Generating Tree Inputs for Testing using Evolutionary Computation Techniques

David Griñán
Universidad Complutense de Madrid
Madrid, Spain
david.grinan.martinez@gmail.com

Alfredo Ibias
Universidad Complutense de Madrid
Madrid, Spain
aibias@ucm.es

***Abstract*—Software Testing usually considers programs with parameters ranging over simple types. However, there are many programs using structured types. The main problem to test these programs is that it is not easy to select a relatively small test suite that can find most of the faults in these programs. In this paper we present a framework to generate test suites for unit testing of methods which have trees as parameters. We combine classical mutation testing with Evolutionary Computation techniques to evolve a population of trees. The final goal is to obtain a set of trees, representing *good* test cases, that will be used as the test suite to test the corresponding method.**

***Index Terms*—Software Testing, Evolutionary Computation, Mutation Testing**

I. Introduction

Software testing [1] is the main validation technique to assess the reliability of complex software systems. In particular, testing is used to validate single methods using *unit testing* techniques. Essentially, testing consists in selecting values for the parameters of the method that we are testing, call this method with these values, observe the returned values, and evaluate whether the returned values correspond to the expected ones. This simple description hides many issues that strongly complicate testing. First, we usually do not have an automatic procedure to check that the produced values are as expected (this is the well-known *oracle problem* [3], [32]). Second, and this is the topic of this paper, selecting values is not trivial because, in principle, not all the values have the same power to find faults. This task is much more difficult if we have to generate test cases for parameters ranging over structured types (e.g. tree-like structures as the ones considered in this paper). The main goal of this paper is to present a novel approach to generate test cases for unit testing of methods with structured type parameters, in particular, tree-like types. Note that lineal structures (such as queues and stacks) are a particular case of tree-like structures. Also note that there are better approaches for unit testing of methods with elementary type parameters.

In order to evaluate the usefulness of our approach, we use a *mutation testing* [8], [20], [26], [37] approach. We introduce small variations in the original program under test, called *mutants*, and apply a test suite to *kill* them. That is, observe results in the mutants different to the ones in the original program. The assumption is that a set of tests killing all mutants (or at least most, after removing *equivalent* mutants [28]) will be able to find most of the faults in the program under test. We will use mutation testing to compute the *fitness function* of our algorithm: the higher the number of killed mutants, the higher the fitness value will be.

Heuristic search algorithms are techniques commonly used in Mathematics and Computer Science either to optimise a function or to find the best possible solution for a given problem. These techniques, also referred to as *metaheuristics*, can be roughly divided into three categories: *global search techniques* such as simulated annealing [29], where only one solution is considered at each step; *evolutive techniques* such as genetic algorithms [14], which handle a population of candidate solutions at each step; and *constructive techniques* such as Ant Colony Optimisation [13], which start with an empty solution and progressively build upon it until achieving a solution for the problem.

In this paper we have used an Evolutionary Computation technique to generate test suites. We have chosen this particular family of techniques because it is well suited for parallel searching and it also combines the knowledge obtained by each member of the population of candidate solutions. Furthermore, by starting with a random set of candidate solutions, the algorithm can quickly obtain a candidate solution that suits our goal. More specifically, we have implemented a variation of a typical Evolutionary approach: the Particle Swarm Optimisation (PSO) algorithm [27]. This variation is the Tree Swarm Optimisation (TSO) algorithm [16]. The main advantage of this model is that it allows us to work with trees as the particles of the search space. Evolutionary algorithms like the one used in this paper are particularly useful to develop complex solutions in a more efficient way. Traditionally, a test suite is generated by a domain expert, which means that this task is complex and might take away a big percentage of a project's budget. To work around this issue, a heuristic search using evolutionary algorithms represents a powerful strategy that can cut in both time and costs. Finally, we would like to mention that the use of metaheuristics in testing is not new [2], [4], [9], [10], [19], [24]. In particular, there is some work on the application of the swarm idea to testing [17], [40]. The novelty of our approach resides in the fact that we are able to

This work has been supported by the Spanish MINECO-FEDER (grant number FAME, RTI2018-093608-B-C31) and the Region of Madrid (grant number FORTE-CM, S2018/TCS-4314).

produce test suites for methods having tree-like structures as parameters, that are *good* test suites to detect faults.

The rest of the paper is organised as follows. In Section II we present some theoretical concepts that we will use along our paper. In Section III we introduce our testing framework. In Section IV we present our experiments and discuss the results. In Section V we review some of the possible threats to the validity of our results. Finally, in Section VI we give the conclusions and outline some directions for future work.

II. Preliminaries

In this section we briefly introduce the concepts that will be used along the work. First, we will define the concepts of (directed) graph and tree.

Definition 1: A *directed graph* is a pair (V, E) where V is a set of vertices (or nodes) and $E \subseteq V \times V$ is a set of edges (or arcs). We say that a graph $G = (V, E)$ is *acyclic* if it does not have cycles, that is, there does not exist a non-empty sequence of edges $(v_0, v_1), (v_1, v_2), \ldots, (v_n, v_{n+1}) \in E$ such that $v_0 = v_{n+1}$. Let $v, v' \in V$ be vertices. We say that v' is *reachable* from v if there exists a non-empty sequence of transitions $(v, v_1), (v_1, v_2), \ldots, (v_n, v') \in E$.

