Benchmark Generator for Software Testers

Javier Ferrer, Francisco Chicano, and Enrique Alba

Departamento de Lenguajes y Ciencias de la Computación
University of Málaga, Spain
{ferrer, chicano, eat} @lcc.uma.es

Abstract. In the field of search based software engineering, evolutionary testing is a very popular domain in which test cases are automatically generated for a given piece of code using evolutionary algorithms. The techniques used in this domain usually are hard to compare since there is no standard testbed. In this paper we propose an automatic program generator to solve this situation. The program generator is able to create Java programs with the desired features. In addition, we can ensure that all the branches in the programs are reachable, i.e. a 100% branch coverage is always possible. Thanks to this feature the research community can test and enhance their algorithms until a total coverage is achieved. The potential of the program generator is illustrated with an experimental study on a benchmark of 800 generated programs. We highlight the correlations between some static measures computed on the program and the code coverage when an evolutionary test case generator is used. In particular, we compare three techniques as the search engine for the test case generator: an Evolutionary Strategy, a Genetic Algorithm and a Random Search.

Keywords: Software Testing, Evolutionary Algorithm, Search Based Software Engineering, Benchmarks

1 Introduction

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [5, 6]. From the first works [10] to nowadays many approaches have been proposed for solving the automatic test case generation problem. It is estimated that half the time spent on software project development, and more than half its cost, is devoted to testing the product [3]. This explains why the Software Industry and Academia are interested in automatic tools for testing.

Evolutionary algorithms (EAs) have been the most popular search algorithms for generating test cases [9]. 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. The objective of an automatic test case generator used for structural testing is to find a test case suite that is able to cover all the software elements. These elements can be statements, branches, atomic conditions, and so on. The performance of an automatic test case generator is usually measured as the percentage of elements that the generated test suite is able to cover in the

test program. This measure is called *coverage*. The coverage obtained depends not only on the test case generator, but also on the program being tested.

Once a test case generating tool is developed, we must face the problem of evaluating and enhancing the tool. Thus, a set of programs with a known maximum coverage would be valuable for comparison purposes. It is desirable to test the tool with programs for which a total coverage can be reached. This kind of benchmark programs could help the tool designer to identify the scenarios in which the tool works well or either has problems in the search. With this knowledge, the designer can improve her/his tool. The problem is that, as far as we know, there is no standard benchmark of programs to be used for comparing test case generating tools, and the software industry is usually reticent to share their source code. Furthermore, it is not usual to find programs in which total coverage is possible. Researches working on automatic test case generators usually provide their own benchmark composed of programs they programmed or belonging to a company with which they are working. This makes hard the comparison between different tools.

In order to alleviate the previous situation we propose, design, and develop a program generator that is able to create programs with certain features defined by the user. The program generator allows us to create a great amount of programs with different features, with the aim of analyzing the behavior of our test case generator in several scenarios. In this way, researches can face their tool with programs containing, for example, a high nesting degree or a high density of conditions. In addition, the program generator is able to write programs in which 100% branch coverage is possible. This way, the branch coverage obtained by different test case generators can be used as a measure of performance of the test case generator on a given program. This automatic program generator is the main contribution of this work. But we also include a study that illustrates how such a tool can be used to identify the program features that more influence have on the performance of a given test case generator. This study could help the researches to propose test case generators that selects the most appropriate algorithm for the search of test cases depending on the value of some static measures taken from the program.

The rest of the paper is organized as follows. In the next section we present the measures that the program generator takes into account and are used in our experimental study. Then, in Section 3, we describe the automatic program generator, the main contribution of the paper. Later, Section 4 describes the empirical study performed and discusses the obtained results and Section 5 outlines some conclusions and future work.

2 Measures

Quantitative models are frequently used in different engineering disciplines for predicting situations, due dates, required cost, and so on. These quantitative models are based on some kinds of measure made on project data or items. Software Engineering is not an exception. A lot of measures are defined in Soft-

ware Engineering in order to predict software quality [14], task effort [4], etc. We are interested here in measures made on source code pieces. We distinguish two kinds of measures: *dynamic*, which requires the execution of the program, and *static*, which does not require this execution.

The static measures used in this study are eight: number of statements, number of atomic conditions per decision, number of total conditions, number of equalities, number of inequalities, nesting degree, coverage, and McCabe's cyclomatic complexity. The three first measures are easy to understand. The number of (in)equalities is the number of times that the operator == (! =) is found in atomic conditions of a program. The nesting degree is the maximum number of conditional statements that are nested one inside another. The cyclomatic complexity is a complexity measure related to the number of ways there exist to traverse a piece of code. This measure determines the minimum number of test cases needed to test all the paths using linearly independent circuits [8].

