SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 0000; **00**:1–31
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe

Evolutionary Algorithms for the Multi-Objective Test Data Generation Problem

Javier Ferrer[1]* , Francisco Chicano[1] and Enrique Alba[1]

[1] *Departamento de Lenguajes y Ciencias de la Computación ,University of Málaga, Spain*

SUMMARY

Automatic Test Data Generation is a very popular domain in the field of Search Based Software Engineering. Traditionally, the main goal has been to maximize coverage. However, other objectives can be defined, like the oracle cost, which is the cost of executing the entire test suite and the cost of checking the system behaviour. Indeed, in very large software systems, the cost spent to test the system can be an issue, and then it makes sense considering two conflicting objectives: maximizing the coverage and minimizing the oracle cost. This is what we do in this paper. We mainly compare two approaches to deal with the Multi-Objective Test Data Generation Problem: a direct multi-objective approach and a combination of a mono-objective algorithm together with multi-objective test case selection optimization. Concretely, in this work we use four state-of-the-art multi-objective algorithms and two mono-objective evolutionary algorithms followed by a multi-objective test case selection based on Pareto efficiency. The experimental analysis compares these techniques on two different benchmarks. The first one is composed by 800 java programs created through a program generator. The second benchmark is composed by 13 real programs extracted from the literature. In the direct multi-objective approach, the results indicate that the oracle cost can be properly optimized; however the full branch coverage of the system poses a great challenge. Regarding the mono-objective algorithms, although they need a second phase of test case selection for reducing the oracle cost, they are very effective maximizing the branch coverage.

Received . . .

KEY WORDS: Multi-objective Test Data Generation; Branch Coverage; Oracle Cost; Evolutionary Testing; Evolutionary Algorithms; Search Based Software Engineering

1. INTRODUCTION

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [1, 2, 3, 4]. From the very first work [5, 6] to nowadays, many approaches have been proposed for solving the automatic test data generation problem (TDGP). This great effort in building computer aided software testing tools is motivated by the cost and importance of the testing phase in the software development cycle. It is estimated that half the time spent on software project development, and more than half its cost, is devoted to testing the product [7]. This explains why Software Industry and Academia are interested in automatic tools for testing.

Evolutionary algorithms (EAs) have been the most popular search-based algorithms for generating test cases [3]. In fact, the term *evolutionary testing* is used to refer to this approach. In the paradigm of *structural testing* a lot of research has been performed using EAs and, in particular, different elements of the structure of a program have been studied in detail. Some examples are the

*Correspondence to: Departamento de Lenguajes y Ciencias de la Computación, Universidad de Málaga, Spain. E-mail: ferrer@lcc.uma.es presence of flags in conditions [8], the coverage of loops [9], the existence of internal states [10], and the presence of possible exceptions [11]. In addition, several evolutionary algorithms have been used as the search engine like scatter search [12], genetic algorithms [13, 14], simulated annealing [15], and tabu search [16].

Traditionally, the solution of the TDGP is a set of test cases whose execution is able to cover all the software elements. Branch coverage is usually the most popular goal. Despite most previous work only considering branch coverage, real-world engineers deal with the tedious and costly task of checking the system behaviour for all the generated test cases. This significant and usually neglected cost is called the oracle cost [17]. Thus, a reformulation of the TDGP to deal with real-world problems is a need, taking into account the oracle cost as another important objective to minimize. The oracle cost can be reduced by minimizing the test suite size. The ideal scenario is to reduce the test suite size without any loss of coverage. However, in certain situations the two objectives are in conflict: minimizing the oracle cost implies minimizing the coverage. When there are multiple conflicting objectives the optimization literature recommends the consideration of a Pareto optimal optimization approach that is able to take into account the need to balance the conflicting objectives. Thus, the TDGP has been reformulated into a multi-objective problem (MOTDGP) in the work by Lakhotia *et al.* [18] and more recently, in 2010, in a work by Harman *et al.* [17].

Our main goal in this work is the comparison between two approaches to deal with the MOTDGP: a direct multi-objective approach (MM) and a combination of a mono-objective algorithm followed by a multi-objective test case selection optimization (mM). The general scheme of the proposed approaches can be seen in Figure 1.

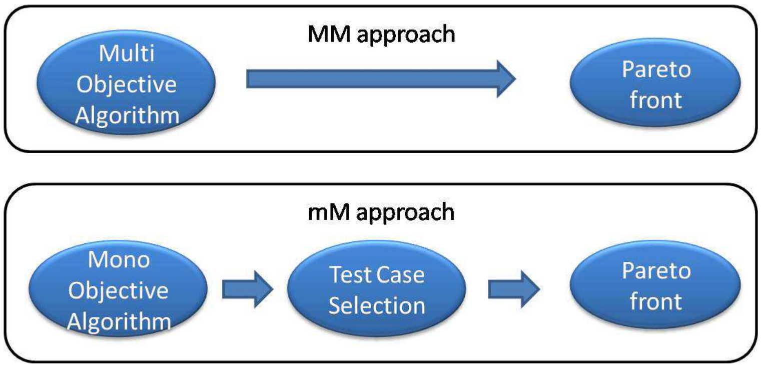


Figure 1. The general scheme of the two proposed approaches.

On the one hand, the MM approach considers the conflicting objectives during all the test data generation process, thus *a priori* it focuses both on the test suite size minimization and the coverage maximization. On the other hand, the mM approach only considers the branch coverage during the test data generation process, thus *a priori* it focuses only on the branch coverage maximization. In order to deal with the optimization of the test suite size, in this second approach an additional second phase of multi-objective test case selection is performed. As nobody has previously compared these approaches yet, we can raise the following research questions and try to answer them in an extensive experimental study.

- How does MM approach deal with MOTDGP?
- Is the MM approach good enough in maximizing the coverage?
- How good is the mM approach performance in optimizing the coverage and the test suite size?
- Which approach is the best?

In order to completely answer the questions we should use all the possible automatic test data generators both in multi and mono-objective or, at least, a large number of them. We can also focus on some test data generators and answer the previous questions on them, taking into account that in this case the results will be valid for the test data generators considered. This is what we do in this paper. In particular, we study the MOTDGP with two objectives, maximizing the branch coverage and minimizing the oracle cost. Among our contributions, we generate the test data and we also minimize the number of tests needed to achieve different values of coverage of the program. The solutions are provided as Pareto fronts. For the MM approach, we use five test data generators: four of them based on evolutionary testing and an additional one based on random search. In the mM approach we use three mono-objective test data generators with a second phase of multi-objective test data selection.

The rest of the paper is organized as follows. In the next section we define the multi-objective test data generation problem. Then, in Section 3 we present some background on multi-objective optimization. Next, in Section 4 we describe the MM approach and the multi-objective algorithms. After that, in Section 5, we describe the mM approach, we provide details regarding the general structure of the test data generator for single objective algorithms and the algorithms used in the experiments. Section 6 is devoted to the experimental methodology where we explain the quality indicators and the benchmark of programs that we use in the experiments. In Section 7, we show the results of the experiments and we answer the proposed research questions. Finally, in Section 8, some conclusions and future work are outlined.

2. MULTI-OBJECTIVE TEST DATA GENERATION PROBLEM

The most popular technique to test software programs consists in executing the program with a set of test data (*software testing*). The engineer selects an initial set of configurations for the program under test (PUT), called test data suite, and s/he checks the PUT behaviour with them. Since the size of the test data suite is an engineer's decision, s/he can control the effort devoted to this task, which is the oracle cost. In order to ensure the correctness of a program with this technique, it would be necessary to execute the PUT with all the possible configurations, but in practise this is unfeasible. The alternative consists in testing the program with a representative set of test data.

Automatic test data generation (*automatic software testing*) consists in proposing an adequate set of test data in an automatic way to test a program, thus preventing the engineer from the task of selecting an adequate set of test data to test the PUT. This automation of the process requires a precise definition of what is an "adequate set" of test data, definition that we will defer until some terms are defined. As we said before, this is a costly and hard task of the software development. Thus, another objective for a software engineer is the minimization of the oracle cost, which can be reduced to the minimization of the test suite size. In the following, we formally define the MOTDGP, but we first need to introduce several terms and notation.

Let P be a program, we denote with B_P the set of branches of the program and with $BranchExec_P(C)$ the set of branches covered in P due to the execution of a given set of test data, C. We define the branch coverage of the test suite C, $BrCov_P(C)$, as the ratio between the traversed branches in the executions of the program P with the set of test data C and the number of branches of the program, i.e.,

$$BrCov_P(C) = \frac{|BranchExec_P(C)|}{|B_P|} \tag{1}$$

The adequacy criterion of branch coverage states that a test suite C for a program P is "adequate" when $BrCov_p(C) = 1$. Nevertheless, it is not always possible to reach such a value of coverage, and in case of reaching it, the cost to test the entire program can be unaffordable. Consequently, a balance between coverage and the cost to achieve such coverage is mandatory. Since the cost of the testing phase depends on the test suite size, minimizing the test suite size, denoted with $|C|$, must be another goal.

Finally we deal with the MOTDGP with two conflicting objectives:

- $\max\ BrCov_P(C)$
- $\min\ |C|$,

that is, maximizing the branch coverage and minimizing the test suite size.

3. MULTI-OBJECTIVE BACKGROUND

In this section, we provide background on multi-objective optimization. In particular, we define the concept of multi-objective optimization problem (MOP), Pareto dominance, and Pareto front. In these definitions we are assuming, without loss of generality, that minimization is the goal for all the objectives.

A general MOP can be formally defined as follows:

Find a vector $\mathbf{x}^* = (x_1^*, x_2^*, \ldots, x_n^*)$ that satisfies the m inequality constraints $g_i(\mathbf{x}) \geq 0, i = 1, 2, \ldots, m$, the p equality constraints $h_i(\mathbf{x}) = 0, i = 1, 2, \ldots, p$, and minimizes the vector function $\mathbf{f}(\mathbf{x}) = (f_1(\mathbf{x}), f_2(\mathbf{x}), \ldots, f_m(\mathbf{x}))^T$, where $\mathbf{x} = (x_1, x_2, \ldots, x_n)^T$ is the vector of decision variables. The set of all the values satisfying the constraints defines the *feasible region* Ω and any point $\mathbf{x} \in \Omega$ is a *feasible solution*.

Taking into account this definition of a MOP, a solution $\mathbf{x}^1 = (x_1^1, x_2^1, ..., x_n^1)$ is said to dominate a solution $\mathbf{x}^2 = (x_1^2, x_2^2, ..., x_n^2)$ denoted with $\mathbf{x}^1 > \mathbf{x}^2$, if and only if $f_i(\mathbf{x}^1) \leq f_i(\mathbf{x}^2)$ for $i = 1, 2, ..., m$, and there exists at least one j $(1 \leq j \leq m)$ such that $f_j(\mathbf{x}^1) < f_j(\mathbf{x}^2)$. Conversely, two points are said to be non-dominated whenever none of them dominates the other. Figure 2 depicts some examples of dominated and non-dominated solutions. In this figure, A dominates C because $f_1(A) < f_1(C)$, and $f_2(A) < f_2(C)$. Meanwhile, A and B are non-dominated solutions because A is better than B in the first objective function ($f_1(A) < f_1(B)$), but B is better than A in the other objective function ($f_2(A) > f_2(B)$).

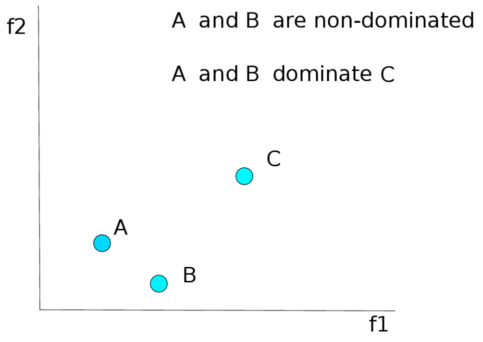


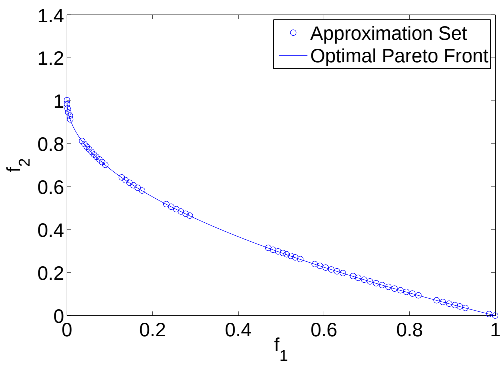
Figure 2. Examples of dominated and non-dominated solutions.

The solution of a given MOP is usually a set of solutions (referred to as the Pareto optimal set) satisfying:

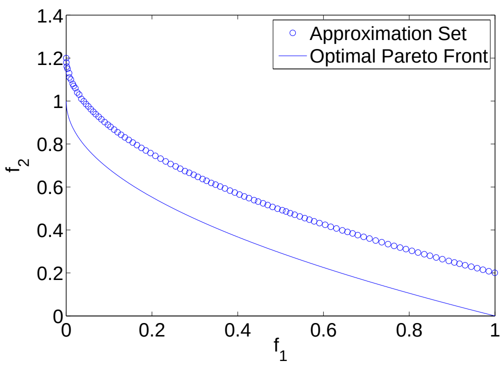
- Every pair of two solutions in the set are non-dominated.
- Any other solution, y, is dominated by at least one solution in the set.