A *tree* is a pair (G, r) where $r \in V$ is the *root* node and $G = (V, E)$ is a directed acyclic graph such that for all nodes $v \in V \backslash \{r\}$ we have that v is reachable from r.

A *connected component* of a graph (V, E) is a graph $G_c = (V_c, E_c)$, with $V_c \subseteq V$ and $E_c \subseteq E$, such that the following two conditions hold:

- There exists $v \in V_c$ such that for all $v' \in V_c \backslash \{v\}$, we have that v' is *reachable* from v. If the graph is acyclic then we may consider that v is the root of the induced tree.
- For all $v' \notin V_c$ we have that there does not exist $v'' \in V_c$ such that either v' is *reachable* from v'' or v'' is *reachable* from v'.

In order to simplify the framework, in this paper we consider the testing of methods that receive a single tree as an input. Note that additional parameters over simple types would be treated using standard software testing techniques. In addition, having several tree-like parameters would be the result of jointly applying the framework presented in this paper to all the parameters. In fact, our restriction does not represent a strong constraint from the theoretical point of view and can be easily overcome in its practical applications. Therefore, we need to generate test cases that consist of a single tree as input. In order to have more than one test, we need to produce test suites, which are a collection of tests. For this task, we recall the concept of forest.

Definition 2: A *forest* is an undirected graph $F = (V, E)$ where each connected component is a tree.

We will identify trees with test cases and forests with test suites. It is important to remark that some data structures (as lists, queues and stacks) can be seen as a special case of trees. Therefore, we will talk about trees in a mathematical way, as defined before, although the implementations can differ for each case.

In order to construct a test suite where each of the test cases corresponds to a tree that will be used as a parameter to call the method, we will use an Evolutionary model: a tree generation heuristic based on the Particle Swarm Optimisation (PSO) algorithm, the Tree Swarm Optimisation (TSO) algorithm [16]. Similarly to an evolutionary computation algorithm, we *evolve* a set of individuals to traverse the search space looking for an optimal solution. For each individual, we keep its best version from all the configurations visited along the evolution process, which in the first iteration will be itself. This information is usually referred to as the *memory* of each individual or particle. In order to perform the evolution process, we will also take into account the best out of the memories of the particles, which is the best individual seen by the population. These components represent the communication between particles in the population.

In a classical PSO algorithm, we start with an initial population of individuals, each one represented by a vector, and we obtain their fitness values. Then, the evolution loop consists in updating each individual by adding a vector called *velocity*. The velocity is defined as the following vector:

$$v_i = v_{i-1} + \alpha\psi \cdot best_p + \beta\phi \cdot best$$

where $best_p$ is the difference between the individual and the best version of it found along the algorithm; $best$ is the difference between the individual and the best individual found along the algorithm, α and β are the user-specified parameters for each memory, and ψ and ϕ are randomisation parameters. The update process is then performed by adding up the velocity vector and the vector representing the particle.

However, we are working with structured types, in particular tree-like types, and we have to adapt the original PSO algorithm (that was developed to work with vectors of numbers) to work with trees (or in our case, trees that represent forests, as we will see later). In order to do so, first, we replace the individuals by trees. Second, the update function, instead of being a simple sum of values, will consist of two consecutive crossovers for each individual of the population:

- a crossover between the best observed individual and the best version of the individual, and
- another crossover between the result of the previous crossover and the current version of the individual.

Essentially, each crossover takes the part of the trees where they match and perform a random update of their different parts. As discussed in [16], the crossover can be performed in multiple ways, specially, using the crossover operators already used in Genetic Programming. However, the only crossover operator, already proposed, that keeps the inspiration of the original PSO algorithm, is the one we defined before.

In Figure 1 we show a graphical representation of a crossover. Consider the two trees in the upper part. The green nodes denote the same information in both trees. When we perform the crossover, we keep the green nodes but the rest of the tree is replaced by a random tree.

There is an implementation detail that we would like to mention. In this paper, we are focusing on the generation of

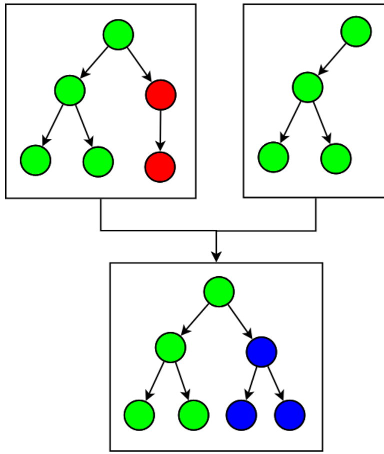

Fig. 1. Cross in the adapted PSO algorithm

test suites, that is, sets of trees. Therefore, it seems natural to work with forests of trees (each tree representing a test case). However, our variant of the PSO algorithm works only with trees. Therefore, at the implementation level, we will consider a special type of trees with a *fake* root such that it has as children all the trees conforming the test suite. We will elaborate on this issue later on.

Our goal behind the use of this approach, instead of a classical Genetic Programming heuristic, is to take advantage of the emergence property this algorithm presents. Thanks to particles communicating amongst themselves, a final test suite can be obtained as the result of individuals constantly exchanging information. It also allows for a more in-depth study of the population with respect to the steps the particles have taken as they move from one element of the search space to another.

