A Multi-Objective Genetic Algorithm to Test Data Generation

Gustavo H.L. Pinto, Silvia R. Vergilio
Federal University of Paraná
Computer Science Department
CP: 19081, Centro Politecnico CEP: 81531-970
Jardim das Américas - Curitiba, Brazil
gustavo@inf.ufpr.br, silvia@inf.ufpr.br.

Abstract

Evolutionary testing has successfully applied search based optimization algorithms to the test data generation problem. The existing works use different techniques and fitness functions. However, the used functions consider only one objective, which is, in general, related to the coverage of a testing criterion. But, in practice, there are many factors that can influence the generation of test data, such as memory consumption, execution time, revealed faults, and etc. Considering this fact, this work explores a multi-objective optimization approach for test data generation. A framework that implements a multi-objective genetic algorithm is described. Two different representations for the population are used, which allows the test of procedural and object-oriented code. Combinations of three objectives are experimentally evaluated: coverage of structural test criteria, ability to reveal faults, and execution time.

1 Introduction

The test data generation to satisfy a test criterion is associated to several limitations, and it can not be completely automated. Because of this, this task usually consumes a lot of effort. Given a criterion C, there is no algorithm to determine whether a test set T is C-adequate, that is, whether T satisfies C, covering all the elements required by C. There is no algorithm even to determine whether such set exists [11].

Due to this fact, the task of test data generation is an open research question. Approaches based on search based algorithms, such as Genetic Algorithms, present promising results, and were grouped in a area, called Evolutionary Testing [32]. The works on this subject differ on the search based technique and on the fitness function used. Most of them are coverage oriented, addressing structural and fault based criteria applied to procedural code [10, 16, 21, 32]. The context of object-oriented programs is a more recent research field with a reduced number of works [25, 26, 27, 28, 31].

A limitation of these existing works is that they formulate the test generation problem as a single objective search problem, which is, in general, the coverage of a path (or required element). However, there are diverse characteristics desired for the test data that should be considered to guide the generation task, such as: ability to reveal certain kind of faults, reduced execution time and memory consumption, and other factors associated to the development environment that can be related to the type of software being developed.

In this way, we notice that the test data generation problem is in fact multi-objective, since it can depend on multiple variables (objectives). For such problems there is no a single solution that satisfies all the objectives. This is because the objective can be in conflict, for example, coverage and execution time.

Considering this fact, this work explores a new multi-objective approach for test data generation. A multi-objective search based framework is described that uses two representations for the test data to allow the test of procedural and object-oriented code, and the integration with the testing tools: Poketool [19], and JaBUTi [30]. The framework implements the NSGA-II (Non-dominated Sorting Genetic Algorithm) algorithm [7], a multi-objective genetic algorithm that evaluates the solutions according to Pareto dominance concepts [23] considering different objectives: coverage of a chosen structural criterion, execution time, and ability to reveal faults. Some experimental results of three case studies show that the implemented algorithm produces a set of solutions that represent a good trade off between the objectives to be satisfied by the test data generated.

The rest of the paper is organized as follows. Section 2 introduces the framework and implemented algorithm. Section 3 presents experimental evaluation results. Section 4

describes related work. Finally, in Section 5 we conclude the paper and present the main contributions of the work.

2 Test Data Generation by a Multi-objective Approach

This section introduces an approach that formulates the test data generation as a multi-objective problem. In fact, diverse factors should be considered to select a test data. For example: to maximize the coverage of a test criterion and the ability to reveal faults; to minimize the execution time and memory consumption; to minimize the size either of the test data or of the test set.

To consider this multi-influence in the test data generation, it was developed a framework that implements a multi-objective genetic algorithm, NSGA-II, a modification of the original NSGA proposed by Deb [7]. The framework is integrated to the testing tools Poketool [19] and JaBUTi [30]. It generates test data to satisfy the structural criteria based on control and data flow, and supported by the tools, respectively for the test of C and Java programs. The fitness function can be based in a combination of the three following objectives: coverage of a given criterion, execution time and ability to reveal faults. The solutions are evaluated according to the Pareto dominance concepts.

2.1 The Framework

The framework has three modules: i) brainModule; ii) testingModule, that controls the integrated testing tools; and iii) configModule, that receives the configuration of parameters from the users. The communication among the modules is illustrated in Figure 1. The parameters provided by the tester are received (‘a’), and formatted by the configModule (‘b’). The brainModule implements the NSGA-II algorithm and is responsible by the evolution process. The test data sets are passed to the testingModule (‘c’), which is responsible for the transformation of the data according to the format used by the integrated tools. After the execution and evaluation of the test data by the tools, the testingModule returns information about the coverage (‘d’) to allow the evolution of the population.