The representation of this set in the objective space is referred to as the *Pareto front*. Generating the *Pareto front* of a problem is the main goal of multi-objective optimization techniques. In theory, a Pareto front could contain a large number of points. In practice, a usable approximate solution will only contain a limited number of them; thus, an important goal is that solutions should be as close as possible to the exact Pareto front and uniformly spread, otherwise, they would not be very useful to the decision maker. Besides, closeness to the Pareto front ensures that we are dealing with optimal solutions, while a uniform spread of the solutions means that we have made a good exploration of the objective space and no regions are left unexplored.

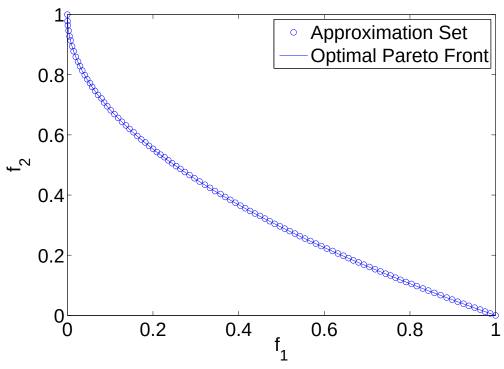
Figure 3 depicts these issues of convergence and diversity. The left front (a) depicts an example of good convergence and bad diversity: the approximation set contains Pareto optimal solutions but there are some unexplored regions of the objective space. The approximation set depicted on the right (b) illustrates poor convergence but good diversity: it has a diverse set of solutions but they are not Pareto optimal. Finally, the lowermost front (c) depicts an approximation front with both good convergence and diversity.



(a) good convergence and bad diversity.



(b) bad convergence and good diversity.



(c) good convergence and good diversity.

Figure 3. Examples of Pareto fronts with different behaviour of convergence and diversity.

4. MM APPROACH

In this work we are dealing with the MOTDGP from two points of view: a direct multi-objective approach (MM) and the application of a mono-objective algorithm followed by a multi-objective test case selection phase (mM). In this section we explain the first approach: the MM approach. We describe how we deal with the MOTDGP and the algorithms used to solve the problem.

The MM approach considers the conflicting objectives during all the test data generation process, thus *a priori*, it focuses on both objectives during all the process. In this approach a solution to the problem is a test suite, that is, a set of test data. These test suites are evaluated according to both objectives. The evaluation of the first objective (coverage) requires, in general, the execution of the test suite over the program under test. The evaluation of the second objective is a simple count of the number of test data in the set. In the following subsection we present the multi-objective algorithms used in the experimental section.

4.1. Multi-objective Algorithms

In this section we describe the five multi-objective algorithms used in the experimental section: NSGA-II, MOCell, SPEA2, PAES and a random search algorithm (RNDMulti).

NSGA-II, proposed by K. Deb *et al.* [19], is a genetic algorithm which is the reference algorithm in multi-objective optimization (with over 4,860 citations at the time of writing[†]). Its pseudocode

[†]Data from Google Scholar: 4,860 citations on July 13^{th} 2011.

is presented in Algorithm 1. NSGA-II makes use of a population (P) of candidate solutions (known as individuals). In each generation, it works by creating new individuals after applying the genetic operators to P, in order to create a new population Q (lines 5 to 8). Then, both the current (P) and the new population (Q) are joined; the resulting population, R, is ordered according to a ranking procedure and a density estimator known as crowding distance (line 13) (for further details, please see [19]). Finally, the population P is updated with the best individuals in R (line 14). These steps are repeated until the termination condition is fulfilled.

Algorithm 1 Pseudocode of NSGA-II.

```
1:  proc Input:(nsga-II)       //Algorithm parameters in 'nsga-II'
2:  P ← Initialize_Population() // P = population
3:  Q ← ∅                      // Q = auxiliary population
4:  while not Termination_Condition() do
5:      for i ← 1 to (nsga-II.popSize / 2) do
6:          parents←Selection(P)
7:          offspring←Recombination(nsga-II.Pc,parents)
8:          offspring←Mutation(nsga-II.Pm,offspring)
9:          Evaluate_Fitness(offspring)
10:         Insert(offspring,Q)
11:     end for
12:     R ← P ∪ Q
13:     Ranking_And_Crowding(nsga-II, R)
14:     P ←Select_Best_Individuals(nsga-II, R)
15: end while
16: end_proc
```

MOCell (Multi-Objective Cellular Genetic Algorithm), introduced by Nebro *et al*. [20], is a cellular genetic algorithm (cGA) which outperforms NSGA-II in some studies [20, 21]. In cGAs, the concept of (small) *neighbourhood* is paramount. This means that an individual may only cooperate with its nearby neighbours in the breeding loop. Overlapped small neighbourhoods of cGAs help in exploring the search space because they induce a slow diffusion of solutions through the population, providing a kind of exploration (diversification). Exploitation (intensification) takes place inside each neighbourhood by applying the typical genetic operations (crossover, mutation, and replacement).

MOCell includes an external archive to store the non-dominated solutions found as the algorithm progresses. This archive is limited in size and uses the crowding distance of NSGA-II to maintain diversity. The pseudocode of MOCell is presented in Algorithm 2, which corresponds with the version called aMOCell4, described in [21].

Algorithm 2 Pseudocode of MOCell.

```
1:  proc Input:(MOCell)       //Algorithm parameters in 'MOCell'
2:  archive ← ∅ //Creates an empty archive
3:  while not Termination_Condition() do
4:      for individual ← 1 to MOCell.popSize do
5:          n_list←Get_neighbourhood(MOCell,position(individual))
6:          parent1←Selection(n_list)
7:          parent2←Selection(archive)
8:          offspring←Recombination(MOCell.Pc,parent1, parent2)
9:          offspring←Mutation(MOCell.Pm,offspring)
10:         Evaluate_Fitness(offspring)
11:         Replacement(position(individual),offspring,MOCell)
12:         Insert_Pareto_Front(offspring, archive)
13:     end for
14: end while
15: end_proc
```

We can observe that, in this version, for each individual we select one parent from its neighbourhood and one from the archive, in order to guide the search towards the best solutions found (lines 5 to 8). Then a new solution is created by applying the genetic operators to these parents. The new solution is used to replace the current solution (line 11), and it is considered for inclusion in the archive (line 12). This constitutes a single iteration of the algorithm. The overall algorithm iterates until a termination condition is fulfilled.

The Strength Pareto Evolutionary Algorithm (SPEA2) is a multi-objective evolutionary algorithm proposed by Zitler *et al.* in [22]. We show the algorithm's pseudocode in Algorithm 3. SPEA2 uses a population and an archive simultaneously in its operation. In it, each individual is assigned a fitness value that is the sum of its strength raw fitness and a density estimation. The strength value of a solution i represents the number of solutions (in either the population or the archive) that are dominated by that solution, that is $S(i) = |\{j | j \in P_t \cup \overline{P_t} \wedge i > j\}|$. The strength raw fitness value of a given solution i, on the contrary, is the sum of strengths of all the solutions that dominate it, and is subject to minimization, that is, $R(i) = \sum_{j \in P_t \cup \overline{P_t}, j > i} S(j)$. The algorithm applies the selection, crossover, and mutation operators to fill an archive of individuals; then, the non-dominated individuals of both the original population and the archive are copied into a new population. If the number of non-dominated individuals is greater than the population size, a truncation operator based on calculating the distances to the k-th nearest neighbour is used (a typical value is $k = 1$), $D(i) = \frac{1}{\sigma_i^k + 2}$, where σ_i^k is the distance from solution i to its k-th nearest neighbour. This way, the individuals having the minimum distance to any other individual are chosen.

Algorithm 3 Pseudocode of SPEA2.

```
1:  proc
2:  $t \leftarrow 0$
3:  Initialize($P_0$, $\overline{P_0}$)
4:  while not EndingCondition($t$,$\overline{P_t}$) do
5:      FitnessAssignment($P_t$, $\overline{P_t}$)
6:      $\overline{P_{t+1}} \leftarrow$ NonDominated($P_t \cup \overline{P_{t+1}}$)
7:      if $|\overline{P_{t+1}}| > \overline{N}$ then
8:          $\overline{P_{t+1}} \leftarrow$ Truncate($\overline{P_{t+1}}$)
9:      else
10:         $\overline{P_{t+1}} \leftarrow$ FillWithDominated($\overline{P_t}$)
11:     end if
12:     Parents $\leftarrow$ BinaryTournament($\overline{P_{t+1}}$)
13:     Offspring $\leftarrow$ Crossover(Parents)
14:     $\overline{P_{t+1}} \leftarrow$ Mutate(Offspring)
15:     $t \leftarrow t + 1$
16: end while
17: end_proc
```

PAES is a metaheuristic proposed by Knowles and Corne [23]. The algorithm is based on a simple (1+1) evolution strategy. To find diverse solutions in the Pareto optimal set, PAES uses an external archive of non-dominated solutions, which is also used to make decisions about new candidate solutions. An adaptive grid is used as a density estimator in the archive. The most remarkable characteristic of PAES is that it does not make use of any recombination operators (crossover). New solutions are generated only by modifying the current solution. The pseudocode of PAES is presented in Algorithm 4. It starts with a random solution (line 3). In each iteration, a new solution is produced by modifying the current solution (line 5). This new solution is included in the archive and it is considered as a potential replacement for the current solution (lines 7 to 14). These steps are repeated until the maximum number of evaluations is reached.

We have included PAES in our study because of its simplicity. PAES does not use any recombination operator, and its only parameter is the number of partitions of the adaptive grid of the archive. Its relative simplicity makes it attractive since there are comparatively few parameters that require tuning in order to know that the algorithm is being applied properly (e.g., population size, crossover probability, mutation probability).

We also apply a random search (RNDMulti). This is merely a 'sanity check'; all metaheuristic algorithms should be capable of comfortably outperform random search for a well-formulated optimization problem. The pseudocode of the RNDMulti is presented in Algorithm 5. The final result of this random search is the set of all the non-dominated solutions found.

Algorithm 4 Pseudocode of PAES.

```
1:  proc Input:(paes)        //Algorithm parameters in ‘paes’
2:  archive ← ∅
3:  currentSolution ← Create_Solution(paes) // Creates an initial solution
4:  while not Termination_Condition() do
5:      mutatedSolution←Mutation(currentSolution)
6:      Evaluate_Fitness(mutatedSolution)
7:      if IsDominated(currentSolution, mutatedSolution) then
8:          currentSolution ← mutatedSolution
9:      else
10:         if Solutions_Are_Nondominated(currentSolution, mutatedSolution) then
11:             Insert(archive, mutatedSolution)
12:             currentSolution ← Select(paes, archive)
13:         end if
14:     end if
15: end while
16: end_proc
```

Algorithm 5 Pseudocode of RNDMulti.

```
1: proc
2: archive ← ∅
3: currentSolution ← Create_Solution() // Creates an initial solution
4: while not Termination_Condition() do
5:     newSolution ← Create_Solution()
6:     Insert(archive, newSolution)
7: end while
8: end_proc
```

5. mM APPROACH

In this section we present the second approach. In this approach we use a mono-objective test data generator to obtain a set of test data with the highest coverage. The mono-objective test data generator deals with only one branch of the program at the same time. This is an advantage to obtain high coverage because the search can focus on covering the most complex branches of the program.

However, the resulting test suite is usually large, redundant and inefficient because these algorithms do not try to minimize the test suite size. One way to reduce the number of test cases in a test suite, and still test the same functionality, is by solving a Multi Objective Test Case Selection Problem (MOTCSP) on the given test suite. This problem was recently formalized by Yoo and Harman in [24] as follows: Given a test suite T and several objective functions F_i, we must find a subset $T' \subseteq T$ such that T' is a Pareto optimal set with respect to the objective functions. The resulting subset of the test suite, T', is composed of the non-dominated solutions considering the objectives as equally important.

In order to solve the MOTCSP we always use in the experimental section the multi-objective algorithm NSGA-II. Our implementation is able to generate a Pareto front from thousands of test cases previously generated by the mono-objective algorithms. But first, we delete repeated test cases from the obtained test suite in order to reduce from thousands of test cases to hundreds of them. Two test cases are repeated when both of them traverse the same branches. We have compared the results obtained with and without this reduction phase, and the results are better when this reduction is applied. Finally, for the mono-objective algorithm involved in the first phase of test data generation, we use three different algorithms: a genetic algorithm, an evolutionary strategy and a random search. In the following we describe in detail the test data generator and the algorithms used as its search engine.

