Reformulating Branch Coverage as a Many-Objective Optimization Problem

Annibale Panichella*, Fitsum Meshesha Kifetew†‡, Paolo Tonella‡
*Delft University of Technology, The Netherlands
†University of Trento, Trento, Italy
‡Fondazione Bruno Kessler, Trento, Italy
a.panichella@tudelft.nl, kifetew@fbk.eu, tonella@fbk.eu

***Abstract*—Test data generation has been extensively investigated as a search problem, where the search goal is to maximize the number of covered program elements (e.g., branches). Recently, the whole suite approach, which combines the fitness functions of single branches into an aggregate, test suite-level fitness, has been demonstrated to be superior to the traditional single-branch at a time approach.**

In this paper, we propose to consider branch coverage directly as a many-objective optimization problem, instead of aggregating multiple objectives into a single value, as in the whole suite approach. Since programs may have hundreds of branches (objectives), traditional many-objective algorithms that are designed for numerical optimization problems with less than 15 objectives are not applicable. Hence, we introduce a novel highly scalable many-objective genetic algorithm, called MOSA (Many-Objective Sorting Algorithm), suitably defined for the many-objective branch coverage problem.

Results achieved on 64 Java classes indicate that the proposed many-objective algorithm is significantly more effective and more efficient than the whole suite approach. In particular, effectiveness (coverage) was significantly improved in 66% of the subjects and efficiency (search budget consumed) was improved in 62% of the subjects on which effectiveness remains the same.

Keywords: Evolutionary testing, many-objective optimization, branch coverage.

I. Introduction

Test automation, and input data generation in particular, has received substantial attention from researchers. When instantiated for branch coverage, the problem can be formulated as: find a set of test cases which maximize the number of covered branches in the software under test (SUT). Solutions based on search meta-heuristics (aka, search-based testing) deal with this problem by targeting one branch at a time and typically using genetic algorithms (GA) to generate test cases that get closer and closer to the target branch [1], [2], under the guidance of a fitness function, measuring the distance between each test case trace and the branch. With this approach, a single-objective GA is performed multiple times, changing the target branch each time, until all branches are covered or the total search budget is consumed. In the end, the final test suite is obtained by combining all generated test cases in a single test suite, including those responsible for accidental coverage.

Such a single-target strategy presents several issues [3]. For example, some targets may be more difficult to cover than others, thus, an improper allocation of the testing budget might affect the effectiveness of the search process (e.g., targeting infeasible goals will by definition fail and the related effort is wasted [3]). To overcome these limitations, Fraser and Arcuri [3] have recently proposed the *whole suite* (WS) approach that uses a search strategy based on (i) a different representation of candidate solutions (i.e., test suites instead of single test cases); and (ii) a new single fitness function that considers all testing goals simultaneously. From the optimization point of view, WS applies the sum scalarization approach that combines multiple target goals (i.e., multiple branch distances) into a single, scalar objective function [4], thus allowing for the application of single-objective meta-heuristics such as standard GA.

Previous works on numerical optimization have shown that the sum scalarization approach to many-objective optimization has a number of drawbacks, among which the main one is that it is not efficient for some kinds of problems (e.g., problems with non-convex region in the search space) [4]. Further studies have also demonstrated that many-objective approaches can be more efficient than single-objective approaches when solving the same complex problem [5], [6], i.e., a many-objective reformulation of complex problems reduces the probability of being trapped in local optima, also leading to a better convergence rate [5]. This remains true even when the multiple objectives are eventually aggregated into a single objective for the purpose of selecting a specific solution at the end of the (many-objective) search [5].

Stemming from the consideration that many-objective approaches can be more efficient when solving complex problems [5], [6], in this paper we propose to explicitly reformulate the branch coverage criterion as a many-objective optimization problem, where different branches are considered as different objectives to be optimized. In this new formulation, a candidate solution is a test case, while its *fitness* is measured according to all (yet uncovered) branches at the same time, adopting the multi-objective notion of optimality. As noted by Arcuri and Fraser [7], the branch coverage criterion lends itself to a many-objective problem, but it poses scalability problems to traditional many-objective algorithms since a typical class can have hundreds if not thousands of objectives (e.g., branches). However, we observe that branch coverage presents some peculiarities with respect to traditional (numerical) problems and we exploit them to overcome the scalability issues associated with traditional algorithms.

In this paper, we introduce a novel highly-scalable many-objective GA, named MOSA (Many-Objective Sorting Algorithm), that modifies the selection scheme used by existing

978-1-4799-7125-1/15/$31.00 ©2015 IEEE

many-objective GA. Results achieved on 64 Java classes extracted from 16 well-known libraries show that MOSA yields significantly better results (i.e., either higher coverage or, if the same coverage is reached, faster convergence) compared to the *whole suite* approach. In particular, coverage was significantly higher in 66% of the subjects and the search budget consumed was significantly lower in 62% of the subjects for which the coverage was the same. The total number of branches/objectives of the subjects ranges from 50 to 1213 (215 on average). To the best of our knowledge, this is the first work in search-based software engineering and test case generation that tackles a many-objective problem with hundreds of objectives. While multiple objectives have been already considered in test case generation, such additional objectives are non-coverage related, dealing for instance with test case length, execution time, memory consumption, etc. [8], [9], [10], [11], [12], [13]. This is the first approach where branches are regarded as many-objectives to be optimized simultaneously.

The remainder of this paper is organized as follows. Section II compares the single-objective formulation of the branch coverage criterion used in WS and the many-objective reformulation proposed in this paper. Section III presents our novel many-objective algorithm MOSA. Section IV describes the design of the empirical study and reports the achieved results. Section V summarizes the main related works, while Section VI concludes the paper.

II. Problem Formulation

This section describes the main characteristics and differences between the single-objective formulation of branch coverage used in the *whole suite* approach and the many-objective reformulation proposed in this paper.

A. Single Objective Formulation

In the *whole suite* approach, a candidate solution is a test suite consisting of a variable number of individual test cases, where each test case is a sequence of method calls again of variable length. The fitness of a test suite is measured with respect to full branch coverage, as the sum of the individual (normalized) branch distances for *all* the branches in the program under test. More formally, the search problem has been formulated [3] as follows:

Problem 1: Let $B = \{b_1, \ldots, b_m\}$ be the set of branches of a program. Find a test suite $T = \{t_1, \ldots, t_n\}$ that covers all the feasible branches, i.e., one that minimizes the following fitness function:

$$\min fitness(T) = \mid M \mid - \mid M_T \mid + \sum_{b \in B} d(b, T) \qquad (1)$$

where M is the total number of methods, M_T is the number of executed methods by all test cases in T and $d(b, T)$ denotes the minimal normalized branch distance for branch $b \in B$.

The fitness function above estimates how close a candidate test suite is to cover all branches of a program. The term $|M| - |M_T|$ accounts for the entry edges of unexecuted methods. The minimal normalized branch distance $d(b, t)$ for each branch $b \in B$ is defined as [3]:

$$d(b,t) = \begin{cases} 0 & \text{if } b \text{ has been covered} \\ \dfrac{D_{min}(t \in T, b)}{D_{min}(t \in T, b) + 1} & \text{if the predicated has been executed at least twice} \\ 1 & \text{otherwise} \end{cases} \qquad (2)$$

where $D_{min}(t \in T, b)$ is the minimal non-normalized branch distance, computed according to any of the available branch distance computation schemes [2]; minimality here refers to the possibility that the predicate controlling a branch is executed multiple times within a test case or by different test cases. The minimum is taken across all such executions.

