Estimating Software Testing Complexity

Javier Ferrer*, Francisco Chicano, Enrique Alba

[a] *Departamento de Lenguajes y Ciencias de la Computación*
Universidad de Málaga, Spain

Abstract

Context: Complexity measures provide us some information about software artifacts. A measure of the difficulty of testing a piece of code could be very useful to take control about the test phase.
Objective: The aim in this paper is the definition of a new measure of the difficulty for a computer to generate test cases, we call it Branch Coverage Expectation (BCE). We also analyze the most common complexity measures and the most important features of a program. With this analysis we are trying to discover whether there exists a relationship between them and the code coverage of an automatically generated test suite.
Method: The definition of this measure is based on a Markov model of the program. This model is used not only to compute the BCE, but also to provide an estimation of the number of test cases needed to reach a given coverage level in the program. In order to check our proposal, we perform a theoretical validation and we carry out an empirical validation study using 2600 test programs.
Results: The results show that the previously existing measures are not so useful to estimate the difficulty of testing a program, because they are not highly correlated with the code coverage. Our proposed measure is much more correlated with the code coverage than the existing complexity measures.
Conclusion: The high correlation of our measure with the code coverage suggests that the BCE measure is a very promising way of measuring the difficulty to automatically test a program. Our proposed measure is useful for predicting the behaviour of an automatic test case generator.

Keywords: Evolutionary Testing, Complexity, Branch Coverage, Search Based Software Engineering, Evolutionary Algorithms, Testability

1. Introduction

Since the birth of Software Industry, there has been a high interest in measuring the effort in terms of time and cost required by a task. Nowadays, software applications are essential for Industry, thus software developers need to measure all sort of elements. Tom DeMarco stated [10]: “You can not control what you can not measure. Measurement is the prerequisite to management control”. The importance of metrics have also been highlighted by the famous physicist Lord Kelvin [33]: “When you can measure what you are speaking about, and express it in numbers, you know something about it; but when you cannot measure it, when you cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind: it may be the beginning of knowledge, but you have scarcely, in your thoughts, advanced to the state of science”. For these reasons, in this work we focus on complexity measures, which quantify the effort required to complete any kind of task.

First, it is needed to define what program complexity means. Basili [5] defines complexity as a measure of the resources used by a system while interacting with a piece of software to perform a given task. If the

*Corresponding author: Javier Ferrer +34952133303
Email addresses: ferrer@lcc.uma.es (Javier Ferrer), chicano@lcc.uma.es (Francisco Chicano), eat@lcc.uma.es (Enrique Alba)

interacting system is a computer, then complexity is defined by the execution time and storage required to perform the computation described by the program. If the interacting system is a programmer then complexity is defined by the difficulty of performing tasks such as coding, debugging, testing or modifying the software. There exist metrics introduced as all-purpose measures of software complexity, however these measures seem to be ineffective in order to measure the testing complexity [16]. The absence of a metric to properly measure the difficulty to test a piece of code encourage us to characterize the testing complexity.

Analyzing the testing complexity, it can be seen as the difficulty for a computer to create a test suite for finding errors in the developed code. Finding errors in early stages of the development is an important task that saves costs of the project. A detailed survey in the United States quantifies the high economic impacts of an inadequate software testing infrastructure [32]. Besides that, it is estimated that half the time spent on the software project development and more than half its cost, is devoted to testing the product [27]. To this end, in recent years researchers have attempted to predict fault-prone software modules using complexity metrics [36]. In addition, the overall experimental results show that complexity metrics are able to predict fault-prone source code [37].

In the last few years, there has been a renewed deal of interest in defining appropriate ways to measure the complexity of software [15, 26]. In most previous works they defined the testing complexity as the number of test cases required [35, 22]. Some works try to compute the lower bound [7] of the test cases required, and other works try to provide better understanding on the testing criterion used to generate those test cases [23]. However, they do not focus on the effort to generate these test cases. In a recent work, Nogueira focuses on the correlation between the complexity of the software under test and the complexity of the test cases [28], but the work did not propose any estimation measure.

We propose in this work a new complexity measure with the aim of helping the tester to find errors in the code. This measure will predict in a better way the behaviour of an automatic test data generator depending on the program under test. This original complexity measure, called "Branch Coverage Expectation", is the main contribution of this paper. The definition of the new measure lies on a Markov model that represents the program. Based on the model of a program, we can also provide an estimation of the number of random test cases that must be generated to obtain a concrete coverage. From these estimations, we can create a theoretical prediction of the evolution of the coverage depending on the number of generated test cases. This second contribution will help the testers to obtain some knowledge about the possible evolution of the testing phase.

The validation of the proposed measure is also addressed in this work. For the theoretical validation of the Branch Coverage Expectation we have used the validation framework proposed by Kitchenham et al. [19]. For the experimental validation we have used Evolutionary and Random Testing techniques, which are the most popular search algorithms for automatically generating test cases [1, 2, 12, 21], to compare our estimation with the real value obtained by several test data generators.

Finally, we also analyze software complexity measures at program level and we discuss a number of issues associated with these known measures. In addition, we have performed an experimental study of correlations with the aim of highlighting the existing relationships among some static measures. We are especially interested in the existing relationships between the static measures and the branch coverage. In this experimental study we have used two large groups of automatically generated programs to serve as a benchmark.

The rest of the paper is organized as follows. In the next section we present the measures that we later use in our experimental study. In Section 3 we explain the Markov model on which two of our main contributions in this paper are based: the definition of the BCE measure and the estimation of the number of test cases required to obtain a particular branch coverage. In Section 4 we explain the details of the automatic test data generator and the benchmark of programs that we use in the experimental section. Later, Section 5 describes the experimental study performed. Towards the end of the article, we describe the threats to the validity of our experimental evaluation in Section 6. Finally, Section 7 outlines some conclusions and future work.

2. Static 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 kind of measure made 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 [30], task effort [8], etc. We are interested here in measures made on source code pieces. We distinguish two kinds of measures: *dynamic*, which require the execution of the program, and *static*, which do not.

Some time ago, project managers began to worry about concepts like productivity and quality, then the lines of code (LOC) metric was proposed. Nowadays, the LOC metric is still the primary quantitative measure in use. An examination of the main metrics reveals that most of them confuse the complexity of a program with its size. The underlying idea of these measures are that a program will be much more difficult to work with than a second one if, for example, it is twice the size, has twice as many control paths leading through it, or contains twice as many logical decisions. Unfortunately, these various ways in which a program may increase in complexity tend to move in unison, making it difficult to identify the multiple dimensions of complexity.

In this section we present the measures used in this study. In a first group we select the main measures that we found in the literature:

- Lines of Code (LOC)
- Source Lines of Code ($SLOC$)
- Lines of Code Equivalent ($LOCE$)
- Total Number of Disjunctions ($TNDj$)
- Total Number of Conjunctions ($TNCj$)
- Total Number of Equalities (TNE)
- Total Number of Inequalities (TNI)
- Total Number of Decisions (TND)
- Number of Atomic Conditions per Decision (CpD)
- Nesting Degree (N)
- Halstead's Complexity (HD)
- McCabe's Cyclomatic Complexity (MC)

Let's have a look at the measures that are directly based on source lines of code (in C-based languages). The LOC measure is a count of the number of semicolons in a method, excluding those within comments and string literals. The $SLOC$ measure counts the source lines that contain executable statements, declarations, and/or compiler directives. However, comments, and blank lines are excluded. The $LOCE$ measure [31] is based on the idea of weighing each source line of code depending on how nested it is. The previous three measures based on the lines of code have several disadvantages:

- Depend on the print length
- Depend of the programmer's style for writing source code
- Depend on how many statements does one put in one line

We have analyzed several measures as the total number of disjunctions (OR operator) and conjunctions (AND operator) that appear in the source code, these operators join atomic conditions. The number of (in)equalities is the number of times that the operator (! =) == is found in atomic conditions of a program. The total number of decisions and the number of atomic conditions per decision do not require any comment. The nesting degree is the maximum number of control flow statements that are nested one inside another. In the following paragraphs we describe the McCabe's cyclomatic complexity and the Halstead complexity measures in detail.