5.1. Test Data Generator

Our test data generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Then, each partial objective can be treated as a separate optimization problem in which a solution to the problem is a test datum and the function to be minimized is a distance between the current test datum and one satisfying the partial objective. In order to solve such minimization problem EAs are used. The main loop of the test data generator is shown in Figure 4.

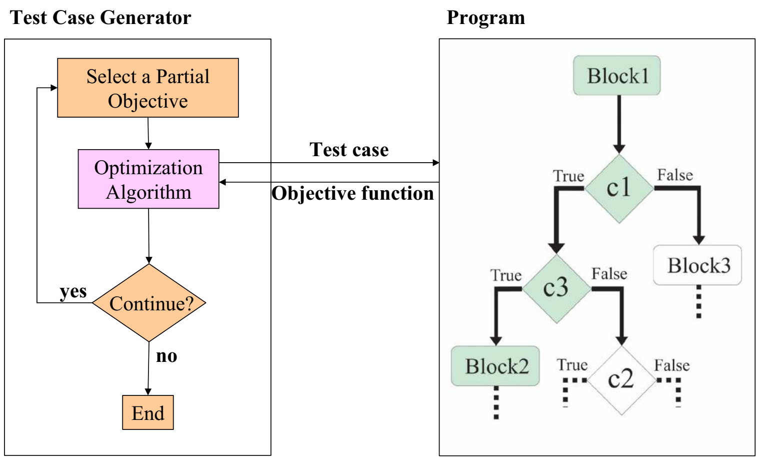


Figure 4. The test data generation process.

In a loop, the test data generator selects a partial objective (a branch) and uses the optimization algorithm to search for test data exercising that branch. When a test datum covers a branch, the test datum is stored in a set associated to that branch. The structure composed of the sets associated to all the branches is called *coverage table*. After the optimization algorithm stops, the main loop starts again and the test data generator selects a different branch. This scheme is repeated until total branch coverage is obtained or a maximum number of consecutive failures of the optimization algorithm is reached. When this happens the test data generator exits the main loop and returns the sets of test data associated to all the branches. In the following two sections we describe two important issues related to the test data generator: the objective function to minimize and the optimization algorithms used.

5.2. Objective Function

We have to solve several minimization problems: one for each branch. Now we need to define an objective function (for each branch) to be minimized. This function will be used for evaluating each test datum, and its definition depends on the desired branch and whether the program flow reaches the branching condition associated to the target branch or not. If the condition is reached we can define the objective function on the basis of the logical expression of the branching condition and the values of the program variables when the condition is reached. The resulting expression is called *branch distance* and can be recursively defined on the structure of the logical expression. That is, for an expression composed of other expressions joined by logical operators the branch distance is computed as an aggregation of the branch distance applied to the component logical expressions.

For the Java logical operators && and || we define the branch distance as:

$$bd(a\&\&b) = bd(a) + bd(b) \quad (2)$$

$$bd(a||b) = \min(bd(a), bd(b)) \quad (3)$$

where a and b are logical expressions.

In order to completely specify the branch distance we need to define its value in the base case of the recursion, that is, for atomic conditions. The particular expression used for the branch distance in this case depends on the operator of the atomic condition. The operands of the condition appear in the expression. A lot of research has been devoted in the past to the study of appropriate branch distances in software testing. An accurate branch distance considering the value of each atomic condition and the value of its operands can better guide the search. In procedural software testing these accurate functions are well-known and popular in the literature. They are based on distance measures defined for relational operators like $<$, $>$, and so on [25]. We use here these distance measures described in the literature.

When a test datum does not reach the branching condition of the target branch we cannot use the branch distance as objective function. In this case, we identify the branching condition c whose value must first change in order to cover the target branch (critical branching condition) and we define the objective function as the branch distance of this branching condition plus the *approximation level*. The approximation level, denoted here with $ap(c, b)$, is defined as the number of branching nodes lying between the critical one (c) and the target branch (b) [26].

In this paper we also add a real valued penalty in the objective function to those test data that do not reach the branching condition of the target branch. With this penalty, denoted by p, the objective value of any test datum that does not reach the target branching condition is higher than the one of any test datum that reaches the target branching condition. The exact value of the penalty depends on the target branching condition and it is always an upper bound of the target branch distance. Finally, the expression for the objective function is as follows:

$$f_b(x) = \begin{cases} bd_b(x) & \text{if } b \text{ is reached by } x \\ bd_c(x) + ap(c,b) * p & \text{otherwise} \end{cases} \tag{4}$$

where c is the critical branching condition, and bd_b, bd_c are the branch distances of branching conditions b and c. The use of the penalty p could be avoided by normalizing the branch distance to the interval $[0, 1)$ (see [27] for example). However, in this work we do not normalize the branch distance, thus, requiring the penalty value p, which is set to $p = 10000$ in the experiments.

Nested branches pose a great challenge for the search. For example, if the condition associated to a branch is nested within three conditional statements, all the conditions of these statements must be true in order for the program flow to proceed onto the next one. Therefore, for the purposes of computing the objective function, it is not possible to compute the branch distance for the second and third nested conditions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMinn calls the *nesting problem* [28]), which forces us to concentrate on satisfying each predicate sequentially.

In order to alleviate the nesting problem, the test data generator selects as objective in each loop one branch whose associated condition has been previously reached by other test data stored in the coverage table. Some of these test data are inserted in the initial population of the EA used for solving the optimization problem. The percentage of individuals introduced in this way in the population is called the *replacement factor* and is denoted by Rf. At the beginning of the generation process some random test data are generated in order to reach some branching conditions.

5.3. Mono-Objective Algorithms

We use two EAs inside the test data generator used in the mM approach: a genetic algorithm and an evolutionary strategy. Let us first describe the general structure of an EA and then we detail the differences between the EAs used here. In Figure 6 we show the main loop of an EA.

Initially, the algorithm creates a population of μ individuals randomly or by using a seeding algorithm. At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation in order to compute a set of λ descendant individuals Q. The objective of the selection operator is to select some individuals from the population to which the other operators will be applied. The recombination operator generates a new individual from several ones by combining their solution components. This operator is able to put together good solution components that are scattered in the population. On the other hand, the mutation operator modifies one single individual and is the source of new different solution components in the population. The individuals created are evaluated according to the fitness function. The last step of the loop is a replacement operation in which the individuals for the new population $P(t + 1)$ are selected from the offspring $Q(t)$ and the old one $P(t)$. This process is repeated until a stop criterion is fulfilled, such as reaching a pre-programmed number of iterations of the algorithm or finding an individual with a preset target

Algorithm 6 Pseudocode of an EA.

```
1:  proc Input: (ea)
2:  t=0:
3:  P(t) ← Create_Population() // P = population
4:  Q ← ∅                    // Q = auxiliar population // Creates an initial solution
5:  while not Termination_Condition() do
6:      for i ← 1 to (ea.popSize) do
7:          parents←Selection(P(t))
8:          offspring←Recombination(ea.Pc,parents)
9:          offspring←Mutation(ea.Pm,offspring)
10:         Evaluate_Fitness(offspring)
11:         Insert(offspring,Q(t))
12:     end for
13:     P(t+1) := Replace (Q(t),P(t))
14:     t= t + 1
15: end while
16: end_proc
```

quality. In this work we use two EAs as the optimization algorithm of the test data generator: an evolutionary strategy (ES) and a genetic algorithm (GA). In the following we focus on the details of the ES. We defer the details of the GA to the parameterization section.

In an ES [29] each individual is composed of a vector of real numbers representing the problem variables ($\mathbf{x}$), a vector of standard deviations (σ) and a vector of angles (ω). These two last vectors are used as parameters for the main operator of this technique: the Gaussian mutation. They are evolved together with the problem variables themselves, thus allowing the algorithm to self-adapt the search to the landscape. The mutation operator is governed by the three following equations:

$$\sigma_i' = \sigma_i \exp(\tau N(0,1) + \eta N_i(0,1)) \quad (5)$$

$$\omega_i' = \omega_i + \varphi N_i(0,1) \quad (6)$$

$$\mathbf{x}' = \mathbf{x} + \mathbf{N}(\mathbf{0}, C(\sigma', \omega')) \quad (7)$$

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω', $N(0,1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix C. The subindex i in the standard normal distribution indicates that a new random number is generated anew for each component of the vector. The notation $N(0,1)$ is used for indicating that the same random number is used for all the components. The parameters τ, η, and φ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [30]. For the recombination operator of an ES there are many alternatives: each of the three real vectors of an individual can be recombined in a different way. In our particular implementation, we use discrete uniform recombination for the solution vector $\mathbf{x}$, where each component is selected from the best parent with a predefined probability, called *bias*. For the vector of standard deviations and angles we use arithmetic recombination. The exact expressions for the components of the vectors are:

$$\mathbf{x}_i = \begin{cases} \mathbf{x}_i^1 & \text{if } U(0,1) < bias \\ \mathbf{x}_i^2 & \text{otherwise} \end{cases} \quad (8)$$

$$\sigma_i = (\sigma_i^1 + \sigma_i^2)/2 \quad (9)$$

$$\omega_i = (\omega_i^1 + \omega_i^2)/2 \quad (10)$$

where the superindices are used to denote the two parent solutions ($\mathbf{x}^1$ is the best one) and $U(0,1)$ denotes a random sample of a uniform distribution in the interval $[0,1)$. With respect to the replacement operator, there is a special notation to indicate whether the old population is taken into account or not to form the new population. When only the new individuals are used, we have a (μ, λ)-ES; otherwise, we have a $(\mu + \lambda)$-ES. Regarding the representation, since all the test programs have integer parameters, each component of the vector solution $\mathbf{x}$ is rounded to the nearest integer and used as actual parameter of the program. There is no limit in the input domain, thus allowing the ES to explore the whole solution space.

We also apply a random algorithm (RNDMono) as search engine of our test data generator (see Algorithm 7). This is again merely a 'sanity check'. The final result of this random search is the set of all the created solutions.

Algorithm 7 Pseudocode of RNDMono.

```
1: proc
2: conditionTable ← $\emptyset$
3: currentSolution ← Create_Solution() // Creates an initial solution
4: while not Termination_Condition() do
5:     newSolution ← Create_Solution()
6:     Insert(conditionTable, newSolution)
7: end while
8: end_proc
```

6. EXPERIMENTAL METHODOLOGY

This section is aimed at presenting the indicators used to measure the quality of the obtained results and the benchmark programs we have used. It also describes how the solutions of the problem have been encoded and the genetic operators employed, the configuration of the algorithms, and the methodology we have followed.

6.1. Quality Indicators

Two different issues are normally considered for assessing the quality of the results computed by a multi-objective optimization algorithm:

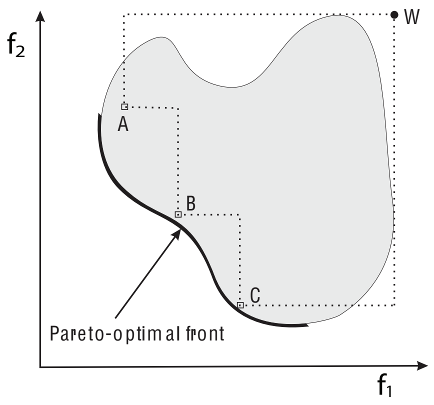
1. To minimize the distance of the computed solution set by the proposed algorithm to the optimal Pareto front (convergence towards the optimal Pareto front).
2. To maximize the spread of solutions found, so that we can have a distribution of vectors as smooth and uniform as possible (diversity).

A number of quality indicators have been proposed in the literature. Among them, we can distinguish between *Pareto compliant* and *non Pareto compliant* indicators [31]. Given two Pareto fronts, A and B, if A dominates B, the value of a Pareto compliant quality indicator is higher for A than for B; meanwhile, this condition is not fulfilled by the non–compliant indicators. Thus, the use of Pareto compliant indicators should be preferable. In this work, we apply the Hypervolume [32] (Pareto compliant), which takes into account the convergence as well as the diversity of the solutions; and Empirical Attainment Surfaces [33], which measures the probability of being dominated by the approximated Pareto front. Both indicators are defined as follows:

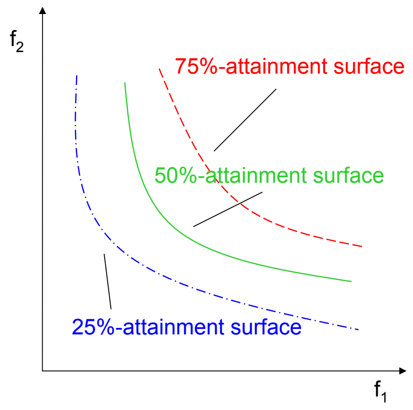
- **Hypervolume (HV).** This indicator calculates the volume (in the objective space) covered by members of a non-dominated set of solutions Q (the region enclosed into the discontinuous line in Figure 5(a), $Q = \{A, B, C\}$) for problems where all objectives are to be minimized. Mathematically, for each solution $i \in Q$, a hypercube v_i is constructed with a reference point W and the solution i as the diagonal corners of the hypercube. The reference point can simply be found by constructing a vector of the worst objective function values. Thereafter, a union of all hypercubes is found and its hypervolume (HV) is calculated:

$$HV = \text{volume}\left(\bigcup_{i=1}^{|Q|} v_i\right). \quad (11)$$

  We apply this metric after a normalization of the objective function values to the range $[0..1]$. A Pareto front with a higher HV than another one could be due to: some solutions in the better front dominate solutions in the other, or, solutions in the better front are more widely



(a) The hypervolume enclosed by the non-dominated solutions.