III. Testing framework

In this section we present our testing framework. Its main goal is to generate *good* test suites for a given program/method that receives trees as inputs. As we have already explained, we decided to rely on evolutionary computation techniques to compute these test suites. In order to have a good fitness function for our evolutionary computation approach, we use mutation testing as the main tool to evaluate the fitness of the population that we are evolving.

Below, we briefly describe the main components of our framework:

- An implementation. This is the system/unit that we are testing (e.g. a method).
- A mutation tool. We need a tool to produce mutants.
- A tree generation heuristic. We need an algorithm to generate and evolve trees with the goal of improving a given fitness function. In our case, it is the TSO algorithm.

Our framework consists of two steps: a *mutant generation* step and a *test suite generation* step. A graphical representation of our framework can be found in Figure 2.

In order to perform the mutant generation step, we follow the usual approach in mutation testing in a white-box setting: we pass our method/program to a mutant generation tool and we produce a population of mutants. Therefore, the result of this step will be a set of mutants of the method that will be used in the next step to obtain the value of the fitness function.

In the test suite generation step we will use a tree generation heuristic (in our case, an evolutionary computation technique) to generate test suites conformed by trees. We work with trees as population members (representing test cases) and our optimisation object is a forest (representing a test suite). However, existing techniques consider trees as first-class citizens and, therefore, we need to encode forests as trees. Each test suite will be represented as a tree, where the root's children will be the roots of the trees that will be used as inputs for the program. With this codification, we represent a forest in a connected way so individuals now represent a set of trees rather than just one tree. A graphical representation of this structure is given in Figure 3.

The tree generation heuristic will use as fitness function the amount of mutants killed by each test suite (or the percentage of killed mutants with respect to the total number of mutants) and will iterate in order to improve the test suite. This fitness function will be used by the heuristic algorithm both to select the best test suite and to measure how effective each of the test suites is. It is important to note that we will not be able to keep test suites that kill fewer mutants but that are the only ones that kill those mutants, because we do not have this information at execution time.

In order to know if a mutant is killed, we will confront the result obtained by the mutant with the one obtained by the original method: if they differ then the mutant is killed. Note that we are considering *strong mutation* because we only check the final result. Even though we are mainly interested in a white-box testing framework, where we have access to the code of the method that we are validating, the idea is that we define a generic, as much as possible, framework that can be used both in white-box and black-box testing. If we consider a *weak mutation* approach, then we need to check the intermediate states but they are usually not available if we consider a black-box testing framework.

IV. Experiments

In this section we report on some experiments that we performed with the goal of evaluating the suitability of our proposed framework. The schema of the experiments (graphically presented in Figure 4) is very similar to our general framework (see Figure 2). There is, however, a slight difference concerning the use of the available mutants. In order to compute an *appropriateness* value for the obtained test suites, we need to confront them against a set of mutants. However, if we use the same sets of mutants that we used to produce the test suite, then there is the obvious risk of *overfitting*. In order to overcome this limitation, we adapted a classical machine learning technique to our setting: cross-validation [39].

