISO – ITU-T, 1996.

[22] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.

[23] J. Kennedy and R. Eberhart. Particle swarm optimization. In *3rd Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948. IEEE Computer Society, 1995.

[24] J. R. Koza. *Genetic programming*. MIT Press, 1993.

[25] H. Liu, F.-C. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering*, 40(1):4–22, 2014.

[26] J. D. McGregor and D. A. Sykes. *A Practical Guide to Testing Object-Oriented Software*. Addison Wesley object technology series. Pearson / Prentice Hall, 2001.

[27] R. I. McKay, N. X. Hoai, P. A. Whigham, Y. S., and M. O'Neill. Grammar-based genetic programming: a survey. *Genetic Programming and Evolvable Machines*, 11(3-4):365–396, 2010.

[28] M. G. Merayo, R. M. Hierons, and M. Núñez. Passive testing with asynchronous communications and timestamps. *Distributed Computing*, 31(5):327–342, 2018.

[29] M. G. Merayo, R. M. Hierons, and M. Núñez. A tool supported methodology to passively test asynchronous systems with multiple users. *Information & Software Technology*, 104:162–178, 2018.

[30] G. J. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. John Wiley & Sons, 3rd edition, 2011.

[31] A. Núñez, M. G. Merayo, R. M. Hierons, and M. Núñez. Using genetic algorithms to generate test sequences for complex timed systems. *Soft Computing*, 17(2):301–315, 2013.

[32] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman. Mutation testing advances: An analysis and survey. volume 112 of *Advances in Computers*, pages 275 – 378. Elsevier, 2019.

[33] S. Splaine and S. P. Jaskiel. *The web testing handbook*. STQE Pub., 2001.

[34] M. Srinivas and L. M. Patnaik. Genetic algorithms: A survey. *IEEE Computer*, 27:17–27, 1994.