(b) 25%-, 50%-, and 75%-attainment surfaces.

Figure 5. Examples of hypervolume and attainment surfaces.

distributed than in the other. Since both properties are considered to be good, algorithms with larger values of HV are considered to be desirable. To apply this quality indicator, it is usually necessary to know the optimal Pareto front (form normalization purposes). Of course, typically, we do not know the location of the optimal front. Therefore, we employ as a *reference Pareto optimal front* the front composed of all the non-dominated solutions out of all the executions carried out (i.e., the best front known until now).

- **Empirical Attainment Surfaces** (EAS): In the related literature, the trade-off between several objectives in a MOP is usually presented by showing one of the approximated Pareto fronts obtained in one single run of a given algorithm. However, the optimization algorithms used are stochastic and therefore there is no warranty that the same result is obtained after a new run of the algorithm. Thus, a single run of a stochastic algorithm gives no information about the average performance of the algorithm. We need a way of representing the results of a multi-objective algorithm that allows us to observe the expected performance and its variability, in the same way as the average and the standard deviation are used in the single-objective case. To do this we use the concept of empirical attainment function (EAF) [33]. In short, the EAF is a function α from the objective space $\mathbb{R}^n$ to the interval $[0, 1]$ that estimates for each vector in the objective space the probability of being dominated by the approximated Pareto front of one single run of the multi-objective algorithm. Given the r approximated Pareto fronts obtained in the different runs, the EAF is defined as:

$$\alpha(z) = \frac{1}{r}\sum_{i=1}^{r} I(A^i \preceq \{z\}) \tag{12}$$

where A^i is the i-th approximated Pareto front obtained with the multi-objective algorithm and I is an indicator function that takes value 1 when the predicate inside it is true, and 0 otherwise. The predicate $A^i \preceq \{z\}$ means A^i dominates solution z. Thanks to the attainment function, it is possible to define the concept of k%-attainment surface [33]. The attainment function α is a scalar field in $\mathbb{R}^n$ and the k%-attainment surface is the level curve with value $k/100$ for α. Informally, the 50%-attainment surface in the multi-objective domain is analogous to the median in the single-objective one. In a similar way, the 25%- and 75%-attainment surfaces can be used as the first and third "quartile fronts" and the region between them could be considered a kind of "interquartile region" (see Fig. 5(b)). When the number of objectives is one, the 50%-attainment surface is the median and the "interquartile region" is the interquartile range.

6.2. Automatic Program Generator

We designed an automatic program generator able to generate programs similar to the ones of the real-world software. To achieve this goal, we focus on measures, made on source code pieces, which do not require the execution of the program, called static measures. Once we have computed these static measures in real-world software, we generate programs having values of the static measures that are similar to the ones of the real-world programs. The main characteristic of our program generator is that it is able to create programs for which total branch coverage is possible, but they do not solve any concrete problem. We propose this generator with the aim of generating a big benchmark of programs with certain characteristics chosen by the user.

In a first approximation we could create a program using a representation based on a syntax tree and a table of variables. The tree stores the sentences that are generated and the table of variables stores basic information about the variables declared and their possible use. With these structures, we are able to generate programs, but we can not ensure that all the branches of the generated programs are reachable. The unreachability of all the branches is a quite common feature of real-world programs, so we could stop the design for the generator at this stage. However, another objective of the program generator is to be able of creating programs that can be used to compare the performance of different algorithms for test data generation. In this case, programs for which total coverage is reachable are desirable, because we can use the coverage obtained by a test suite as a measure of its quality.

Let us illustrate this with an example. Let us suppose that a given tool for automatic test case generation is able to find test suites that cover 80% of the branches of program A and 90% of the branches of program B (a test suite for each program). It seems that the tool is more effective in program B, since it is able to generate a test suite with higher coverage. Now, imagine that 20% of the branches of program A are unreachable and all the branches of program B are reachable. Then, the tool obtained the maximum possible coverage in program A but not in program B. Thus, we would say that the tool is more effective for program A.

This example shows that coverage is not a good measure of the performance of an automatic tool for test case generation if we do not know the maximum reachable coverage for each program. In previous work, alternative measures, such as corrected coverage [34] have been used to alleviate this problem. In this work we adopt a different approach. Since we automatically generate the programs of the benchmark, we decided to generate programs for which the maximum coverage is always 100%. This way, we can safely use the coverage as a measure of performance of the automatic test data generators and we can compare these generators among them using coverage.

An alternative approach would be to generate programs for which there is no warranty that 100% of branch coverage can be obtained and then analyze these programs in order to find the maximum possible coverage and use a corrected coverage measure. However, the drawback of this alternative approach is that the size of the benchmark would be limited, since we should check all the programs by hand (the automatic determination of the maximum branch coverage is an undecidable problem), and we would lose statistical confidence in the results. It would be unviable to generate 800 programs (as we do in this work) and analyze them manually.

With the goal of generating programs for which total coverage is reachable, we thought in the way the variables are treated in symbolic execution [35, 36] and some methods of formal derivation of programs [37, 38, 39]. Unlike the formal derivation of programs, our generator is not guided by a specification. Therefore, at the end, we introduce logic predicates in the program generation process in order to generate programs for which total coverage is always ensured.

The program generator is parameterizable, the user can fix several parameters of the program under construction (*PUC*). Thus, we can assign the probability distributions of the number of sentences of the *PUC*, the number of variables, the maximum number of atomic conditions per condition, and the maximum nesting degree. Another parameter the user can tune is the percentage of control structures or assignment sentences that will appear in the code. By tuning this parameter the program will contain the desired density of decisions.

Once the parameters are fixed, the program generator builds the general scheme of the PUC. It stores in the syntax tree the program structure, and creates a main method where the local variables are first declared. Then, the program is built through a sequence of basic blocks of sentences where, according to a probability, the program generator decides which sentence will be added to the program. The creation of the entire program is done in a recursive way. The user can decide whether all the branches of the generated program are reachable or not.

If total reachability is desired, logic predicates are used to represent the set of possible values that the variables can take at a given point of the PUC. Using these predicates we can know the range of values that a variable can take. This range of values is useful to build a new condition that can be *true* or *false*. For example, if at a given point of the program we have the predicate $x \leq 3$ we know that a forthcoming condition $x \leq 100$ will be always true and if this condition appears in an `if` statement, the `else` branch will not be reachable. The predicates are thus used to guide the program construction to obtain a 100% coverable program.

In general, at each point of the program the predicate is different. During the program construction, when a sentence is added to the program, we need to compute the predicate at the point after the new sentence. For this computation we distinguish two cases. First, if the new sentence is an assignment then the new predicate CP' is computed after the previous one CP by updating the values that the assigned variable can take. For example, if the new sentence is $x = x + 7$ and $CP \equiv x \leq 3$, then we have $CP' \equiv x \leq 10$.

Second, if the new sentence is a control statement, an `if` statement for example, then the program generator creates two new predicates called True-predicate (TP) and False-predicate (FP). The TP is obtained as the result of the AND operation between CP and the condition related to the control statement. The FP is obtained as the result of the AND operation between the CP and the negated condition. In order to ensure that all the branches can be traversed, we check that both, TP and FP are not equivalent to *false*. If any of them were false, this new predicate is not valid and a new control structure would be generated.

Once these predicates are checked, the last control statement is correct and new sentences are generated for the two branches. The predicates are computed inside the branches in the same way. After the control structure is completed, the last predicates of the two branches are combined using the OR operator and the result is the predicate after the control structure. In Figure 6 we illustrate the previous explanation with one example.

```
/* $CP_1 \equiv x \leq 3$ */
if (x < 0)
{
    /* $CP_2 \equiv TP_1 \equiv x \leq 3 \wedge x < 0 \equiv x < 0$ */
    y=5;
    /* $CP_3 \equiv x < 0 \wedge y = 5$ */
}
else
{
    /* $CP_4 \equiv FP_1 \equiv x \leq 3 \wedge x \geq 0 \equiv 0 \leq x \leq 3$ */
    x=x-3;
    /* $CP_5 \equiv -3 \leq x \leq 0$ */
}
/* $CP_6 \equiv x < 0 \wedge y = 5 \vee -3 \leq x \leq 0$ */
```

Figure 6. Illustration of the predicates transformation.

6.3. Benchmark of Test Programs

In the experimental section we use two benchmarks. The first one is composed of 800 synthetic programs[‡]. They are described in the next section. The second one is composed of 13 real-world programs that are described in Section 6.3.2.

6.3.1. Synthetic Programs The program generator can create programs having the same value for the static measures, as well as programs having different values for the measures. In addition, the generated programs are characterized by having a 100% coverage, thus all possible branches are reachable.

Our program generator takes into account the desired values of some static measures. The static measures selected are: the number of atomic conditions, the nesting degree, the number of sentences and the number of variables. The main features of the generated programs are: they deal with integer input parameters, their conditions are joined by whichever logical operator, they are randomly generated and all their branches are reachable.

The methodology applied for the program generation was the following. First, we analyzed a set of Java source files from the JDK 1.5 (java.util.*, java.io.*, java.sql.*, etc.) and we computed the static measures on these files. Next, we used the ranges of the most interesting values, obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. This way, we generated programs with the values in these ranges, e.g., nesting degree in 1-4 (25% for each value), atomic conditions per condition in 1-4 (68.43% with 4 conditions per decision), and statements in 25, 50, 75 or 100 (25% for each value). The percentage of control flow statements is 32.23% (in this work we use IF statements), this means that the test case generator should cover around 64 different branches (32 true and 32 false) in programs with 100 statements. The previous values are realistic with respect to the static measures, making our study meaningful. Besides, we generated 50 programs for each size and nesting degree (50 x 4 sizes x 4 nesting degrees = 800), which is a total of 800 Java programs.

6.3.2. Real Programs In order to improve the interest of our work we propose an additional benchmark of real programs. It is composed of 13 real programs extracted from the literature [40, 41, 42]. Some of them have been extracted from the book *C Numerical Recipes*, available on-line at http://www.nr.com/. They deal with real and integer input values and some of them also contain loops. The programs are listed in Table I, where we inform on the maximum nesting degree, the lines of code (LOC), the number of branches, and the number and type of input arguments.

Table I. Characteristics of the Real Programs

Name	ND	LOC	Branches	Arguments	Description
calday	2	47	22	3 Integer	Calculate the day of the week
complex	3	74	24	6 Integer	Calculate complex arithmetic functions
gcd	2	28	8	2 Integer	Greatest common denominator
line	8	92	36	8 Integer	Check if two rectangles overlap
numbers	3	71	28	1 Integer	Parse a big number from integer to string
qformula	2	24	4	3 Double	Solve Real Equations
qformulas	2	22	6	3 Integer	Solve Integer Equations
remainder	6	49	18	2 Integer	Calculate the remainder of an integer division
tmichael	5	69	20	3 Integer	Classify triangles in 4 types: Michael
tmyers	6	54	12	3 Integer	Classify triangles in 4 types: Myers
triangle	4	53	28	3 Integer	Classify triangles in 4 types: Our implementation
tsthamer	3	76	26	3 Integer	Classify triangles in 5 types: Sthamer
twegener	3	46	26	3 Double	Classify triangles in 5 types: Wegener

[‡]They are available at http://neo.lcc.uma.es/mase/index.php/component/content/article/48-problems/121-source-of-800-sythetic-programs

6.4. *Solution Encoding, Genetic Operators and Configuration*

Here we detail the configuration of the operators and the encoding of the solutions used in the algorithms.

6.4.1. Details of the Mono-objective algorithms: In this work, each solution is encoded as an integer/real vector of length n (the number of arguments). As we said in Section 5.1 the generator breaks down the global objective (to cover all the branches) into several partial objectives consisting of dealing with only one branch of the program. Thus, two stopping conditions exist: one for partial objectives and the other one for the whole test data generation process. The search for one partial objective stops when 1000 evaluations are performed while the test data generation process ends after 150000 evaluations.

In our GA we use as recombination operator the uniform crossover (UX), in which each component of the new solution is randomly selected from the two parents. The formal definition is the same as Equation (8) with $bias = 0.5$. The mutation operator adds a random value to the components of the vector. That is,

$$x_i = x_i + U(-500, 500) \tag{13}$$

where the probability distribution of these random values is a uniform distribution in the range $[-500, 500]$. However, not all the components of the individual are perturbed, only half of them are. In our ES, we use a discrete crossover operator and a Gaussian mutation. We show in Table II a summary of the parameters used by the two EAs in the experimental section.