From an optimization point of view, the *whole suite* approach considers all the branches at the same time and aggregates all corresponding branch distances into a unique fitness function, by summing up the contributions from the individual branch distances. In other words, multiple search goals (the branches to be covered) are reduced to a single search goal by means of an aggregated fitness function. Using this kind of approach, often named *scalarization* [4], a problem which involves multiple goals is transformed into a traditional single-objective, scalar one, thus allowing for the application of single-objective meta-heuristics such as standard genetic algorithms.

B. Many-Objective Formulation

In this paper we reformulate the branch coverage criterion as a many-objective optimization problem, where the objectives to be optimized are the branch distances of all the branches in the class under test. More formally, in this paper we consider the following reformulation:

Problem 2: Let $B = \{b_1, \ldots, b_m\}$ be the set of branches of a program. Find a set of non-dominated test cases $T = \{t_1, \ldots, t_n\}$ that minimize the fitness functions for all branches $b_1, \ldots, b_m$, i.e., minimizing the following m objectives:

$$\begin{cases} \min f_1(t) = al(b_1, t) + d(b_1, t) \\ \vdots \\ \min f_m(t) = al(b_m, t) + d(b_m, t) \end{cases} \qquad (3)$$

where each $d(b_i, t)$ denotes the normalized branch distance of test case t for branch b_i, while $al(b_i, t)$ is the corresponding approach level (i.e., the minimum number of control dependencies between the statements in the test case trace and the branch). Vector $\langle f_1, \ldots, f_m \rangle$ is also named fitness vector.

In this formulation, a candidate solution is a test case, not a test suite, and it is scored by *branch distance + approach level*, computed for all branches in the program. Hence, its fitness is a vector of m values, instead of a single aggregate score. In many-objective optimization, candidate solutions are evaluated in terms of *Pareto dominance* and *Pareto optimality* [4]:

Definition 1: A test case x dominates another test case y (also written $x \prec y$) if and only if the values of the objective

functions satisfy the following conditions:

$$\forall i \in \{1, \ldots, m\} \;\; f_i(x) \leq f_i(y)$$
and
$$\exists j \in \{1, \ldots, m\} \text{ such that } f_j(x) < f_j(y)$$

Conceptually, the definition above indicates that x is preferred to (dominates) y if and only if x is better on one or more objectives (i.e., it has a lower branch distance + approach level for one or more branches) and it is not worse for the remaining objectives. Among all possible test cases, the optimal test cases are those non-dominated by any other possible test case:

Definition 2: A test case x^* is Pareto optimal if and only if it is not dominated by any other test case in the space of all possible test cases (feasible region).

Single-objective optimization problems have typically one solution (or multiple solutions with the same optimal fitness value). On the other hand, solving a multi-objective problem may lead to a set of Pareto-optimal test cases (with different fitness vectors), which, when evaluated, correspond to trade-offs in the objective space. While in many-objective optimization it may be useful to consider all the trade-offs in the objective space, especially if the number of objectives is small, in the context of coverage testing we are interested in finding only the test cases that contribute to maximize the total coverage by covering previously uncovered branches, i.e., test cases having one ore more objective scores equal to zero, i.e., $f_i(t) = 0$. These are the test cases that intersect any of the m Cartesian axes of the vector space where fitness vectors are defined. Such test cases are candidates for inclusion in the final test suite and represent a specific sub-set of the Pareto optimal solutions.

III. Algorithm

This section briefly summarizes the main previous works on many-objective optimization and highlights their limitations in the context of many-objective branch coverage. Then, it introduces our novel many-objective genetic algorithm, suitably defined for the many-objective branch coverage problem.

A. Existing Many Objective Algorithms

Multi-objective algorithms have been successfully applied within the software engineering community to solve problems with two or three objectives, such as software refactoring, test case prioritization, etc. However, it has been demonstrated that classical multi-objective evolutionary algorithms, such as the Non-dominated Sorting Genetic Algorithm II (NSGA-II) [14] and the improved Strength Pareto Evolutionary Algorithm (SPEA2) [15], are not effective in solving optimization problems with more than three-objectives.

To overcome their limitations, new algorithms have been recently proposed that modify the Pareto dominance relation to increase the selection pressure. For example, Laumanns *et* al. [16] proposed the usage of an ϵ-dominance relation (ϵ-MOEA) instead of the classical one. Although this approach has been shown to be helpful in obtaining a good approximation of an exponentially large Pareto front in polynomial time, it presents drawbacks and in some cases it can slow down the optimization process significantly [17]. Zitzler and Künzli [18] proposed the usage of the hypervolume indicator instead of the Pareto dominance when selecting the best solutions to form the next generation. Even if the new algorithm, named IBEA (Indicator Based Evolutionary Algorithm), was able to outperform NSGA-II and SPEA2, the computation cost associated with the exact calculation of the hypervolume indicator in a high-dimensional space (i.e., with more than five objectives) is too expensive, making it infeasible with hundreds of objectives as in the case of branch coverage. Yang *et* al. [19] introduced a Grid-based Evolutionary Algorithm (GrEA) that divides the search space into hyperboxes of a given size and uses the concepts of grid dominance and grid difference to determine the mutual relationship of individuals in a grid environment. Di Pierro *et* al. [20] used a preference order-approach (POGA) as an optimality criterion in the ranking stage of NSGA-II. This criterion considers the concept of efficiency of order in subsets of objectives and provides a higher selection pressure towards the Pareto front than Pareto dominance-based algorithms. Yuan *et* al. [21] proposed θ-NSGA-III, an improved version of the classical NSGA-II, where the non-dominated sorting scheme is based on the concepts of θ-dominance to rank solutions in the environmental selection phase, which ensures both convergence and diversity.

All the many-objective algorithms mentioned above have been investigate mostly for numerical optimization problems with less than 15 objectives. Moreover, they are designed to produce a rich set of optimal trade-offs between different optimization goals, by considering both proximity to the real Pareto optimal set and diversity between the obtained trade-offs [16]. As explained in Section II-B, this is not the case for branch coverage, since we are interested in finding only the test cases having one ore more objective scores equal to zero (i.e., $f_i(t) = 0$), while the trade-offs are useful just for maintaining diversity during the optimization process. Hence, there are two main peculiarities that have to be considered in many-objective branch coverage as compared to more traditional many-objective problems: (i) not all Pareto optimal test cases (trade-offs between objectives) have a practical utility, hence, the search has to focus on a specific sub-set of the optimal solutions (those intersecting the m axes); (ii) for a given level of branch coverage, shorter test cases (i.e., test cases with a lower number of statements) are preferred.

