International Journal of Software Engineering and Its Applications
Vol. 4, No. 4, October 2010

Correlation Between Static Measures and Code Coverage in Evolutionary Test Data Generation

Javier Ferrer, Francisco Chicano and Enrique Alba

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

Abstract

Evolutionary testing is a very popular domain in the field of search based software engineering that consists in automatically generating test data for a given piece of code using evolutionary algorithms. One of the most important measures used to evaluate the quality of the generated test suites is code coverage. In this paper we first analyze if there exists a correlation between some static measures computed on the test program and the code coverage when an evolutionary test data generator is used. In particular, we use and compare three techniques for the search engine of the test data generator: an Evolutionary Strategy, a Genetic Algorithm, and a Random Search. We have also developed a program generator that is able to create Java programs with the desired values for the given static measures. Our experimental study includes a benchmark of 1800 programs automatically generated. In addition to the correlations study we also analyze the subset of programs for which one algorithm is better than another one. This second analysis could be the basis for the development of a software tool that automatically decides the suitable test data generation search engine according to the static measures computed on the test object.

Keywords: Evolutionary testing, branch coverage, evolutionary algorithms

1.Introduction

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [13, 14]. From the first works [20] to nowadays many approaches have been proposed for solving the automatic test data generation problem. 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 algorithms for generating test data [17]. 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 with a focus on different elements of the structure of a program studied in detail. Some examples are the presence of flags in conditions [5], the coverage of loops [10], the existence of internal states [31], and the presence of possible exceptions [27]. In addition, several evolutionary algorithms have been used as search engine like scatter search [8], genetic algorithms [1, 3], simulated annealing [30] or tabu search [11].

The objective of an automatic test data generator used for structural testing is to find a test data suite that is able to cover all the software elements. These elements can be instructions, branches, atomic conditions, and so on. The performance of an automatic test data 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 data generator, but also on the program being tested. Then, we can raise the following research questions:

– *RQ1:* Is there any static measure of the test program having a clear correlation with the coverage percentage of a given algorithm?
– *RQ2:* How many of these measures exist and how they correlate with coverage?
– *RQ3:* Is it possible to use the static measures to determine *a priori* which test data generation algorithm is the best for a given program?

As we said before, coverage depends also on the test data generator. Then, in order to completely answer the questions we should use all the possible automatic test data generators or, at least, a large number of them. We can also focus on some test data generators and answer to 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 influence on the coverage of a set of static software measures when we use three test data generators: two of them based on evolutionary testing and an additional one based on random search. In a first step, we study the correlations between a dynamic measure (coverage) and static ones. This way, we answer RQ1 and RQ2. In a second step, we analyze the results in order to study which automatic test data generation method is more suitable for a given program.

The rest of the paper is organized as follows. In the next section we present the measures that we use in our study. Then, we detail the general structure of the test data generator used in Section 3. After that, Section 4 describes the experiments performed and discusses the results obtained. Finally, in Section 5 some conclusions and future work are outlined.

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 performed on project data or items. Software Engineering is not an exception. A lot of measures are defined in Software Engineering in order to predict software quality [25], task effort [9], etc. We are interested here in measures performed on source code pieces. We distinguish two kinds of measures: *dynamic*, which requires the execution of the program, and *static*, which does not require the execution.

The measures used in this study are eight: number of sentences, number of atomic conditions per decision, total number of decisions, number of equalities, number of inequalities, nesting degree, McCabe’s cyclomatic complexity, and branch coverage. 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. In the following paragraphs we describe in more detail the coverage and the McCabe's cyclomatic complexity.

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* [28], which is the percentage of branches exercised in a program. This coverage measure is used in most of the related papers in the literature [11, 26].

Cyclomatic complexity is a complexity measure of code related to the number of ways there exists 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 [16]. Cyclomatic complexity is computed using the control flow graph of the program: the nodes of the graph correspond to indivisible groups of sentences of a program, and a directed edge connects two nodes if the second sentence might be executed immediately after the first sentence. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program, and is formally defined as follows:

$$v(G) = E - N + 2P; \tag{1}$$

where E is the number of edges of the graph, N is the number of nodes of the graph, and P is the number of connected components.

In Figure 1, we show an example of control flow graph (G). It is assumed that each node can be reached by the entry node and each node can reach the exit node. The maximum number of linearly independent circuits in G is 9-6+2=5, so this is the cyclomatic complexity.

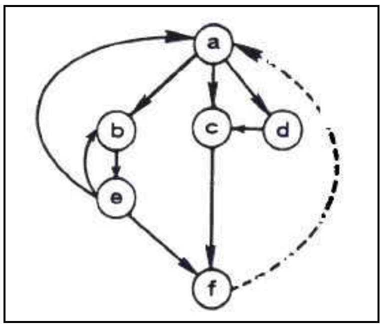


Fig. 1: The original graph of McCabe's article

The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [6, 15]. 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. For example, a 2008 study by metric-monitoring software supplier Energy [12] analyzed classes of open-source Java applications and divided them into two sets based on how commonly faults were found in them. They found strong correlation between cyclomatic complexity and their faultiness, with classes with a combined complexity of 11 having a probability of being fault-prone of just 0.28, rising to 0.98 for classes with a complexity of 74.

In addition to this correlation between complexity and errors, a connection has been found between complexity and difficulty to understand software. Nowadays, the subjective reliability of software is expressed in statements such as “I understand this program well enough to know that the tests I have executed are adequate to provide my desired level of confidence in the software”. For that reason, we make a hard link between complexity and difficulty of discovering errors.

Since McCabe proposed the cyclomatic complexity, it has received several criticisms. Weyuker [29] concluded that one of the obvious intuitive weaknesses of the cyclomatic complexity is that it makes no provision for distinguishing between programs which perform very little computation and those which perform massive amounts of computation, provided that they have the same decision structure. Piwowarski [21] noticed that cyclomatic complexity is the same for *N* nested if statements and *N* sequential if statements.

In connection with our research questions, Weyuker’s critic is not relevant, since coverage does not take into account the amount of computation of a block of statements. However, Piworarski’s critic is important in our research because the *nesting degree* of a program is inversely correlated with branch coverage (see Section 4.2).