Halstead complexity measures are software metrics [14] introduced by Maurice Howard Halstead in 1977. Halstead's Metrics are based on arguments derived from common sense, information theory and psychology. The metrics are based on four easily measurable properties of the program, which are:

- *n1* = the number of distinct operators
- *n2* = the number of distinct operands
- *N1* = the total number of operators
- *N2* = the total number of operands

From these values, six measures can be defined:

- Halstead Length (HL): $N = N1 + N2$
- Halstead Vocabulary (HV): $n = n1 + n2$
- Halstead Volume (HVL): $V = N * \log_2 n$
- Halstead Difficulty (HD): $HD = \frac{n_1}{2} * \frac{N_2}{n_2}$
- Halstead Level (HLV): $L = \frac{1}{HD}$
- Halstead Effort (HE): $E = HD * V$
- Halstead Time (HT): $T = \frac{E}{18}$
- Halstead Bugs (HB): $B = \frac{V}{3000}$

The most basic one is the Halstead Length, which simply totals the number of operators and operands. A small number of statements with a high Halstead Volume would suggest that the individual statements are quite complex. The Halstead Vocabulary gives a clue on the complexity of the statements. For example, it highlights if a small number of operators are used repeatedly (less complex) or if a large number of different operators are used, which will inevitably be more complex. The Halstead Volume uses the length and the vocabulary to give a measure of the amount of code written. The Halstead Difficulty uses a formula to assess the complexity based on the number of unique operators and operands. It suggests how difficult the code is to write and maintain. The Halstead Level is the inverse of the Halstead Difficulty: a low value means the program is prone to errors. The Halstead Effort attempts to estimate the amount of work that it would take to recode a particular method. The Halstead Time is the time to implement or understand a program and it is proportional to the effort. The experiments were used for calibrating this quantity but nowadays it is not true that dividing the effort by 18 gives an approximation for the time in seconds. The Halstead Bugs attempts to estimate the number of bugs that exist in a particular piece of code.

McCabe's cyclomatic complexity is a complexity measure 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 [25]. 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 (basic blocks), and a directed edge connects two nodes if the second group of sentences might be executed immediately after the first one. Cyclomatic complexity may also be applied to individual functions, modules, methods or classes within a program, and is formally defined as follows:

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

where Ed is the number of edges of the graph, Nd is the number of nodes of the graph and P is the number of connected components.

The correlation between the cyclomatic complexity and the number of software faults has been studied in some research articles [6, 18]. Most such studies find a strong positive correlation between the cyclomatic complexity and the errors: the higher the complexity the larger the number of faults. For example, a 2008 study by metric-monitoring software supplier Energy [11] analyzed classes of open-source Java applications and divided them into two sets based on how common mistakes were found in them. They found a strong correlation between the 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 on it". For that reason, we make a close link between complexity and difficulty of discovering errors. Software complexity metrics developed by Halstead and McCabe are related to the difficulty programmers experience in locating errors in code [9]. They can be used in providing feedback to programmers about the complexity of the code they have developed and to managers about the resources that will be necessary to maintain particular sections of code.

Since McCabe proposed the cyclomatic complexity, it has received several criticisms. Weyuker [34] 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. Piwarski [29] noticed that cyclomatic complexity is the same for N nested `if` statements and N sequential `if` statements. Moreover, we find the same weaknesses in the group of Halstead's metrics. No notice is made for the nesting degree, which may increase the effort required by the program severely. The solution of both McCabe's and Halstead's weakness is a factor to consider that a nested statement is more complex. For example, we have also studied the LOCE measure that takes into account whether a statement is nested or not.

The proposed existing measures of decision complexity tend to be based upon a graph theoretical analysis of a program control structure like McCabe's complexity. Such measures are meaningful at the program and subprogram level, but metrics computed at those levels will depend on program or subprogram size. However, the values of these metrics primarily depend upon the number of decision points within a program. This suggests that we can compute a size-independent measure of decision complexity by measuring the density of decisions within a program. In addition we have considered making the LOCE measure size-independent. The resulting expression takes into account the nesting degree and the density of the sentences. Following this assumption, we consider in this paper two measures derived from some of the first group:

- Density of Decisions (DD) = TND/LOC.
- Density of LOCE (DLOCE) = $LOCE/LOC$.

Finally, we present the dynamic measure used in the study: Branch Coverage. Before defining a coverage measure, it is necessary 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 (sentences) that are executed. In this work we use *Branch Coverage*, which is the percentage of branches of the program that are traversed. This coverage measure is used in most of the related articles in the literature. We formally define the *Branch Coverage* as follows: Let P be a program, we denote with B_P the set of branches of the program and with $BranchExec_P(C)$ the set of branches covered in P due to the execution of a given test suite, C. We define the branch coverage of the test suite C, $BrCov_P(C)$, as the ratio between the traversed branches in the executions of the program P

with the test suite C and the number of branches of the program, i.e.,

$$BrCov_P(C) = \frac{|BranchExec_P(C)|}{|B_P|}. \tag{2}$$

The adequacy criterion of branch coverage states that a test suite C for a program P is "adequate" when $BrCov_p(C) = 1$.

3. New complexity measure: Branch Coverage Expectation

This section is aimed at presenting a new complexity measure that might help testers to estimate the difficulty of testing a piece of code. The definition of the new measure lies on a Markov chain that represents the program. In this section we briefly explain the characteristics of a Markov chain and the way we generate a model of a given program. The Markov model of the program can be used not only to compute the BCE, but also to estimate the number of random test cases that must be generated to achieve a concrete value of branch coverage. We first introduce the required concepts of Markov chains [20].

3.1. Markov Chain

A *first order Markov chain* is a random sequence of states X_t where each state depends only on the previous one. That is, $P(X_{t+1} = j|X_k; -\infty < k \leq t) = P(X_{t+1} = j|X_t)$ for all $t \in \mathbb{N}$. We consider here that the set of possible states is finite and, without loss of generality, we label the states using elements of the set $[n] = \{1, ..., n\}$. The conditional probabilities of a first order Markov chain $P(X_{t+1} = j|X_t = i) = P_{ij}(t)$ are called *one-step transition probabilities* and the matrix $\mathbf{P}(t) = [P_{ij}(t)]$ is the so-called *transition probability matrix*. We will assume here that these probabilities do not depend on the step t, and thus, $P_{ij}(t) = P_{ij}$ for all t. The Markov chains fulfilling this property are called *homogeneous*. Two properties of the transition probability matrices are:

$$P_{ij} \geq 0, \tag{3}$$

$$\sum_{j=1}^{n} P_{ij} = 1. \tag{4}$$

Matrices fulfilling the above equations are called *stochastic*. Let us denote with the column vector $\mathbf{q}(t)$ the probability distribution of the states at step t. The component $q_i(t)$ is the probability of having state i at step t. A state which is reached infinitely often in a finite Markov chain is called *positive-recurrent*. If every state in a Markov chain can be reached from every other state, then we say that the Markov chain is *irreducible*. For irreducible Markov chains having only positive-recurrent states the probability distribution of the states $\mathbf{q}(t)$ tends to a given probability distribution π as the time tends to infinite. This probability distribution π is called the *stationary distribution* and can be computed solving the following linear equations:

$$\pi^T P = \pi^T, \tag{5}$$

$$\pi^T \mathbf{1} = 1. \tag{6}$$

3.2. Definition of BCE

In our case the Markov model is built from the Control Flow Graph (CFG) of the program, where the states of the Markov chain are the basic blocks of the program. A basic block (BB) is a portion of the code that is executed sequentially with no interruption. It has one entry point and one exit point, meaning that only the last instruction can be a jump. Whenever the first instruction in a basic block is executed, the rest of the instructions are necessarily executed exactly once, in order. In order to completely characterize a Markov chain we must assign a value to the edges among vertices. The transition probabilities of all branches are computed according to the logical expressions that appear in each condition. We recursively define this probability as follows:

$$P(c1\&\&c2) = P(c1) * P(c2), \tag{7}$$
$$P(c1\,|\,|\,c2) = P(c1) + P(c2) - P(c1) * P(c2), \tag{8}$$
$$P(\neg c1) = 1 - P(c1), \tag{9}$$
$$P(a < b) = \frac{1}{2}, \tag{10}$$
$$P(a \leq b) = \frac{1}{2}, \tag{11}$$
$$P(a > b) = \frac{1}{2}, \tag{12}$$
$$P(a \geq b) = \frac{1}{2}, \tag{13}$$
$$P(a == b) = q, \tag{14}$$
$$P(a! = b) = 1 - q, \tag{15}$$

where $c1$ and $c2$ are conditions.

We establish a 1/2 probability when the operators are ordering relational operators ($<, \leq, >, \geq$). Despite that the actual probability in a random situation is not always 1/2, we have selected the value with the lowest error rate. In the case of equalities and inequalities the probabilities are q and $1 - q$, respectively, where q is a parameter of the measure and its value should be adjusted based on the experience. Satisfying an equality is, in general, a hard task and, thus, q should be close to zero. This parameter could be highly dependent on the data dependencies of the program. The quality of the complexity measure depends on a good election for q. We delay to future work the thorough analysis of this parameter. Based on a previous phase for setting parameters, we use $q = 1/16$ for the experimental analysis.