B. MOSA: a New Many Objective Sorting Algorithm

Previous research in many-objective optimization [22] has shown that many-objective problems are particularly challenging because the proportion of non-dominated solutions increases exponentially with the number of objectives, i.e., all or most of the individuals are non-dominated. As a consequence, it is not possible to assign a preference among individuals for selection purposes and the search process becomes equivalent to a random one [22]. Thus, problem/domain specific knowledge is needed to impose an order of preference over test cases that are non-dominated according to the traditional non-dominance relation. For branch coverage, this means focusing the search effort on the test cases that are closer to one or more uncovered branches of a program. To this aim, we propose the following *preference criterion* in order to impose an order of preference among non-dominated test cases:

Definition 3: Given a branch b_i, a test case x is preferred over another test case y (also written $x \prec_{b_i} y$) if and only if

Algorithm 1: MOSA

```
   Input:
   B = {b_1, ..., b_m} the set of branches of a program.
   Population size M
   Result: A test suite T
1  begin
2      t ⟵ 0 // current generation
3      P_t ⟵ RANDOM-POPULATION(M)
4      archive ⟵ UPDATE-ARCHIVE(P_t)
5      while not (search_budget_consumed) do
6          Q_t ⟵ GENERATE-OFFSPRING(P_t)
7          R_t ⟵ P_t ⋃ Q_t
8          𝔽 ⟵ PREFERENCE-SORTING(R_t)
9          P_{t+1} ⟵ ∅
10         d ⟵ 0
11         while | P_{t+1} | + | 𝔽_d | ⩽ M do
12             CROWDING-DISTANCE-ASSIGNMENT(𝔽_d)
13             P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d
14             d ⟵ d + 1
15         Sort(𝔽_d) //according to the crowding distance
16         P_{t+1} ⟵ P_{t+1} ⋃ 𝔽_d[1 : (M − | P_{t+1} |)]
17         archive ⟵ UPDATE-ARCHIVE(archive ⋃ P_{t+1})
18         t ⟵ t + 1
19     T ⟵ archive
```

Algorithm 2: PREFERENCE-SORTING

```
   Input:
   A set of candidate test cases T
   Result: Non-dominated ranking assignment 𝔽
1  begin
2      𝔽_0 // first non-dominated front
3      for b_i ∈ B and b_i is uncovered do
4          // for each uncovered branch we select the best test case according to
           the preference criterion
5          t_best ⟵ test case in T with minimum objective score for b_i
6          𝔽_0 ⟵ 𝔽_0 ⋃ {t_best}
7      T ⟵ T − {t_best}
8      if T is not empty then
9          𝔾 ⟵
           FAST-NONDOMINATED-SORT(T, {b ∈ B|b is uncovered})
10         d ⟵ 0 //first front in 𝔾
11         for All non-dominated fronts in 𝔾 do
12             𝔽_{d+1} ⟵ 𝔾_d
```

the values of the objective function for b_i satisfy the following condition:

$$f_i(x) < f_i(y)$$

where $f_i(x)$ denotes the objective score of test case x for branch b_i (see Section II). The *best test case* for a given branch b_i is the one preferred over all the others for such branch ($x_{\text{best}} \prec_{b_i} y,\ \forall y \in T$). The *set of best test cases* across all uncovered branches ($\{x \mid \exists\ i : x \prec_{b_i} y,\ \forall y \in T\}$) defines a subset of the Pareto front that is given priority over the other non-dominated test cases in our algorithm. When there are multiple test cases with the same minimum fitness value for a given branch b_i, we use the test case length (number of statements) as a secondary preference criterion.

Our preference criterion provides a way to distinguish between test cases in a set of non-dominated ones, i.e., in a set where test cases are incomparable according to the traditional non-dominance relation, and it increases the selection pressure by giving higher priority to the best test cases across uncovered branches. Since none of the existing many-objective algorithms considers this preference ordering, which is a peculiarity of the branch coverage criterion, in this paper we present a novel many-objective genetic algorithm, which we name MOSA (Many Objective Sorting Algorithm), incorporating the proposed *preference criterion* during the selection process.

As shown in Algorithm 1, MOSA starts with an initial set of randomly generated test cases that forms the initial *population* (line 3 of Algorithm 1). The population then evolves toward nearby better test cases through subsequent iterations, called *generations*. To produce the next generation, MOSA first creates new test cases, called *offspring*, by combining parts from two selected test cases (*parents*) in the current generation using the *crossover* operator and randomly modifying test cases using the *mutation* operator (function GENERATE-OFFSPRING, at line 6 of Algorithm 1).

A new population is generated using a *selection* operator, which selects from parents and offspring according to the values of the objective scores. Such a selection is performed by considering both the non-dominance relation and the proposed *preference criterion* (function PREFERENCE-SORTING, at line 8 of Algorithm 1). In particular, the PREFERENCE-SORTING function, whose pseudo-code is provided in Algorithm 2, determines the test case with the lowest objective score (branch distance + approach level) for each uncovered branch b_i , i.e., the test case that is closest to cover b_i (lines 2-7 of Algorithm 2). All these test cases are assigned rank 0 (i.e., they are inserted into the first non-dominated front $\mathbb{F}_0$), so as to give them a higher chance of surviving in to the next generation (*elitism*). The remaining test cases (those not assigned to the first rank) are ranked according to the traditional non-dominated sorting algorithm used by the NSGA-II [14], starting with a rank equal to 1 and so on (line 8-12 of Algorithm 2). It is important to notice that the routine FAST-NONDOMINATED-SORT assigns the ranks to the remaining test cases by considering only the non-dominance relation for the *uncovered* branches, i.e., by focusing the search toward the interesting sub-region of the search space.

Once a rank is assigned to all candidate test cases, the *crowding distance* is used in order to make a decision about which test case to select: the test cases having a higher distance from the rest of the population are given higher probability of being selected. Specifically, the loop at line 11 in Algorithm 1 and the following lines 15 and 16 add as many test cases as possible to the next generation, according to their assigned ranks, until reaching the population size. The algorithm first selects the non-dominated test cases from the first front ($\mathbb{F}_0$); if the number of selected test cases is lower than the population size M, the loop selects more test cases from the second front ($\mathbb{F}_1$), and so on. The loop will stop when adding test cases from current front $\mathbb{F}_d$ exceeds the population size M. At end of the loop (lines 15-16), when the number of selected test cases is lower than the population size M, the algorithm selects the remaining test cases from the current front $\mathbb{F}_d$ according to the descending order of crowding distance.

As a further peculiarity with respect to other many-objective algorithms, MOSA uses a second population, called *archive*, to keep track of the best test cases that cover branches of the program under test. Specifically, after each generation MOSA stores every test case that covers previously uncovered branches in the *archive* as a candidate test case to form the final test suite (line 4 and 17 of Algorithm 1). To this aim, at the end

Algorithm 3: UPDATE-ARCHIVE

Input:
A set of candidate test cases T
Result: An archive A

```
1  begin
2      A ⟵ ∅
3      for b_i ∈ B do
4          best_length ⟵ ∞
5          t_best ⟵ ∅
6          for t_j ∈ T do
7              score ⟵ objective score of t_j for branch b_i
8              length ⟵ number of statements in t_j
9              if score == 0 and length ≤ best_length then
10                 t_best ⟵ {t_j}
11                 best_length ⟵ length
12         if t_best ≠ ∅ then
13             A ⟵ A ⋃ t_best
```

of each generation function UPDATE-ARCHIVE (reported in Algorithm 3 for completeness) updates the set of test cases stored in the *archive* with the new test cases forming the last generation. This function considers both the covered branches and the length of test cases when updating the *archive*: for each covered branch b_i it stores the shortest test case covering b_i in the *archive*.