3.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 the function to be minimized is a distance between the current test data and one satisfying the partial objective. In order to solve such minimization problem Eolutionary Algorithms Algorithms (EAs) could be used. The main loop of the test data generator is shown in Fig. 2.

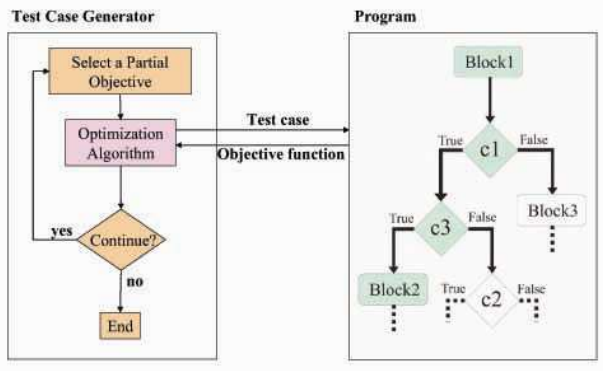


Fig. 2: The test data generation process

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 data covers a branch, the test data 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 are reached (10 consecutive failures in this work). 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 rest of this section we describe two important issues related to the test data generator: the objective function to minimize and the optimization algorithms used.

3.1 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 data, and its definition depends on the desired branch and whether the program flow reaches the branching decision associated to the target branch or not. If the decision is reached we can define the objective function on the basis of the logical expression of the branching decision and the values of the program variables when the decision 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. For the Java logical operators & and | we define the branch distance as[1]:

$$bd(a\&b) = bd(a) + bd(b) \tag{2}$$

$$bd(a\,|\,b) = \min(bd(a), bd(b)) \tag{3}$$

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 taking into account 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 [19]. We use here these distance measures described in the literature.

When a test data does not reach the branching decision of the target branch we cannot use the branch distance as objective function. In this case, we identify the branching decision c whose value must first change in order to cover the target branch (critical branching decision) and we define the objective function as the branch distance of this branching decision 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) [28].

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

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

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

Nested branches pose a great challenge for the search. For example, if the decision associated to a branch is nested within three conditional statements, all the decisions 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 decisions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMinn calls the *nesting problem* [18]), 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 decision 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*. In the experimental section of this work we use *Rf* = 25%. At the beginning of the generation process some random test data are generated in order to reach some branching decisions.

3.2 Optimization Algorithm

EAs [4] are metaheuristic search techniques loosely based on the principles of natural evolution, namely, adaptation, and survival of the fittest. These techniques have been shown to be very effective in solving hard optimization tasks. They are based on a set of tentative solutions (individuals) called *population*. The problem knowledge is usually enclosed in an objective function, the so-called *fitness function*, which assigns a quality value to the individuals. In Fig. 3 we show the main loop of an EA.

```
t := 0;
P(t) = Generate ();
Evaluate (P(t));
while not StopCriterion do
   P′(t) := VariationOps (P(t));
   Evaluate (P′(t));
   P(t+1) := Replace (P′(t),P(t));
   t := t+1;
endwhile;
```

Fig. 3: Pseudocode of an EA

Initially, the algorithm creates, randomly or by using a seeding algorithm, a population of μ individuals, each one representing an input of the test program. At each step, the algorithm applies stochastic operators such as selection, recombination, and mutation (we call them variation operators in Fig. 3) in order to compute a set of λ

[1] These operators are the Java *and*, *or* logical operators without shortcut evaluation. For the sake of clarity we omit here the definition of the branch distance for other operators.

descendant individuals $P'(t)$. 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 $P'(t)$ 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 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 these two EAs.

In an ES [23] each individual represents a test data input, being composed of a vector of real numbers representing the problem variables (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:

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

$$\omega_i' = \omega_i + \varphi N_i(0,1) \tag{6}$$

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

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω', $N(0, 1)$ is the standard univariate normal distribution, and $\mathrm{N}(0,C)$ is the multivariate normal distribution with mean 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 [24]. 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 x, where each component is selected form 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:

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

$$\sigma_i = (\sigma_i^1 + \sigma_i^2)/2 \tag{9}$$

$$\omega_i = (\omega_i^1 + \omega_i^2)/2 \tag{10}$$

where the subindices are used to denote the two parent solutions 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 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.

In our GA the individuals are vectors of integer values that represent a test data input. As the recombination operator we use 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,

$$x_i = x_i^1 + U(-500, 500)$$

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.

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.	Population	25 indivs.
Selection	Random, 5 indivs.	Selection	Random, 5 indivs.
Mutation	Gaussian	Mutation	Add $U(-500, 500)$
Crossover	discrete (bias = 0.6) + arith. + arith.	Crossover	Uniform
Replacement	Elitist	Replacement	Elitist
Stopping condition	5000 evaluations	Stopping condition	5000 evaluations

4. Experimental Section

In this section we present the experiments performed to answer the research questions and the results obtained. In the next section we explain how the benchmark of test programs was generated. In the remaining sections we show the empirical results and the conclusions obtained, answering the research questions. In Subsection 4.2 we answer RQ1 and RQ2 by studying the correlations between the static measures and coverage. Then, RQ3 is accomplished in Subsections 4.3 and 4.4, where we first compare the different test data generators and then we characterize the programs for which one algorithm is better than another one. In Subsection 4.2 we answer RQ1 and RQ2 by studying the correlations between the static measures and coverage. Then, RQ3 is accomplished in Subsections 4.3 and 4.4, where we first compare the different test data generators and then we characterize the programs for which one algorithm is better than another one.

4.1 Benchmark of Test Programs

In order to accomplish our empirical studies we first need a large number of test programs. For the study to be well-founded, we require a big number of programs having the same value for the static measures, as well as programs having different values for the measures. It is not easy to find such a variety of programs in the related literature. Thus, we decided to automatically generate the programs. This way, it is possible to randomly generate programs with the desired values for the static measures and, more important, we can generate different programs with the same values for the static measures.