Then, once we have the CFG completed with the transition probabilities, the generation of the transition matrix is automatic. This matrix relates the states and the probability to move from one to another. We assume, without loss of generality, that there is only one entry and exit basic block in the code. Then, in order to obtain a positive-recurrent irreducible Markov chain we add a fictional link from the exit to the entry basic block (labelled as BB_1) having probability 1. We then compute the stationary probability π and the frequency of appearance of each basic block in one single execution of the program ($E[BB_i]$). The stationary probability of a basic block is the probability of appearance in infinite program executions starting in any state. On the other hand, the frequency of appearance of a basic block is the mathematical expectation of traversing the basic block in one single execution and is computed as:

$$E[BB_i] = \frac{\pi_i}{\pi_1}, \tag{16}$$

where π_1 is the stationary probability of the entry basic block, BB_1.

Thus, the expectation of traversing a branch (i, j) is computed from the frequency of appearance of the previous basic block and the probability to take the concrete branch from the previous basic block as:

$$E[BB_i, BB_j] = E[BB_i] * P_{ij} \tag{17}$$

Finally, we define the *Branch Coverage Expectation* (BCE) as the average of the values $E[BB_i, BB_j]$ with a value lower than 1/2. If a program has a low value of BCE then a random test case generator is supposed to require a large number of test cases to obtain full branch coverage. The BCE is bounded in the interval (0, 1/2]. Formally, let A be the set of edges with $E[BB_i, BB_j] < 1/2$:

$$A = \{(i, j) | E[BB_i, BB_j] < \frac{1}{2}\}. \tag{18}$$

Then, the BCE is defined as:

$$BCE = \frac{1}{|A|} \sum_{(i,j)\in A} E[BB_i, BB_j]. \tag{19}$$

In the experimental section we analyze the new complexity measure over program artifacts, nevertheless we illustrate here its computation based on the piece of code shown in Figure 1. First, we compute the Control Flow Graph (CFG) of this piece of code, which can be seen in Figure 2. This CFG is composed of BBs and transitions among the BBs. Interpreted as a Markov chain, the basic blocks are the states, and the transitions are defined by the probabilities to move from one basic block to another. These probabilities depend on the condition associated to a concrete branch. For example, to move from BB_1 to BB_2 in our example, the condition $(x < 0)||(y < 2)$ must be true, then according to equations (2) to (10) the probability of this transition is:

$$P((x<0)||(y<2)) = P(x<0) + P(y<2) - P(x<0) * P(y<2) = \frac{1}{2} + \frac{1}{2} - \frac{1}{2} * \frac{1}{2} = \frac{3}{4} = 0.75.$$

```
/* BB1 */
if (x < 0) || (y < 2)
{
    /* BB2 */
    y=5;
}
else
{
    /* BB3 */
    x=y-3;
    while (y > 5) || (x > 5)
    {
           /* BB4 */
           y=x-5;
    }
    /* BB5 */
    x=x-3;
}
/* BB6 */
```

Figure 1: A piece of code to illustrate the computation of Branch Coverage Expectation

Once we have computed all the transition probabilities, we build the transition matrix that represents the Markov chain.

$$P = \begin{pmatrix} 0.0 & 0.75 & 0.25 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1 \\ 0.0 & 0.0 & 0.0 & 0.75 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.75 & 0.25 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 & 0.0 & 1 \\ 1 & 0.0 & 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix}$$

We can now compute the stationary probabilities π and the frequency of appearance $E[BB_i]$ of the basic blocks in one execution of the program (see Table 1). It is sure that the control flow of the program traverses

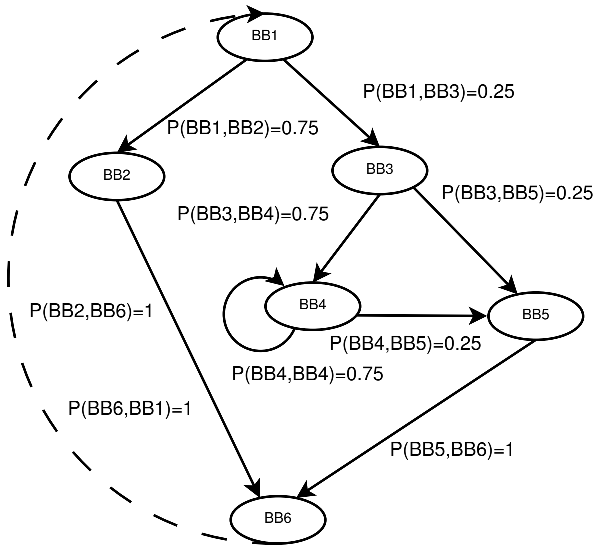

Figure 2: The CFG and the probabilities used to build a Markov Chain of the piece of code of Figure 1

exactly once the BB_1 and BB_6 in one run. In this way, the start and the end of the program always have a $E[BB_i] = 1$. An example of the computation of the mathematical expectation is:

$E(BB_2) = \frac{\pi_2}{\pi_1} = \frac{0.1875}{0.2500} = 0.75.$

Table 1: Stationary probabilities and the frequency of appearance of the basic blocks of the piece of code shown above.

	Stationary Probabilities π_i	Frequency of Appearance $E[BB_i]$
BB1	0.2500	1.00
BB2	0.1875	0.75
BB3	0.0625	0.25
BB4	0.1875	0.75
BB5	0.0625	0.25
BB6	0.2500	1.00

The stationary probability and the frequency of appearance of the BBs in a single execution of the piece of code can be seen in Table 1. Now, we are able to compute the probability of appearance of a branch in one single run. For example the expectation of traversing the branch $BB_3 - BB_4$ is:

$E[BB_3, BB_4] = E(BB_3) * P_{34} = \frac{1}{4} * \frac{3}{4} = \frac{3}{16} = 0.1875.$

In Figure 3 we show the mathematical expectations of traversing all the branches of the CFG of our example in one single execution. So, finally we can compute the BCE by averaging the expectations of traversing the branches which have a value lower than 1/2. We have excluded those values equals to 1/2 because both branches have the same value. In case all branches have the expectation of 1/2, then the BCE is 1/2. In addition, a program with a Branch Coverage Expectation value of 1/2 would be the easiest one to be tested. In this example the value of BCE is :

BCE $= \frac{E[BB_1,BB_3]+E[BB_3,BB_4]+E[BB_3,BB_5]+E[BB_4,BB_5]+E[BB_5,BB_6]}{5} = \frac{\frac{1}{4}+\frac{3}{16}+\frac{1}{16}+\frac{3}{16}+\frac{1}{4}}{5} = \frac{3}{16} = 0.1875.$

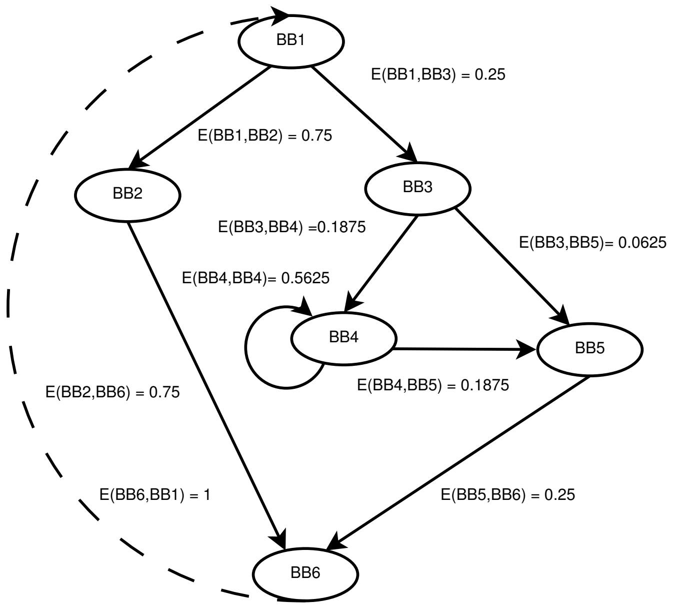

Figure 3: The CFG and the expectations of traversing each branch in the piece of code of Figure 1

Based on the model of a program, we can also provide an estimation of the number of random test cases that must be generated to obtain a concrete coverage. Following with the example, executing the branch (BB_3, BB_5) is more difficult according to the expectations than the others. The inverse of this expectation, is the expected number of random test cases that must be generated to execute the branch. In this example, the number of expected test cases needed to traverse the branch between BB3 and BB5 is:

Number of test cases (BB3,BB5)=$\frac{1}{\frac{1}{16}} = 16$.

From these estimations, we can create a theoretical prediction of the evolution of the coverage depending on the number of generated test cases. This contribution could help the testers to obtain some knowledge about the possible evolution of the testing phase. In the experimental section (Section 5.3) we compare our theoretical prediction with the results obtained by a test case generator.

