

On the Correlation between Static Measures and Code Coverage using Evolutionary Test Case Generation

Javier Ferrer, Francisco Chicano, y Enrique Alba

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

Resumen. Evolutionary testing is a very popular domain in the field of search based software engineering that consists in automatically generating test cases 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 want to analyze if there exists a correlation between some static measures computed on the test program and the code coverage when an evolutionary test case generator is used. In particular, we use Evolutionary Strategies (ES) as search engine of the test case generator. We have also developed a program generator that is able to create Java programs with the desired values of the static measures. The experimental study includes a benchmark of 3600 programs automatically generated to find correlations between the measures. The results of this study can be used in future work for the development of a tool that decides the test case generation method according to the static measures computed on a given program.

Palabras clave: Evolutionary testing, branch coverage, evolutionary algorithms, evolutionary strategy

1 Introduction

Automatic software testing is one of the most studied topics in the field of Search-Based Software Engineering (SBSE) [?]. From the first works [10] to nowadays many approaches have been proposed for solving the automatic test case 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 [4]. 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 [?]. In fact, the term *evolutionary testing* was coined to refer to this approach. In the paradigm of *structural testing* a lot of research has been performed using EAs and, in particular, different elements of the structure of a program have been studied in detail. Some examples are the presence of flags in conditions [2], the coverage of loops [5], the existence of internal states [18] and the presence of possible exceptions [15].

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 instructions, 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. Then, we can ask the following research questions:

- *RQ1*: Is there any static measure of the test program having a clear correlation with the coverage percentage?
- *RQ2*: Which are these measures and how they correlate with coverage?

As we said before, coverage depends also on the test case generator. Then, in order to completely answer the questions we should use all the possible automatic test case generators or, at least, a large number of them. We can also focus on one test case generator and answer to the previous questions on this generator. 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 an evolutionary test case generator. This study can be used to predict the value of a dynamic measure, coverage, from static measures. This way, it is possible to develop tools taking into account the static measures to decide which automatic test case 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 evolutionary test case 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

The measures used in this study are six: number of sentences, number of atomic conditions per condition, total number of conditions, nesting degree, coverage, and McCabe’s cyclomatic complexity. The three first measures are easy to understand. 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*, which is the percentage of branches exercised in a program. This coverage measure is used in most of the related papers in the literature.

Cyclomatic complexity is a complexity measure of code related to the number of ways there are to traverse a piece of code. This determines the minimum number of inputs needed to test all the ways to execute the program. 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 $9-6+2=5$, and this is the cyclomatic complexity.

The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [3, 7]. Most such studies find a strong positive correlation between the cyclomatic complexity and the defects: the higher the complexity

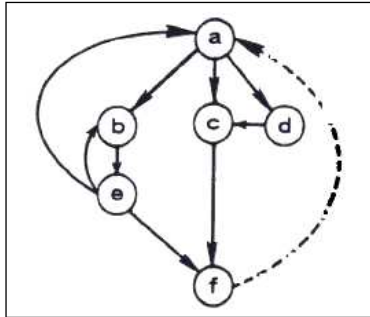


Fig. 1. The original graph of the McCabe’s article

the larger the number of faults. For example, a 2008 study by metric-monitoring software supplier Energy [6], 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 [17] 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. Piwowski [12] 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 made by a block of statements. However, Piwowski’s critic is important in our research because the nesting degree of a program is inverse correlated with the branch coverage as we will show in the experimental section.

3 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 EAs are used. The main loop of the test data generator is shown in Fig. 2.