In order to define a coverage measure, we first need to determine which kind of element is going to be "covered". Different coverage measures can be defined depending on the kind of element to cover. *Statement coverage*, for example, is defined as the percentage of statements that are executed. In this work we use *branch coverage*, which is the percentage of branches of the program that are traversed. This coverage measure is used in most of the related papers in the literature.

3 Automatic Program Generator

We have designed a novel automatic program generator able to generate programs with values for the static measures that are similar to the ones of the real-world software, but the generated programs do not solve any concrete problem. Our program generator is able to create programs for which total branch coverage is possible. 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 general tree and a table of variables. The tree stores the statements 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, programs for which total coverage is reachable are desirable. With this goal in mind we introduce logic predicates in the program generation process.

The program generator is parameterizable, the user can set several parameters of the program under construction (PUC). Thus, we can assign through several probability distributions the number of statements of the PUC, the number of variables, the maximum number of atomic conditions per decision, and

the maximum nesting degree by setting these parameters. The user can define the structure of the PUC and, thus, its complexity. Another parameter the user can tune is the percentage of control structures or assignment statements that will appear in the code. By tuning this parameter the program will contain the desired density of decisions.

Once the parameters are set, the program generator builds the general scheme of the PUC. It stores in the used data structure (a general tree) the program structure, the visibility, the modifiers of the program, and creates a main method where the local variables are first declared. Then, the program is built through a sequence of basic blocks of statements where, according to a probability, the program generator decides which statement 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 can be traversed (using logic predicates) or this characteristic is not ensured.

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 which is 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. Thus, the predicates are 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 statement is added to the program, we need to compute the predicate at the point after the new statement. For this computation we distinguish two cases. First, if the new statement 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 statement is $x = x + 7$ and $CP \equiv x \leq 3$, then we have $CP' \equiv x \leq 10$.

Second, if the new statement 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 generated 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 statements are generated for the two branches and 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.

4 Experimental Section

In this section we present the experiments performed on a benchmark of programs created by the program generator. First, we describe how our test case generator works. Then, we explain how the benchmark of test programs was generated. In the remaining sections we show the empirical results and the conclusions obtained. Particularly, in Subsection 4.3 we study the correlations between some static measures and branch coverage when three different automatic test data generators are used.

4.1 Test Case Generator

Our test case 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 the function to be minimized is a distance between the current test case and one satisfying the partial objective. In order to solve such minimization problem Evolutionary Algorithms (EAs) are used.

In a loop, the test case generator selects a partial objective (a branch) and uses the optimization algorithm to search for test cases exercising that branch. When a test case covers a branch, the test case 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 case 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 cases associated to all the branches. In the following section we describe two important issues related to the test case generator: the objective function to minimize and the parameters of the optimization algorithms used.

Objective Function Following on from the discussion in the previous section, 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 case, 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 defined recursively 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.

When a test case 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) [15].

In this paper we also add a real valued penalty in the objective function to those test cases that do not reach the branching condition of the target branch. With this penalty, denoted by p, the objective value of any test case that does not reach the target branching condition is higher than the one of any test case 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:

$$f_b(x) = \begin{cases} bd_b(x) & \text{if } b \text{ is reached by } x \\ bd_c(x) + ap(c, b) + p & \text{otherwise} \end{cases} \quad (1)$$

where c is the critical branching condition, and bd_b, bd_c are the branch distances of branching conditions b and c.

To finish this section, we show in Table 1 a summary of the parameters used by the two EAs in the experimental section.

Table 1. Parameters of the two 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.	5000 evals.	5000 evals.

4.2 Benchmark of Test 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. The main advantage of these programs is that algorithms can be tested and analyzed in a fair way. This kind of programs are not easy to find in the literature.

Our program generator takes into account the desired values for the number of atomic conditions, the nesting degree, the number of statements and the number of variables. With these parameters the program generator creates a program with a defined control flow graph containing several conditions. The main features of 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 (size, nesting degree, complexity, number of decisions and atomic conditions per decision), 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, atomic conditions per decisions in 1-4, and statements in 25, 50, 75, 100. The previous values are realistic with respect to the static measures, making the following study meaningful.

Finally, we generated a total of 800 Java programs using our program generator and we applied our test case generator using an ES and a GA as optimization algorithms. We also add to the study the results of a random test case generator (RND). This last test case generator proceeds by randomly generating test cases until total coverage is obtained or a maximum of 100,000 test cases are generated. Since we are working with stochastic algorithms, we perform in all the cases 30 independent runs of the algorithms to obtain a very stable average of the branch coverage. The experimental study requires a total of $800 \times 30 \times 3 = 72,000$ independent runs of the test case generators.