Table II. Parameters of the two mono-objective EAs used in the experimental section

	ES	GA
Population	25 indivs.	25 indivs.
Selection	Random, 5 indivs.	Random, 5 indivs.
Mutation	Gaussian	Add $U(-500, 500)$
Crossover	discrete (bias = 0.6) + arith. + arith.	Uniform
Replacement	Elitist	Elitist
Stopping cond.	1000 evals.	1000 evals.
Total Evals.	150000 evals.	150000 evals.

After the execution of the test data generator, we obtain a huge table of coverage where the test data that satisfy a concrete branch during the execution are saved. This table is filtered in order to remove those test data for which a different test exist in the table traversing the same branches, as explained in Section 5. Then, a test data selection is performed over this set using a standard NSGA-II.

6.4.2. Details of the Multi-objective algorithms: In the multi-objective approach, each individual is encoded as a set of test data. In Table III can be seen the parameters of the multi-objective EAs used in the experimental section. As genetic operators, we have used *binary tournament* as the selection scheme. This operator works by randomly choosing two individuals from the population and the one dominating the other is selected; if both solutions are non-dominated one of them is randomly selected.

We created some crossover operators to increase the efficiency of the algorithm. The best results were obtained with the *union crossover*. It takes two solutions, C_1 and C_2, and creates a new one C that is the union of both, that is: $C = C_1 \cup C_2$. If the resulting solution C has more coverage than C_1 and C_2 then C is the new offspring. Otherwise, the solution with more coverage (C_1 or C_2) is the new child.

Table III. Parameters of the Multi-objective EAs used in the experimental section

	NSGA-II	MOCell	SPEA2	PAES
Population	20 indivs.	20 indivs.	20 indivs.	20 indivs.
Selection	BT, 2 indivs.	BT, 2 indivs.	BT, 2 indivs.	BT, 2 indivs.
Mutation	Adaptive Mutation	Adaptive Mutation	Adaptive Mutation	Adaptive Mutation
Crossover	Union Crossover	Union Crossover	Union Crossover	-
Replacement	Elitist	Elitist	Elitist	Elitist
Total Evals.	150000 evals.	150000 evals.	150000 evals.	150000 evals.

Finally, the mutation operator adds new test data to the solution with proability 0.6, deletes one test datum with probability 0.2 and keeps the individual unchanged with proability 0.2. In the case of adding test data, the number of new test data is 30% of the test data present in the solution.

If the resulting individual has the same coverage and more test data, at the end of the iteration, the algorithm deletes it from the population because this solution is dominated.

All the multi-objective algorithms have been implemented using jMetal [43], a Java framework aimed at the development, experimentation, and study of metaheuristics for solving multi-objective optimization problems.

7. EXPERIMENTAL ANALYSIS

In this section we present the results of the two proposed approaches. In the first subsection we analyze the MM approach and we compare the performance of the multi-objective algorithms. In the second subsection we study the mM approach and we compare the performance of the mono-objective algorithms used as the base for the approach. Then, in a third subsection we compare the two proposed approaches for the academic benchmark, and finally, in the last subsection, we compare both approaches with a benchmark of real programs.

For the study we use the 800 Java programs automatically generated and another benchmark composed of 13 real programs. Both benchmarks were described in Section 6.3. Since we are dealing with stochastic algorithms, we need to perform several independent runs of each algorithm and program, 30 in our case, in order to obtain a very stable average of the measures. All test data generators used in this work proceed by generating test data until a maximum of 150,000 test data are generated. We also perform a multiple comparison statistical test for each program on the obtained results to compare the algorithms among them. We set a confidence level of 95% (*p*-value under 0.05) for the whole comparison (all the algorithms acting on a program) and we used the Bonferroni correction for each particular comparison.

7.1. Evaluation of the MM approach

In this section, we analyze the behaviour of the multi-objective algorithms with the aim of highlighting the algorithm that works better. We have analyzed 800 programs, so we cannot represent all HV values for all the programs. For this reason, we summarize in Table IV the times one algorithm has better median HV than the others. We have classified the results according to the nesting degree and the size of the PUT. For this indicator, the higher the value, the better the quality of the obtained results. Thus, by looking at the tables, we can see that MOCell was usually the algorithm computing clearly the best results regarding HV. However, when the programs are small (25-50 statements) and complex (nesting degree four), the NSGA-II algorithm has a better behaviour. We must highlight the big difference between MOCell (443), NSGA-II (198) and the others altogether (43).

Then, we compare the HV values of all the programs and independent executions with the Kruskal-Wallis test. In each cell of a table of statistics we have a pair (number, triangle). The number indicates how many programs are significantly different, and the triangle indicates that the program in the row is significantly better (▲) or worse (▽) than the program in the column. The results are summarized in Table V. Although the previous values set a clear tendency, the absence of

Table IV. Programs in which the median Hypervolume of one algorithm is better than the others

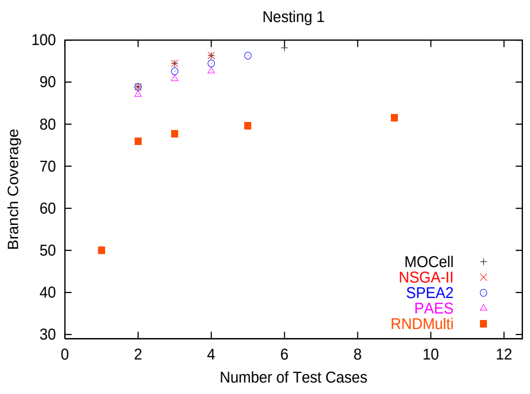
Nesting degree	Statements	MOCell	NSGA-II	SPEA2	PAES	RNDMulti
1	25	10	1	0	0	0
	50	24	9	0	2	2
	75	34	6	1	1	0
	100	38	4	0	1	0
	Total	106	20	1	4	2
2	25	13	5	1	2	3
	50	35	13	0	0	0
	75	37	12	0	0	0
	100	40	10	0	0	0
	Total	125	40	1	2	3
3	25	18	11	3	1	2
	50	33	15	0	0	0
	75	32	16	1	0	0
	100	30	19	0	0	0
	Total	116	61	4	1	2
4	25	17	20	3	2	2
	50	23	25	2	1	0
	75	27	19	2	0	1
	100	29	13	10	0	0
	Total	96	77	17	3	3
Total		443	198	23	10	10

significant differences between MOCell, NSGA-II and SPEA2, does not allow us to say that MOCell is better than the other two. However, we can mention that RNDMulti is the worst algorithm in all the programs (800) and PAES is worse than MOCell in 18 programs, NSGA-II in 9 programs, and SPEA2 in only 2 programs.

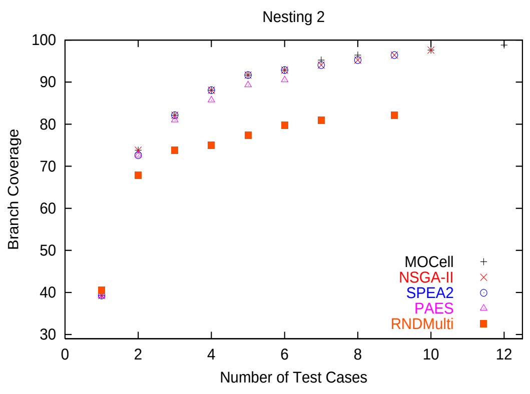
Table V. Number of programs where there exists significant difference among the HV obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell
MOCell	800▲	18▲	0	0	–
NSGA-II	800▲	9▲	0	–	0
SPEA2	798▲	2▲	–	0	0
PAES	750▲	–	2▽	9▽	18▽
RNDMulti	–	750▽	798▽	800▽	800▽

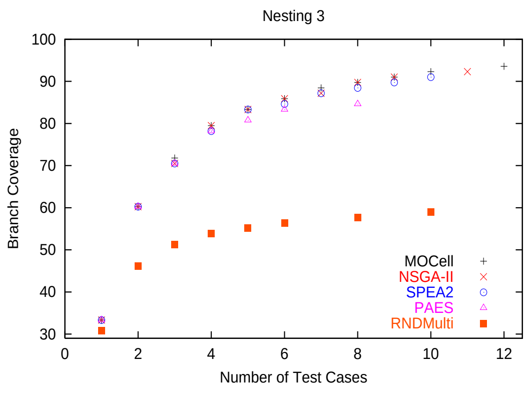
With the aim of showing an example of the computed fronts for the instances, we selected one program for each nesting degree, which can represent the typical behaviour of the different algorithms in this kind of instance. In Figure 7 are depicted the 50%-attainment surfaces of these selected programs. In the instance with low nesting degree, MOCell dominates the others and has a good performance because it reaches almost the same or better coverage with the same test data. NSGA-II has a similar behaviour except in the right extreme of the figure where it is not able to reach the same maximum coverage as MOCell. On the other hand, in the program with nesting degree 4, NSGA-II is the algorithm that is able to reach the best coverage and dominates all the other fronts. The other two multi-objective algorithms (SPEA2 and PAES) have problems finding the solutions with high coverage, in the upper-right bound of the figure, and are worse than MOCell and NSGA-II. RNDMulti is always the worst. MOCell has been able to find non-dominated solutions in the right area where SPEA2, PAES and RNDMulti have not found any of them (solutions in the extremes of the front). This is related to a better exploration of the search space by MOCell. Specifically, this is one of the properties of the cellular GA model, on which MOCell is based. This fact has been reported in many studies on single-objective optimization (see [44]). There is only one exception, when a program has nesting degree 4 and it is more difficult to obtain high coverage, NSGA-II has the best performance.



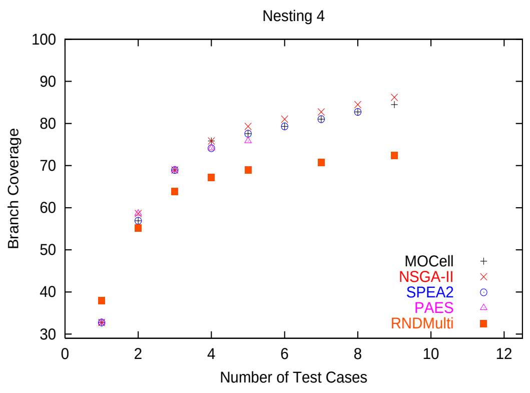
(a) Program with nesting degree 1



(b) Program with nesting degree 2



(c) Program with nesting degree 3



(d) Program with nesting degree 4

Figure 7. 50%-attainment surfaces: coverage against the number of test cases.

We have also analyzed the reduction obtained in the number of test cases, since one of our goals is to minimize the number of test cases. We analyze the reduction experienced using our approaches compared with the use of all the generated test cases. It is very difficult to analyze this reduction because not all the algorithms achieve a 100% coverage in all the programs. For this reason, we cannot simply average the number of test cases, but we must take into account the maximum obtained coverage in order to give the real reduction made by the multi-objective algorithm. The total reduction is from thousands of test cases generated to around ten, but this reduction could also be easily computed based in the *table of coverage* of the algorithms by choosing one test case per branch. The drawback of the latter approach is that the minimization of the test suite would be far from optimal. For this reason, we establish a theoretical upper bound of the required number of test cases needed. This upper bound is the number of branches that were achieved by the algorithm. We compute the real oracle cost of the test suites generated by any algorithm according to the next expression:

$$\begin{aligned} upper\_bound(P,A) &= B_P * MaxCov(P,A) \\ oracle\_cost(P,A) &= \frac{tc(P,A)}{upper\_bound(P,A)} \end{aligned} \tag{14}$$

where P is a program, A is an algorithm, B_P is the number of branches of the program P, $MaxCov(P, A)$ is the maximum coverage obtained by the algorithm A in the program P, and $tc(P, A)$ is the number of test cases needed by the algorithm A to obtain the maximum coverage in program P.

We can state that the oracle cost of the test suite generated by all the multi-objectives algorithms can be reduced by our approach, only 15.12% of the test cases are needed in comparison with the computed upper bound. This reduction is computed in the case of the maximum coverage, and hence the largest number of computed test cases. But we must bear in mind that our solution is a complete Pareto front offered to the expert to make a decision about the test suite that best fits his/her needs, therefore a similar percentage of reduction is carried out for each couple *coverage-number of test cases* that appears in the Pareto front.

In the TDGP, it is particularly hard to achieve a 100% branch coverage, specially if one uses a multi-objective algorithm because its execution is not entirely guided to obtain a total coverage. The multi-objective approach deals with all the branches at the same time, this provokes a lack of information. In addition, the search does not spend most of its effort to cover the most complex branches. In Table VI we show the average of maximum coverage (among the solutions in the front) obtained with the solutions for all the programs with different nesting degree. We highlight the maximum values in the table for each nesting degree. As we expected, MOCell's performance is the best on nesting degree 1, 2 and 3. On the other hand, NSGA-II obtains the best coverage with nesting degree 4. Since the differences are low, we compared the coverage values of all the programs and independent executions with the Kruskal-Wallis test. The results are summarized in Table VII. As we expected, MOCell obtains significant differences in more programs with respect to PAES and RNDMulti, than NSGA-II and SPEA2.