3.3. Validation of the Branch Coverage Expectation

Software applications are essential for Industry and software measurement is a key factor in understanding and controlling software development practices. Consequently, measures must represent accurately those attributes which they quantify. Thus, validation is critical when a new measure is introduced. The software measurement validation implies two basic methods, theoretical and empirical validation. Theoretical methods allow us to say that a measure is valid with respect to certain criteria, meanwhile empirical methods only provide evidence of validity or invalidity. In the experimental section we obtain evidences of the validity of the proposed measure, but first, in this section we focus on theoretical validity using the framework proposed by Kitchenham et al. [19]. The requirements defined when validating a measure are attribute validity, unit validity, instrument validity, and protocol validity:

- **Attribute validity**: Attributes are the properties that an entity possesses. For a given attribute, there is a relationship of interest in the empirical world that we want to capture formally in the mathematical world. The attribute we consider for our measure is the testing complexity of a piece of software. Two measures are defined to estimate testing complexity: “branch coverage expectation”(BCE) and “number of expected test cases”(related to the inverse of BCE). And both are estimated by capturing transition probabilities at each edge among vertices (decisions and conditions of branches). The measure is able to satisfy the proposed criteria. There could be two different programs for which the measure results in different values. Our measure also obey the *Representation Condition.* The BCE value of two programs is the same, when they are the same except they have different labels. So, different programs can have the same BCE value.

- **Unit validity**: A measure maps an empirical attribute to the formal, mathematical world. A measurement unit determines how we measure an attribute. We define the unit of BCE by reference to a wider theory explained in the previous subsection 3.1 (Markov Chain). We must highlight that the inverse of the BCE measure is related to the number of test cases needed to achieve full coverage by a random test case generator. Then, the measurement unit used by the BCE is $number\ of\ test\ cases^{-1}$. In conclusion, we use an alternative unit that is valid because is an admissible transformation from an original unit (number of test cases).

- **Instrument validity**: The instrument model as it defines how to capture the data, is also theory based. The validity again depends on the validity of the underlying theory. It can be defined by reference to properties of the control flow graph. Our instrument model is valid because the underlying theory-based model is valid. For example, we can use a thermometer to measure temperature, or a software program to count the number of lines of code in a program.

- **Protocol validity**: Measurement protocols let us measure a specific attribute on a specific entity consistently and repeatedly. We can measure a specific attribute of a program consistently, repeatable, and the measurement is independent of the measurer. Our measurement protocols are unambiguous, self-consistent, and prevent problems such as double counting. The same measurement could be done with a different measurer obtaining the same results. A protocol that does not violate these criteria is usually validated by peer acceptance rather than logical or empirical studies.

Empirical validation of our proposed measure is required, so we are introducing in the next section a tool for generating test data, a tool for generating synthetic programs, and a benchmark of real and synthetic programs, which will help us to compare our estimation of BCE with the result of branch coverage obtained by the execution of our testing tool. Branch Coverage is the dynamic measure used in this work to measure the difficulty of testing a program. The test data generator goal is generating a test suite that covers all the source code, with the aim of helping the tester to find errors in the code.

Finally, we claim that our measure is valid because we are unable to invalidate it using the Kitchenham et al. framework.

4. Experimental Benchmark

In this section we outline the test case generation tool [13] used for empirical validation of our proposed measure. We also explain a novel automatic program generator. We have developed this generator to create the variety of programs that we use in the experimental section. Then, we outline the main characteristics of the used benchmark.

4.1. Evolutionary Test Case Generator

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

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. After the optimization algorithm stops, the main loop starts again and the test case generator selects a different branch. This scheme is repeated until total branch coverage is obtained or a maximum number of consecutive failures of the optimization algorithm is reached.

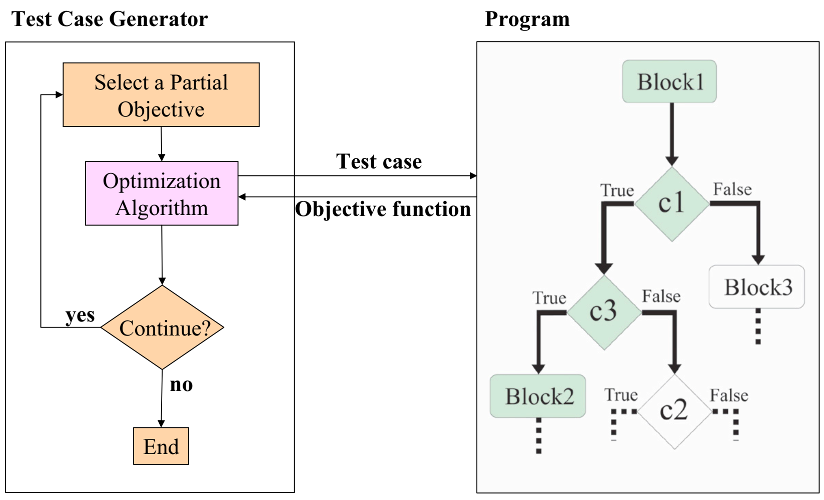

Figure 4: The test case generation process

When this happens the test data generator exits the main loop and returns the sets of test cases associated to all the branches.

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. In this work we use two EAs as the optimization algorithm of the test case generator: an evolutionary strategy (ES) and a genetic algorithm (GA). We show in Table 2 a summary of the parameters used by the two EAs in the experimental section.

Table 2: Parameters of the two EAs used in the experimental section.

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

4.2. Automatic Program Generator

We have designed an automatic program generator able to generate programs with values for the static measures that are similar to the ones of the real-world software, but the generated programs do not solve any concrete problem. Our program generator is able to create programs for which total branch coverage is possible. We propose this generator with the aim of generating a big benchmark of programs with certain characteristics chosen by the user.

In a first approximation we could create a program using a representation based on a general tree and a table of variables. The tree stores the sentences that are generated and the table of variables stores basic information about the variables declared and their possible use. With these structures, we are able to generate programs, but we can not ensure that all the branches of the generated programs are reachable. The unreachability of all the branches is a quite common feature of real-world programs, so we could stop the design for the generator at this stage. However, another objective of the program generator is to be able of creating programs that can be used to compare the performance of different algorithms, programs for which total coverage is reachable are desirable. With this goal in mind we introduce logic predicates in the program generation process.

The program generator is parameterizable, the user can set several parameters of the program under construction (PUC). Thus, we can assign through several probability distributions the number of sentences of the PUC, the number of variables, the maximum number of atomic conditions in a decision, and the maximum nesting degree by setting these parameters. The user can define the structure of the PUC and, thus, its complexity. Another parameter the user can tune is the percentage of control structures or assignment sentences that will appear in the code. By tuning this parameter the program will contain the desired density of decisions.

Once the parameters are set, the program generator builds the general scheme of the PUC. It stores in the used data structure (a general tree) the program structure, the visibility, the modifiers of the program, and creates a main method where the local variables are first declared. Then, the program is built through a sequence of basic blocks of sentences where, according to a probability, the program generator decides which sentence will be added to the program. The creation of the entire program is done in a recursive way. The user can decide whether all the branches of the generated program are reachable (using logic predicates).

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

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

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

Once these predicates are checked, the last control statement is correct and new sentences are generated for the two branches and the predicates are computed inside the branches in the same way. After the control structure is completed, the last predicates of the two branches are combined using the OR operator and the result is the predicate after the control structure. In Figure 5 we illustrate the previous explanation with one example.

At a certain point of the program's execution our current predicate (CP_1) is $x \leq 3$. The new sentence is an `if` statement with an associated decision $x < 0$. Then, the program generator creates two new predicates. The first one (CP_2) is $CP_1 \wedge x < 0 \equiv x < 0$. The second one is the AND operation between CP_1 and the negation of $x < 0$, which is $x \geq 0$. The resulting expression is $CP_4 \equiv 0 \leq x \leq 3$. Next, the program generator modifies the predicates according to the assignment sentences. Finally, the resulting expression after the execution of the `if` statement is $CP_6 \equiv x < 0 \wedge y = 5 \vee -3 \leq x \leq 0$. This expression is the OR operation between CP_3 (true branch) and CP_5 (false branch). Those values that satisfy the logic predicate may participate in the following generated sentences.

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

Figure 5: Illustration of the predicates transformation

4.3. Benchmark of Test Programs

The program generator can create programs having the same value for the static measures, as well as programs having different values for the measures. The programs we generated for this paper can be separated in two groups. One group is characterized by being 100% coverable (called 100%CP), thus all the branches are reachable. The main advantage of these programs is that algorithms can be tested and analyzed on fair way. This kind of programs is not easy to find in the literature. On the other hand, the other group of programs do not guarantee a 100% branch coverage, called ¬100%CP, this fact makes them similar to the real world programs.