4.3 Correlation between Coverage and Static Measures

After the execution of all the independent runs for the three algorithms in the 800 programs, in this section we analyze the correlation between the static measures and the coverage. We use the Spearman's rank correlation coefficient ρ to study the degree of correlation between two variables.

First, we study the correlation between the number of statements and the branch coverage. We obtain a correlation of 0.173, 0.006, and 0.038 for these two variables using the ES, GA and RND, respectively. In Table 2 we show the average coverage against the number of statements for all the programs and algorithms. It can be observed that the number of statements is not a significant parameter for GA and RND, however, if we use ES, the influence on coverage exists. Thus, we can state that the test case generator obtains better coverage on average with ES in large programs.

Table 2. Relationship between the number of statements and the average coverage for all the algorithms. The standard deviation is shown in subscript

# Statements	ES	GA	RND
25	$77.31_{13.31}$	$87.59_{12.66}$	$76.85_{16.17}$
50	$78.40_{12.15}$	$88.58_{11.36}$	$74.82_{16.10}$
75	$80.92_{10.21}$	$89.51_{9.54}$	$78.56_{13.75}$
100	$82.93_{10.36}$	$89.95_{8.75}$	$77.90_{14.02}$
ρ	0.173	0.006	0.038

Second, we analyze the nesting degree. In Table 3, we summarize the coverage obtained in programs with different nesting degree. If the nesting degree is increased, the branch coverage decreases and vice versa. It is clear that there is an inverse correlation between these variables. The correlation coefficients are -0.526 for ES, -0.314 for GA, and -0.434 for RND, which confirms the observations. These correlation values are the highest ones obtained in the study of the different static measures, so we can state that the nesting degree is the

parameter with the highest influence on the coverage that evolutionary testing techniques can achieve.

Table 3. Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript

Nesting degree	ES	GA	RND
1	$88.53_{6.85}$	$94.05_{5.84}$	$86.13_{10.81}$
2	$82.38_{8.36}$	$90.28_{8.22}$	$79.87_{13.07}$
3	$76.92_{10.12}$	$87.72_{11.01}$	$73.46_{13.99}$
4	$71.74_{13.41}$	$83.57_{13.38}$	$68.67_{16.03}$
ρ	-0.526	-0.314	-0.434

Now we study the influence on coverage of the number of equalities and inequalities found in the programs. It is well-known that equalities and inequalities are a challenge for automatic software testing. This fact is confirmed in the results. The correlation coefficients are -0.184, -0.180, and -0.207 for equalities and -0.069, -0.138, and -0.095 for inequalities using ES, GA, and RND, respectively. We conclude that the coverage decreases as the number of equalities increases for all the algorithms. However, in the case of the number of inequalities only the GA is affected by them.

Let us analyze the rest of static measures of a program. We studied the correlation between the number of atomic conditions per decision and coverage. After applying Spearman's rank correlation we obtained low values of correlation for all the algorithms $(0.031, -0.012, 0.035)$. From the results we conclude that there is no correlation between these two variables. This could seem counterintuitive, but a large condition with a sequence of logical operators can be easily satisfied due to OR operators. Otherwise, a short condition composed of AND operators can be more difficult to satisfy.

Now we analyze the influence on the coverage of the number of total conditions of a program. The correlation coefficients are -0.004 for ES, -0.084 for GA, and -0.082 for RND. At a first glance, it seems that the number of total conditions has no influence on the coverage, nevertheless, calculating the density of conditions (number of total conditions / number of statements), one can realize that the influence exists. The correlation coefficients between the coverage and the density of conditions are -0.363 for ES, -0.233 for GA, and -0.299 for RND. In Figure 1.a) the tendency is clear: a program with a high density of conditions is more difficult to test, especially for the ES.

Finally, we study the relationship between the McCabe's cyclomatic complexity and coverage. In Figure 1.b), we plot the average coverage against the cyclomatic complexity for GA in all the programs. In general we can observe that there is no clear correlation between both parameters. The correlation coefficients are 0, -0.085, and -0.074 for ES, GA, and RND, respectively. These values are low, and confirm the observations: McCabe's cyclomatic complexity and branch coverage are not correlated.

intended

Furthermore, the correlation coefficients are lower than the coefficients we have obtained with other static measures like the nesting degree, the density of conditions, the number of equalities, and the number of inequalities. This