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

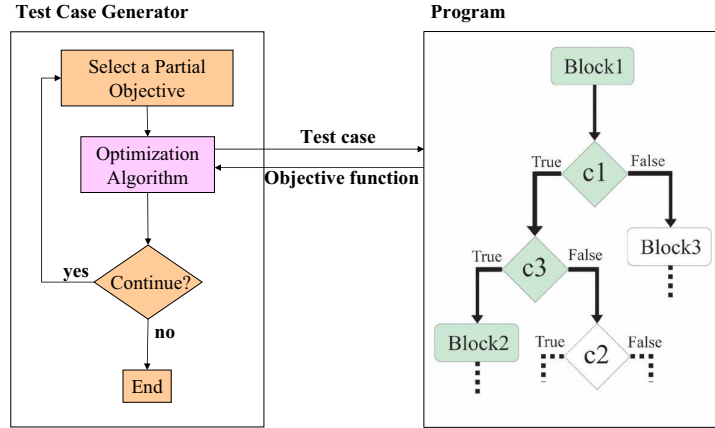


Fig. 2. The test case generation process

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 rest of this section we describe two important issues related to the test case generator: the objective function to minimize and the optimization algorithm 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 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. For the Java logical operators $\&$ and $|$ we define the branch distance as¹:

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

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

where a and b are logical expressions.

In order to completely specify the branch distance we need to define its value in the base case of the recursion, that is, for atomic conditions. The particular expression used for the branch distance in this case depends on the operator of the atomic condition. The operands of the condition appear in the expression. A lot of research has been devoted in the past to the study of appropriate branch distances in software testing. An accurate branch distance 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

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.

are well-known and popular in the literature. They are based on distance measures defined for relational operators like $<$, $>$, and so on [9]. We use these distance measures used in the literature.

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) [16].

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 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:

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

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

Nested branches pose a great challenge for the search. For example, if the condition associated to a branch is nested within three conditional statements, all the conditions of these statements must be true in order for the program flow to proceed onto the next one. Therefore, for the purposes of computing the objective function, it is not possible to compute the branch distance for the second and third nested conditions until the first one is true. This gradual release of information might cause efficiency problems for the search (what McMin calls the *nesting problem* [11]), which forces us to concentrate on satisfying each predicate sequentially.

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

3.2 Optimization Algorithm

EAs [1] 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.