2.2 ConfigModule

The configModule needs the following parameters: a) information related to the evolution process: maximum number of generations or maximum execution time; crossover, mutation and relation rates (related to the application of the genetic operators); number of individuals in the population; b) information to generate the population: number of test data in each individual, type of the input variables or number maximum of methods calls; c) objectives to be used in the fitness evaluation; d) information about the test tools: language and chosen test criterion.

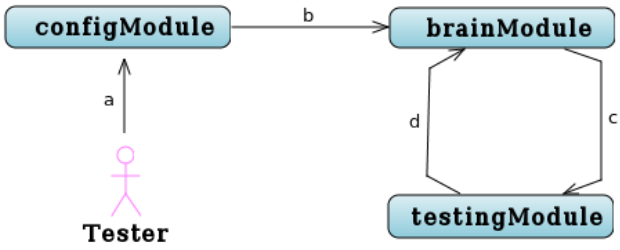

Figure 1. Modules of the framework

2.3 BrainModule

The brainModule is responsible for the: initial generation of the individuals that are test data sets, evolution of the population by the application of the genetic algorithms and the fitness evaluation. It implements the multi-objective algorithm. Important implementation aspects of any metaheuristic algorithm are: the representation of the population and the choice of the fitness function, which are presented next.

2.3.1 Representation of the population

Works on search based test, in general, represents the input data as an individual of the population. However, in this work the individual represents a set of test data to be evolved. The maximum size of such set is determined by the tester. This structure is presented in Figure 2, in which the individual is composed by n test data. With this representation different solutions can be explored in the search space, and in the future, the size of the set can be an objective to be minimized.

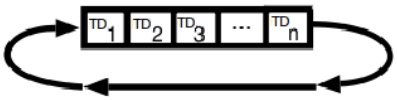

Figure 2. An individual in the population

Two possible representations for the test data in each individual can be used, depending on the chosen language and tool, representing test data for either procedural or object-oriented programs. In the procedural context, the input consists in a sequence of values to execute the program. The type and number of the values are fixed and provided by test and vary according to the program being tested. Because of this, the codification of the input is simple.

In the context of object-oriented code, however, the test data is not a simple sequence of values. It contains se-

quences of invocations to constructors and methods, including the correspondent parameters. To represent each test data, the framework used the following grammar, introduced by Tonella [28]:

```
<chromosome> ::= <actions> @ <values>
<actions>    ::= <action> {: <actions>}?
<action>     ::= $id = constructor ( {<parameters>}? )
              |  $id = class # null
      |  $id . method ( {<parameters>? )
<parameters> ::= <parameter> {, <parameters>}?
<parameter>  ::= builtin-type {<generator>}?
              |  $id
<generator>  ::= [ low ; up ]
      |  [ genClass ]
<values>     ::= <value> {, <values>}?
<value>      ::=  integer
      |   real
      |   boolean
      |   string
```

The test data is composed by two parts, separated by the character '@'. The first part, the non-terminal <actions> contains a sequence of constructors and methods invocations, separated by ':'.

Each <action> can either construct a new object, assigned to a variable $id, or result in a method invocation. A special case involves the value null, that corresponds to a reference for a non existing object (if no constructor for this object is invoked). The value null is preceded by the class of $id (separated by the character '#') and can be used as a parameter only if its type matches to that of the formal parameter. Parameters of the methods and constructors are primitive types of the language such as int, real, boolean or a variable $id. The second part of the chromosome contains input values (parameters) used in each invocation, separated by the character ','.

Examples of test data for the program TriTyp.java are presented below. This program receives three integer values and checks if the values correspond to a triangle. In the affirmative case the programs returns the type of triangle formed. The program has six public methods: setI(), setJ(), setK(), type(), equals() e toString(). In the examples, the methods invocations are separated by ':'. The first one is the constructor. After '@' we find the parameters to each method.

2.3.2 Genetic operators

The genetic operators transform the population through successive generations and help to maintain diversity and adaptation characteristics obtained by the solutions of previous generations. In the context of procedural code, the operators are applied by considering the type of each provided input. Some examples are presented in Table 1. If the values are integer they can be summed or subtracted to compose another test data.