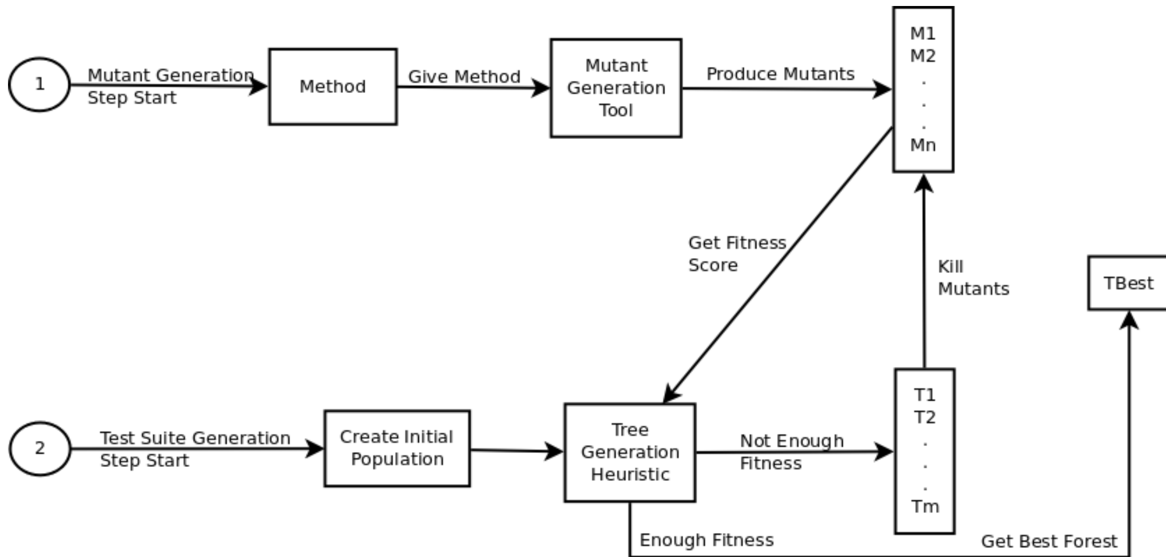

Fig. 2. Schema of the proposed testing framework

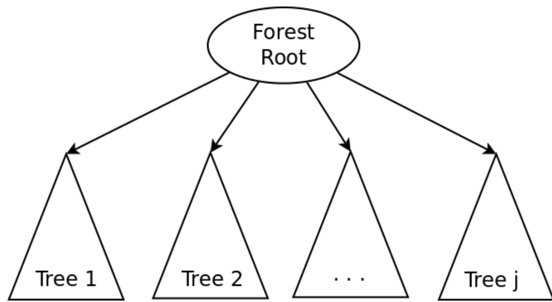

Fig. 3. Forest used in our approach

```
Data: tree input T
li = [];
for node n in T do
    li.append([n.depth(), n.orderInDepth()]);
end
li.sortBy(orderInDepth);
r = 0.0;
for orderInDepth o in li do
    r += sum({d : [d,o] in li})/size({d : [d,o] in li});
end
return returnFunct(r);
```

Algorithm 1: Pseudo-code of the methods algorithm.

Our implementation of cross-validation works as follows. We initially split the elements generated by the mutant generation step into 10 sets. These 10 sets will conform two sets: a *training set*, including 9 of these sets, and a *test set*, including the remaining set. As usual, the purpose of the training set is to generate a good solution (in this case, a good test suite) and the purpose of the test set is to *test* how good the solution is when working on *new* elements. We repeat this process 10 times, considering all the possible combinations. In the first iteration we chose the first set of mutants as test set while the sets $2, 3 \ldots, 10$ conform the training set. In the next iteration we use the second set of mutants as the test set and the set of mutants $1, 3 \ldots 10$ as a training set. The last iteration considers that the tenth set of mutants is the test set while the first 9 set of mutants conform the training set. We combine all the obtained results to compute an average value of the performance of the process for all test sets. This process is depicted in Figure 5.

In our experiments, we test three small but not trivial Java methods: given a tree, they produce a list of pairs where, for each node of the tree, a pair is added to the list with two values: the *depth* in the tree of the node and the *order*, from left to right, of the node between its depth. Then, with this list, we sort it using the order of the nodes. Once the list is sorted, we compute, for each order, the *mean* of the depths, that is, we sum the values of the depths and divide them by the number of elements. Finally, we sum the means of all the orders, obtaining a real number. Then, with this number, each method performs differently:

- Real method: it returns the obtained number, that is, its result is a real number.
- Integer method: it returns the truncation of the obtained number, that is, its result is an integer number.
- Boolean method: it returns true if the truncation of the obtained number is even, and false otherwise. Then, its result is a boolean value.

A pseudo-code of the methods algorithm is displayed in Algorithm 1.

We tried these three methods using common operations but different return type in order to test how well our algorithm performs in different difficulty scenarios. With the real method it will be easier to detect a mutation, because the possible output values are a huge range of values, while with the boolean method will be harder to detect a mutation because the