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

```

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

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 the following we focus on the details of the specific EAs used in this work to perform the test case generation.

We used an Evolutionary Strategy (ES) for the search of test cases for a given branch. In an ES [13] each individual is composed of a vector of real numbers representing the problem variables (\mathbf{x}), a vector of standard deviations (σ) and, optionally, 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. 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. However, this operator is less important than the mutation. The mutation operator is governed by the three following equations:

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

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

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

where $C(\sigma', \omega')$ is the covariance matrix associated to σ' and ω' , $N(0, 1)$ is the standard univariate normal distribution, and $\mathbf{N}(\mathbf{0}, C)$ is the multivariate normal distribution with mean $\mathbf{0}$ and covariance matrix C . The subindex i in the standard normal distribution indicates that a new random number is generated anew for each component of the vector. The notation $N(0, 1)$ is used for indicating that the same random number is used for all the components. The parameters τ , η , and φ are set to $(2n)^{-1/2}$, $(4n)^{-1/4}$, and $5\pi/180$, respectively, as suggested in [14]. 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, each component of the vector solution \mathbf{x} is rounded to the nearest integer and used as actual parameter of the method under test. There is no limit in the input domain, thus allowing the ES to explore the whole solution space. This contrasts with other techniques such as genetic algorithm with binary representation that can only explore a limited region of the search space.

In the experimental section we have used two ESs: a (1+5) ES without crossover, that we call (1+5)-ES, and a (25+5)-ES with uniform crossover, called (25+5)-ES_c. In the latter algorithm, random selection was used.

4 Experimental Section

In order to study the correlations between the static measures and the coverage, we first need a large number of test programs. For the study to be well-founded, we require a lot 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, most 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 we want to analyze the correlations between some static measures of the programs and code 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 per condition, 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 conditions. The main features of the generated programs are:

- They deal with integer input parameters.
- Their conditions are joined by whichever logical operator.
- They are randomly generated.

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 [?] 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, in particular, the package `java.util`; and we computed the static measures on these files. In Table ?? we show a summary of this analysis. Next, we used the ranges of the most interesting values obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. This way, we generate programs that are realistic with respect to the static measures, making the following study meaningful. Finally, we generated a total of 3600 Java programs using our program generator and we applied our test case generator with (1+5)-ES and (25+5)-ES_c to all of them 5 times. We need 5 independent runs of each algorithm because they are stochastic algorithms: one single run is not meaningful; instead, several runs are required and the average and the standard deviation are used for comparison purposes. The experimental study requires a total of $3600 \cdot 5 \cdot 2 = 36000$ independent runs of the test case generator.

4.1 Results

After the execution of all the independent runs for the two algorithms in the 3600 programs, in this section we analyze the linear correlation between the static measures and the coverage.