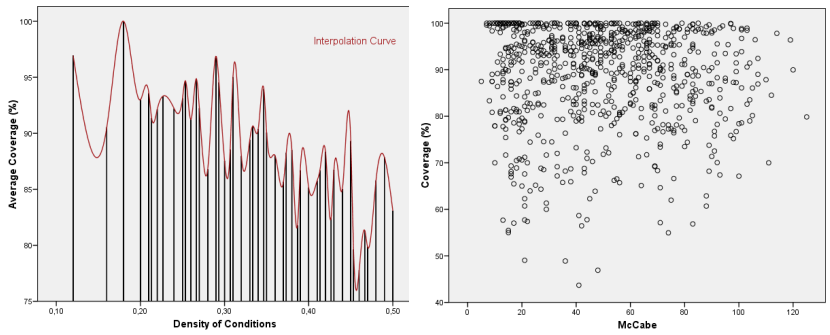

Fig. 1. Average branch coverage against the Density of conditions for GA and all the programs (a) and average branch coverage against the McCabe's cyclomatic complexity for GA and all the programs (b)

is somewhat surprising, because it would be expected that a higher complexity implies also a higher difficulty to find an adequate test case suite. The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [2, 7]. Most such studies find a strong positive correlation between the cyclomatic complexity and the defects: the higher the complexity the larger the number of faults. However, we cannot say the same with respect to the difficulty in automatically finding a test case suite. McCabe's cyclomatic complexity cannot be used as a measure of the difficulty of getting an adequate test suite.

We can go one step forward and try to justify this unexpected behavior. We have seen in the previous paragraphs that the nesting degree is the static measure with the highest influence on the coverage. The nesting degree has no influence on the computation of the cyclomatic complexity. Thus, the cyclomatic complexity is not taking into account the information related to the nested code, it is based on some other static information that has a lower influence on the coverage. This explanation is related to one of the main criticisms that McCabe complexity has received since it was proposed, namely: Piwowarski [11] noticed that cyclomatic complexity is the same for N nested `if` statements and N sequential `if` statements.

5 Conclusions

In this work we have developed a program generator that is able to create programs for which total branch coverage is reachable. To achieve this desirable characteristic, we have used logic predicates to represent the set of possible values that the variables can get at a given point of the PUC. We illustrate the potential of the generator developing a study that could hardly be done in any other way. We created a benchmark of 800 programs and analyzed the correlations between the static measures taken on the programs and the branch coverage obtained by three different test case generators. In the empirical study we included eight static measures: number of statements, number of atomic con-

ditions per decision, number of total conditions, density of conditions, nesting degree, McCabe's cyclomatic complexity, number of equalities, and number of inequalities. The results show that the nesting degree, the density of conditions and the number of equalities are the static measures with a higher influence on the branch coverage obtained by automatic test case generators like the ones used in the experimental section. This information can help a test engineer to decide which test case generator s/he should use for a particular test program.

As future work we plan to advance in the flexibility of the program generator. We should add more parameters to the program generator with the aim of giving total control of the structure of the program under construction to the user. We should also modify the program generator to be able to create Object Oriented programs in order to broaden the scope of applicability.

References

1. T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation.* Oxford University Press, New York NY, 1997.
2. V. Basili and B. Perricone. Software errors and complexity: an empirical investigation. *ACM commun*, 27(1):42–52, 1984.
3. B. Beizer. *Software testing techniques.* Van Nostrand Reinhold Co., New York, USA, 2nd edition, 1990.
4. B. Boehm, C. Abts, A. W. Brown, S. Chulani, B. K. Clark, E. Horowitz, R. Madachy, D. J. Reifer, and B. Steece. *Software cost estimation with COCOMO II.* Prentice-Hall, 2000.
5. M. Harman. The current state and future of search based software engineering. In *Proceedings of ICSE/FOSE '07*, pages 342–357, Minneapolis, Minnesota, USA, 20-26 May 2007.
6. M. Harman and B. F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, December 2001.
7. T. Khoshgoftaar and J. Munson. Predicting software development errors using software complexity metrics. *Jnl. on Selected Areas in Communications*, 1990.
8. T. J. McCabe. A complexity measure. *IEEE Trans. on Software Engineering*, 2(4):308–320, 1976.
9. P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
10. W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.
11. P. Piwarski. A nesting level complexity measure. *SIGPLAN*, 17(9):44–50, 1982.
12. I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution.* Fromman-Holzboog Verlag, Stuttgart, 1973.
13. G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, vol. 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.
14. I. Samoladas, G. Gousios, D. Spinellis, and I. Stamelos. *Open Source Development, Communities and Quality*, vol.275, chapter The SQO-OSS Quality Model, pages 237–248. 2008.
15. J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.