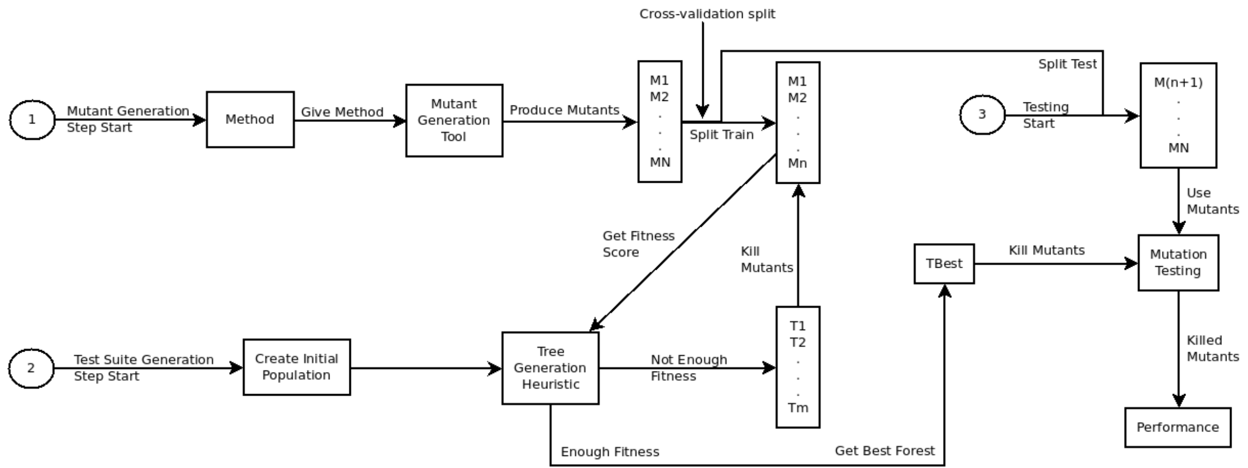

Fig. 4. Schema of the experiments flow

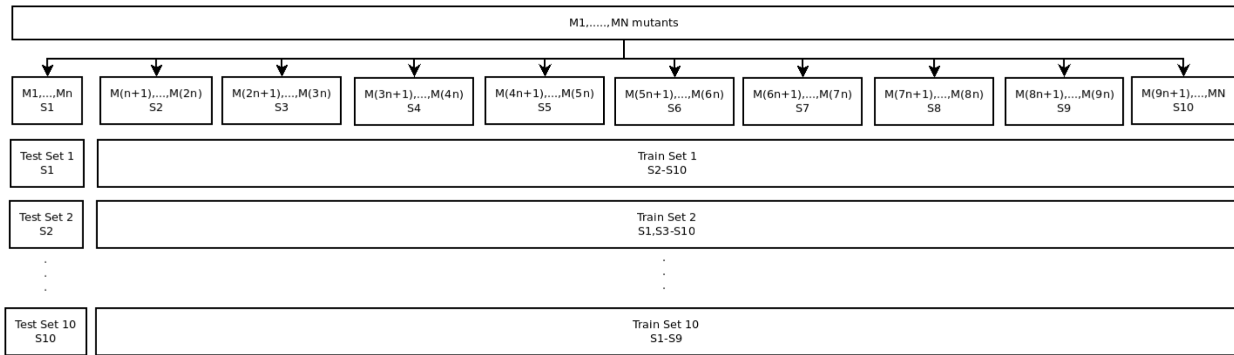

Fig. 5. Schema of the cross validation

possible values are 2, and therefore the possibility that, when tested with the same tree, both the mutated and the original method return the same result is higher.

Formally, the main differences between the output spaces of the three methods are:

- Real method: the output space is an infinite non-countable set, and therefore, its cardinality is $|\mathbb{R}| = \aleph_1$.
- Integer method: the output space is an infinite countable set, and therefore, its cardinality is $|\mathbb{Z}| = \aleph_0$.
- Boolean method: the output space is a finite countable set, and its cardinality is $|\mathbb{B}| = 2$.

These differences show that the sensibility to mutations of each method is different. In fact, it is proportional to the cardinality of their output spaces: output spaces with high cardinality will yield high *sensibility* to mutations and vice versa. Thus, methods with lower cardinality will have more cases of Failed Error Propagation when mutated and, therefore, it will be harder to detect those mutations [25]. This leads to fewer inputs that reveal the fault introduced by the mutation, hindering the selection of *good* test suites. As a conclusion, output spaces with low cardinality will suppose a higher challenge for our experiments, as we will see later on.

We have implemented our methods in Java and, therefore, we have to use a mutant generation tool for Java code. Although there are several academic and industrial tools to generate mutants from Java code, we have decided to stick with the *classical* system: MuJava [33]. We have done so because it is the latest updated tool that gives the source code of the generated mutants, which are the two requisites we need for our experiments. We explored other alternatives, specifically, recently updated alternatives, but the only tool we found was Pitest [7]. The problem with Pitest was that, although it produces more mutants, it is impossible to get the source code of those mutants. Instead, Pitest gives you an HTML file with the mutations performed, which is not enough to reproduce them. Therefore, we revert to MuJava due to its capacity to give the source and binary code of the mutants. Other alternatives where discontinued before MuJava was, therefore we assumed that they will be worse due to not having the most recent mutation operators implemented.

We used the tool to generate 575 mutants of the real and integer methods, and 593 mutants of the boolean method. We split those mutants into 10 sets of $57-58$ mutants ($59-60$ for the boolean method). With these sets, we build the two comparison sets through cross-validation, obtaining training sets of $517-518$ mutants ($533-534$ for the boolean method) and test sets of $58-57$ mutants ($60-59$ for the boolean method).