In summary, generation by generation MOSA focuses the search towards the uncovered branches of the program under test (both PREFERENCE-SORTING and FAST-NONDOMINATED-SORT routines analyze the objective scores of the candidate test cases considering the uncovered branches only); it also stores the shortest covering test cases in an external data structure (i.e., the *archive*) to form the final test suite. Finally, since MOSA uses the *crowding distance* when selecting the test cases, it promotes *diversity*, which represents a key factor to avoid premature convergence toward suboptimal regions of the search space [23].

C. Graphical Interpretation of Preference Criterion

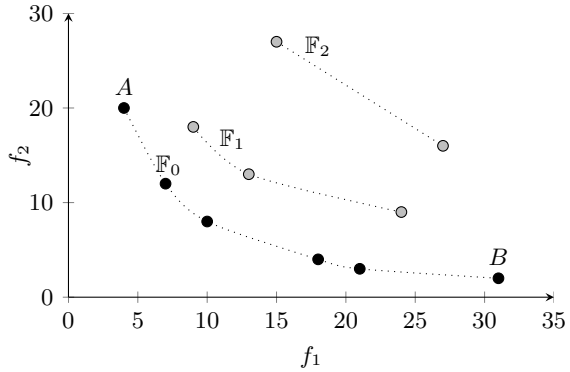
Let us consider the simple program shown in Figure 1-a and let us assume that the uncovered goals are the true branches of nodes 1 and 3, whose branch predicates are `(a == b)` and `(b == c)` respectively. According to the proposed many-objective formulation, the corresponding problem has two residual optimization goals, which are $f_1 = al(b_1) + d(b_1) = abs(a - b)$ and $f_2 = al(b_2) + d(b_2) = abs(b - c)$. Hence, any test case produced at a given generation t corresponds to some point in a two-dimensional objective space as shown in Figure 1-b and 1-c. Unless both a and b are equal, the objective function f_1 computed using the combination of approach level and branch distance is greater than zero. Similarly, the function f_2 is greater than zero unless b and c are equal.

Let us consider the scenario reported in Figure 1-b where no test case is able to cover the two uncovered branches (i.e., in all cases $f_1 > 0$ and $f_2 > 0$). If we use the traditional non-dominance relation between test cases, all test cases corresponding to the black points in Figure 1-b are non-dominated and form the first non-dominated front $\mathbb{F}_0$. Therefore, all such test cases have the same probability to be selected to form the next generation, even if test case A is the closest to the Cartesian axis f_2 (i.e., closest to cover branch b_2) and test case B is the closest to the Cartesian axis f_1 (branch b_1). Since there is no preference among test cases in $\mathbb{F}_0$, it might happen that A and/or B are not kept for the next generation, while other, less useful test cases in $\mathbb{F}_0$ are preserved. This scenario is quite common in many-objective optimization, where the number of non-dominated solutions increases exponentially with the number of objectives [22]. However, from the branch coverage point of view the two test cases A or B are better (fitter) than all other test cases, because they are the closest to cover each of the two uncovered branches.

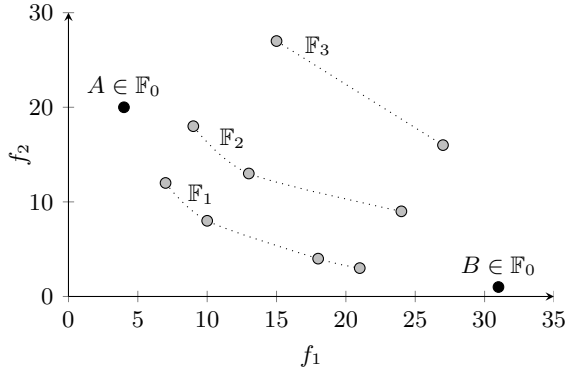
Our novel *preference criterion* gives a higher priority to test cases A and B with respect to all other test cases, guaranteeing their survival in the next generation. In particular, using the new ranking algorithm proposed in this paper, the first non-dominated front $\mathbb{F}_0$ will contain only test cases A and B (see Figure 1-c), while all other test cases will be assigned to other, successive fronts.

	Instructions
s	`int example(int a, int b, int c)`
	`{`
1	`if (a == b)`
2	`  return 1;`
3	`if (b == c)`
4	`  return -1;`
5	`return 0;`
	`}`

(a) Example program



(b) Ranking based on the traditional non-dominance relation



(c) Ranking based on the proposed *preference criterion*

Fig. 1. Graphical comparison between the non-dominated ranks assignment obtained by the traditional *non-dominated sorting algorithm* and the ranking algorithm based on *preference criterion* proposed in this paper.

IV. Empirical Evaluation

The goal of the empirical evaluation is to assess the *effectiveness* and *efficiency* of MOSA, in comparison with state-of-the-art single-objective approaches, and in particular the whole test suite optimization (WS) approach implemented by EvoSuite [3]. Specifically, we investigated the following research questions:

- **RQ1 (effectiveness)**: *What is the coverage achieved by MOSA vs. WS?*
- **RQ2 (efficiency)**: *What is the rate of convergence of MOSA vs. WS?*

RQ1 aims at evaluating the benefits introduced by the many-objective (re)formulation of branch coverage and to what extent the proposed MOSA algorithm is able to cover more branches if compared to an alternative, state-of-the-art whole suite optimization approach. With **RQ2** we are interested in analyzing to what extent the proposed approach is able to reduce the cost required for reaching the highest coverage.

A. Prototype Tool

We have implemented MOSA in a prototype tool by extending the EvoSuite test data generation framework. In particular, we implemented an extended many-objective GA as described in Section III. All other details (e.g. test case encoding schema, genetic operators, etc.) are those implemented in EvoSuite [3].

The experimental results reported in this section are obtained by using this prototype tool. The tool, along with a replication package, is available for download here: http://selab.fbk.eu/kifetew/mosa.html.

B. Subjects

In our empirical evaluation we used 64 Java classes from 16 widely used open source projects, many of which were used to evaluate the whole suite approach [3]. We tried to incorporate a diverse set of classes with varying levels of complexity and functionality. Table II (columns 2, 3, and 4) summarizes the details of the subjects. As running experiments on all classes from all the projects is computationally expensive, we selected classes randomly from the respective projects, with the only restriction that the total number of branches in the class should be at least 50. As can be seen from Table II, the total number of branches ranges from 50 to 1213 (on average around 215 branches). In our proposed many-objective formulation of the test data generation problem, each branch represents an objective to be optimized. Hence the set of classes used in our experiments present a significant challenge for our algorithm, in particular with respect to scaling to a large number of objectives.

C. Metrics

For comparing the two techniques, we use coverage as a measure of *effectiveness* and consumption of search budget as a measure of *efficiency*. Coverage (branch coverage) of a technique for a class is computed as the number of branches covered by the technique divided by the total number of branches in the class. Efficiency (search budget) is measured in the *number of executed statements*. Efficiency is used as a secondary metric, hence we compare efficiency only in cases where there is no statistically significant difference in effectiveness (i.e., coverage).