The automatic program generation raises a non-trivial question: are the generated programs "realistic"? That is, could them be found in real-world? Using automatic program generation it is not likely to find programs that are similar to the ones who a programmer would make. This is especially true if the program generation is not driven by a specification. However, this is not a drawback in our study, since our analysis is based only on some static measures of the programs and branch coverage. In this situation, "realistic programs" means programs that have similar values for the considered static measures as the ones found in real-world; and we can easily fulfil this requirement.

Our program generator takes into account the desired values for the number of atomic conditions, the nesting degree, the number of sentences and the number of variables. With these parameters and other (less important) ones, the program generator creates a program with a defined control flow graph containing several decisions. 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.

Due to the randomness of the generation, the static measures could take values that are different from the ones specified in the configuration file of the program generator. For this reason, in a later phase, we used the free tool CyVis to measure the actual values for the static measures. CyVis [22] is a free software tool for metrics collection, analysis, and visualization of Java based programs.

The methodology applied for the program generation is 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, e.g., the number of sentences (10-294), McCabe's complexity (1-80) or nesting degree (1-7), 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 generate programs that are realistic with respect to the static measures, making the following study meaningful. Finally, we generated a total of 1800 Java programs using our program generator and we applied our test data generator using an ES and a GA as optimization algorithms. We also add to the study the results of a random test data generator (RND). This last test data generator proceeds by randomly generating test data until total coverage is obtained or a maximum of 100,000 test data are generated. We selected this high number of test data as stopping condition because it is higher than the number of test data generated by any of the EA based test data generators in the empirical study. 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 $1800 \times 30 \times 3 = 162{,}000$ independent runs of test data generators.

4.2 Correlation between Coverage and Static Measures

After the execution of all the independent runs for the three algorithms in the 1800 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 sentences and the branch coverage. We obtain a correlation of 0.047, 0.069, and 0.073 for these two variables using the ES, GA, and RND, respectively. In Fig. 4 we plot the average coverage against the number of sentences for ES and all the programs. It can be observed that the number of sentences is not a significant parameter and it has no influence on the coverage measure. The results obtained with the other two algorithms are similar and we omit the corresponding plots.

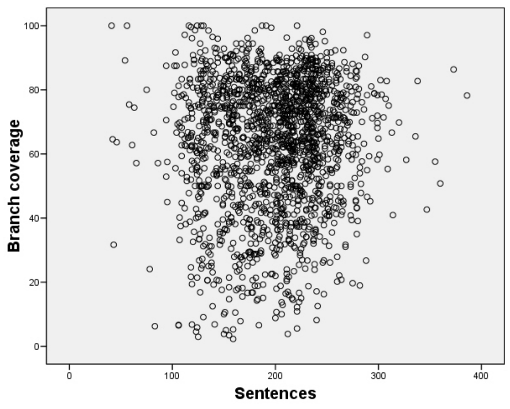


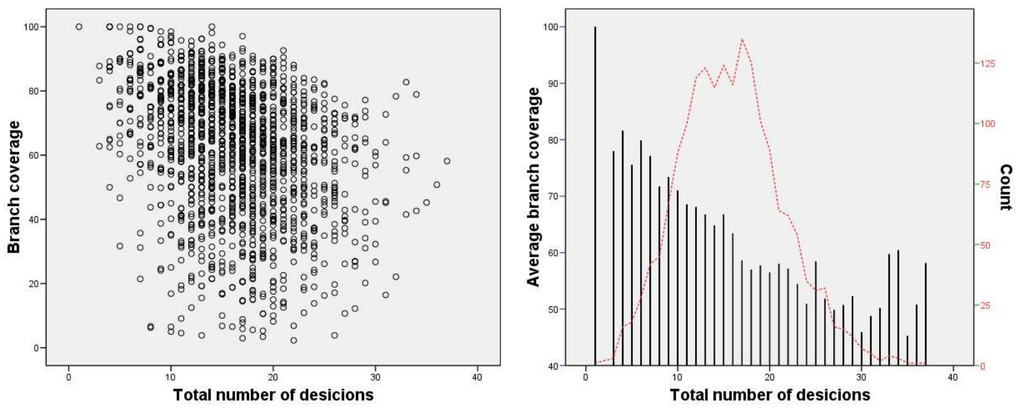
Fig. 4: Average branch coverage against the number of sentences for ES and all the programs

Second, we study 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 (0.026, 0.032, and 0.031). In Table 2 we show the coverage obtained for all the programs with different number of atomic conditions per decision when ES, GA, and RND are used. From the results we conclude that there is no correlation between these two variables. The minimum values for coverage are reached with 1 and 7 atomic conditions per decision. This could seem counterintuitive, but a large decision with a sequence of logical operators can be easily satisfied due to OR operators. Otherwise, a short decision composed of AND operators can be more difficult to satisfy.

Table 2: Relationship between the number of atomic conditions per decision and the average coverage. The standard deviation is shown in subscript

At. Conds.	ES	GA	RND
1	$60.11_{21.96}$	$58.87_{22.16}$	$58.86_{22.11}$
2	$62.37_{20.92}$	$60.99_{21.49}$	$60.94_{21.29}$
3	$63.03_{21.12}$	$62.19_{21.08}$	$61.96_{21.10}$
4	$66.68_{18.48}$	$66.22_{18.41}$	$65.86_{18.58}$
5	$66.56_{17.89}$	$65.71_{18.12}$	$65.53_{18.30}$
6	$63.02_{19.64}$	$61.99_{19.76}$	$61.92_{19.99}$
7	$58.41_{21.14}$	$57.80_{20.96}$	$57.38_{21.04}$
ρ	0.026	0.032	0.031