The methodology applied for the program generation was the following. First, we analyzed a set of Java source files from the JDK 1.5 (java.util.*, java.io.*, java.sql.*, etc.) and we computed the static measures on these files. Next, we used the ranges of the most interesting measures, obtained in this previous analysis as a guide to generate Java source files having values in the same range for the static measures. These values are realistic with respect to the static measures, making the following study meaningful. 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 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 and they are randomly generated. This way, we generated programs with the values in the ranges shown in Table 3.

Table 3: Range of values for some static measures from the two benchmarks of programs.

	100% CP	$\neg 100\% CP$
SLOC	33-150	33-235
Nesting	1-4	1-7
Conditions	1-4	1-7
Decisions	3-50	1-37
McCabe	5-125	3-127

Finally, we generated a total of 2600 (800 in 100%CP and 1800 in ¬100%CP) Java programs using our program generator. With the aim of studying the BCE, we applied our test case generator using an ES and a GA as optimization algorithms and a random test case generator (RND). The test case generators proceed by generating test data until total coverage is obtained or a maximum of 150,000 test cases are generated. Since we are working with stochastic algorithms, we perform in all the cases 30 independent runs of the algorithms to obtain a very stable average of the branch coverage. The experimental study requires a total of $2600 \times 30 \times 3 = 234,000$ independent runs of the test case generators. We need to perform a statistical analysis of the obtained results to compare them with a certain level of confidence. The statistical test that we have carried out is the non-parametric Kruskal-Wallis test used to compare the average of the algorithms. We always consider in this work a confidence level of 95% (i.e., p-value under 0.05) in the statistical tests, which means that the differences are unlikely to have occurred by chance with a confidence of 95%.

4.3.1. Real Programs

In order to improve the interest of our work we propose an additional benchmark of real programs. It is composed of 10 real programs extracted from the literature [3, 17, 24]. Some of them have been extracted from the book *C Numerical Recipes*, available on-line at `http://www.nr.com/`. They deal with real and integer input values and some of them also contain loops. The programs are listed in Table 4, where we inform on the maximum nesting degree, the lines of code (LOC), the number of branches, and the number and type of input arguments.

Table 4: Characteristics of the Real Programs.

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

5. Experimental Results

In this section we describe the experimental analysis performed and we interpret the relationship of the studied measures. We divide the main study in four subsections. In the first one, we analyze the static measures in order to reveal their correlations. In the second subsection, we highlight the existing relationship between the static measures and the code coverage. We analyze which of them are more appropriate for estimating the difficulty for a computer to generate an adequate test suite. In the third subsection, we use the Markov model of programs to predict the relationship between coverage and the number of required test cases. We analyze if our theoretical prediction is similar to a real execution of an automatic test case generator. Finally, we perform the study on the real world programs.

For computing correlations among measures we use the Spearman's correlation coefficient ρ. This coefficient takes into account the rank of the values of the samples instead of the samples themselves.

5.1. Analysis of the Correlation Between the Static Measures

In this section we analyze the existing relationship among the static measures of the generated programs. With this previous study we want to clarify the possible similarities and differences of the analyzed static measures in this paper.

In this study there are three different measures that try to rate the complexity of a program: McCabe's complexity, the Halstead Difficulty and LOCE. In Tables 5 and 6 we show a comparison of correlation between these measures, the nesting degree and the group of measures derived from some of the first group (density of decisions and density of LOCE). We show these measures because we expect that they are highly correlated with the branch coverage. In addition, we include the BCE measure in order to compare it with the other studied measures.

Table 5: Correlation coefficient of the most interesting static measures in the 100%CP benchmark. We highlight the highest value per row.

	100%CP						
	MC	HD	LOCE	N	DD	DLOCE	BCE
MC	•	0.796	0.965	0.266	0.519	0.408	0.025
HD	0.796	•	0.786	−0.108	0.052	−0.035	0.284
LOCE	0.965	0.786	•	0.344	0.515	0.474	-0.038
N	0.266	−0.108	0.344	•	0.765	0.877	-0.540
DD	0.519	0.052	0.515	0.765	•	0.912	-0.377
DLOCE	0.408	−0.035	0.474	0.877	0.912	•	-0.485
BCE	0.025	0.284	−0.038	−0.540	−0.377	−0.485	•

Table 6: Correlation coefficient of the most interesting static measures in the ¬100%CP benchmark. We highlight the highest value per row.

	¬100%CP						
	MC	H	LOCE	N	DD	DLOCE	BCE
MC	•	0.698	0.571	0.257	0.432	0.351	-0.142
HD	0.698	•	0.359	0.062	0.023	0.014	0.051
LOCE	0.571	0.359	•	0.692	0.590	0.833	-0.461
N	0.257	0.062	0.692	•	0.708	0.870	-0.575
DD	0.432	0.023	0.590	0.708	•	0.774	-0.426
DLOCE	0.351	0.014	0.833	0.870	0.774	•	−0.556
BCE	−0.142	0.051	−0.461	−0.575	−0.426	−0.556	•

As we can see, the three measures that rate the complexity (MC, HD and LOCE) are highly correlated among them in the 100%CP benchmark and less correlated in the ¬100%CP one. This gives us a clue about the similarities that the complexity measures have among them. On the other hand, the nesting degree is lowly correlated with McCabe's complexity and Halstead Difficulty. In the case of LOCE, the correlations with the nesting degree are 0.344 in 100%CP and 0.692 in ¬100%CP. This was expected because the LOCE measure weighs the nested statements. In addition, we can remark that the Halstead Difficulty does not correlate with the density of decisions in 100%CP and ¬100%CP (0.052 and 0.023, respectively).

We can also observe that the static measures that are highly correlated in one benchmark are highly correlated in the other one too. This is the case of the relationship between nesting, density of decisions and density of LOCE that must be emphasized. All of them are highly correlated, being the correlation between nesting degree and density of LOCE 0.877 in 100%CP and 0.870 in ¬100%CP, nesting degree and density of decisions 0.765 in 100%CP and 0.708 in ¬100%CP and density of decisions and density of LOCE 0.912 in 100%CP and 0.774 in ¬100%CP.

Once we have analyzed the other static measures, we report the correlation coefficients of our proposal and the most important static measures studied in this paper. The nesting degree must be emphasized because it is the most correlated static measure with the BCE, -0.540 in 100%CP and -0.575 in ¬100%CP, what means that the nesting degree is the most similar measure. In addition, we can see that our proposal is not correlated with McCabe's and Halstead's complexities.

This analysis of the three rates of complexity is not complete if we do not highlight the static measures that are more correlated with these complexity measures (all correlation coefficients can be seen in Tables A.10 and A.11). McCabe's complexity is highly correlated with the number of conjunctions, disjunctions, equalities and inequalities (0.934, 0.925, 0.829 and 0.811 in 100%CP and 0.937, 0.936, 0.803 and 0.827 in ¬100%CP, respectively). These high coefficients were expected because McCabe's complexity depends on the CFG of the program. Halstead Difficulty is highly correlated with the other Halstead measures. In addition, it is highly correlated with McCabe's complexity (0.796 in 100%CP and 0.698 in ¬100%CP). LOCE is highly correlated with the total number of decisions and SLOC (0.976 and 0.974) in 100%CP and in ¬100%CP (0.814 and 0.717). These results were expected because SLOC and LOCE are very similar measures and the total number of decisions gives us an idea of the length of the code. The Halstead Length is highly correlated with LOC and SLOC, with a minimum value of correlation of 0.906. Moreover, the other Halstead measures are highly correlated too, except Halstead Difficulty and Level. This indicates that several Halstead measures are similar to a simple count of lines of code.

In this subsection we have provided an overview of static measures that are part of our study. Now, we know the measures that are similar and those that are different. In the next section we show the measures that are more correlated with the branch coverage, which is the way we measure the difficulty of testing a program.

5.2. Correlation Between Coverage and Static Measures

In the previous section we showed the basic relationship among the static measures, in this section we include the branch coverage in the study. The existing correlations between the branch coverage and the static measures studied give us an idea of which static measures are useful to determine *a priori* the complexity of the automatic test data generation task. In this study we have applied three different test case generators, two based on evolutionary techniques (ES, GA) and one based on random testing (RND).

Table 7: Relationship between the most important static measures and the average branch coverage for all the algorithms. We highlight the high value of correlation for each algorithm and benchmark.