TABLE I. Parameter settings

Parameter	Value
Population size	50
Crossover rate	0.75
Mutation rate	1/size
Search budget (statements executed)	1,000,000
Timeout (seconds)	600

D. Experimental Protocol

For each class, each search strategy (WS or MOSA) is run and the effectiveness (coverage) and efficiency (budget consumption) metrics are collected. On each execution, an overall time limit is imposed so that the run of an algorithm on a class is bounded with respect to time. Hence, the search stops when either full branch coverage is reached, the search budget is finished, or the total allocated time is elapsed. To allow reliable detection of statistical differences between the two strategies, each run is repeated 100 times. Consequently we performed a total of 2 (search strategies) $\times$ 64 (classes) $\times$ 100 (repetitions) = 12,800 experiments. Statistical significance is measured with the non-parametric Wilcoxon Rank Sum test [24] with a p-value threshold of 0.05. Significant p-values indicate that the null hypothesis can be rejected in favour of the alternative hypothesis, i.e., that one of the algorithms reaches a higher branch coverage (**RQ1**) or a lower number of executed statements (**RQ2**). Other than testing the null hypothesis, we used the Vargha-Delaney ($\hat{A}_{12}$) statistic [25] to measure the effect size, i.e., the magnitude of the difference between the coverage levels (or number of executed statements for **RQ2**) achieved with different algorithms. The Vargha-Delaney ($\hat{A}_{12}$) statistic also classifies the obtained effect size values into four different levels (*negligible*, *small*, *medium* and *large*) that are easier to interpret.

There are several parameters that control the performance of the algorithms being evaluated. We adopted the default parameter values used by EvoSuite[3], as it has been empirically shown [26] that the default values, which are also commonly used in the literature, give reasonably acceptable results. The values for the important search parameters are listed in Table I.

E. Results

Table II summarizes the results (coverage) of the experiment. The coverage, averaged over 100 runs, that was achieved by each strategy is shown along with p-values obtained from the Wilcoxon test. Effect size metrics are also shown, indicating the magnitude of the difference. We can see from the table that out of the 64 classes, WS was statistically significantly better in 9 cases while MOSA was better in 42 cases. In the remaining 13 cases, no statistically significant difference was observed. In particular, from the analysis of the results reported in Table II we can notice that using MOSA coverage increases between 2% and 53%. It is worth noticing that such a result indicates a notable improvement if we consider the actual number of branches in our subjects. For example, if we consider class `Conversion` extracted from the *Apache*

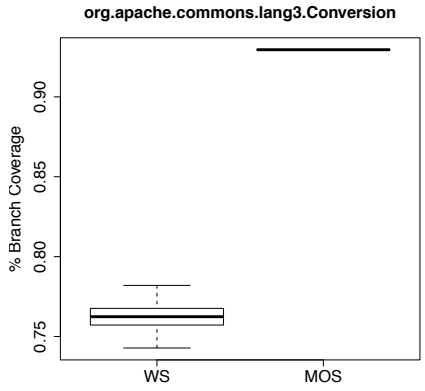


Fig. 2. Comparison of coverage achieved by WS and MOSA over 100 independent runs on *Conversion.java*.

TABLE III. BUDGET CONSUMED BY EACH APPROACH TO ACHIEVE THE BEST COVERAGE. P-VALUES AND EFFECT SIZE STATISTICS ARE ALSO SHOWN. $\hat{A}_{12} < 0.5$ MEANS MOSA IS BETTER THAN WS, $\hat{A}_{12} > 0.5$ MEANS WS IS BETTER THAN MOSA AND $\hat{A}_{12} = 0.5$ MEANS THEY ARE EQUAL. STATISTICALLY SIGNIFICANT VALUES ARE PRINTED IN BOLDFACE.

No	Class	WS	MOS	p-value	$\hat{A}_{12}$	Magnitude
1	Monitor	**294**	906	**0.000**	**0.94**	**Large**
2	TDoubleShort-MapDecorator	193701	**158788**	**0.003**	**0.38**	**Small**
3	TShortByte-MapDecorator	206375	**157995**	**0.000**	**0.32**	**Medium**
4	TShortHash	203317	179764	0.078	0.43	Negligible
5	LinearMath	99049	**58050**	**0.000**	**0.26**	**Large**
6	Option	311052	347356	0.436	0.53	Negligible
7	TreeList	425663	**286717**	**0.000**	**0.27**	**Medium**
8	FunctionUtils	408573	416136	0.719	0.49	Negligible
9	SchurTransformer	42033	**31865**	**0.029**	**0.41**	**Small**
10	BrentOptimizer	89938	**78094**	**0.000**	**0.33**	**Medium**
11	SAXOutputter	353790	**195672**	**0.000**	**0.07**	**Large**
12	JDOMResult	29544	20135	0.691	0.48	Negligible
13	NamespaceStack	273263	**137880**	**0.000**	**0.20**	**Large**
	Overall Average	202814	159181			
	No. cases significantly better	1/13	8/13			

commons library, we can observe that WS is able to cover 584 branches (averaged over 100 runs) while MOSA covers on average 712 branches using the same search budget. In other cases the improvements are even larger. For example, if we consider class `ExpressionParser` extracted from *JSci* library, we can notice that WS covers on average 53 branches against 237 branches covered on average by MOSA. This notable improvement is also highlighted by the effect size values obtained when applying the Vargha-Delaney ($\hat{A}_{12}$) statistic (see Table II). In the majority of cases where MOSA outperforms WS, the magnitude of the difference in terms of coverage is *large*, i.e., in 34 out of 42 cases. While in the few cases where WS is statistically better than MOSA we observe the effect size is *small* or *negligible* in the majority of cases, i.e., in 7 cases out of 9. Figure 2 provides a graphical comparison of the coverage values produced by WS and MOSA over 100 runs for one of the subject: `Conversion`. It can be seen from the boxplot that the distribution of coverage values obtained by MOSA over all independent runs is substantially higher than the distribution achieved by WS. Specifically, in all the runs MOSA reached a coverage of 92.58% while WS yielded a lower coverage raging between 74% and 78%.

For the 13 classes on which there was no significant difference in coverage between WS and MOSA, we compared the amount of search budget consumed by each search strategy (**RQ2**). Since for these 13 classes none of the two strategies achieved 100% coverage, they both consume the entire search budget. For this reason, we recorded the amount of search budget consumed to achieve the best coverage. This serves as an indicator of the convergence rate of the search strategy. Table III summarizes the result of the comparison of the budget consumed to achieve the final (highest) coverage. Out of 13 cases, WS consumed a significantly lower budget in only one case, while MOSA consumed significantly lower budget in 8 cases. In the remaining 4 cases, there was no statistically significant difference in budget consumption as well. Furthermore we can see from Table III that MOSA reached the best overall mean efficiency across the 13 classes, with a reduction in budget consumption of 22%, with the minimum reduction of 18% yielded for `TDoubleShortMapDecorator` and the maximum one of 50% achieved for `NamespaceStack`. The $\hat{A}_{12}$ effect size values shown in Table III confirm this analysis: in the majority of cases where MOSA outperforms WS, the magnitude of the difference in terms of efficiency is either *large* (3 out of 8 cases) or *medium* (3 out of 8 cases).