Table 1. Range of values obtained in java.util library

Parameter	Minimum	Maximum
Number of Sentences	10	294
Nesting Degree	1	7
McCabe Cyclomatic Complexity	1	80

We use the Pearson correlation coefficient to study the degree of linear correlation between two variables. This coefficient is usually represented by r , and is computed with the following formula:

$$r = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{(n-1)S_x S_y}$$

where x_i and y_i are the values of the samples, n is the number of cases, S_x and S_y are the standard deviations of each variable.

First, we study the correlation between the number of sentences and the branch coverage. We obtain a correlation of 13.4% for these two variables using the (25+5)-ES_c algorithm and 18.8% with the (1+5)-ES algorithm². In Figure 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 in the coverage measure. The results obtained with ES_c are similar and we omit the corresponding graph.

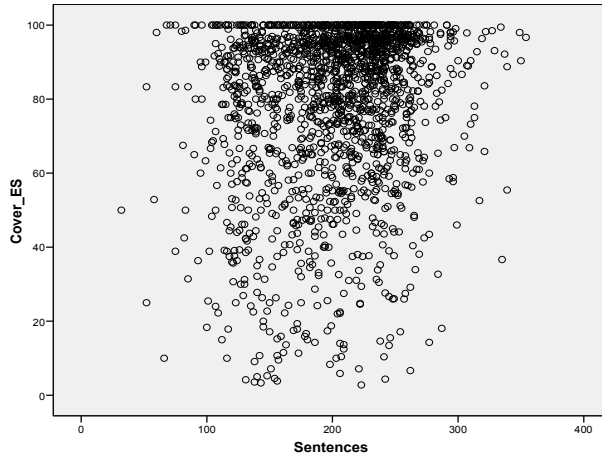


Fig. 4. Average branch coverage against the number of sentences for ES in all the programs

In second place, we study the correlation between the number of atomic conditions per condition and coverage. In Table 1 we show the average coverage obtained for all the programs with the same number of atomic conditions per condition when ES and ES_c are used. From the results we conclude that there is no linear correlation between these two variables. The minimum values for coverage are reached with 1 and 7 atomic conditions per condition. This could seem counterintuitive, but a large condition with a sequence of

² In order to save room, in the following we use ES_c and ES to refer to (25+5)-ES_c and (1+5)-ES, respectively

logical operators, can be easily satisfied due to OR operators. Otherwise, a short condition composed of AND operators can be more difficult to satisfy.

Table 2. Correlation between the number of atomic conditions per condition and average coverage for both algorithms

At. Conds.	ES_c	ES
1	83.59% _{21.69}	77.74% _{23.49}
2	82.85% _{20.33}	78.54% _{21.82}
3	85.15% _{19.82}	81.39% _{21.53}
4	88.04% _{16.42}	83.23% _{19.87}
5	85.06% _{19.19}	80.50% _{21.53}
6	84.16% _{19.58}	79.12% _{23.09}
7	81.24% _{21.18}	76.26% _{23.74}
r	-0.50 %	0.30 %

Now we analyze the influence on coverage of total number of conditions of a program. In Figure 5, we can observe that programs with a small number of conditions reach a higher coverage than the programs with a large number of conditions. This could be interpreted as large programs with many conditions are more difficult to test. If there are a lot of conditions, this generates a lot of different paths in the control flow graph, this fact is also weighted in the cyclomatic complexity. The correlation coefficient is -16.4% for ES and -21.6% for ES_c .

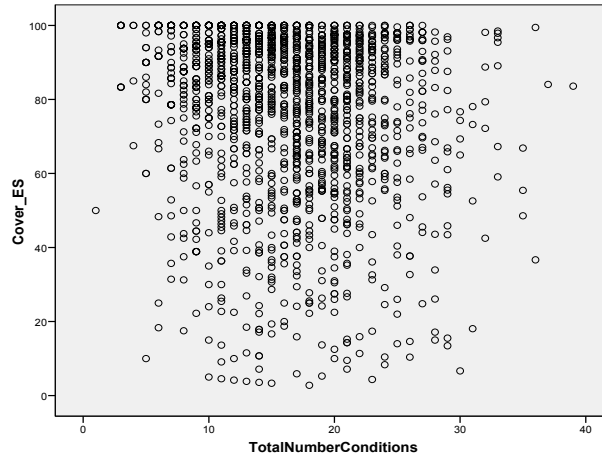


Fig. 5. Average branch coverage against the total number of conditions for ES in all the programs

Let us analyze the nesting degree. In Table 2, we summarize the coverage obtained 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 -45.7% and -47% for ES and ES_c respectively, what confirms the observations. As we said in Section 3.1, nested branches pose a great challenge for the search.

Table 3. Correlation between nesting degree and average coverage for both algorithms

Nesting	ES_c	ES
1	96.03% _{5.74}	93.56% _{8.15}
2	94.07% _{8.85}	89.97% _{11.29}
3	90.02% _{13.67}	85.29% _{16.71}
4	85.06% _{16.43}	80.09% _{19.18}
5	77.83% _{22.53}	72.02% _{23.90}
6	73.02% _{24.80}	67.47% _{24.66}
7	64.45% _{25.96}	58.41% _{27.16}
r	-47,00 %	-45,70 %

Finally, we study the correlation between the McCabe cyclomatic complexity and coverage. In Figures 6 and 7, we plot the average coverage against the cyclomatic complexity for ES and ES_c in all the programs. In general we can observe that there is no clear correlation between both parameters. The correlation coefficients are -4.8% and -8.6% for ES and ES_c , respectively. These values are very low, and confirms the observations: McCabe’s cyclomatic complexity and branch coverage are not correlated. 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. However, this is not true: McCabe’s cyclomatic complexity cannot be used as a measure of the difficulty to get an adequate test suite. We can go one step forward and try to explain this unexpected behaviour. Cyclomatic complexity is not correlated with branch coverage because complexity does not take into account the nesting degree, which is the parameter with a higher influence on branch coverage.