The mutation occurs according to a probability. In such case, a new value is generated considered the type of the input value. Duplicate test data in the individual are eliminated.

Table 1. Illustrating crossover in C programs

parent 1	parent 2	descendent 1	descendent 2
W Ol8AS+Jj	UvHW]BYYMO	W Ol8B | YYMO	UvHW] | AS+Jj
K0?y-n UnZ	XJc_Is.OO)	K0?yI | s.OO)	XJc_I | OOUnZ
-7093	-4950	-12043	-2143
+4899	-7968	-3069	+12867

In the context of object-oriented programs, the operators are applied as follows. The mutation operator changes, inserts or removes inputs, constructors and methods invocations. For example, the test data:

```
$t= TriTyp():$t.setI(int):$t.setJ(int):$t.type() @ 9, 3
```

can be changed by:

```
$t=TriTyp():$t.setI(int):$t.setJ(int):$t.type() @ 6, 3
$t=TriTyp():$t.setJ(int):$t.type() @ 3
```

The crossover operator selects points that can be in the sequence of methods invocations or in the part that contains the parameters. For example:

Parents:

```
$t=TriTyp():$t.setI(int):$t.setJ(int):$t.setK(int) @ 9, 3, 8
$t=TriTyp():$t.setI(int):$t.setJ(int):$t.setK(int) @ 4, 1, 4
```

Descendents:

```
$t=TriTyp():$t.setI(int):$t.setJ(int):$t.setK(int) @ 1, 3, 8
$t=TriTyp():$t.setI(int):$t.setJ(int):$t.setK(int) @ 4, 9, 4
```

The operators of relation change test data between individuals to ensure diversity. They apply: half relation and random relation. Consider the initial sets A = {a, b, c, d, e, f} and B = {g, h, i, j, k, l}. The operator of *half relation* derives two new sets. The first one containing the first half of A and the second half of B. The second one containing the remaining parts of A and B. The following sets are generated: HR_i = {a, b, c, j, k, l} and HR_{ii} = {g, h, i, d, e, f}.

The operator *Random relation* randomly selects index for the elements that will be changed. For example, for the random values 3, 4 e 6, corresponding to the indexes of the elements in A and B, the following sets are obtained: RR_i = {a, b, i, j, e, l} and RR_{ii} = {g, h, c, d, k, f}.

2.3.3 Fitness Evaluation

Each individual can be evaluated according two or more objectives chosen by the tester. In this version of the framework, three functions are available.

- Coverage of a structural criterion: the tester can choose a criterion implemented by the integrated tools. The first one, Poketool, supports the test of C programs with the control-flow based criteria all-nodes and all-edges, and with the Potential Uses criteria family, which are data-flow based criteria [19]. JaBUTi supports all-nodes, all-edges, all-uses and all-potential uses, for the test of Java programs [30] and derives the required elements directly from the Java bytecode. The coverage of each set x (individual) to be maximized is given by:

$$Fitness_x = \frac{\text{nr. elements covered}_x \text{ * 100}}{\text{total number of required elements}}$$

- Execution time: the execution time of the test data set x is given by the sum of the execution time of each test data on x.

- Ability to reveal faults: to evaluate this ability for a test data set x it was used mutation test and the essential operators [2] implemented by the tool Proteum [8]. If the test data is capable to kill the mutants generated by a mutation operator, we consider that this data test is capable to reveal the fault described by the corresponding operator. The ability is given by the mutation score of x provided by Proteum.

2.4 Evolution Process

The evolution process follows the steps, according to Algorithm 1.

In the initial step, the test data are generated by using the chosen representation, according to the language. The individuals in the population are test data sets. The initial population is randomly generated (Line 1.2). A loop starts the evolution process (1.3). The fitness of each individual is given by the tools, by evaluating each test data in the set (1.8,1.10) After this, the dominated and non-dominated solutions are identified (1.12) according to the Pareto dominance criterion. Then, the criterion given by the multiple objectives is applied (1.13) and the selected individuals are used in other evolution, the evolution of the test data (1.14-1.25).

In such process, the selection based on a roulette (1.16) is used to select the test data in the individual that will suffer the action of the genetic operators (mutation (1.19) and crossover (1.20)). This means that the test data are also evolved, and it is supposed that the individuals, the test data sets will save the best test data trough the iterations. At the end of an iteration, the test data are changed among individuals by the relation operators (1.21), to maintain diversity. This is related to the evolution of the test data sets.