Table VI. Relationship between the nesting degree and the average maximum coverage for the multi-objective algorithms. The standard deviation is shown in subscript.

Nesting degree	MOCell	NSGA-II	SPEAII	PAES	RNDMulti
1	$98.10_{2.08}$	$97.90_{2.22}$	$97.53_{2.34}$	$93.08_{5.30}$	$81.36_{12.74}$
2	$94.77_{3.44}$	$94.42_{3.49}$	$93.56_{3.75}$	$87.59_{6.31}$	$75.04_{14.00}$
3	$90.66_{5.83}$	$90.41_{5.46}$	$89.29_{5.65}$	$81.55_{7.68}$	$69.77_{13.87}$
4	$85.50_{9.45}$	$85.77_{8.18}$	$84.61_{8.12}$	$75.87_{9.22}$	$63.87_{15.95}$
Total	$92.26_{7.54}$	$92.12_{6.99}$	$91.24_{7.24}$	$84.52_{9.72}$	$72.51_{15.57}$

Table VII. Number of programs where there exists a significant difference among the coverage values obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell
MOCell	800▲	800▲	2▲	0	–
NSGA-II	800▲	799▲	0	–	0
SPEA2	800▲	782▲	–	0	2▽
PAES	711▲	–	782▽	799▽	800▽
RNDMulti	–	711▽	800▽	800▽	800▽

If we consider the HV obtained (Table IV), the significant HV differences (Table V), the attainment surfaces and the average maximum coverage achieved showed in Table VI, it is clear that the ranking of the performance of the algorithms is: MOCell is the best, second NSGA-II, third SPEA2, fourth PAES, and finally RNDMulti, the worst one, as expected.

7.2. Evaluation of mM approach

In this section we analyze the mM approach. First of all, we study the values of HV. We show in Table VIII the programs in which one algorithm has a better value of HV.

Table VIII. Programs in which the median Hypervolume of one algorithm is better than the others.

Nesting degree	Statements	GA	ES	RNDMono
1	25	3	3	0
	50	7	18	1
	75	7	26	3
	100	9	33	3
	Total	26	80	7
2	25	13	8	1
	50	23	17	0
	75	23	22	1
	100	18	29	1
	Total	77	76	3
3	25	23	6	0
	50	31	16	0
	75	30	16	0
	100	21	29	0
	Total	105	67	0
4	25	37	3	0
	50	41	6	0
	75	39	11	0
	100	34	14	0
	Total	151	34	0
Total		359	257	10

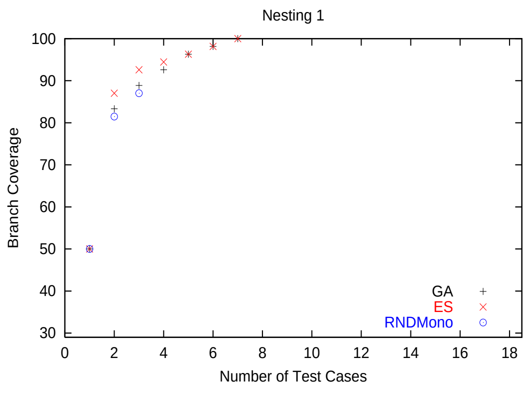
It is noteworthy that when the nesting degree is the smallest (1) the ES obtains better results and when the nesting degree is large (3 and 4) the GA is better than the others. In other words, when the program is more complex, the GA is clearly the best. The ES is better in large programs (100 statements) except when the program has nesting degree four. Then, we compared the HV values of all the programs and independent executions with the Kruskal-Wallis test. The results indicate that there is no significant difference between GA and ES (Table IX). As we expected, the results of RNDMono are worse than ES in 786 programs and GA in 765 programs.

Table IX. Programs where a significant difference exists among the HV obtained.

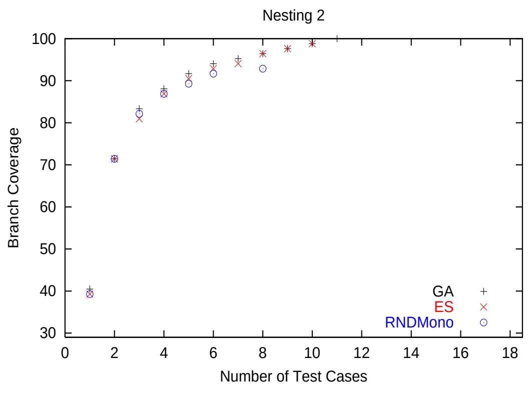
	RNDMono	ES	GA
GA	765▲	0	–
ES	786▲	–	0
RNDMono	–	786▽	765▽

Second, we show the 50%-attainment surfaces of four representative programs with different nesting degree in Figure 8. In the instance with nesting degree 1, the attainment surfaces are very similar between GA and ES. RNDMono is far from the behaviour of the others. In the instance with nesting degree 2, the three algorithms obtain similar results. The instances with nesting degree 3 and 4, represent the general behaviour of the algorithms in most of the programs. The RNDMono is far from the others, the ES obtains similar values of coverage to the GA with the same number of test cases, but GA can achieve the best value of coverage. The GA is the best algorithm in maximum obtained coverage. This is related to a better exploitation of the search space by GA.

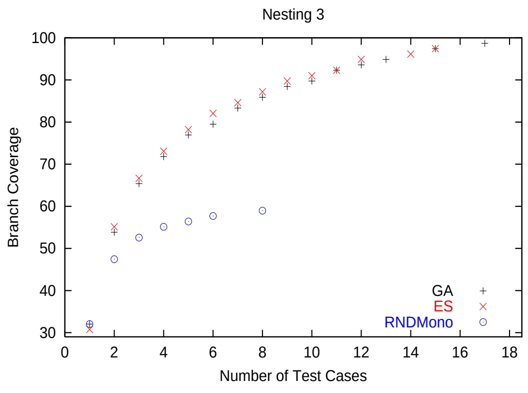
In order to highlight the reduction of the test cases needed to achieve the maximum coverage, we have applied Equation (14). We can state that the oracle cost of the test suite generated by the three



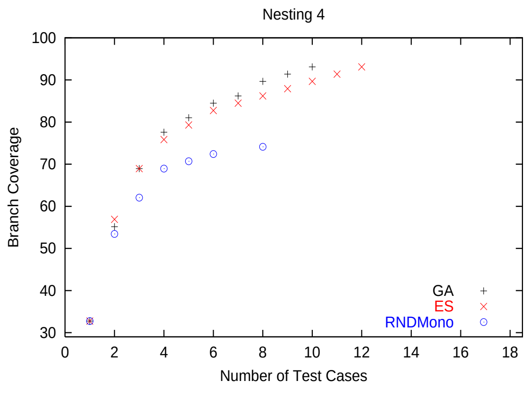
(a) Program with nesting degree 1



(b) Program with nesting degree 2



(c) Program with nesting degree 3



(d) Program with nesting degree 4

Figure 8. 50%-attainment surfaces: coverage against the number of test cases.

studied mono-objective algorithms can be reduced by our approach, only 19.32% of the test cases are needed in comparison with the computed upper bound. This percentage of test cases needed to achieve a concrete coverage is larger than the one obtained with the MM approach (15.12%).

Now, let us analyze the best value of coverage obtained with the three algorithms. In Table X we show the average of maximum coverage of the three algorithms. As is known, achieving a total coverage is a great challenge for the search, for this reason, we consider that an algorithm must focus on obtaining a high value of coverage. In this sense, the GA and ES obtain very good values of coverage, both above 90% in all the cases. However, the average coverage obtained by the GA is always the best. This advantage of the GA increases in programs with higher nesting degree where high values of coverage are very difficult to obtain. We performed a statistical test (Table XI), but, although the GA obtains the best results, significant differences only exist in 23 programs between GA and ES. Thus, it seems that GA is the best in obtaining a high value of coverage, specifically in more complex programs.

7.3. *MM vs. mM approaches*

In the previous sections we have performed a comparison between the algorithms used in each of the approaches. In the mM approach, GA seems to be the best algorithm in most of the programs and ES is the best algorithm in programs with the lowest nesting degree. Regarding the MM approach, MOCell was the best in most of the programs, except in a few programs with high nesting degree. In this section we compare all the algorithms together, with the aim of showing what technique is the overall best.

Table X. Relationship between the nesting degree and the average maximum coverage for the mono-objective algorithms. The standard deviation is shown in subscript.

Nesting degree	GA	ES	RNDMono
1	$99.19_{2.20}$	$98.70_{2.63}$	$85.33_{10.51}$
2	$98.85_{2.02}$	$97.87_{2.44}$	$79.20_{12.14}$
3	$98.52_{2.09}$	$95.66_{4.54}$	$71.94_{13.36}$
4	$96.89_{4.80}$	$93.19_{6.66}$	$66.42_{14.80}$
Total	$98.36_{3.13}$	$96.36_{4.90}$	$75.72_{14.65}$

Table XI. Number of programs where there exists a significant difference between the coverage obtained.

	RNDMono	ES	GA
GA	800▲	23▲	–
ES	800▲	–	23▽
RNDMono	–	800▽	800▽

First of all, we analyze the HV indicator. In Table XII we summarize the number of times where the HV value of an algorithm is better than the rest. The results show that, on the one hand, MOCell is better for programs with low nesting degree (1-2). On the other hand, the GA is better for programs with high nesting degree (3-4). The performance of the MOCell algorithm and the GA is similar but they work better in different kind of programs. This performance depends on the maximum nesting degree of the program. NSGA-II and the ES have similar performances among them; however they are clearly worse than MOCell and GA. Finally, the performance of SPEA2, PAES, RNDMono, and RNDMulti is clearly worse than the previous algorithms (MOCell, GA, NSGA-II and ES).

Table XII. Programs in which the median Hypervolume of one algorithm is better than the others.

ND	Statements	MOCell	NSGA-II	SPEA2	PAES	RNDMulti	GA	ES	RNDMono
1	25	1	0	0	0	0	0	0	0
	50	5	0	0	0	0	0	0	0
	75	16	2	0	1	0	0	0	0
	100	14	2	0	0	0	0	2	0
	Total	36	4	0	1	0	0	2	0
2	25	4	1	0	0	0	0	0	0
	50	16	4	0	0	0	4	2	0
	75	24	6	0	0	0	4	1	0
	100	26	8	0	0	0	6	4	0
	Total	70	19	0	0	0	14	7	0
3	25	4	0	1	0	0	6	1	0
	50	10	1	0	0	0	18	4	0
	75	14	4	0	0	0	20	9	0
	100	13	10	0	0	0	13	11	0
	Total	41	15	1	0	0	57	25	0
4	25	2	1	1	0	0	19	1	0
	50	6	5	0	0	0	34	0	0
	75	7	2	0	0	0	33	6	0
	100	3	6	4	0	0	27	7	0
	Total	18	14	5	0	0	113	14	0
Total		165	52	6	1	0	184	48	0

In order to clarify the obtained results, we have performed the statistical test to check if there exist significant differences among the HV values. In Table XIII, we can see that there is just a small significant difference among the main evolutionary algorithms. However, there are significant differences between the worse algorithms (the two random algorithms and PAES), and the rest. In

Table XIII we show that the HV values of GA are significantly better than the others, except the ES. The same observation can be made on ES: it is significantly better than the others (except the GA). NSGA-II, MOCell and SPEA2 are worse than GA and ES, but for most of the programs their HV values are better than the random algorithms. In some programs, there are significant differences between MOCell and PAES and also between NSGA-II and PAES.

In summary, the mM approach using the evolutionary algorithms (GA and ES) always achieves good HV values. We observed that MOCell, NSGA-II and SPEA2 are significantly better than PAES in more programs than GA and ES. In this case, the HV values of the mM approach are worse, concretely they do not get a good diversity because the Pareto fronts are computed from a finite subset of test cases obtained by the mono-objective algorithms. However, the MM approach takes better care of the convergence as well as the diversity of the Pareto front, consequently their HV values will be better. For the purpose of illustrating this issue we plot in Figure 9 the 50%-attainment surfaces for the best algorithms: MOCell, NSGA-II, GA, and ES.