	Real Method		**Integer Method**		**Boolean Method**	
Trial Number	**% Mutants killed by Framework test suite**	**% Mutants killed by Random test suite**	**% Mutants killed by Framework test suite**	**% Mutants killed by Random test suite**	**% Mutants killed by Framework test suite**	**% Mutants killed by Random test suite**
1	0.8573913043478261	0.8104347826086956	0.8347826086956521	0.8330434782608696	0.7032040472175379	0.6711635750421585
2	0.8817391304347826	0.8782608695652174	0.8834782608695653	0.7652173913043478	0.6812816188870152	0.6897133220910624
3	0.8539130434782609	0.7773913043478261	0.8991304347826087	0.8556521739130435	0.7453625632377741	0.7099494097807757
4	0.88	0.8765217391304347	0.8278260869565217	0.8956521739130435	0.7571669477234402	0.6998313659359191
5	0.8991304347826087	0.8556521739130435	0.8834782608695653	0.7356521739130435	0.7453625632377741	0.684654300168634
6	0.8904347826086957	0.8417391304347827	0.8539130434782609	0.8278260869565217	0.6509274873524452	0.7032040472175379
7	0.8660869565217392	0.8956521739130435	0.8869565217391304	0.8504347826086956	0.7487352445193929	0.7048903878583473
8	0.8782608695652174	0.808695652173913	0.8643478260869565	0.8417391304347827	0.6998313659359191	0.6964586846543002
9	0.8347826086956521	0.8347826086956521	0.8747826086956522	0.8956521739130435	0.7403035413153457	0.7048903878583473
10	0.8556521739130435	0.8486956521739131	0.8834782608695653	0.7947826086956522	0.7504215851602024	0.6172006745362564
11	0.9026086956521739	0.8573913043478261	0.8504347826086956	0.8765217391304347	0.7318718381112985	0.6711635750421585
12	0.8817391304347826	0.88	0.8243478260869566	0.8243478260869566	0.7032040472175379	0.6644182124789207
13	0.8921739130434783	0.8469565217391304	0.7373913043478261	0.8660869565217392	0.7369308600337268	0.684654300168634
14	0.88	0.831304347826087	0.8852173913043478	0.8452173913043478	0.718381112984823	0.6441821247892074
15	0.888695652173913	0.8782608695652174	0.8869565217391304	0.8104347826086956	0.7470489038785835	0.6930860033726813
16	0.9095652173913044	0.7443478260869565	0.8226086956521739	0.7565217391304347	0.7369308600337268	0.6812816188870152
17	0.8991304347826087	0.782608695652174	0.8852173913043478	0.8556521739130435	0.7352445193929174	0.6711635750421585
18	0.888695652173913	0.8121739130434783	0.8782608695652174	0.8139130434782609	0.7453625632377741	0.7318718381112985
19	0.9043478260869565	0.8678260869565217	0.8695652173913043	0.8608695652173913	0.7403035413153457	0.6526138279932546
20	0.8747826086956522	0.8660869565217392	0.8939130434782608	0.808695652173913	0.7487352445193929	0.6930860033726813
21	0.8817391304347826	0.8747826086956522	0.831304347826087	0.8121739130434783	0.6981450252951096	0.6593591905564924
22	0.8556521739130435	0.8817391304347826	0.8573913043478261	0.7860869565217391	0.7284991568296796	0.5345699831365935
23	0.8765217391304347	0.8573913043478261	0.8660869565217392	0.8156521739130435	0.7386172006745363	0.7335581787521079
24	0.8765217391304347	0.8469565217391304	0.8243478260869566	0.8869565217391304	0.6795952782462057	0.6930860033726813
25	0.8973913043478261	0.8191304347826087	0.8765217391304347	0.8121739130434783	0.7436762225969646	0.657672849915683
26	0.8939130434782608	0.8434782608695652	0.88	0.88	0.6947723440134908	0.6795952782462057
27	0.8521739130434782	0.8504347826086956	0.888695652173913	0.8452173913043478	0.7048903878583473	0.7133220910623946
28	0.871304347826087	0.8834782608695653	0.8139130434782609	0.8765217391304347	0.7504215851602024	0.688026981450253
29	0.7930434782608695	0.8365217391304348	0.8365217391304348	0.8660869565217392	0.7824620573355818	0.6897133220910624
30	0.8521739130434782	0.8678260869565217	0.8452173913043478	0.8156521739130435	0.6981450252951096	0.7082630691399663
31	0.8539130434782609	0.8417391304347827	0.8121739130434783	0.8608695652173913	0.7318718381112985	0.6644182124789207
32	0.8539130434782609	0.8765217391304347	0.88	0.8591304347826086	0.7352445193929174	0.6863406408094435
33	0.831304347826087	0.8695652173913043	0.8330434782608696	0.8660869565217392	0.7217537942664418	0.654300168634064
34	0.8695652173913043	0.8695652173913043	0.8347826086956521	0.8243478260869566	0.684654300168634	0.6694772344013491
35	0.88	0.8469565217391304	0.8034782608695652	0.8382608695652174	0.7217537942664418	0.6762225969645869
36	0.8782608695652174	0.8660869565217392	0.8695652173913043	0.8608695652173913	0.6593591905564924	0.7284991568296796
37	0.8139130434782609	0.8643478260869565	0.8852173913043478	0.8208695652173913	0.7723440134907251	0.7200674536256324
38	0.8904347826086957	0.831304347826087	0.8452173913043478	0.8852173913043478	0.6644182124789207	0.6458684654300169
39	0.9008695652173913	0.8678260869565217	0.8330434782608696	0.831304347826087	0.7234401349072512	0.7369308600337268
40	0.8678260869565217	0.8208695652173913	0.8765217391304347	0.8573913043478261	0.7537942664418212	0.7065767284991569
41	0.9060869565217391	0.9026086956521739	0.8173913043478261	0.8643478260869565	0.6779089376053963	0.7116357504215851
42	0.8347826086956521	0.7930434782608695	0.8295652173913044	0.8121739130434783	0.7571669477234402	0.6728499156829679
43	0.8834782608695653	0.84	0.8852173913043478	0.8626086956521739	0.7032040472175379	0.7133220910623946
44	0.8991304347826087	0.8747826086956522	0.8539130434782609	0.8173913043478261	0.6694772344013491	0.7622259696458684
45	0.8226086956521739	0.8782608695652174	0.8417391304347827	0.8260869565217391	0.7706576728499157	0.7453625632377741
46	0.8904347826086957	0.8208695652173913	0.84	0.8486956521739131	0.7757166947723441	0.6677908937605397
47	0.88	0.8817391304347826	0.8121739130434783	0.8034782608695652	0.7268128161888702	0.654300168634064
48	0.8817391304347826	0.831304347826087	0.8417391304347827	0.7895652173913044	0.7689713322091062	0.718381112984823
49	0.8678260869565217	0.8608695652173913	0.808695652173913	0.8539130434782609	0.6779089376053963	0.688026981450253
50	0.9026086956521739	0.8417391304347827	0.7878260869565218	0.8643478260869565	0.7436762225969646	0.6745362563237775

TABLE I
RESULTS OF THE EXPERIMENTS.