Finally, the set of the best solutions are added to the new population (1.27), and are returned as output by the algorithm (1.28). The evolution continues until the stop criterion is reached, that is, a number maximum of generations (or the execution time determined by the tester) is reached. After the loop, the individuals are analysed and the non-dominated solutions are added to the set of non-dominated solutions found during the role process.

```
    input  : code under test (cut, configuration file
    output: a set of non-dominated solutions
1.1 nondominated[] ← ∅;
1.2 pop[] ← generateRandomPop();
1.3 while not numGenerations or maxExecutionTime
    do
1.4     for i ← 0 to sizeof(pop[]), do
1.5         ind ← pop[i];
1.6         for j ← 0 to ind.c(), do
1.7             testData ← ind.getTestData(j);
1.8             evaluate(testData);
1.9         end
1.10        evaluate(ind);
1.11    end
1.12    nonDominated[] ←
        selectNonDominated(pop);
1.13    indSel[] ← multi-objectiveSelecion(pop);
1.14    for i ← 0 to indSel[], do
1.15        ind ← indSel[i];
1.16        selData[] ← rouletteSelecion(ind);
1.17        for j ← 0 to sizeof(selData[]), do
1.18            testData ← selData[j];
1.19            mutation(testData);
1.20            crossover(testData, selData[j + 1]);
1.21            selData[j] ← testData;
1.22        end
1.23        ind ← selData[];
1.24        indSel[i] ← ind;
1.25    end
1.26    relate(indSel);
1.27    pop[] ← indSel[];
1.28    foreach solution in nonDominated[] do
        print(solution);
1.29 end
```

Algorithm 1: Pseudocode of the Implemented Algorithm

3 Evaluation Studies

The implemented approach was evaluated in three case studies. They were chosen to make possible an evaluation in both test contexts, including procedural and object-oriented

programs. Other points to be explored by the case studies were: the use of different test criteria (based on control and data flow) and the combination of the three objectives implemented by the framework to guide the test data generation.

The configuration was empirically adjusted. In each case study, the algorithm was executed with variations in the number of generations, number of individuals in the population, number of test data, and etc. The rates related to the genetic operators were fixed following suggestions found in the literature [9], and after this, adjusted. They are: 0.8 for crossover rate; 0.2 for mutation; and 0.7 for the relation operator. Chosen the configuration, the results obtained by NSGA-II were compared to a random strategy (RS) by configuring the number of generations equal to 1.

In both strategies, the algorithms were executed 10 times. After all the executions, to allow a visual comparison, a front was obtained composed only by the non-dominated solutions, considering the solutions of all executions. In many cases, mainly considering RS, after this step only one solution was obtained. This means that this solution dominates the solutions found in all executions. Next, the solutions found and the configuration used for each case study are presented.

3.1 Case Study 1

This study used two object-oriented programs and two objectives: coverage of a structural criterion and execution time. The programs are: Find.java and TriTyp.java[1]. As mentioned in Section 3 the program TriTyp.Java determines if three integer numbers form a triangle and in the affirmative case returns the type of formed triangle. The other program Find.Java searches for an entry in a vector. The following criteria implemented by JaBUTi were used: all-edges$_{ei}$ (AEi) and all-uses$_{ei}$(USi). The configuration used is in Table 2. The solutions found for each program and criterion are in Table 3.

Table 2. Configurations of Case Study 1

Program	Criterion	numGenerations	numIndividuals	numTestData
TriTyp	AEi	100	100	50
TriTyp	USi	100	100	50
Find	AEi	50	50	50
Find	USi	50	100	50

We can observe, as it was expected, that it is more difficult to satisfy the data-flow based criterion. For program TriTyp and the criterion all-edges, NSGA-II presented 5 non-dominated solutions and RS one. The RS solution is dominated by the NSGA solutions. The NSGA solutions

[1]Used in diverse works reported in the literature [4, 24]. Available on http://www.infcr.uclm.es/www/mpolo/stvr/.

Table 3. Results of Case Study 1

Program	Criterion	NSGA-II		RS	
		Cov. (%)	Exec.Time (ms)	Cov. (%)	Exec.Time(ms) (ms)
TriTyp	AEi	81	319	65	284
		76	297		
		74	281		
		72	278		
		52	344		
	USi	69	315	42	349
				40	331
Find	AEi	95	297	88	327
		93	292		
	USi	88	303	86	247
		87	220	76	240
		77	200		

vary from 244 to 319ms and coverage of 52 to 81%. The algorithm could select the most relevant points in the search space. For the criterion US, NSGA-II presented only one solution that also dominates both solutions found by RS. Similar result was obtained to program Find. NSGA-II presented a greater number of solutions, which dominate the solutions found by RS.