	100%CP			¬100%CP		
	ES	GA	RND	ES	GA	RND
MC	-0.150	-0.226	-0.074	-0.177	-0.168	-0.173
HD	0.070	-0.101	0.077	0.069	0.067	0.079
LOCE	-0.186	-0.251	-0.133	-0.461	-0.452	-0.476
N	-0.543	-0.381	-0.434	-0.563	-0.554	-0.589
DD	-0.439	-0.304	-0.311	-0.476	-0.473	-0.497
DLOCE	-0.504	-0.345	-0.397	-0.577	-0.564	-0.602
BCE	0.510	0.375	0.534	0.714	0.698	0.732

The first question we should answer is if there exists a link between the coverage and the traditional measures of code complexity: McCabe's, Halstead's, and LOCE. In Table 7 we show the correlation coefficients for the most important static measures and the branch coverage obtained with three automatic test data generators. The correlations between Halstead's Difficulty and the coverage are very low, so the answer is no in this case. The correlation coefficients of McCabe's complexity are higher than Halstead Difficulty but too low. This result was expected because, as we showed in the previous section, Halstead Difficulty is highly correlated with McCabe's complexity. Finally, the correlation coefficients of LOCE indicate that it is more correlated with the branch coverage because this measure takes into account the nested statements. After analyzing these results, we realise that the traditional complexity measures (MC, HD, and LOCE) are not useful to measure the difficulty of testing a program.

In the second group of measures, there exist higher correlations with branch coverage. The nesting degree is the static measure with the highest correlation coefficient with branch coverage in the 100%CP benchmark for the evolutionary test case generators. On the other hand, DLOCE is more correlated than the nesting degree in the ¬100%CP benchmark. Despite that the total number of decisions is not correlated with coverage, as can be seen in Tables A.10 and A.11, the density of decisions correlates with the obtained coverage, as we show in Table 7. Moreover, the density of decisions is also more correlated than the traditional complexity measures. In Figure 6 the trend indicates that the programs with a high density of decisions are more difficult to test because a lower coverage is obtained.

After analyzing the LOCE measure, we supposed that if the influence of the LOC were removed by dividing LOCE by LOC, it could be obtained a measure with a high influence of the nested level (DLOCE) (recall that that the LOCE measure weighs those nested statements). As the nesting degree is highly correlated with the branch coverage, the DLOCE would have high correlation too. After doing the correlation

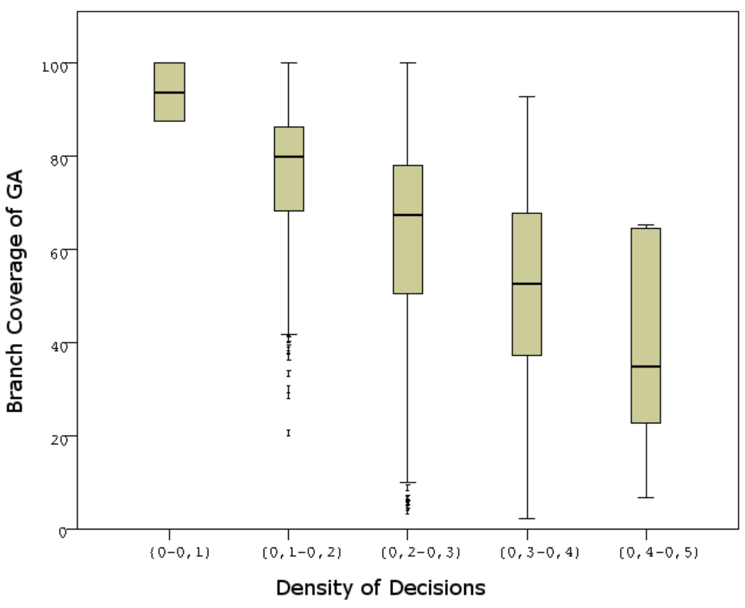

Figure 6: Boxplots showing the branch coverage against the Density of Decisions for GA in ¬100%CP

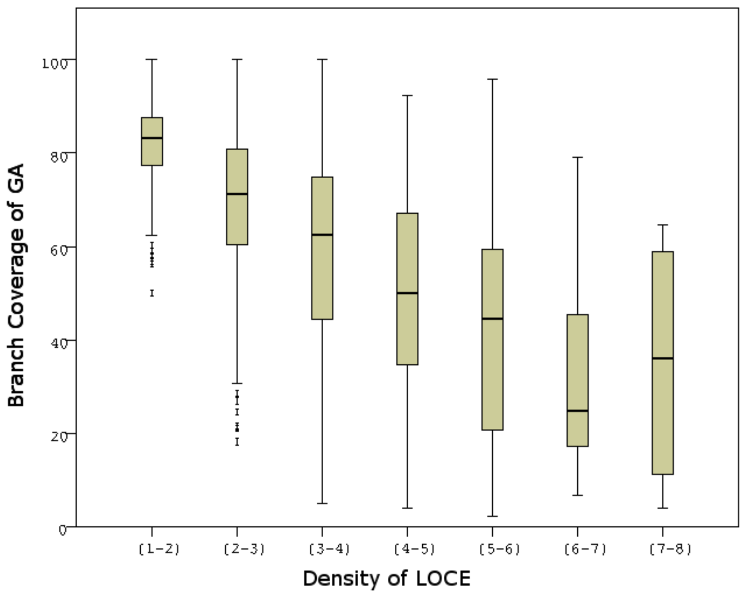

Figure 7: Boxplots showing the branch coverage against the DLOCE for GA in ¬100%CP

test, our expectations were true, as one can see in Table 7. These results are similar to the results obtained with the nesting degree. In the case of the benchmark ¬100%CP, DLOCE has more influence than N in general. In Figure 7, we can see that the coverage clearly increases as the DLOCE decreases with the exception of the programs with DLOCE between 7 and 8.

Let us analyze the nesting degree. In Table 8, we summarize the obtained coverage in programs with different nesting degree in the two benchmarks of programs. If the nesting degree is increased, the branch coverage decreases and vice versa. It is clear that there is an inverse correlation among these variables.

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 feature with the highest influence on the coverage that evolutionary and random testing techniques can achieve. Nested branches pose a great challenge for the search. The high correlation value of the nesting degree supports that assertion.

Table 8: Relationship between the nesting degree and the average coverage for all the algorithms. The standard deviation is shown in subscript. We highlight the highest values of branch coverage for each algorithm and benchmark.

	100%CP			¬100%CP		
Nesting degree	ES	GA	RND	ES	GA	RND
1	$96.30_{4.83}$	$96.67_{5.78}$	$86.13_{10.81}$	$82.32_{8.16}$	$82.51_{7.89}$	$81.36_{7.97}$
2	$92.28_{7.66}$	$95.33_{7.11}$	$79.87_{13.06}$	$73.43_{11.70}$	$73.92_{11.58}$	$71.86_{11.66}$
3	$83.92_{12.06}$	$92.68_{10.28}$	$73.46_{13.99}$	$69.85_{15.35}$	$70.33_{15.46}$	$68.20_{15.23}$
4	$81.44_{14.32}$	$85.41_{13.96}$	$68.67_{16.03}$	$62.55_{17.93}$	$62.37_{18.07}$	$59.83_{17.82}$
5	-	-	-	$53.81_{21.09}$	$54.48_{21.57}$	$51.83_{20.80}$
6	-	-	-	$50.32_{21.14}$	$50.78_{21.90}$	$46.33_{20.93}$
7	-	-	-	$44.31_{20.57}$	$45.33_{20.77}$	$42.77_{19.68}$
ρ	-0.543	-0.381	-0.434	-0.563	-0.554	-0.589

Finally, we analyze the BCE measure, the new measure proposed to estimate the difficulty to generate an adequate test suite. In the 100%CP benchmark the correlation between this new measure and the coverage was 0.510 for ES, 0.375 for GA and 0.534 for RND, as we can see in Table 7. The obtained correlation coefficients when an RND generator is used are higher because the Markov model is inspired on it. In addition, in the ¬100%CP the correlations are even higher: 0.714 for ES, 0.698 for GA and 0.732 for RND. This promising measure is more correlated with the coverage (specially in the RND generator) than the nesting degree and the other static measures. This suggests that it is the best static complexity measure for measuring the difficulty of testing a program by an automatic test data generator.