According to the results and the analyses reported above we can answer the research questions considered in this experiment as follows:

> *In summary, we can conclude that MOSA achieves higher branch coverage than WS (**RQ1**) and that when both achieve the same coverage, MOSA converges to such coverage more quickly than WS (**RQ2**).*

F. Qualitative analysis

Figure 3 shows an example of branch covered by MOSA but not by WS for the class `MatrixUtils` extracted from the *Apache commons math* library, i.e., the false branch of line 165 of method `createFieldMatrix`. The related branch condition checks the size of the input matrix `data` and returns an object of class `Array2DRowFieldMatrix` or of class `BlockFieldMatrix` depending on the outcome of the branch condition. At the end of the search process, the final test suite obtained by WS has a whole suite fitness $f = 34.17$. Within the final test suite, the test case closest to cover the considered goal is shown in Figure 3-b, i.e., a test case with maximum branch distance, $d = 1.0$, for the branch under analysis. This test case executes method `createFieldMatrix` indirectly by calling a second method `createFieldDiagonalMatrix` as reported in Figure 3-a. However, by analyzing all the test cases generated by WS during the search process we found that TC1 is not the closest test case to the false branch of line 165 across all generations. For example, at some generation WS generated a test case TC2 with a lower branch distance $d = 0.9997$ that is reported in Figure 3-c. As we can see, TC1 executes the line 162 because the input data is `null`, while the test case TC2 executes the branch condition in line 165 and the corresponding true branch in line 166. However, TC2 was generated within a candidate test suite with a poor whole suite fitness $f = 170.0$, which was also the worst candidate test suite in its generation. Thus, in the next generation this test suite (with the promising TC2) is not selected to form the next generation and this promising test case is lost. By manual

TABLE II. COVERAGE ACHIEVED BY WS AND MOSA ALONG WITH p-VALUES RESULTING FROM THE WILCOXON TEST. NUMERIC AND VERBAL EFFECT SIZE ($\hat{A}_{12}$) VALUES ARE ALSO SHOWN. $\hat{A}_{12} > 0.5$ MEANS MOSA IS BETTER THAN WS; $\hat{A}_{12} < 0.5$ MEANS WS IS BETTER THAN MOSA, AND $\hat{A}_{12} = 0.5$ MEANS THEY ARE EQUAL. SIGNIFICANTLY BETTER VALUES ARE SHOWN IN BOLDFACE.

No	Subject	Class	Branches	WS	MOS	p-value	$\hat{A}_{12}$	Magnitude
1	Guava	Utf8	63	85.16%	**90.24%**	**0.00000**	**0.73**	**Medium**
2	Guava	CacheBuilderSpec	139	94.22%	**98.95%**	**0.00000**	**1.00**	**Large**
3	Guava	BigIntegerMath	133	**91.35%**	90.97%	**0.02562**	**0.41**	**Small**
4	Guava	Monitor	191	10.47%	10.47%	NaN	0.50	Negligible
5	Tullibee	EReader	306	29.78%	**46.16%**	**0.00000**	**0.95**	**Large**
6	Tullibee	EWrapperMsgGenerator	67	88.34%	**95.90%**	**0.00000**	**0.86**	**Large**
7	Trove	TDoubleShortMapDecorator	59	88.31%	87.81%	0.88950	0.51	Negligible
8	Trove	TShortByteMapDecorator	59	87.88%	87.56%	0.56929	0.52	Negligible
9	Trove	TByteIntHash	87	92.91%	**94.26%**	**0.00813**	**0.61**	**Small**
10	Trove	TCharHash	60	89.52%	**90.72%**	**0.00000**	**0.68**	**Medium**
11	Trove	TFloatCharHash	87	**90.05%**	86.69%	**0.00000**	**0.29**	**Medium**
12	Trove	TFloatDoubleHash	87	**91.38%**	86.92%	**0.00000**	**0.23**	**Large**
13	Trove	TShortHash	60	89.87%	89.37%	0.11116	0.44	Negligible
14	Trove	TDoubleLinkedList	277	87.14%	**93.17%**	**0.00000**	**0.95**	**Large**
15	Trove	TByteFloatHashMap	293	85.63%	**90.98%**	**0.00000**	**0.95**	**Large**
16	Trove	TByteObjectHashMap	242	87.80%	**93.45%**	**0.00000**	**0.88**	**Large**
17	Trove	TFloatObjectHashMap	242	88.55%	**93.65%**	**0.00000**	**0.91**	**Large**
18	JSci	LinearMath	262	66.26%	68.01%	0.46728	0.53	Negligible
19	JSci	SpecialMath	196	85.94%	**87.98%**	**0.00000**	**0.94**	**Large**
20	JSci	ExpressionParser	435	12.23%	**54.47%**	**0.00000**	**0.96**	**Large**
21	JSci	SimpleCharStream	82	33.67%	**70.48%**	**0.00000**	**0.99**	**Large**
22	NanoXML	XMLElement	304	65.80%	**76.36%**	**0.00000**	**0.97**	**Large**
23	CommonsCli	HelpFormatter	142	**86.86%**	85.34%	**0.00062**	**0.36**	**Small**
24	CommonsCli	Option	96	95.64%	95.62%	0.84901	0.50	Negligible
25	CommonsCodec	DoubleMetaphone	498	84.95%	**92.32%**	**0.00000**	**1.00**	**Large**
26	CommonsPrimitives	RandomAccessByteList	81	95.33%	**96.09%**	**0.00000**	**0.74**	**Medium**
27	CommonsCollections	TreeList	215	94.00%	94.02%	0.81068	0.49	Negligible
28	CommonsCollections	SequencesComparator	89	96.16%	**96.63%**	**0.01327**	**0.53**	**Negligible**
29	CommonsLang	ArrayUtils	1119	67.08%	**71.64%**	**0.00000**	**1.00**	**Large**
30	CommonsLang	BooleanUtils	271	84.89%	**93.34%**	**0.00000**	**1.00**	**Large**
31	CommonsLang	CompareToBuilder	249	87.01%	**89.70%**	**0.00000**	**0.96**	**Large**
32	CommonsLang	HashCodeBuilder	116	85.57%	**90.27%**	**0.00000**	**1.00**	**Large**
33	CommonsLang	Conversion	766	76.24%	**92.95%**	**0.00000**	**1.00**	**Large**
34	CommonsLang	NumberUtils	383	81.52%	**89.59%**	**0.00000**	**1.00**	**Large**
35	CommonsLang	StrBuilder	567	95.47%	**98.18%**	**0.00000**	**1.00**	**Large**
36	CommonsLang	DateUtils	314	91.78%	**95.48%**	**0.00000**	**1.00**	**Large**
37	CommonsLang	Validate	98	90.67%	**96.31%**	**0.00000**	**1.00**	**Large**
38	CommonsMath	FunctionUtils	64	65.09%	65.39%	0.05707	0.55	Negligible
39	CommonsMath	TricubicSplineInterpolatingFunction	80	75.05%	**79.20%**	**0.00000**	**0.80**	**Large**
40	CommonsMath	DfpDec	138	67.78%	**65.46%**	**0.00512**	**0.61**	**Small**
41	CommonsMath	MultivariateNormalMixtureExpectationMaximization	66	**46.97%**	46.70%	**0.04442**	**0.48**	**Negligible**
42	CommonsMath	IntervalsSet	50	**86.10%**	85.64%	**0.04371**	**0.42**	**Small**
43	CommonsMath	MatrixUtils	143	72.94%	**78.57%**	**0.00000**	**0.87**	**Large**
44	CommonsMath	SchurTransformer	92	91.25%	89.02%	0.62217	0.52	Negligible
45	CommonsMath	AbstractSimplex	59	**67.71%**	65.39%	**0.00276**	**0.37**	**Small**
46	CommonsMath	BrentOptimizer	65	96.89%	96.91%	0.56565	0.51	Negligible
47	Javex	Expression	173	82.47%	**89.64%**	**0.00000**	**1.00**	**Large**
48	JDom	AttributeList	133	76.03%	**78.32%**	**0.00000**	**0.90**	**Large**
49	JDom	SAXOutputter	89	97.09%	97.33%	0.09856	0.56	Negligible
50	JDom	XMLOutputter	62	95.05%	**95.16%**	**0.01327**	**0.53**	**Negligible**
51	JDom	JDOMResult	50	58.82%	58.00%	0.08275	0.49	Negligible
52	JDom	NamespaceStack	80	76.90%	77.03%	0.30896	0.54	Negligible
53	JDom	Verifier	277	82.81%	**89.24%**	**0.00000**	**1.00**	**Large**
54	JodaTime	BasePeriod	79	93.41%	**95.39%**	**0.00000**	**0.82**	**Large**
55	JodaTime	BasicMonthOfYearDateTimeField	63	94.89%	**95.51%**	**0.00090**	**0.63**	**Small**
56	JodaTime	LimitChronology	112	**76.46%**	75.04%	**0.00187**	**0.37**	**Small**
57	JodaTime	PeriodFormatterBuilder	579	79.85%	**95.68%**	**0.00000**	**0.99**	**Large**
58	JodaTime	MutablePeriod	76	86.80%	**100.00%**	**0.00000**	**1.00**	**Large**
59	JodaTime	Partial	134	84.83%	**88.07%**	**0.00000**	**0.77**	**Large**
60	Tartarus	englishStemmer	290	81.45%	**83.11%**	**0.00000**	**0.95**	**Large**
61	Tartarus	italianStemmer	228	67.22%	**70.71%**	**0.00000**	**0.91**	**Large**
62	Tartarus	turkishStemmer	514	63.67%	**69.01%**	**0.00000**	**0.98**	**Large**
63	XMLEnc	XMLChecker	1213	**35.66%**	34.79%	**0.01285**	**0.40**	**Small**
64	XMLEnc	XMLEncoder	138	88.26%	**90.70%**	**0.00000**	**0.82**	**Large**
		Overall Average		78.86%	83.08%			
		No. cases significantly better		9/64	42/64			