3.2 Case Study 2

In this study, two objectives were evaluated: the coverage of the data-flow based criterion all-potential-uses (PU) (or the coverage of the criterion all-edges (AE)) implemented by Poketool, and the ability of reveal faults, described by the essential operators of Proteum. It was used the program compress.c[2]. The program replaces characters that appear more than three times in a string by a sequence that indicates the number of repetitions, and returns a string with a lower number of characters.

In the case, for both criteria all the parameters were set with 50, except the number of individuals in the population for criterion PU, which was set with 100. The obtained results are in Table 4.

Table 4. Results of Case Study 2

Program	Criterion	NSGA-II		RS	
		Cov. (%)	Exec.Time (ms)	Cov. (%)	Exec.Time (ms)
compress	AE	100	77	100	70
	PU	83.33	77	83.33	70

We observe that both strategies reach the same coverage for both structural criterion. The obtained value for the coverage is optimal (or near to the optimal, since the criterion PU requires around 10% of infeasible elements [29]). However, the test set generated by NSGA-II has greater ability to reveal faults.

[2]Available on the book of Kernighan [17] and also used by other authors in the literature [29, 33].

3.3 Case Study 3

Again, with the program compress.c, three objectives were investigated: the coverage of the criterion all-potential-uses (PU), the ability to reveal faults described by the essential operators, and execution time. The parameters used are: number of generations and individuals equal to 100, and number of test data 10. The obtained results are in Table 5.

Considering more than two objectives does not imply lower efficiency. NSGA-II obtained solutions with the same coverage reached with two objectives, and at the same time to minimize the third objective, execution time. The same does not happen with RS. The solutions obtained by NSGA dominate the solution found by RS.

Table 5. Results of Case Study 3

NSGA-II			RS		
Cov. (%)	Fault (%)	Exec.Time (ms)	Cov. (%)	Fault (%)	Exec.Time (ms)
83.33	77	80	78.89	70	87
83.33	76	77			
78.89	77	76			

3.4 Analysis

By analysing the results of all case studies we can observe that the solutions found by NSGA-II dominate all the solutions found by RS, independently of the test criterion and objective being considered. The implemented algorithm is capable to improve the quality of the data generated, and offer a set of good solutions to be used according to the test goals.

To compare the strategies, the indicator Hypervolume [36] was used. This indicator shows how the solutions found by the algorithms are spread in the search space. The values of Hypervolume were compared through the Mann-Withney U-test [6]. It is a non-parametric test, used to check the null hypothesis of two samples being similar. It was adopted a significance level (p-value) of 0.05. The lower the value the greater the difference between the algorithms. All the obtained p-values are always < 0.05, indicating the the NSGAII presents statistically better solutions.

With respect to the costs, given by the execution time. The studies were done in a computer Intel(R) Xeon(R)- 7 GB, operational system Debian 5.0. We observe that greater numbers of individuals and test data contribute to increase costs of both strategies. It seems that the test criterion used does not influence significantly on this cost. The mean time for all executions of the programs is presented in Table 6. We observe that the cost is almost three times greater. So we suggest that the strategies can be used in a complementary way. NSGA-II can be used to improve the random sets, mainly considering strength criteria, such as data flow based.

Other observed point, to be investigated in future experiments, is that the coverage of the criterion and ability to reveal faults seems to be dependent. Improving coverage implies to reveal more faults. Hence, a combination that makes the use of a multi-objective approach more suitable is that one that includes the execution time.

Table 6. Costs of the algorithms

Program	Criterion	Result
TriTyp	02:40:23	00:40:10
Find	01:22:38	00:19:23
compress	01:37:11	00:23:34
compress(3)	01:30:22	00:19:21

4 Related Work

The generation of test data sets using search based algorithms has been largely explored in the literature [20]. The existing works differ on the technique used, such as Genetic Algorithms, Genetic Programming, and etc. Other difference is on the fitness evaluation. Most works are oriented to the coverage of structural and fault-based criteria in the context of procedural programs [10, 16, 21, 32]. In the context of object-oriented programs, few works are found [25, 26, 27, 28, 31]. They also have the goal of satisfying structural criteria, mainly the all-edges criterion.