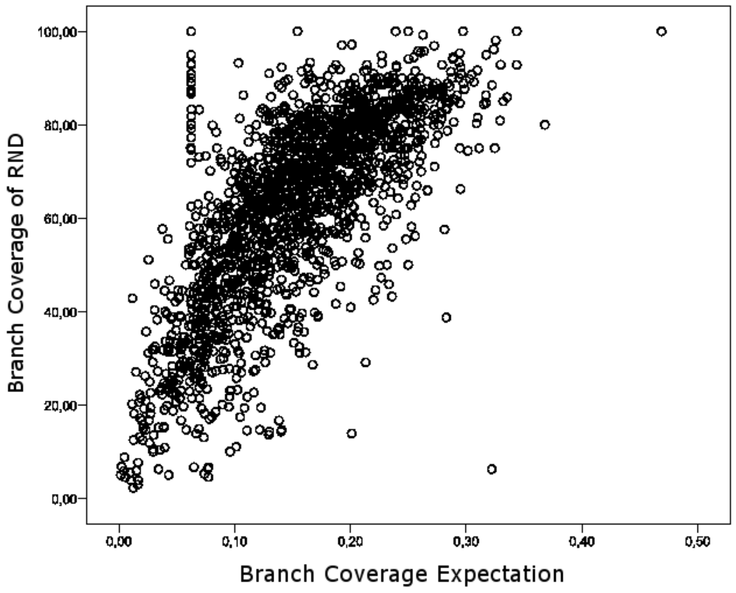

Figure 8: Average Branch Coverage of RND against the BCE measure

In Figure 8 we show the obtained average branch coverage with the random test data generator against the BCE measure. The trend is clear: the lower the value of Branch Coverage Expectation, the lower the coverage. We have opened a way to estimate the difficulty to test a program that is better than using the existing complexity measures or other known static measures like the nesting degree.

5.3. Another use of the Branch Coverage Expectation

As we detailed in Section 3 for each branch (BB_i, BB_j) the expected number of test cases required to traverse it is $1/E[BB_i, BB_j]$. Then, given a number of test cases x, we can compute the number of branches that would be theoretically traversed if the tester execute x random test cases, according to this equation:

$$f(x) = \left| \left\{ (i,j) \,\middle|\, \frac{1}{E[BB_i, BB_j]} < x \right\} \right|. \tag{20}$$

Thanks to this estimation, we propose a theoretical prediction about the behaviour of an automatic test data generator based on random testing.

In Figure 9 we show a plot for a particular program with the expected theoretical behaviour together with the experimental data obtained using the average branch coverage of the 30 independent executions of an RND generator for that program. The features of this test program are shown in Table 9. The resulting curves show that our theoretical prediction and the experimental data are very similar. The theoretical prediction is more optimistic because it does not take into account data dependencies. At the first steps of the algorithm, the experimental behaviour is better than the theoretical prediction, but in the region of high coverage (close to 90%), the behaviour of the RND test case generator is worse than expected. One explanation for this behaviour could be the presence of data dependencies in the program, which is not considered in the theoretical approach in order to keep it simple.

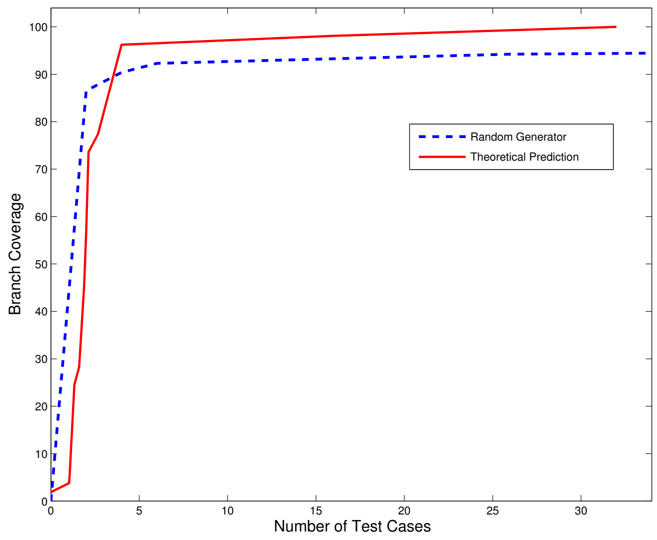

Figure 9: Coverage against the number of test cases of the random generator and the theoretical model

This new proposal is useful to decide which is the best way of generating a test suite for a piece of work. It could be useful to decide the parameters of an evolutionary test data generator prior to its execution, for example, the stopping condition.

5.4. Validation on Real Programs

In this section we want to make some validation of our proposed measure on real programs. We study 10 real programs extracted from the literature and with characteristics similar to the artificial programs used in

Table 9: Static measures for a representative program.

Features	Value
Nesting	1
Atomic Conditions	4
Total Decisions	26
Equalities	3
Inequalities	8
McCabe	62
Halstead Difficulty	32.53
LOCE	107
Density of Decisions	0.37
Density of LOCE	1.51

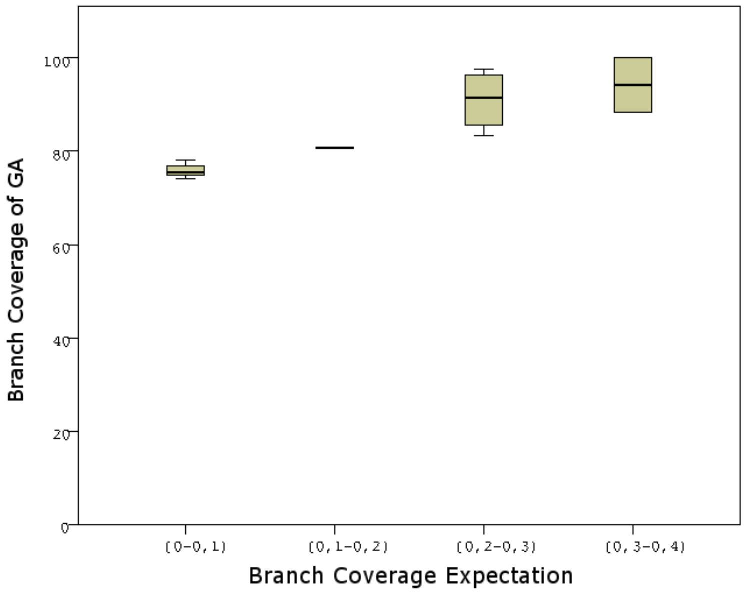

Figure 10: Average Branch Coverage of GA against the Branch Coverage Expectation for the real programs.

the previous sections. The reader must take into account that the number of programs used in the previous sections gives us the chance to average among 2,600 programs and extract statistically more reliable results. Despite the fact that in this section we only analyze the proposed testing measure over 10 programs, most of the conclusions are similar to the ones we have obtained with the synthetic programs.

In Figure 10 we show the average coverage obtained with the GA against the BCE. Once again we can see that the higher the Branch Coverage Expectation the higher the coverage. Relying on this figure we can state that there is a strong correlation between the obtained coverage and the BCE. Besides showing the figure, we have computed the Spearman's correlation coefficient. The coefficients are 0.770 and 0.758 for GA and RND, respectively. These values of correlation are even higher than the values obtained with the synthetic programs. Thanks to the experiments on real programs we can state that the proposed measure (BCE) is useful in order to measure the difficulty to automatically generate an adequate test suite.

When we have analyzed the behaviour of the ES algorithm, we obtained that the correlation does not exist (-0.013). We can try to justify this unexpected result because the ES has problems when it deals with a few complex branches. This algorithm achieves high coverage in most programs, but in a few, it obtains less coverage than expected. It is not able to cover some complex branches. This statement is supported by the value of correlation between the coverage obtained with ES and the estimation of test cases needed introduced in the previous section. They are correlated with a value of -0.582. This means that the most complex branch to cover in a program has high influence in the computation of the coverage.

6. Threats to Validity

In this section we are going to analyze all threats that might have an impact on the validity of the results. Our model is built with the aim of the simplicity. We think that the simplicity is necessary to generate an understandable and useful model, for this reason there are some threats that should be evaluated in order to analyze how they affect the theoretical model.

Data dependencies pose a great challenge for the exact techniques because the analysis of the code can be extremely complex. When a later condition depends on a previous one, there exists a data dependency among the variables that are part of these conditions. Thus, to satisfy a concrete condition could be necessary to traverse several previous branches. This prerequisite is not taken into account in our model. For that reason, the results of our theoretical model are not exact and might be optimistic, specially when nested complex branches occur in the code.

In this study we have applied three different test data generators, two based on evolutionary techniques (ES, GA) and one based on random testing (RND) for experimental validation. Although we cannot ensure that three algorithms could describe the behaviour of all automatic test data generators, at least we have made a fair comparison with different metaheuristic algorithms (GA and ES) and simple techniques such as RND as a sanity check. Nevertheless, our measure seems to predict better the effort required by the the random algorithm, which is still used as core of test data generators. Besides that, we have used an automatic generated benchmark of programs which could introduce bias in the experiments (as all generated programs), but we have tried to minimize its impact by generating programs with different characteristics.