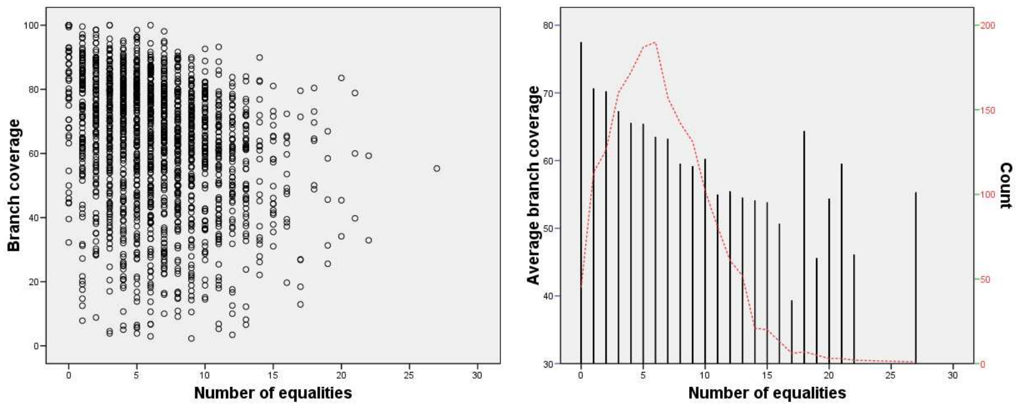
Now we analyze the influence on the coverage of the total number of decisions of a program. In Figure 5(a), we can observe that in programs with a small number of decisions ES reaches a higher coverage than in programs with a large number of decisions. This could be interpreted as large programs with many decisions are more difficult to test. If there are many decisions, then many different paths in the control flow graph will exist, this fact is also taken into account in the cyclomatic complexity. The correlation coefficients are −0.371 for ES, −0.351 for GA, and −0.348 for RND. With the aim of showing up the trend that we found with the correlation test, we plot in Fig. 5(b) the average code coverage against the total number of decisions in a different way. We complement this plot with the counting function. This function is useful to see the regions of the plot where most of the programs are concentrated. In this particular case, most of the programs contain between 10 and 25 decisions. Looking at this region, we can observe that the coverage decreases when the number of decisions increases.



(a) Coverage against the total number of decisions (b) Coverage against the total number of decisions with counting function

Fig. 5: Average branch coverage against the total number of decisions for ES and all the programs

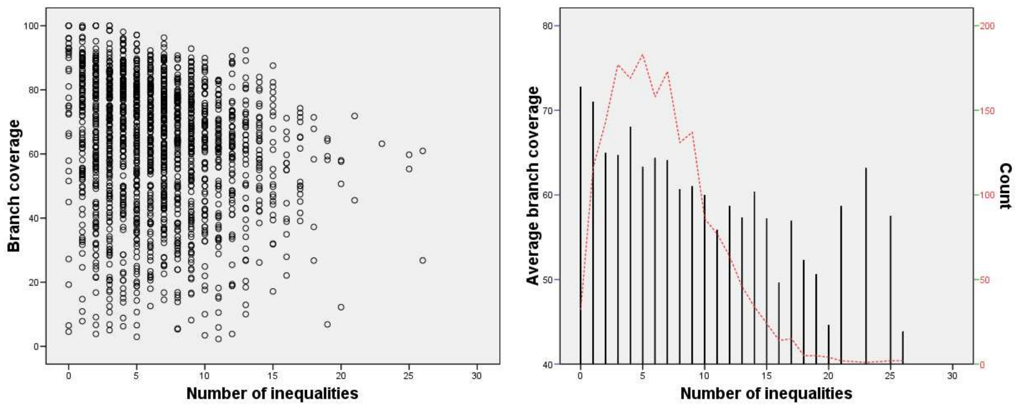
Now we study the influence on coverage of the number of equalities and inequalities found in the programs. It is well-known that equalities and inequalities are a challenge for automatic software testing. This fact is confirmed in the results. The correlation coefficients are −0.291, -0.272, and -0.270 for equalities and -0.219, -0.203, and -0.204 for inequalities using ES, GA, and RND, respectively. In Fig. 6(a), we plot the average coverage of all the programs against the number of equalities when the ES algorithm is used. In addition, Fig. 6(b) shows the average coverage of all the programs with the same number of equalities together with the counting function. The same information is showed in Figs. 7 (a) and (b) for the number of inequalities. If we compare both figures, they are quite similar, although the trend is clearer with the equalities because they are slightly more correlated with coverage. We conclude that the coverage decreases as the number of (in)equalities increases.



(a) Coverage against the number of equalities

(b) Coverage against the number of equalities with counting function

Fig. 6: Average branch coverage against the number of equalities for ES in all the programs



(a) Coverage against the number of inequalities

(b) Coverage against the number of inequalities with counting function

Fig. 7: Average branch coverage against the number of inequalities for ES in all the programs

Let us 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 −0.590 for ES, −0.590 for GA, and −0.589 for RND, what confirms the observations. These correlation values are the highest ones obtained in the study of the different static measures, so we can say that the nesting degree is the parameter with the highest influence on the coverage that evolutionary testing techniques can achieve. As we said in Section 3.1, nested branches pose a great challenge for the search. The high correlation value of the nesting degree supports that statement.

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	$82.33_{8.00}$	$81.54_{8.10}$	$81.37_{7.97}$
2	$72.94_{11.60}$	$72.10_{11.43}$	$71.86_{11.66}$
3	$68.95_{15.10}$	$68.26_{15.23}$	$68.20_{15.23}$
4	$61.18_{17.82}$	$59.85_{17.73}$	$59.83_{17.82}$
5	$52.53_{20.19}$	$52.16_{20.85}$	$51.83_{20.80}$
6	$48.76_{20.78}$	$47.05_{21.06}$	$46.33_{20.93}$
7	$43.43_{19.96}$	$42.47_{19.51}$	$42.77_{19.68}$
ρ	-0.590	-0.590	-0.589

Finally, we study the relationship between the McCabe's cyclomatic complexity and coverage. In Fig. 8, we plot the average coverage against the cyclomatic complexity for ES in all the programs. Since the figure does not show a clear trend, we plotted together the average coverage and the McCabe's cyclomatic complexity with the counting function in Fig. 9. In general we can observe that there is no clear correlation between both parameters. The correlation coefficients are −0.193, −0.173, and −0.173 for ES, GA, and RND, respectively. These values are low, and confirm the observations: McCabe's cyclomatic complexity and branch coverage are not highly correlated. Furthermore, the correlation coefficients are lower than the coefficients we have obtained with other static measures like the nesting degree, the total number of decisions, the number of equalities and the number of inequalities. This is somewhat surprising, we would expect a positive correlation between the complexity of a program and the difficulty to get an adequate test suite (represented here as the coverage). However, this is not true: McCabe's cyclomatic complexity cannot be used as a measure of the difficulty to get an adequate test suite.