In order to analyse the effectiveness of our test suites, we checked for equivalent mutants using the Trivial Compiler Equivalence. We used the Trivial Equivalent mutant Detector (TeD) from [28], as it is optimised for MuJava mutants. However, we detected no equivalent mutants. We also checked for duplicated mutants, and we detected 40 for the real and integer methods, and 44 for the boolean method. However, we checked that those duplicated mutants corresponded to different potential errors of a programmer, and therefore we decided to keep them in order to measure how many potential errors each test suite can find.

For the test suite generation step, as we have explained before, we used as tree generation heuristic a version of the PSO algorithm adapted to consider that particles are trees. We used as fitness function of our algorithm the number of mutants killed by each test suite and the algorithm evolves them towards the test suite that kills more mutants.

In order to test how *good* our generated test suite is, we take the test set and check how many mutants our best test suite kills, comparing the results with how many mutants a randomly generated test suite kills. It is worth mentioning that many studies claim that random selection is not much worse than *intelligent* approaches [6], [34], [38]. Therefore, an indicative of the usefulness of our proposal is to show that we are able to consistently outperform it. In order to have a greater challenge, and give a certain initial advantage to the

random approach, we set that the elements of the randomly generated test suite will have the maximum number of possible nodes (in our case, this is a total of 10 nodes).

We executed our experiments 50 times and we obtained the results displayed in Table I. On average, 81.62774788% of the mutants where killed by the test suite obtained by our framework while a randomly generated test suite killed 79.078460298% of the mutants. We think that this difference shows experimental evidence to claim that our framework is performing better than a randomly generated test suite that has the advantage of having at least the same size than the test suite that we build.

Analysing the results by method, we can see how the differences in the output cardinality lead to different results:

- Real method: 87.35652173913043% of the mutants were killed by the test suite obtained using our framework, while a randomly generated test suite killed only 84.87304347826087% of the mutants.
- Integer method: 85.07478260869563% of the mutants were killed by the test suite obtained using our framework, while a randomly generated test suite killed only 83.71478260869567% of the mutants.
- Boolean method: 72.45193929173693% of the mutants were killed by the test suite obtained using our framework, while a randomly generated test suite killed only 68.64755480607081% of the mutants.

The consistent differences between the means of killed mutants for the different methods (specially between the real and integer methods, which have exactly the same mutants) are an empirical evidence of how the cardinality of the outputs influences the *sensibility* of the method to mutations, and therefore, the difficulty of finding a good test suite for it.

Over the obtained results, we performed a statistical hypothesis test whose null hypothesis was that a random test suite and a test suite obtained using our framework give similar results, that is, both kill a similar amount of mutants. We applied a one-way ANOVA test where we tested whether the results of both test suites are similar in average. Then, we computed the p-values for each method, obtaining the following results:

- Real method: p-value of $3.92 \cdot 10^{-5} < 0.05$.
- Integer method: p-value of $0.049 < 0.05$.
- Boolean method: p-value of $2.77 \cdot 10^{-7} < 0.05$.

As it can be seen, in all cases the null hypothesis was rejected with a p-value lower than 0.05, thus allowing us to state that our framework gets different results than using a random test suite with a confidence higher than 0.95. In order to double-check our results, we also performed a t-test and obtained the same p-values. Therefore, we can claim with a high certainty that, statistically, our framework performs better than using a random test suite, in any scenario.

V. 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*, which consider uncontrolled factors that might be responsible for the obtained results, 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.

The main threat to *external validity*, which concerns conditions that allow us to generalise our findings to other situations, is the different possible methods to which we could apply our framework. Such a threat cannot be entirely addressed since the population of possible tree-input methods is unknown and it is not possible to sample from this (unknown) population. In order to diminish this risk, we considered different methods during the development of the work, focusing on different parameters, but the results were very similar (that is the reason why in this paper we report only on the results of the experiments that modify the output range).

Finally, we considered threats to *construct validity*, which are related to the *reality* of our experiments, that is, whether our experiments reflect real-world situations or not. In our work, the main construct threat is what would happen if we use our framework with much more complex methods. Although this is a potential threat, we consider that it is minor because our methods fulfil the requisites that any method receiving a tree input should fulfil and, therefore, are as valid as any other method receiving trees as inputs.

VI. Conclusions

We have proposed a new framework to test methods that accept structured types as inputs. Our framework strongly relied on a classical testing technique, mutation testing, to evaluate the suitability of the obtained tests and in Evolutionary techniques, in particular in evolutionary computation, to produce the test sets. Specifically, we have used a novel variant of PSO to deal with trees as individuals of the population. With this new framework, new test suites can be generated for a given method without having to design an ad-hoc battery of tests, which is a difficult and time consuming task. The complexity of the problem confronted in this paper is associated with the inherent difficulty to analyse methods having as parameters structured elements instead of simple numeric values. In addition to present the new framework, in this paper we have reported some of our experiments. The goal of the experiments was to show that the test suites generated by using our framework were better, concerning their capabilities to kill mutants, than randomly generated test suites. We think that the results show that this is indeed the case.