investigation we verified that this scenario is quite common, especially for classes with a large number of branches to cover. As we can see from this example, the whole suite fitness is really useful in increasing the global number of covered goals, but when aggregating the branch distances of uncovered branches, the individual contribution of single promising test cases may remain unexploited.

Unlike WS, MOSA selects the best test case (and not test suites) within the current population for each uncovered branch. Therefore, in a similar scenario it would place TC2 in the first non-dominated front $\mathbb{F}_0$ according to the proposed *preference criterion*. Thus, generation by generation test case TC2 will be assigned to front $\mathbb{F}_0$ until it is replaced by a new test case that is closer to covering the target branch. Eventually, MOSA covers the false branch of line 165, while WS does not.

```
public static <...> createFieldMatrix(T[] data)
    throws NullArgumentException {
162    if (data == null) {
163        throw new NullArgumentException();
164    }
165       return (data.length * data[0].length <= 4096) ?
166          new Array2DRowFieldMatrix<T>(data) :
167                          new BlockFieldMatrix<T>(data);
168    }

public static <T extends FieldElement<T>> FieldMatrix<T>
    createFieldDiagonalMatrix(final T[] diagonal) {
    final FieldMatrix<T> m =
        createFieldMatrix(diagonal[0].getField());
        for (int i = 0; i < diagonal.length; ++i) {
            m.setEntry(i, i, diagonal[i]);
        }
    return m;
}
```

(a) Target branch

```
Fraction[] fractionArray0 = null;
FieldMatrix<Fraction> fieldMatrix0 =
       MatrixUtils.createFieldDiagonalMatrix(fractionArray0);
```

(b) TC1 with branch distance $d = 1.0$

```
FractionField fractionField0=FractionField getInstance();
Fraction[] fractionArray0 = new Fraction[1];
Fraction fraction0 = fractionField0.getZero();
fractionArray0[0] = fraction0;
FieldMatrix<Fraction> fieldMatrix1 =
    MatrixUtils.createFieldDiagonalMatrix(fractionArray0);
```

(c) TC2 with branch distance $d = 0.9997$

Fig. 3. Example of uncovered branch for *MatrixUtils*

G. Threats to Validity

Threats to *construct validity* regard the relation between theory and experimentation. For measuring performance of the compared techniques, we used metrics that are widely adopted in the literature: branch coverage and number of statements executed. In the context of test data generation, these metrics give reasonable estimates of the effectiveness and efficiency of the test data generation techniques.

Threats to *internal validity* regard factors that could influence our results. To deal with the inherent randomness of GA, we repeated each execution 100 times and reported average performance together with sound statistical evidence. Another potential threat arises from GA parameters. While different parameter settings could potentially result in different results, determining the best configuration is an extremely difficult and resource intensive task. Furthermore, such attempts to find the best configuration may not always pay off in practice as compared to using default configurations widely used in the literature [26]. Hence, we used default values suggested in related literature.

Threats to *conclusion validity* stem from the relationship between the treatment and the outcome. In analyzing the results of our experiments, we have used appropriate statistical tests coupled with enough repetitions of the experiments to enable the statistical tests. In particular, we have used the Wilcoxon test for testing significance in the differences and the Vargha-Delaney effect size statistic for estimating the magnitude of the observed difference. We drew conclusions only when results were statistically significant according to these tests.

There is a potential threat to *external validity* with respect to the generalization of our results. We carried out experiments on 64 Java classes taken from 16 widely used open source projects with a total number of branches ranging from 50 to 1213. While these classes exhibit a reasonable degree of diversity, further experiments on a larger set of subjects would increase the confidence in the generalization of our results. We also evaluated the performances of the compared techniques with subjects having less than 50 branches, obtaining consistent results.

V. Related Work

The application of search algorithms for test data generation has been the subject of increasing research efforts. As a result, several techniques and tools have been proposed. Existing works on search based test data generation rely on the single objective formulation of the problem, as discussed in Section II. In the literature, two variants of the single objective formulation can be found: (i) targeting one branch at a time [2], and (ii) targeting all branches at once (whole suite approach [3]). The first variant (i.e., targeting one branch at a time) has been shown to be inferior to the whole suite approach [3], [7], mainly because it is significantly affected by the inevitable presence of unreachable or difficult targets. Consequently, we focused on the whole suite formulation as a state-of-the-art representative of the single objective approach.

In the related literature, previous works applying multi-objective approaches in evolutionary test data generation have been reported. However, all previous works considered branch coverage as a unique objective, while other additional domain-specific goals have been considered as further objectives the tester would like to achieve [8], such as memory consumption, execution time, test suite size, etc. For example, Harman *et al.* [9] proposed a search-based multi-objective approach in which the first objective is branch coverage (each goal is still targeted individually) and the second objective is the number of collateral targets that are accidentally covered. Moreover, no specific heuristics are used to help cover these other targets. Ferrera *et al.* [10] proposed a multi-objective approach that considers two conflicting objectives: the coverage (to maximize) and the oracle cost (to minimize). They also used the *targeting one branch at a time* approach for maximizing the branch coverage criterion, i.e., their approach selects one branch at time and then runs GA for finding the test case with minimum oracle cost that covers such a branch. Pinto and Vergilio [11] considered three different objectives when generating test cases: structural coverage criteria (*targeting one branch at a time* approach), ability to reveal faults, and execution time. Oster and Saglietti [12] considered other two objectives to optimize: branch coverage (to maximize) and number of test cases required to reach the maximum coverage (to minimize). Lakhotia *et al.* [13] experimented bi-objective approaches by considering as objectives branch coverage and dynamic memory consumption for both real and synthetic programs. Even if they called this bi-objective formulation as *multi-objective branch coverage* they still represent branch coverage with a single objective function, by considering one branch at a time.