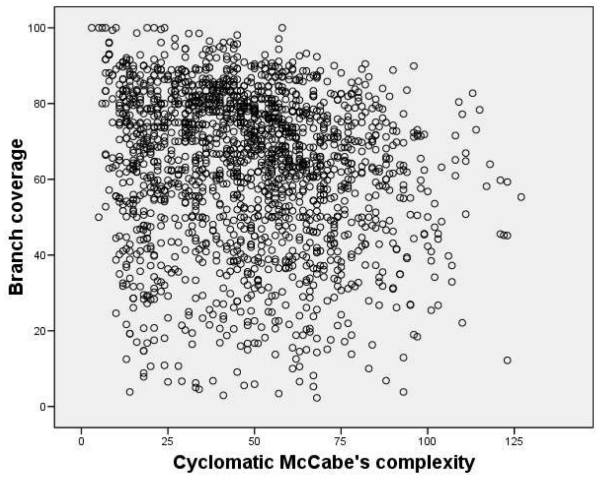


Fig. 8: Average branch coverage against the McCabe's cyclomatic complexity for ES and all the programs

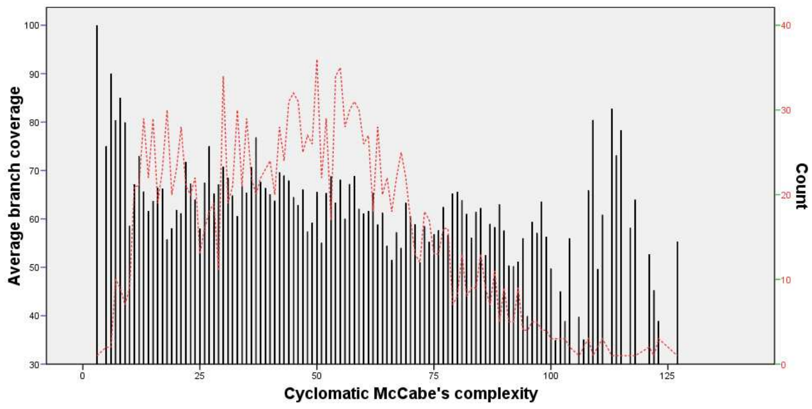


Fig. 9: Average branch coverage against the McCabe's cyclomatic complexity for ES and all the programs with the counting function

We can go one step forward and try to justify this unexpected behaviour. 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 measure 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.

4.3 Comparison of the Algorithms

In the previous section we have analyzed the correlations between the static measures and the branch coverage. For the study we used the 1800 programs that compose the benchmark. In the related literature it is usual to associate the coverage obtained by an evolutionary testing technique with its success. That is, it is considered that if technique A reaches a low branch coverage, say 30%, then technique A is not very good. However, this is not necessarily true. It could happen that the other 70% of the branches in the program cannot be exercised by any test data. This situation can also happen in our large benchmark. For this reason, we characterize now the coverage and the static measures of the programs for which we know that total branch coverage can be obtained or the maximum possible coverage is not easy to reach. We define in the following two subsets of the entire benchmark.

The first sub-benchmark is composed of all the programs for which 100% of branch coverage was obtained in at least one run of one algorithm. This subset contains 41 programs and will be denoted with 100-PCC (100 Per Cent Coverage) to distinguish it from the entire benchmark that we call 1800-JP (1800 Java Programs). On the one hand, this benchmark allows us to study which are the static measures with highest influence on the appearance of infeasible branches. On the other hand, it allows us to compare in a fair way the three algorithms we are running, since in 100-PCC the coverage percentage is really a measure of success.

The second sub-benchmark is composed of the programs that are not invariant for the algorithms. First, we must clear what we mean with "invariant". In the context of the experiments done, we say that a program is invariant for an algorithm if the average coverage obtained by the algorithm for this problem is the same as the maximum coverage obtained, which implies that for all the independent runs of the algorithm over the program the coverage was the same. That is, for each program and algorithm we performed 30 independent runs, so we have 30 coverage results. We compare the average of these 30 coverage values and the maximum value. If they coincide we say that the program is invariant for the algorithm. This does not necessarily means that it is easy for the algorithm to get the maximum possible coverage of the program. It just means that it is highly probably that the algorithm cannot reach a higher coverage for this program with the parameterization used. Then, the second sub-benchmark, denoted with NI-JP (Non-Invariant Java Programs), is composed of those programs that pose some challenge for the algorithms. There are 1455 programs fulfilling this condition, which is a large number of programs.

Once we have presented the benchmarks, let us show the results. The static measures of the programs and the coverage obtained by the three test data generators and the three benchmarks are shown in Table 4. We performed a Mann-Whitney statistical test to check the significance of the differences between the results of the sub-benchmarks and the entire benchmark. We use a confidence level of 95% and we highlight the values for which a statistically significant difference with the same parameter in the 1800-JP benchmark exists. In Table 5 we show the correlation coefficients between the static measures and the branch coverage for the three algorithms. For each measure we highlight the correlation with higher absolute value.

Table 4: Comparing the static measures and coverage of the three benchmarks

	1800-JP	100-PCC	NI-JP
Coverage ES	$63.26_{20.29}$	$93.72_{7.37}$	$62.73_{19.35}$
Coverage GA	$62.34_{20.44}$	$90.11_{9.97}$	$61.59_{19.53}$
Coverage RND	$62.16_{20.48}$	$88.81_{11.47}$	$61.37_{19.57}$
McCabe	$48.05_{23.84}$	$25.68_{16.17}$	$50.09_{23.96}$
Sentences	$198.31_{47.74}$	$158.29_{53.00}$	$199.62_{47.94}$
Nesting	$3.70_{1.79}$	$2.22_{1.39}$	$3.85_{1.75}$
Conditions	$3.71_{1.87}$	$2.90_{1.65}$	$3.76_{1.87}$
Num. equals.	$6.48_{3.92}$	$2.32_{2.25}$	$6.73_{3.99}$
Num. inequals.	$6.55_{4.08}$	$2.71_{2.47}$	$6.83_{4.12}$
Num. decisions	$15.85_{5.61}$	$8.95_{3.68}$	$16.45_{5.57}$