A problem is that most of these works are not integrated with a test tool, and most algorithms implemented need the code is available for the test. In addition to this, they formulate the test generation problem as related to a single objective, by using only a fitness function. As we mention before, there are several factors that can be considered. This task is in fact a multi-objective problem.

In the Software Engineering area, the use of multi-objective optimization is an emergent research area [15]. It has been explored to reduce project costs and to configure software project times [1]; to select requirements for software release [35], to software refactoring and design [3, 13].

In the test activity these algorithms were used for selection of test cases [34] and also to test data generation. Cao et al. [5] present a method based on similarity of executed paths to generate test data to cover a specific path. Ghiduk et al. [12] introduce an algorithm to ensure coverage of a data-flow based criterion and to reduce the number of generated test data. With the same goal, the work [22] implements multi-objective algorithms to maximize coverage of control and data flow based criteria, and to minimize the number of test data. A work that is very similar to ours is the work of Harmam et al. [14]. In such work, the authors also used a

multi-objective Genetic Algorithm to maximize the coverage of branches and to reduce consumption of memory.

Our work, differently of the abovementioned ones, is integrated with two test tools, which allows the use of a multi-objective genetic algorithm in two contexts, procedural and object-oriented programs. The criteria implemented in the context of object-oriented code derive their test requirements from the Java bytecode. The representation used for the individuals allows the test data generation even if the source code is not available. This is very common in the test of most components. In addition to this, the the multi-objective approach considers other objectives and factors that can influence the test data generation and are required for the test data: execution time and kind of faults to be revealed.

5 Conclusions

This work explored a multi-objective approach for test data generation and described a framework that implements the algorithm NSGA-II considering possible combinations of three objectives: coverage of structural criteria, execution time, and ability to reveal faults.

The framework implements two kind of representations for the population and the NSGA-II algorithm. The first representation allows the test of C programs and the integration with the tool Poketool that implements the criteria based on control and data flow. The second representation allows the test of object-oriented code and the integration with the tool JaBUTi, which implements control and data flow based criteria that derive their test requirements directly from the Java bytecode. The framework allows the test data generation even if the source code is not available.

The use of the implemented algorithm in both contexts was evaluated in three case studies and compared with a random strategy. The results show that the multi-objective algorithm get better solutions and improvements for all the objectives, and a large variety of solutions to be chosen according to the testing purposes. The solutions found by the random strategy are always dominated by the NSAGA-II. In spite of more expensive, the use of the framework contributes to reduce the tester's effort. It is justified if there are different characteristics desired for the test data to be generate, and that they are in conflict.

Other research studies include the implementation and evaluation of other objectives in the framework, as well as, other meta-heuristic algorithms. These improvements will allow the conduction of new evaluation experiments. A possible context to be explored is the integration of the framework with the JaBUTi/AJ [18], to permit the test data generation for aspect-oriented programs.

References

[1] E. Alba and F. Chicano. Software project management with GAs. *Information Sciences*, 177(11):2380–2401, June 2007.

[2] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Software Testing, Verification Reliability*, 11(2):113–136, 2001.

[3] M. Bowman, L. C.Briand, and Y. Labiche. Multi-objective genetic algorithm to support class responsibility assignment. In *IEEE International Conference on Software Maintenance*, pages 124–133, October 2007.

[4] L. C. Briand, Y. Labiche, and Z. Bawar. Using machine learning to refine black-box test specifications and test suites. In *The Eighth International Conference on Quality Software*, pages 135–144, 2008.

[5] Y. Cao, C. Hu, and L. Li. Search-based multi-paths test data generation for structure-oriented testing. In *ACM/SIGEVO Summit on Genetic and Evolutionary Computation*, pages 25–32, 2009.

[6] W. J. Conover. On methods of handling ties in the wilcoxon signed-rank test. *Journal of the American Statistical Association*, pages 985–988, December 1973.

[7] K. Deb and N. Srinivas. Multiobjective optimization using nondominated sorting in genetic algorithms. In *IEEE Transactions on Evolutionary Computation*, pages 221–248, 1994.

[8] M. E. Delamaro and J. C. Maldonado. Proteum - a tool for the assesment of test adequacy for C programs. *In Conference on Performability in Computing Systems (PCS 96)*, pages 79–95, July 1996.