It is important to notice that all previous multi-objective approaches for evolutionary test data generation used the

targeting one branch at a time approach [2]. The branch distance of a single targeted branch is one objective, considered with additional non-coverage goals. From all these studies, there is no evidence that the usage of additional (non-coverage) objectives provides benefits in terms of coverage with respect to the traditional single-objective approach based on branch coverage alone [10]. Moreover, the number of objectives considered in these studies remains limited to a relatively small number.

Unlike previous multi-objective approaches to evolutionary test data generation, in this paper we proposed to consider the branch coverage by itself as a many-objective problem, where the goal is to minimize simultaneously the distances between the test cases and the uncovered branches in the class under test.

VI. Conclusions and Future Work

We have reformulated branch coverage as a many-objective problem, where different branches are considered as different objectives to be optimized. Our novel many-objective genetic algorithm, MOSA, exploits the peculiarities of branch coverage with respect to traditional many-objective problems to overcome scalability issues when dealing with hundreds of objectives (branches).

An empirical study conducted on 64 Java classes extracted from widely used open source libraries demonstrated that the proposed algorithm, MOSA, (i) yielded strong, statistically significant improvements (i.e., either higher coverage or faster convergence) with respect to the whole suite approach, and (ii) is highly scalable to programs with even more than one thousand of branches. Specifically, the improvements can be summarized as follows: coverage was significantly higher in 66% of the subjects and the search budget consumed was significantly lower in 62% of the subjects for which coverage was the same.

Based on the promising results achieved in this paper, we intend to investigate different directions for future work. First, we will replicate the study, considering further java classes to corroborate the results reported in this paper. Second, we will investigate whether other adequacy testing criteria (e.g., statement coverage, mutation testing, etc.) can be reformulated as many-objective problems and then solved using the proposed novel algorithm (MOSA). We also plan to combine the proposed many-objective branch coverage with other non-coverage criteria, such as execution time [11] and memory consumption [13].

References

[1] P. Tonella, "Evolutionary testing of classes," in *ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA '04. ACM, 2004, pp. 119–128.

[2] P. McMinn, "Search-based software test data generation: a survey," *Softw. Test. Verif. Reliab.*, vol. 14, no. 2, pp. 105–156, 2004.

[3] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Software Eng.*, vol. 39, no. 2, pp. 276–291, 2013.

[4] K. Deb and K. Deb, "Multi-objective optimization," in *Search Methodologies*. Springer US, 2014, pp. 403–449.

[5] J. Knowles, R. A. Watson, and D. Corne, "Reducing local optima in single-objective problems by multi-objectivization," in *Evolutionary Multi-Criterion Optimization*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2001, vol. 1993, pp. 269–283.

[6] J. Handl, S. C. Lovell, and J. Knowles, "Multiobjectivization by decomposition of scalar cost functions," in *Parallel Problem Solving from Nature*. Springer Berlin Heidelberg, 2008, vol. 5199, pp. 31–40.

[7] A. Arcuri and G. Fraser, "On the effectiveness of whole test suite generation," in *Search-Based Software Engineering*, ser. Lecture Notes in Computer Science. Springer International Publishing, 2014, vol. 8636, pp. 1–15.

[8] M. Harman, "A multiobjective approach to search-based test data generation," in *Association for Computer Machinery*. ACM Press. To, 2007, pp. 1029–1036.

[9] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *3rd International Conference on Software Testing, Verification, and Validation Workshops (ICSTW)*, April 2010, pp. 182–191.

[10] J. Ferrer, F. Chicano, and E. Alba, "Evolutionary algorithms for the multi-objective test data generation problem," *Software Practise & Experience*, vol. 42, no. 11, pp. 1331–1362, Nov. 2012.

[11] G. Pinto and S. Vergilio, "A multi-objective genetic algorithm to test data generation," in *22nd IEEE International Conference on Tools with Artificial Intelligence (ICTAI)*, vol. 1, Oct 2010, pp. 129–134.

[12] N. Oster and F. Saglietti, "Automatic test data generation by multi-objective optimisation," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, vol. 4166, pp. 426–438.

[13] K. Lakhotia, M. Harman, and P. McMinn, "A multi-objective approach to search-based test data generation," in *9th Conference on Genetic and Evolutionary Computation*, ser. GECCO '07. ACM, 2007, pp. 1098–1105.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast elitist multi-objective genetic algorithm: NSGA-II," *IEEE Trans. on Evolutionary Computation*, vol. 6, pp. 182–197, 2000.

[15] E. Zitzler, M. Laumanns, and L. Thiele, "Spea2: Improving the strength pareto evolutionary algorithm," Tech. Rep., 2001.

[16] M. Laumanns, L. Thiele, K. Deb, and E. Zitzler, "Combining convergence and diversity in evolutionary multiobjective optimization," *Evolutionary Computation*, vol. 10, no. 3, pp. 263–282, Sep. 2002.

[17] C. Horoba and F. Neumann, "Benefits and drawbacks for the use of epsilon-dominance in evolutionary multi-objective optimization," in *10th Conference on Genetic and Evolutionary Computation*, ser. GECCO '08. New York, NY, USA: ACM, 2008, pp. 641–648.

[18] E. Zitzler and S. Künzli, "Indicator-based selection in multiobjective search," in *8th International Conference on Parallel Problem Solving from Nature (PPSN VIII*. Springer, 2004, pp. 832–842.

[19] S. Yang, M. Li, X. Liu, and J. Zheng, "A grid-based evolutionary algorithm for many-objective optimization," *IEEE Trans. on Evolutionary Computation*, vol. 17, no. 5, pp. 721–736, Oct 2013.

[20] F. di Pierro, S.-T. Khu, and D. Savic, "An investigation on preference order ranking scheme for multiobjective evolutionary optimization," *IEEE Trans. on Evolutionary Computation*, vol. 11, no. 1, pp. 17–45, Feb 2007.

[21] Y. Yuan, H. Xu, and B. Wang, "An improved nsga-iii procedure for evolutionary many-objective optimization," in *14th Conference on Genetic and Evolutionary Computation*, ser. GECCO '14. ACM, 2014, pp. 661–668.

[22] C. von Lücken, B. Barán, and C. Brizuela, "A survey on multi-objective evolutionary algorithms for many-objective problems," *Computational Optimization and Applications*, vol. 58, no. 3, pp. 707–756, 2014.

[23] F. M. Kifetew, A. Panichella, A. D. Lucia, R. Oliveto, and P. Tonella, "Orthogonal exploration of the search space in evolutionary test case generation," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA)*, 2013, pp. 257–267.

[24] W. J. Conover, *Practical Nonparametric Statistics*, 3rd ed. Wiley, 1998.

[25] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[26] A. Arcuri and G. Fraser, "Parameter tuning or default values? an empirical investigation in search-based software engineering," *Empirical Software Engineering*, vol. 18, no. 3, pp. 594–623, 2013.