Table 5: Correlation coefficients of the static measures against the coverage for the three benchmarks and the three algorithms

ρ	ES coverage			GA coverage			RND coverage		
	1800-JP	100-PCC	NI-JP	1800-JP	100-PCC	NI-JP	1800-JP	100-PCC	NI-JP
McCabe	-0.193	-0.263	-0.182	-0.173	-0.261	-0.153	-0.173	-0.305	-0.153
Sentences	0.047	-0.043	0.031	0.069	-0.048	0.058	0.073	-0.014	0.062
Nesting	-0.590	-0.096	-0.546	-0.590	-0.076	-0.544	-0.589	-0.151	-0.543
Conditions	0.026	-0.079	0.046	0.032	-0.064	0.055	0.031	-0.058	0.053
NumEquals	-0.291	-0.157	-0.280	-0.272	-0.199	-0.254	-0.270	-0.200	-0.251
NumNotEquals	-0.219	-0.240	-0.192	-0.203	-0.230	-0.168	-0.204	-0.301	-0.170
TotalNumDes	-0.371	-0.369	-0.357	-0.351	-0.339	-0.327	-0.348	-0.381	-0.324

Let us first focus on the comparison between the results of 100-PCC and 1800-JP. At a first glance, we can observe that the coverage values of the algorithms in 100-PCC are higher than the same values in 1800-JP: they have increased around 30%. This is not surprising since in 100-PCC all the programs with unfeasible branches have been removed. If we focus on the coverage for 100-PCC we can also compare the algorithms. From the comparison we conclude that, in general, the best algorithm is ES, followed by GA, and, finally, RND. This result supports some other ones published in the literature [2].

If we focus on the static measures, we can observe that, in general, they decrease when we consider the 100-PCC benchmark. Especially interesting is the decrease of the nesting degree (from 3.70 to 2.22), which is the most influent measure when the coverage is considered. In this case we observe in Table 5 that the absolute value of the correlation coefficient has been reduced (from -0.590 to -0.096 in ES). The reason is that the programs that can be totally covered have very low values of nesting degree, and a reduction in the number of possible values in one variable reduces the correlation coefficient with any other variable.

The total number of decisions is reduced to its half on average (from 15.85 to 8.95), being the unique static measure that maintained the coefficients of correlation close to the original ones. Thus, we can assure that there is a strong correlation between branch coverage and the total number of decisions.

Let us focus on the number of equalities. The average number of equalities have been reduced about one third, which means that the programs for which total coverage is possible have on average just over two equalities. The correlation coefficient varies from −0.291 to −0.157 in ES. The last correlation coefficient is the lowest one in absolute value of the three algorithms, which indicates that when there are few equalities ES works better. We then conclude that the ES is quite sensitive to the number of equalities that appear in the code in both benchmarks. With respect to the number of inequalities, the RND algorithm changed the correlation coefficients from −0.203 to −0.301, what indicates that the inequalities are a challenge for that algorithm, even in 100-PCC.

To finish this analysis, we study the McCabe's cyclomatic complexity. The average complexity is reduced to a half in 100-PCC, which means that the control flow graph is simpler. This is obvious because all the static measures tightly correlated with the McCabe's complexity, like total number of decisions, have decreased to more than half its value. With respect to the correlation coefficients, they increase in absolute value, especially in the case of RND (from −0.173 to −0.305).

Let us now focus on the comparison between the results of NI-JP and 1800-JP. In this case the average of the static measures is quite similar in both benchmarks, so there is no much to say. However, there are small differences. In particular, all the static measures rise and all the coverage percentages decrease in NI-JP. In addition, all the correlation coefficients slightly decrease (in absolute value) in general. The observations support the previous findings: as the static measures increase the coverage decreases for the three algorithms. Once again the performance of the algorithms is ordered in the same way: according to the average coverage, ES is the best algorithm followed by GA and RND.

4.4 Characterizing Programs According to the Algorithms Results

In this section we focus on the algorithm used in the test data generator's core with the aim of characterizing the programs for which one algorithm is better than another. In this study, we compare the algorithms in a pairwise way. For each pair of algorithms, we select the programs where the difference between the average coverage obtained by the algorithms is higher than a given value δ. Then, we analyze the features of the selected programs to conclude some rules that can help a tester to decide which test data generator is more appropriate for their programs. The values of δ used are δ=15% and δ=0%. For each comparison we highlight with a gray background the values for which a statistically significant difference exists. We remove from this study two static measures that have a low correlation with the branch coverage in any scenario: the number of sentences and the number of atomic conditions per decision.

First, we compare ES and RND in Table 6. Let us analyze the features of the programs for which δ = 15%. The first observation we can highlight is that the number of programs for which ES is better than RND (44) is higher than the number of programs for which the opposite happens (19). The differences in the static measures for the two different scenarios are not statistically significant. The programs for which the ES is better have a slightly higher value for the nesting degree. The remaining static measures are slightly reduced for these programs. Now we focus on the case in which δ=0%. Again, the number of programs for which ES is better is higher (922 against 489 for the contrary situation). In this case we find statistically significant differences in the McCabe's complexity, the number of (in)equalities, and the total number of decisions. Based on these differences we conclude that the performance of ES improves as the values of the mentioned measures decrease.

Table 6: Average and standard deviation of the static measures and coverage for the programs involved in the pairwise comparison between ES and RND.