This is only the first step of, what we expect to be, a long research line. We have already identified several directions for future work concerning applicability, scalability, suitability and adaptability of our framework. First, we plan to apply our framework to test methods with parameters belonging to different (structured and simple) types. Second, we would like to consider more complex methods and check whether our

technique scales well. We will consider classical algorithms manipulating tree-like data structures [31]. In this line, it will be important to use recent advances on mutation testing [5], [15], [18] to support our framework. Concerning suitability, we have two orthogonal lines of work. First, we would like to compare our PSO approach with other metaheuristics, specially Genetic Programming [30]. In addition, recent contributions dealing with the integration of Computational Intelligence and Complex Events Processing [11], [12] might be related to our work. Second, we would like to consider methods with existing test suites, produced by an expert, and compare the quality of the existing test suites and the ones produced by our framework. Finally, concerning adaptability, we would like to assess the usefulness of our methodology in other frameworks. In particular, we will consider more complicated testing frameworks where communications are asynchronous [21], [35], [36] and/or data can be distributed among different components [22], [23].

REFERENCES

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

[2] C. Andrés, M. G. Merayo, and M. Núñez. Multi-objective genetic algorithms: Construction and recombination of passive testing properties. In *22nd Int. Conf. on Software Engineering & Knowledge Engineering, SEKE'10*, pages 405–410. Knowledge Systems Institute, 2010.

[3] 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.

[4] 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.

[5] 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.

[6] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive Random Testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[7] H. Coles, T. Laurent, C. Henard, M. Papadakis, and A. Ventresque. PIT: a practical mutation testing tool for java (demo). In *25th Int. Symp. on Software Testing and Analysis, ISSTA'16*, pages 449–452. ACM Press, 2016.

[8] P. Delgado-Pérez, I. Medina-Bulo, and J. J. Domínguez-Jiménez. Mutation testing. In *Encyclopedia of Information Science and Technology*, pages 7212–7221. IGI Global, 3rd edition, 2014.

[9] P. Delgado-Pérez, I. Medina-Bulo, and M. Núñez. Using evolutionary mutation testing to improve the quality of test suites. In *19th IEEE Congress on Evolutionary Computation, CEC'17*, pages 596–603. IEEE Computer Society, 2017.

[10] K. Derderian, M. G. Merayo, R. M. Hierons, and M. Núñez. A case study on the use of genetic algorithms to generate test cases for temporal systems. In *11th Int. Conf. on Artificial Neural Networks, IWANN'11, LNCS 6692*, pages 396–403. Springer, 2011.

[11] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and F. Cuartero. An intelligent transportation system to control air pollution and road traffic in cities integrating CEP and colored petri nets. *Neural Computing and Applications*, 32(2):405–426, 2020.

[12] G. Díaz, H. Macià, V. Valero, J. Boubeta-Puig, and G. Ortiz. Facilitating the quantitative analysis of complex events through a computational intelligence model-driven tool. *Scientific Programming*, 2019:2604148:1–2604148:17, 2019.

[13] M. Dorigo and T. Stützle. *Ant Colony Optimization*. MIT Press, 2004.

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

[15] 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.

[16] 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.

[17] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr. Swarm testing. In *22nd ACM SIGSOFT Int. Symposium on Software Testing and Analysis, ISSTA'12*, pages 78–88. ACM Press, 2012.

[18] L. Gutiérrez-Madroñal, A. García-Domínguez, and I. Medina-Bulo. Evolutionary mutation testing for IoT with recorded and generated events. *Software - Practice & Experience*, 49(4):640–672, 2019.

[19] M. Harman and P. McMinn. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[20] R. M. Hierons, M. G. Merayo, and M. Núñez. Mutation testing. In Phillip A. Laplante, editor, *Encyclopedia of Software Engineering*, pages 594–602. Taylor & Francis, 2010.

[21] R. M. Hierons, M. G. Merayo, and M. Núñez. An extended framework for passive asynchronous testing. *Journal of Logical and Algebraic Methods in Programming*, 86(1):408–424, 2017.

[22] 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.

[23] R. M. Hierons and M. Núñez. Implementation relations and probabilistic schedulers in the distributed test architecture. *Journal of Systems and Software*, 132:319–335, 2017.

[24] 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.

[25] 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.

[26] 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.

[27] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Int. Conf. on Neural Networks, ICNN'95*, pages 1942–1948, 1995.

[28] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman. Detecting trivial mutant equivalences via compiler optimisations. *IEEE Transactions on Software Engineering*, 44(4):308–333, 2018.

[29] S. Kirkpatrick, C. D. Gelatt Jr., and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

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

[31] R. Lafore. *Data Structures and Algorithms in Java*. Sams Publishing, 2nd edition, 2002.

[32] 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.

[33] Y.-S. Ma, J. Offutt, and Y. R. Kwon. MuJava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[34] R. Majumdar and F. Niksic. Why is random testing effective for partition tolerance bugs? *Proceedings of the ACM on Programming Languages*, 2:46:1–46:24, 2017.

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

[36] 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.

[37] 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.

[38] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification & Reliability*, 28(4), 2018.

[39] M. Stone. Cross-validatory choice and assessment of statistical predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974.

[40] A. Windisch, S. Wappler, and J. Wegener. Applying particle swarm optimization to software testing. In *9th Genetic and Evolutionary Computation Conference, GECCO'07*, pages 1121–1128. ACM Press, 2007.