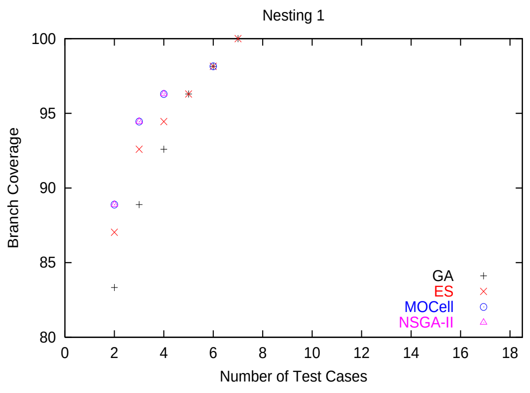
Table XIII. Programs where a significant difference exists among the HV obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell	RNDMono	GA	ES
MOCell	800▲	235▲	0	0	–	800▲	39▽	7▽
NSGA-II	800▲	197▲	0	–	0	800▲	29▽	5▽
SPEA2	800▲	61▲	–	0	0	800▲	13▽	2▽
PAES	799▲	–	61▽	197▽	235▽	645▲	36▽	18▽
RNDMulti	–	799▽	800▽	800▽	800▽	24▽	782▽	795▽
ES	795▲	18▲	2▲	5▲	7▲	737▲	0	–
GA	782▲	36▲	13▲	29▲	39▲	689▲	–	0
RNDMono	24▲	645▽	800▽	800▽	800▽	–	689▽	737▽

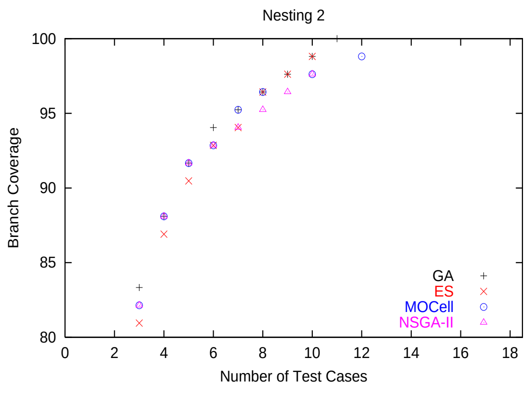
We focus on the most interesting area (80%-100% coverage) of the plots in Figure 9. In all the pictures, we appreciate that MOCell and NSGA-II have the best fronts in the programs with nesting degree 1, 2 and 3, although they do not obtain the best coverage in all cases. In addition, we must highlight that the fronts of GA and ES are dominated in this case. We find the exception when the program has nesting degree four, where the GA is the best algorithm because its solutions dominate the others. The second in performance is the ES; close to the values of GA. The other algorithms only find solutions with middle values of coverage and more test cases.

At this stage of the study, we know that the MM approach provides more diversity in the solutions. In other words, it is able to find a test suite with few test cases, but the obtained coverage is not very large. On the other hand, the mM approach is able to better explore the search space to find solutions with a high coverage, but it needs more test cases than the MM approach. The MM approach obtains worse average coverage because nested statements pose a great challenge for the search. We think that the main reason for this fact is that the multi-objective algorithms deal with all the branches at the same time and less information is obtained to guide the search.

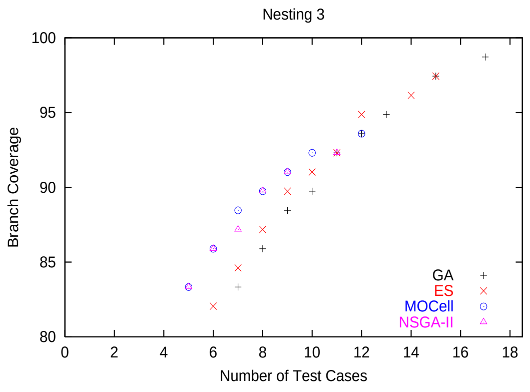
As we previously said, automatically generating a test suite that covers the entire program is a hard task. When a program has high nesting degree and the decisions are very complex, the task of covering all the program code requires a lot of effort. It is important for an algorithm to be able to find test cases to cover all the program's branches. We show in Table XIV a comparison of the average maximum coverage obtained for all the algorithms and all the programs. It is clear that the best algorithm, if coverage is the main objective, is GA. It obtains the best results in all the groups of programs with different nesting degree, and therefore in the complete benchmark. The performance of ES is also very good because it is always better than the multi-objective algorithms. If the nesting degree increases, the distance between the average coverage of GA and ES increases. In other words, the ES has a similar performance to GA in low complexity programs and it is worse than GA in complex programs. On the other hand, MOCell and NSGA-II have almost the same coverage; it only varies at the decimal level. SPEA2 is only 1% worse in the entire benchmark with respect to MOCell and NSGA-II. The results of the PAES algorithm are in the middle between the best (GA) and the worst (RNDMulti).



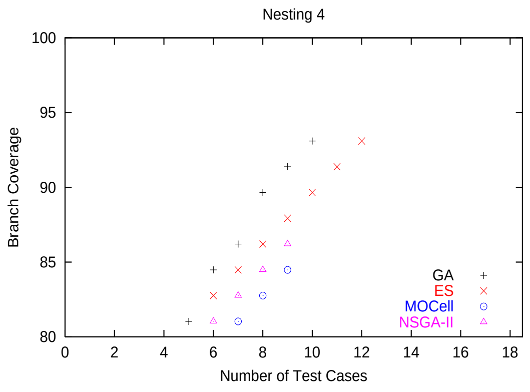
(a) Program with nesting degree 1



(b) Program with nesting degree 2



(c) Program with nesting degree 3



(d) Program with nesting degree 4

Figure 9. 50%-attainment surfaces: coverage against the number of test cases of all the algorithms.

Table XIV. Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript.

ND	GA	ES	RNDMono	MOCell	NSGA-II	SPEA2	PAES	RNDMulti
1	$99.19_{2.20}$	$98.70_{2.63}$	$85.33_{10.51}$	$98.10_{2.08}$	$97.90_{2.22}$	$97.53_{2.34}$	$93.08_{5.30}$	$81.36_{12.74}$
2	$98.85_{2.02}$	$97.87_{2.44}$	$79.20_{12.14}$	$94.77_{3.44}$	$94.42_{3.49}$	$93.56_{3.75}$	$87.59_{6.31}$	$75.04_{14.00}$
3	$98.52_{2.09}$	$95.66_{4.54}$	$71.94_{13.36}$	$90.66_{5.83}$	$90.41_{5.46}$	$89.29_{5.65}$	$81.55_{7.68}$	$69.77_{13.87}$
4	$96.89_{4.80}$	$93.19_{6.66}$	$66.42_{14.80}$	$85.50_{9.45}$	$85.77_{8.18}$	$84.61_{8.12}$	$75.87_{9.22}$	$63.87_{15.95}$
Total	$98.36_{3.13}$	$96.36_{4.90}$	$75.72_{14.65}$	$92.26_{7.54}$	$92.12_{6.99}$	$91.24_{7.24}$	$84.52_{9.72}$	$72.51_{15.57}$

In order to provide a high level of confidence to these results, we have performed statistical tests. The results are shown in Table XV. There are some differences among the best algorithms; we can take as a reference the column of the GA values. This column can be seen as a ranking of performance of all algorithms. The GA has the best results and outperforms the rest of the algorithms in average maximum coverage. ES is the second in average maximum coverage (with significant difference), next MOCell, then NSGA-II, and finally SPEA2. As we expected, the statistical test does not show significant differences among MOCell, NSGA-II and SPEA2, but if the number of independent runs were higher, the significant differences would appear. We should highlight that PAES is not much better than the random algorithms. The differences in average maximum coverage shown in Table XIV have been confirmed by the statistical tests: using a GA is the best way to obtain high branch coverage.

Finally, we have considered in this experimental study the obtained HV, the significant differences, the attainment surfaces and the average maximum coverage achieved with all the

Table XV. Number of programs where a significant difference exists among the coverage obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell	RNDMono	GA	ES
MOCell	800▲	797▲	0	0	–	800▲	350▽	30▽
NSGA-II	800▲	786▲	0	–	0	800▲	438▽	87▽
SPEA2	800▲	691▲	–	0	0	800▲	613▽	322▽
PAES	503▲	–	691▽	786▽	797▽	85▲	800▽	800▽
RNDMulti	–	503▽	800▽	800▽	800▽	7▽	800▽	800▽
ES	800▲	800▲	322▲	87▲	30▲	800▲	1▽	–
GA	800▲	800▲	613▲	438▲	350▲	800▲	–	1▲
RNDMono	7▲	85▽	800▽	800▽	800▽	–	800▽	800▽

algorithms and the benchmark of 800 programs. After analyzing the experimental results we can state that the GA is the best mono-objective algorithm and MOCell is the best multi-objective algorithm. We expected that an algorithm like MOCell would be clearly superior to all the mono-objective ones in the MOTDGP, but in fact this is not true. In addition, the GA is clearly superior in HV and average maximum coverage when we are testing programs with high nesting degree. This fact is due to a better exploration of the search space because this algorithm is able to find solutions for the most complex branches that appear in the code. It uses most of its evaluations in most complex branches, in order to achieve a high coverage. However, the multi-objective algorithms deal with all the branches at the same time, for this reason they do not use most of its evaluations trying to cover a concrete complex branch. This fact suggests that if there exist hard requirements of coverage and the program has high nesting degree, we should use the GA as search engine of an automatic test data generator. Nevertheless, a second phase of multi-objective test case selection must be performed in order to minimize the oracle cost. On the other hand, if there are cost requirements, we highly recommend the use of MOCell algorithm.

7.4. *Real Programs*

In this section we analyze the two proposed approaches using some real programs. We study 13 real programs extracted from the literature and with characteristics similar to the artificial programs used in the previous sections. The reader must take into account that the number of programs used in the previous sections gives us the chance to average among 800 programs and extract statistically more reliable results. Despite the fact that in this section we only analyze the performance of the proposed approaches and algorithms over 13 programs, most of the conclusions are similar to the ones we have been obtained with the synthetic programs.

Once again we start the analysis with the HV indicator. In Table XVI we summarize the number of programs where the HV value of an algorithm is better than the others. There are six programs where an algorithm is the best. The GA outperforms the other algorithms in four programs, then the ES in two programs, and the MOCell in only one program. In the previous results these three algorithms also obtain the best results.

Table XVI. Real programs in which the median Hypervolume of one algorithm is better than the others and average maximum coverage of all the real programs.

-	MOCell	NSGA-II	SPEA2	PAES	RNDMulti	GA	ES	RNDMono
HV Better	1	0	0	0	0	4	2	0
Avg. Max. Cov.	87.26	91.35	89.31	72.43	76.84	94.14	92.27	80.09

In order to validate these previous results we compared the HV values of all the real programs using the multiple comparison statistical test. In Table XVII we show the existing differences among the HV value of all the algorithms. We can observe that the GA outperforms the other algorithms in at least one program. Then, there is a group of algorithms composed by ES, NSGA-II, SPEA2, MOCell and RNDMono that are better than PAES and RNDMulti, but the statistical test does not show significant differences among them. In addition, we can analyze the PAES column in order to obtain an informal ranking of algorithms according to the HV indicator.

Table XVII. Real programs where a significant difference exists among the HV obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell	RNDMono	GA	ES
MOCell	1▲	1▲	0	0	–	0	2▽	0
NSGA-II	3▲	3▲	0	–	0	0	1▽	0
SPEA2	3▲	2▲	–	0	0	0	1▽	0
PAES	0	–	2▽	3▽	1▽	1▽	7▽	6▽
RNDMulti	–	0	3▽	3▽	1▽	1▽	7▽	4▽
ES	4▲	6▲	0	0	0	0	1▽	–
GA	7▲	7▲	1▲	1▲	2▲	3▲	–	1▲
RNDMono	1▲	1▲	0	0	0	–	3▽	0

Let us analyze the average maximum coverage obtained by the algorithms when are applied to real programs (Table XVI). We must highlight that the GA and the ES are the best algorithms in coverage for the real programs. In contrast, the PAES has obtained the worst results, even worse than the random algorithms. In Table XVIII, we show the results of a statistical test to compare the maximum coverage. Once again the GA is the best algorithm: it obtains significant differences in 40 comparisons. NSGA-II obtains significant differences in 30 comparisons. Next, SPEA2 and ES are better than the others in 18 comparisons. Most of these significant differences are obtained in comparison with PAES or the random algorithms. Only a few differences exist between the best algorithms. Nevertheless, the performance of GA seems to be better than the other algorithms. On the other hand, the PAES has obtained the worst results. We think that these results are due to the absence of crossover operator and the nature of the selected programs (i.e., number of equalities in the code).

Table XVIII. Number of real programs where a significant difference exists among the coverage obtained.

	RNDMulti	PAES	SPEA2	NSGA-II	MOCell	RNDMono	GA	ES
MOCell	6▲	12▲	0	0	–	0	2▽	0
NSGA-II	12▲	13▲	1▲	–	0	4▲	0	0
SPEA2	7▲	13▲	–	0	1▲	0	3▽	0
PAES	0	–	13▽	13▽	12▽	4▲	13▽	12▽
RNDMulti	–	0	7▽	12▽	6▽	3▽	12▽	9▽
ES	9▲	12▲	0	0	0	0	3▽	–
GA	12▲	13▲	3▲	0	2▲	7▲	–	3▲
RNDMono	3▲	4▽	0	4▽	0	–	7▽	0

Finally, with the aim of showing an example of the computed fronts for the instances, we selected the *line* program. This program can represent the typical behaviour of the different algorithms in this kind of instance. In Figure 10 the 50%-attainment surfaces of the best algorithms are depicted. In this instance, GA dominates the others and has a good performance because it always reaches the best coverage with the same test data. MOCell is the only algorithm able to obtain all the points of the front. This is a desirable property for a solution of a multi-objective problem.