	ES > RND+15%	RND > ES+15%	ES > RND	RND > ES
Coverage ES	$65.38_{15.22}$	$45.78_{18.56}$	$64.00_{19.66}$	$60.46_{18.54}$
Coverage RND	$42.51_{17.08}$	$68.98_{15.80}$	$60.01_{20.25}$	$63.93_{17.94}$
McCabe	$44.30_{20.84}$	$46.26_{21.77}$	$48.82_{23.43}$	$52.92_{24.37}$
Nesting	$4.82_{1.57}$	$4.68_{1.56}$	$3.82_{1.78}$	$3.94_{1.68}$
Num. equals.	$5.55_{3.94}$	$6.32_{3.42}$	$6.58_{3.96}$	$7.04_{4.00}$
Num. inequals.	$5.95_{3.19}$	$6.16_{3.53}$	$6.64_{4.07}$	$7.21_{4.14}$
Num. decisions	$15.16_{4.98}$	$17.53_{5.21}$	$16.19_{5.44}$	$17.09_{5.72}$
Num. of progs.	44	19	922	489

Now we compare the ES and GA algorithms in Table 7. As in the previous comparison, when δ = 15% the differences in the static measures are not statistically significant and the number of programs for which ES obtained higher coverage is greater than the number of programs in the contrary situation. We can observe again that the programs for which the ES is better have a slightly higher value for the nesting degree and the remaining static measures are slightly reduced for these programs. Let us turn our attention to the programs for which δ= 0%. The number of programs for which ES is better is higher than the number of programs for which GA is better (870 against 516). We find statistically significant differences in the McCabe's complexity, the number of (in)equalities, and the total number of decisions. As in the ES-RND comparison, we conclude that the performance of ES improves as the values of the mentioned measures decrease.

Table 7: Average and standard deviation of the static measures and coverage for the programs involved in the pairwise comparison between ES and GA

	ES > GA+15%	GA > ES+15%	ES > GA	GA > ES
Coverage ES	$68.29_{15.16}$	$45.47_{21.59}$	$62.93_{19.97}$	$64.19_{18.39}$
Coverage GA	$46.61_{16.58}$	$70.07_{17.22}$	$59.53_{20.23}$	$61.69_{18.02}$
McCabe	$43.04_{19.37}$	$48.90_{22.05}$	$48.48_{23.48}$	$53.87_{24.40}$
Nesting	$5.04_{1.48}$	$4.70_{1.70}$	$3.85_{1.79}$	$3.92_{1.69}$
Num. equals.	$4.42_{3.68}$	$6.90_{4.43}$	$6.59_{3.94}$	$7.11_{4.12}$
Num. inequals.	$5.92_{3.19}$	$6.50_{3.97}$	$6.55_{4.02}$	$7.49_{4.25}$
Num. decisions	$15.58_{4.31}$	$18.20_{5.82}$	$16.12_{5.47}$	$17.38_{5.64}$
Num. of progs.	26	10	870	516

Finally, we compare the GA and RND algorithms in Table 8. In this case the number of programs for which one algorithm is better is very low for δ = 15% (6 for GA and 10 for RND). The differences in the average static measures are not statistically significant. If we focus on the situation in which δ = 0%, the large number of programs involved allows us to draw more reliable conclusions. The number of programs for which GA is better is higher than the number of programs with the opposite situation happens (727 against 469). The differences in the average static measures are not statistically significant. This fact indicates that both algorithms have the same performance in the same region of the static measures space. In other words, none of the static measures used clearly indicates which is the best algorithm to apply out from GA and RND.

Table 8: Average and standard deviation of the static measures and coverage for the programs involved in the pairwise comparison between GA and RND

	GA > RND+15%	RND > GA+15%	GA > RND	RND > GA
Coverage GA	$60.35_{7.93}$	$45.11_{14.46}$	$63.33_{18.37}$	$59.83_{19.36}$
Coverage RND	$32.65_{14.71}$	$67.83_{17.71}$	$61.28_{18.82}$	$62.31_{19.02}$
McCabe	$48.67_{24.94}$	$42.60_{20.89}$	$51.19_{24.56}$	$50.80_{23.05}$
Nesting	$5.00_{0.89}$	$5.20_{1.62}$	$3.88_{1.77}$	$3.87_{1.69}$
Num. equals.	$5.17_{3.31}$	$6.30_{2.98}$	$6.88_{4.20}$	$6.83_{3.79}$
Num. inequals.	$7.33_{5.78}$	$7.00_{2.79}$	$7.08_{4.33}$	$6.89_{3.91}$
Num. decisions	$14.17_{5.11}$	$17.60_{4.45}$	$16.81_{5.65}$	$16.56_{5.46}$
Num. of progs.	6	10	727	469

To conclude this section, we analyze the subset of programs for which one algorithm is better than the other two (in average coverage). This yields three subsets of programs whose features are shown in Table 9. A statistical test shows that the only statistically significant differences appear between GA and ES or RND and ES. For this reason, we only highlight the values of the GA and RND columns that are significantly different from the corresponding value of the ES column.

Table 9: Average and standard deviation of the static measures and coverage for the programs in which one algorithm is better than the other two.

	ES wins	GA wins	RND wins	1800-JP
Coverage	$63.34_{19.92}$	$65.35_{17.21}$	$64.42_{18.38}$	-
McCabe	$48.63_{23.32}$	$54.00_{25.93}$	$52.87_{22.39}$	$48.05_{23.84}$
Nesting	$3.86_{1.79}$	$3.78_{1.72}$	$3.91_{1.64}$	$3.70_{1.79}$
Num. equals.	$6.58_{3.91}$	$7.28_{4.38}$	$7.13_{3.71}$	$6.48_{3.92}$
Num. inequals.	$6.59_{4.06}$	$7.72_{4.18}$	$7.23_{3.92}$	$6.55_{4.08}$
Num. decisions	$16.10_{5.53}$	$17.75_{5.62}$	$17.04_{5.42}$	$15.85_{5.61}$
Num. of progs.	716	202	228	1800

McCabe's complexity is greater in the programs where GA obtains the better coverage. Regarding the nesting degree there is no statistically significant difference, moreover this is shown in the slightly difference that exists on average. If we focus on the number of equalities and inequalities, GA and RND algorithms seem to be better (with respect to ES) when a high number of them appear in the programs. These values are high in comparison with the average obtained for all the programs. This fact can yield in the following tentative rule: for programs with a low number of (in)equalities a test data generator based on ES is the most appropriate one, but when many (in)equalities are present in the code it is better to apply GA or RND. Let us focus on the total number of decisions. We can observe that the programs for which GA obtains the best coverage contains a higher number of decisions compared to the programs for which ES is the best algorithm. As the number of (in)equalities, this feature allows us to determine if ES is the appropriate algorithm or if GA or RND would be better instead.