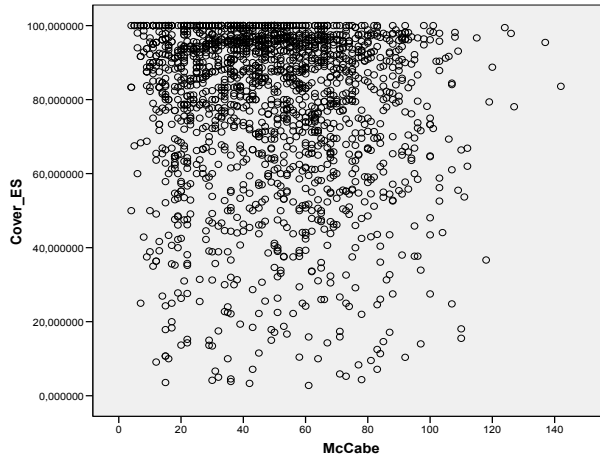


Fig. 6. Average branch coverage against the McCabe’s cyclomatic complexity for ES in all the programs

5 Conclusions

In this work we have analyzed the correlation between the branch coverage obtained using automatically generated test suites and five static measures: number of sentences, number

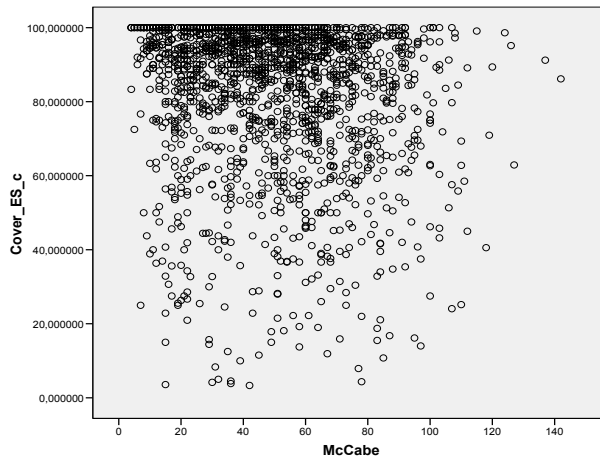


Fig. 7. Average branch coverage against the McCabe's cyclomatic complexity for ES_c in all the programs

of atomic conditions per condition, number of total conditions, nesting degree and McCabe's cyclomatic complexity. The results show that there is a small correlation between the branch coverage and the number of sentences, the number of conditions per conditions and the number of total conditions. On the other hand, the nesting degree is the measure that is more (inverse) correlated to the branch coverage: the higher the nesting degree the lower the coverage. Finally, there is no correlation between McCabe's cyclomatic complexity and branch coverage, that is, cyclomatic complexity does not reflect the difficulty of the automatic generation of test suites.

As future work we plan to advance in the analysis of static and dynamic measures of a program in order to propose a reliable measure of the difficulty of generating adequate test suites. In addition, we want to make an exhaustive study of the static measures in object-oriented programs, using a larger number of static measures. Finally, we would like to design new static measures able to reflect the real difficulty of automatic testing for specific test case generators.

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. T. Bäck, D. B. Fogel, and Z. Michalewicz. *Handbook of Evolutionary Computation*. Oxford University Press, New York NY, 1997.
2. André Baresel, David Wendell Binkley, Mark Harman, and Bogdan Korel. Evolutionary testing in the presence of loop-assigned flags: A testability transformation approach. In *International Symposium on Software Testing and Analysis (ISSTA 2004)*, pages 108–118, 2004.
3. Victor Basili and Barry Perricone. Software errors and complexity: an empirical investigation. *ACM commun*, 27(1):42–52, 1984.
4. Boris Beizer. *Software testing techniques*. Van Nostrand Reinhold Co., New York, NY, USA, 2nd edition, 1990.

5. 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.
6. Mark Dixon. An objective measure of code quality. *Technical Report*, February 2008.
7. T.M. Khoshgoftaar and J.C. Munson. Predicting software development errors using software complexity metrics. *IEEE Journal on Selected Areas in Communications*, 1990.
8. Phil McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, June 2004.
9. 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.
10. Webb Miller and David L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Software Eng.*, 2(3):223–226, 1976.
11. David Binkley Phil McMinn 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.
12. Paul Piwarski. A nesting level complexity measure. *SIGPLAN*, 17(9):44–50, 1982.
13. I. Rechenberg. *Evolutionsstrategie: Optimierung technischer Systeme nach Prinzipien der biologischen Evolution*. Fromman-Holzboog Verlag, Stuttgart, 1973.
14. G. Rudolph. *Evolutionary Computation 1. Basic Algorithms and Operators*, volume 1, chapter 9, Evolution Strategies, pages 81–88. IOP Publishing Lt, 2000.
15. 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.
16. Joachim Wegener, Andre Baresel, and Harmen Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, December 2001.
17. E.J. Weyuker. Evaluating software complexity measures. *IEEE Trans. Software Eng.*, 14(9):1357–1365, 1988.
18. 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.