8. CONCLUSIONS

In this paper we have studied the Multi-Objective Test Data Generation Problem with the aim of analyzing the performance of a direct multi-objective approach (MM) versus the application of mono-objective algorithms followed by a test case selection (mM). Previous results in the literature have only focused on the coverage of a program while the oracle cost is a significant cost that has

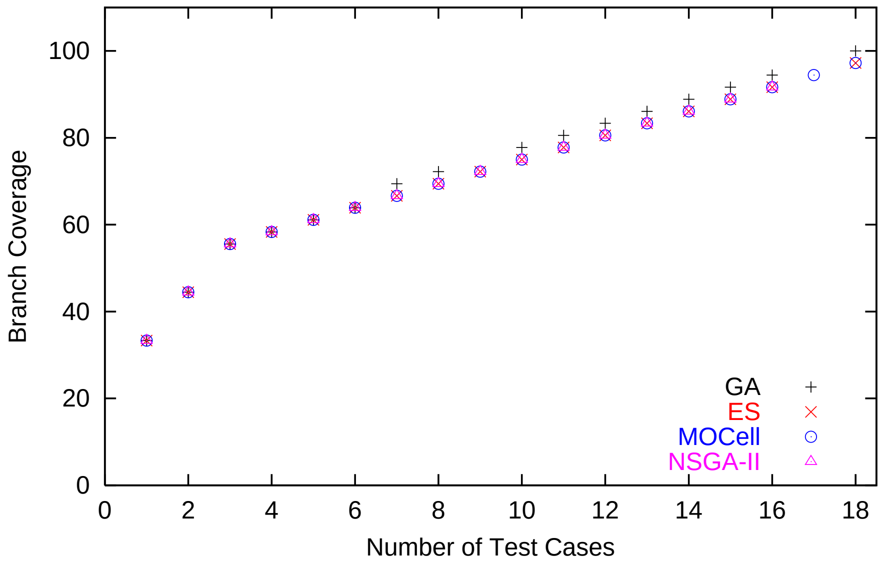


Figure 10. 50%-attainment surfaces: coverage against the number of test cases for the program *line*.

been ignored in most of the previous studies. For this reason, in this work we have dealt with the coverage and the oracle cost as equally important targets.

Our study has been performed on 800 synthetic programs. We designed a program generator able to produce programs ensuring a 100% of branch coverage. This kind of programs is very useful because all the branches are reachable and we can compare the algorithms in a fair way using coverage. In addition, we have also analyzed the two proposed approaches with a benchmark of 13 real and popular programs in the literature. We have evaluated four state-of-the-art multi-objective optimization algorithms: MOCell, NSGA-II, SPEA2, and PAES, two mono-objective algorithms GA, ES, and two random algorithms as merely a ‘sanity check’. This comparison has been done on the basis of three quality indicators: the hypervolume, the 50%-empirical attainment surface, and the average maximum coverage obtained by those algorithms. We can see a final ranking of algorithms in Table XIX.

In terms of convergence towards the optimal Pareto front, GA and MOCell have been the best solvers in our comparison. On the one hand, MOCell has obtained the best fronts in programs with nesting degree 1 and 2, values commonly found in practice. On the other hand, the GA is the best algorithm for facing programs with high nesting degree and it is the algorithm which is significantly better in most of the programs, attending to the HV indicator and to the average maximum coverage. This fact indicates that GA is the best alternative if the tested program has a high nesting degree or we need a high coverage. But, if we have time restrictions, we highly recommend the use of MOCell as a search engine for an automatic test data generator. Although the multi-objective approach is working very well in most of the programs, we realized that dealing with only one branch at the same time (mono-objective approach) can be more effective when the program under test has high nesting degree. In addition, we must highlight that both approaches (MM and mM) are quite good at reducing the number of test cases needed to obtain a given coverage. The oracle cost can be greatly reduced because the mM approach only needs 19.32% of the upper bound of test cases needed for obtaining the maximum value of coverage, and the MM approach is even better, only needing a 15.12% of the test cases. This improvement justifies the use of our approaches to deal with the MOTDGP.

Future work will verify these findings with still larger real-world software. Also, we should find a set of representative software programs because research community has not established a standard benchmark of well-known programs. We also want to advance in designing better evolutionary

Table XIX. Ranking of algorithms according to maximum coverage and hypervolume grouped by nesting degree.

Coverage					
Rank	ND 1	ND 2	ND 3	ND 4	All
1	GA	GA	GA	GA	GA
2	ES	ES	ES	ES	ES
3	MOCell	MOCell	MOCell	NSGA-II	MOCell
4	NSGA-II	NSGA-II	NSGA-II	MOCell	NSGA-II
5	SPEA2	SPEA2	SPEA2	SPEA2	SPEA2
6	PAES	PAES	PAES	PAES	PAES
7	RNDMono	RNDMono	RNDMono	RNDMono	RNDMono
8	RNDMulti	RNDMulti	RNDMulti	RNDMulti	RNDMulti
Hypervolume					
Rank	ND 1	ND 2	ND 3	ND 4	All
1	MOCell	MOCell	GA	GA	GA
2	NSGA-II	GA	MOCell	MOCell	MOCell
3	ES	NSGA-II	NSGA-II	ES	ES
4	GA	ES	ES	NSGA-II	NSGA-II
5	PAES	SPEA2	SPEA2	SPEA2	SPEA2
6	SPEA2	PAES	PAES	PAES	PAES
7	RNDMono	RNDMono	RNDMono	RNDMono	RNDMono
8	RNDMulti	RNDMulti	RNDMulti	RNDMulti	RNDMulti

operators in order to deal with programs with high nesting degree that could pose a challenge for our algorithms, especially for the multi-objective ones.

9. ACKNOWLEDGEMENTS

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2008-06491-C04-01 (the M* project). It has been also partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

REFERENCES

1. Xanthakis S, Ellis C, Skourlas C, Gall AL, Katsikas S, Karapoulios K. Application of genetic algorithms to software testing. *Proceedings of the 5th International Conference on Software Engineering and Applications*, Toulouse, France, 1992; 625–636.
2. Harman M, Jones BF. Search-based software engineering. *Information & Software Technology* December 2001; **43**(14):833–839.
3. McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* June 2004; **14**(2):105–156.
4. Harman M. The current state and future of search based software engineering. *Proceedings of International Conference on Software Engineering / Future of Software Engineering 2007 (ICSE/FOSE '07)*, IEEE Computer Society: Minneapolis, Minnesota, USA, 2007; 342–357.
5. Miller W, Spooner DL. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.* 1976; **2**(3):223–226.
6. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering* August 1990; **16**(8):870–879.
7. Beizer B. *Software testing techniques*. 2nd edn., Van Nostrand Reinhold Co.: New York, NY, USA, 1990.
8. Baresel A, Binkley DW, Harman M, Korel B. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. *International Symposium on Software Testing and Analysis (ISSTA 2004)*, 2004; 108–118.
9. Díaz E, Blanco R, Tuya J. Tabu search for automated loop coverage in software testing. *Proceedings of the International Conference on Knowledge Engineering and Decision Support (ICKEDS)*, Porto, 2006; 229–234.
10. Zhan Y, Clark JA. The state problem for test generation in simulink. *GECCO'06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, ACM Press, 2006; 1941–1948.
11. Tracey N, Clark J, Mander K, McDermid J. Automated test-data generation for exception conditions. *Software Practice and Experience* 2000; **30**(1):61–79.

12. Blanco R, Tuya J, Adenso-Díaz B. Automated test data generation using a scatter search approach. *Inf. Softw. Technol.* 2009; **51**(4):708–720.
13. Ahmed MA, Hermadi I. GA-based multiple paths test data generator. *Computers & Operations Research* 2008; **35**(10):3107–3124.
14. Alshraideh M, Bottaci L. Search-based software test data generation for string data using program-specific search operators. *Softw. Test. Verif. Reliab.* 2006; **16**(3):175–203.
15. Xiao M, El-Attar M, Reformat M, Miller J. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering* 2007; **12**(2):183–239.
16. Díaz E, Tuya J, Blanco R, Dolado JJ. A tabu search algorithm for structural software testing. *Computers & Operations Research* 2008; **35**(10):3052 – 3072.
17. Harman M, Kim SG, Lakhotia K, McMinn P, Yoo S. Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem. *Proceedings of the 3rd International Workshop on Search-Based Software Testing (SBST) in conjunction with ICST 2010*, IEEE: Paris, France, 2010; 182–191.
18. Lakhotia K, Harman M, McMinn P. A multi-objective approach to search-based test data generation. *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, ACM: New York, NY, USA, 2007; 1098–1105.
19. Deb KD, Pratap A, Agarwal S, Meyarivan T. A fast and elitist multiobjective genetic algorithm : NSGA-II. *IEEE Transactions on Evolutionary Computation,* August 2002; **6**(2):182–197.
20. Nebro AJ, Durillo JJ, Luna F, Dorronsoro B, Alba E. MOCell: A cellular genetic algorithm for multiobjective optimization. *Int. J. Intell. Syst.* 2009; **24**(7):726–746.
21. Nebro AJ, Durillo JJ, Luna F, Dorronsoro B, Alba E. Design issues in a multiobjective cellular genetic algorithm. *Evolutionary Multi-Criterion Optimization. 4th International Conference, EMO 2007, Lecture Notes in Computer Science,* vol. 4403, Obayashi S, Deb K, Poloni C, Hiroyasu T, Murata T (eds.), Springer, 2007; 126–140.
22. Zitzler E, Laumanns M, Thiele L. SPEA2: Improving the strength pareto evolutionary algorithm. *Technical Report 103*, Gloriastrasse 35, CH-8092 Zurich, Switzerland 2001.
23. Angeline PJ, Michalewicz Z, Schoenauer M, Yao X, Zalzala A (eds.). *The Pareto Archived Evolution Strategy: A New Baseline Algorithm for Pareto Multiobjective Optimisation*, vol. 1, IEEE Press: Mayflower Hotel, Washington D.C., USA, 1999.
24. Yoo S, Harman M. Pareto efficient multi-objective test case selection 9-12 July 2007; :140–150.
25. Michael CC, McGraw G, Schatz MA. Generating software test data by evolution. *IEEE Transactions on Software Engineering* December 2001; **27**(12):1085–1110.
26. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* December 2001; **43**(14):841–854.
27. Arcuri A. It really does matter how you normalize the branch distance in search-based software testing. *Software Testing, Verification and Reliability* 2011: (doi 10.1002/stvr.457)
28. McMinn P, Binkley D, Harman M. Empirical evaluation of a nesting testability transformation for evolutionary testing. *ACM Trans. Softw. Eng. Methodol.* June 2009; **18**:11:1–11:27.
29. Rechenberg I. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag: Stuttgart, 1973.
30. Rudolph G. *Evolutionary Computation 1. Basic Algorithms and Operators,* vol. 1, chap. 9, Evolution Strategies. IOP Publishing Lt, 2000; 81–88.
31. Knowles J, Thiele L, Zitzler E. A Tutorial on the Performance Assessment of Stochastic Multiobjective Optimizers. *TIK Report 214,* Computer Engineering and Networks Laboratory (TIK), ETH Zurich February 2006.
32. Zitzler E, Thiele L. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE Trans. Evolutionary Computation* 1999; **3**(4):257–271.
33. Knowles J. A summary-attainment-surface plotting method for visualizing the performance of stochastic multiobjective optimizers. *International Conference on Intelligent Systems Deisgn and Applications*, 2005; 552–557.
34. Alba E, Chicano F. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research* 2008; **35**(10):3161–3183.
35. King JC. A new approach to program testing. *SIGPLAN Not.* April 1975; **10**:228–233.
36. King JC. Symbolic execution and program testing. *Commun. ACM* July 1976; **19**:385–394.
37. Dijkstra EW. *A Discipline of Programming*. Prentice Hall, 1976.
38. Gris D. *The science of programming*. Springer-Verlag, 1981.
39. Kaldewaij A. *Programming: The derivation of algorithms*. Prentice-Hall, 1990.
40. May PS. Test Data Generation: Two Evolutionary Approaches to Mutation Testing. PhD Thesis, Computing Laboratory 2007.
41. Arcuri A. Evolutionary repair of faulty software. *Applied Soft Computing* 2011; **11**:3494–3514.
42. Jones BF, Sthamer HH, Eyres DE. Automatic structural testing using genetic algorithms, 1996.
43. Durillo JJ, Nebro AJ, Luna F, Dorronsoro B, Alba E. jMetal: A Java Framework for Developing Multi-Objective Optimization Metaheuristics. *Technical Report ITI-2006-10,* Departamento de Lenguajes y Ciencias de la Computación, University of Málaga, E.T.S.I. Informática, Campus de Teatinos December 2006.
44. Alba E, Dorronsoro B. *Cellular Genetic Algorithms, Operations Research/Computer Science Interfaces*, vol. 42. Springer-Verlag Heidelberg, 2008.