As summary, from the previous analysis we can extract a heuristic rule to determine when ES is the best algorithm to apply and when it is better to use GA or RND. If one program has a low number of (in)equalities and a high total number of decisions then ES seems to be the best algorithm. On the contrary, if one program has a high number of (in)equalities and a low total number of decisions (that is, a high percentage of the conditions are equalities or inequalities) then GA or even RND could be better. Although the statistically significant differences in Table 9 support the previous rule, the differences are not very large on average, what suggests that many exceptions to the rule can be found.

5.Conclusions

In this work we have used a benchmark composed of 1800 Java programs to correlate the features of the programs with the branch coverage obtained by three different test data generators. For the features of the programs we selected seven static measures: number of sentences, number of atomic conditions per decision, number of total decisions, nesting degree, McCabe's cyclomatic complexity, number of equalities, and number of inequalities. In addition to the correlation analysis we characterized the features of the programs for which one algorithm is better than another one or the other two. With this information we were able to state some rules that can help a test engineer to decide which test data generator s/he should use for a particular test program. The results show that the nesting degree, the total number of decisions, and the number of (in)equalities are the static measures with the highest influence on the branch coverage obtained by automatic test data generators like the ones used in the experimental section. We have also observed that ES is a good algorithm for the test data generation process when the test object has a low number of (in)equalities and a high total number of decisions. On the contrary, if the number of (in)equalities is high and the number of decisions low then algorithms like GA or RND could be better. However, the differences are slight and more experiments must be done in order to find an approximate rule to determine a priori the performance of an algorithm over a test object.

As future work we plan to advance in the analysis of static and dynamic measures of a program. We should add more static measures to the study like the number of (in)equalities joined by AND operators or the number of decisions affected by the input, in order to be able of determining the best algorithm depending of the characteristic of the program under test should also add more test data generation algorithms to the study in order to characterize their scope of applicability. Finally, we would like to perform the same study on real-world software.

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 also been partially funded by the Andalusian Government under contract P07-TIC-03044 (DIRICOM project).

References

[1] Moataz A. Ahmed and Irman Hermadi. GA-based multiple paths test data generator. Computers & Operations Research, 35(10):3107–3124, 2008.
[2] E. Alba and F. Chicano. Observations in using parallel and sequential evolutionary algorithms for automatic software testing. *Computers & Operations Research*, 35(10):3161–3183, October 2008.
[3] Mohammad Alshraideh and Leonardo Bottaci. Search-based software test data generation for string data using program-specific search operators: Research articles. *Softw. Test. Verif. Reliab.*, 16(3):175–203, 2006
[4] T. Bӓck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.
[5] André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop–assigned flags: A testability transformation approach. *(ISSTA 2004)*, pages 108–118, 2004.
[6] Victor Basili and Barry Perricone. Software errors and complexity: an empirical investigation. *ACM commun*, 27(1):42–52, 1984.
[7] Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.
[8] Raquel Blanco, Javier Tuya, and Belarmino Adenso-Díaz. Automated test data generation using a scatter search approach. *Inf. Softw. Technol.*, 51(4):708–720, 2009.
[9] Barry Boehm, Chris Abts, A. Winsor Brown, Sunita Chulani, Bradford K. Clark, Ellis Horowitz, Ray Madachy, Donald J. Reifer, and Bert Steece. *Software cost estimation with COCOMO II*. Prentice-Hall, 2000.
[10] Eugenia Díaz, Raquel Blanco, and Javier Tuya. Tabu search for automated loop coverage in software testing. In *Proceedings of the International Conference on Knowledge Engineering and Decision Support (ICKEDS)*, pages 229–234, Porto, 2006.
[11] Eugenia Díaz, Javier Tuya, Raquel Blanco, and José Javier Dolado. A tabu search algorithm for structural software testing. *Computers & Operations Research*, 35(10):3052 – 3072, 2008.
[12] Mark Dixon. An objective measure of code quality. *Technical Report*, February 2008.

[13] Mark Harman and Bryan F. Jones. Search-based software engineering. *Information & Software Technology*, 43(14):833–839, December 2001.
[14] T.M. Khoshgoftaar and J.C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 1990.
[15] Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.
[16] Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
[17] Phil McMinn, David Binkley, and Mark Harman. Testability transformation for efficient automated test data search in the presence of nesting. *Proceedings of the Third UK Software Testing Workshop 2005*, pages 165–182, 2005.
[18] Christoph C. Michael, Gary McGraw, and Michael A. Schatz. Generating software test data by evolution. *IEEE Transactions on Software Engineering*, 27(12):1085–1110, December 2001.
[19] Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.
[20] Paul Piwarski. A nesting level complexity measure. *SIGPLAN*, 17(9):44–50, 1982.
[21] Vinay Iyer Pradeep Selvaraj. Cyvis. Sourceforge, 2005.
[22] I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biol- ogischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.
[23] G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.
[24] Ioannis Samoladas, Georgios Gousios, Diomidis Spinellis, and Ioannis Stamelos. *Open Source Development, Communities and Quality*, volume 275 of *IFIP International Federation for Information Processing*, chapter The SQO-OSS Quality Model: Measurement Based Open Source Software Evaluation, pages 237–248. Srpinger, 2008.
[25] Nigel Tracey, John Clark, Keith Mander, and John McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE Conference on Automated Software Engineering*, pages 285–288, Hawaii, USA, October 1998.
[26] Nigel Tracey, John Clark, Keith Mander, and John McDermid. Automated test-data generation for exception conditions. *Software Practice and Experience*, 30(1):61–79, 2000.Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.
[27] E.J. Weyuker. Evaluating software complexity measures.*IEEE Trans. Software Eng.*, 14(9) : 1357 –1365, 1988.
[28] Man Xiao, Mohamed El-Attar, Marek Reformat, and James Miller. Empirical evaluation of optimization algorithms when used in goal-oriented automated test data generation techniques. *Empirical Software Engineering*, 12(2):183–239, 2007.
[29] Yuan Zhan and John A. Clark. The state problem for test generation in simulink. In *GECCO'06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1941–1948. ACM Press, 2006.