[9] J. D. Farmer, N. Packard, and A. Perelson. *Introduction to genetic algorithms*. MIT press, 1 edition, 1997.

[10] L. P. Ferreira and S. R. Vergilio. TDSGen: An environment based on hybrid genetic algorithms for generation of test data. In *Software Engineering and Knowledge Engineering*, pages 312–317, 2005.

[11] P. G. Frankl and E. J. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions Software Engineering*, 14(10):1483–1498, 1988.

[12] A. S. Ghiduk, M. J. Harrold, and M. R. Girgis. Using genetic algorithms to aid test-data generation for data-flow coverage. In *Proceedings of the 14th Asia-Pacific Software Engineering Conference (APSEC '07)*, pages 41–48, December 2007.

[13] L. Grunske. Identifying "good" architectural design alternatives with multi-objective optimization strategies. In *Proceedings of the 28th International Conference on Software Engineering*, pages 849–852, May 2006.

[14] M. Harman, K. Lakhotia, and P. McMinn. A multi-objective approach to search-based test data generation. In *Genetic and Evolutionary Computation Conference*, 2007.

[15] M. Harman, S. A. Mansouri, and Y. Zhang. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. Technical Report TR-09-03, Department of Computer Science, King's College London, April 2009.

[16] B. F. Jones, H. H. Sthamer, and D. E. Eyres. Automatic structural testing using genetic algorithms. *Software Engineering Journal*, pages 299–306, 1996.

[17] B. W. Kernighan. *The C Programming Language*. Prentice-Hall, Englewood Cliffs New Jersey, 1978.

[18] O. Lemos, A. Vincenzi, J. Maldonado, and P. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of System and Software*, 80(6):862–882, 2007.

[19] J. C. Maldonado, M. L. Chaim, and M. Jino. Briding the gap in the presence of infeasible paths: Potential uses testing criteria. In *XII International Conference of the Chilean Science Computer Society*, pages 323–340, October 1992.

[20] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 2(14):105–156, 2004.

[21] C. C. Michael, G. McGraw, and M. A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.

[22] N. Oster and F. Saglietti. Automatic test data generation by multi-objective optimisation. In *SAFECOMP*, pages 426–438, 2006.

[23] V. Pareto, editor. *Manuel D"Economie Politique*. Ams Pr, 1927.

[24] M. Polo, M. Piattini, and I. García-Rodríguez. Decreasing the cost of mutation testing with second-order mutants. *Software Testing Verification Reliability*, 19(2):111–131, 2009.

[25] J. C. B. Ribeiro. Search-based test case generation for object-oriented Java software using strongly-typed genetic programming. In *Genetic and Evolutionary Computation Conference*, 2008.

[26] R. Sagarna, A. Arcuri, and X. Yao. Estimation of distribution algorithms for testing object oriented software. In *Proceedings of the IEEE Congress on Evolutionary Computation (CEC '07)*, pages 438–444, September 2007.

[27] A. Seesing and H.-G. Gross. A genetic programming approach to automated test generation for object-oriented software. *International Transactions on System Science and Applications*, 1(2):127–134, October 2006.

[28] P. Tonella. Evolutionary testing of classes. In *ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[29] S. R. Vergilio and M. J. J. C. Maldonado. Infeasible paths in the context of data flow based testing. *Journal of the Brazilian Computer Society*, 12(1), 2006.

[30] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong. Establishing structural testing criteria for Java bytecode. *Software: Practice and Experience*, 36(14):1513–1541, 2006.

[31] S. Wappler and F. Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, 2005.

[32] J. Wegener, A. Baresel, and H. Sthamer. Using evolutionary testing to improve efficiency and quality in software testing. research and technology. In *2nd Asia-Pacific Conference on Software Testing Analysis and Review (AsiaSTAR*, 2002.

[33] E. J. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, 1993.

[34] S. Yoo and M. Harman. Pareto efficient multi-objective test case selection. In *International Symposium on Software Testing and Analysis*, 2007.

[35] Y. Zhang, M. Harman, and S. A. Mansouri. The multi-objective next release problem. In *GECCO '07: Proceedings of the 9th annual conference on Genetic and evolutionary computation*, pages 1129–1137, 2007.

[36] E. Zitzler, D. Brockhoff, and L. Thiele. The hypervolume indicator revisited: On the design of pareto-compliant indicators via weighted integration. In *Conference on Evolutionary Multi-Criterion Optimization (EMO 2007)*, pages 862–876, 2006.