Third, in our model we compute the transition probabilities according to the general probability theory, but in the base case of recursion, when a value of probability must be given to single atomic conditions we set arbitrary values. In this case, we set 1/2 probabilities in all the cases except when the logical operator is equal or inequal. Despite that, in our opinion half probability is a good value, it would be necessary a calculation of this probability for each condition because this value is not constant. This lack of accuracy is the prize we pay to maintain the model simple. In addition, setting values has another disadvantage, if there is a condition that can never be covered, our model would give a probability greater than 0. For example, if there is a condition like ($x^2 < 0$), our model will establish a probability 1/2 of being traversed but it will never be covered. In order to alleviate this problem we should analyze each single condition to take into account uncoverable branches in our model.

After describing the significant weaknesses of our model, we think that solving some of these disadvantages require a sophistication of the model. We plan to analyze whether it is worthwhile to make the model more complex. Nowadays, we think that it is preferable to maintain the simplicity of the model and avoid the complexity of analyzing the code repeatedly since this could be very slow in large programs.

7. Conclusions and Future Work

In this paper we dealt with the testing complexity from an original point of view: a program is more complex if it is more difficult to be automatically tested. Therefore, we defined the "Branch Coverage Expectation" in order to provide some knowledge about the difficulty of testing programs. The foundation of this measure is based on a Markov model of the program. The Markov model provides a theoretical background. The analysis of this measure indicates that it is more correlated with branch coverage than the other studied static measures. This means that this is a good way of estimating the difficulty of testing a program. We think, supported by the results, that this measure is useful for predicting the behaviour of an automatic test case generator.

The Markov model of the program can also be used to provide an estimation of the number of test cases needed to cover a concrete percentage of the program. We have compared our theoretical prediction with an average of real executions of a test case generator. The results show that our prediction is very similar to the evolution of a real execution of the test case generator. This theoretical prediction could be very useful to set some parameters of the test case generator prior to its execution, for example, the stopping condition. In conclusion, this model can help to predict the evolution of the testing phase, which consequently can save time and cost of the entire project.

In this study, we also analyzed the static features and the most common complexity measures in Software Engineering. This analysis was performed in two automatically generated benchmarks of programs. We studied the correlations among static measures of a program and we determined which of them can be useful to estimate the complexity of a program. At the end, the studied complexity measures like McCabe's and Halstead's seemed to be useless for this task. Instead, the nesting degree, the density of decisions and the density of LOCE were the static measures more correlated with branch coverage, although none of them is so correlated as the "Branch Coverage Expectation".

As future work we want to improve our model without losing its simplicity. We plan to advance in the knowledge of the features of a program that occurs in a condition. The computation of the probabilities associated to a concrete decision is a great challenge to improve our measure. In addition, we will take into account the data dependencies in the probabilities computed for the Markov model. This fact will provide more precision in the transition probabilities. Besides that, we plan to consider the amount of resources needed to execute different test cases, in particular when loops are involved in the execution of a test case. Finally, we would like to apply our complexity measure to real-world software and compare the results with the real difficulty of testing the program by an expert.

8. Acknowledgements

This work has been partially funded by the Spanish Ministry of Science and Innovation and FEDER under contract TIN2011-28194 (the roadME project).

References

[1] Ali, S., Briand, L.C., Hemmati, H., Panesar-Walawege, R.K.. A systematic review of the application and empirical investigation of search-based test case generation. IEEE Transactions on Software Engineering 2010;36(6):742–762.

[2] Anand, S., Burke, E., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P.. An orchestrated survey on automated software test case generation. Journal of Systems and Software 2013;.

[3] Arcuri, A.. Evolutionary repair of faulty software. Applied Soft Computing 2011;11:3494–3514.

[4] Bäck, T., Fogel, D.B., Michalewicz, Z.. Handbook of Evolutionary Computation. New York NY: Oxford University Press, 1997.

[5] Basili, V.. Quantitative software complexity models: A panel summary. Tutorial on models and methods for software management and engineering. IEEE Computer society press 1980;.

[6] Basili, V., Perricone, B.. Software errors and complexity: an empirical investigation. ACM communication 1984;27:42–52.

[7] Bertolino, A., Marré, M.. How many paths are needed for branch testing? J Syst Softw 1996;35(2):95–106.

[8] Boehm, B., Abts, C., Brown, A.W., Chulani, S., Clark, B.K., Horowitz, E., Madachy, R., Reifer, D.J., Steece, B.. Software cost estimation with COCOMO II. Prentice-Hall, 2000.

[9] Curtis, B., Sheppard, S.B., Milliman, P.. Third time charm: Stronger prediction of programmer performance by software complexity metrics. Piscataway, NJ, USA: IEEE Press, 1979.

[10] DeMarco, T.. Controlling Software Projects: Management, Measurement, and Estimates. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1986.

[11] Dixon, M.. An objective measure of code quality. Technical Report 2008;1.

[12] Ferrer, J., Chicano, F., Alba, E.. Evolutionary algorithms for the multi-objective test data generation problem. Software Practice and Experience 2012;42(11):1331–1362.

[13] Ferrer, J., Chicano, J.F., Alba, E.. Correlation between static measures and code coverage in evolutionary test data generation. International Journal of Software Engineering and Its Applications 2010;4(4):57–79.

[14] Halstead, M.H.. Elements of software science. Elsevier North-Holland 1977;.

[15] Hassan, A.E.. Predicting faults using the complexity of code changes. In: ICSE '09: Proceedings of the 31st International Conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2009. p. 78–88.

[16] Honglei, T., Wei, S., Yanan, Z.. The research on software metrics and software complexity metrics. In: Computer Science-Technology and Applications, 2009. IFCSTA '09. International Forum on. volume 1; 2009. p. 131–136.

[17] Jones, B., Sthamer, H.H., Eyres, D.. Automatic structural testing using genetic algorithms. Software Engineering Journal 1996;11(5):299–306.

[18] Khoshgoftaar, T., Munson, J.. Predicting software development errors using software complexity metrics. IEEE Journal on Selected Areas in Communications 1990;.
[19] Kitchenham, B., Pfleeger, S.L., Society, Z.C.. Towards a framework for software measurement validation. IEEE Transactions on Software Engineering 1995;21:929–944.
[20] Kobayash, H., Mark, B.L., Turin, W.. Probability, Random Processes, and Statistical Analysis. Cambridge University Press, 2011.
[21] Lakhotia, K., Harman, M., Gross, H.. Austin: An open source tool for search based software testing of C programs. Information and Software Technology 2013;55(1):112 – 125.
[22] Lam, S.S.B., Raju, M.L.H.P., M, U.K., Ch, S., Srivastav, P.R.. Ni. Procedia Engineering 2012;30:191–200.
[23] Malevris, N., Yates, D.. The collateral coverage of data flow criteria when branch testing. Information and Software Technology 2006;48(8):676 – 686.
[24] May, P.S.. Test Data Generation: Two Evolutionary Approaches to Mutation Testing. Ph.D. thesis; Computing Laboratory; 2007.
[25] McCabe, T.J.. A complexity measure. IEEE Trans on Software Engineering 1976;2(4):308–320.
[26] Misra, S., Misra, A.K.. Evaluation and comparison of cognitive complexity measure. SIGSOFT Softw Eng Notes 2007;32(2):1–5.
[27] Myers, G., Badgett, T., Sandler, C.. The Art of Software Testing. New York: John Wiley and Sons, 2011.
[28] Nogueira, A.F.. Predicting software complexity by means of evolutionary testing. In: ASE'12. New York, NY, USA: ACM; 2012. p. 402–405.
[29] Piwarski, P.. A nesting level complexity measure. SIGPLAN 1982;17(9):44–50.
[30] Samoladas, I., Gousios, G., Spinellis, D., Stamelos, I.. Open Source Development, Communities and Quality; Springer; volume 275 of *IFIP International Federation for Information Processing*. p. 237–248.
[31] Software, C.. Source monitor. 2012.
[32] Tassey, G.. The economic impacts of inadequate infrastructure for software testing. Technical Report; NIST; 2002.
[33] Thomson, W.. Mathematical and Physical Papers. Cambridge University Press, 1882.
[34] Weyuker, E.. Evaluating software complexity measures. IEEE Transactions on Software Engineering 1988;14(9):1357–1365.
[35] Yoo, S., Harman, M.. Regression testing minimization, selection and prioritization: a survey. Software Testing, Verification and Reliability 2012;22(2):67–120.
[36] Yu, L., Mishra, A.. Experience in predicting fault-prone software modules using complexity metrics. Quality Technology and Quantitative Management 2012;9(4):421–433.
[37] Zhou, Y., Xu, B., Leung, H.. On the ability of complexity metrics to predict fault-prone classes in object-oriented systems. Journal of Systems and Software 2010;83(4):660–674.

Appendix A. Tables

In this appendix we show the tables of correlation coefficients among all the measures analyzed in both benchmarks of programs, 100%CP and ¬100%CP.

Table A.10: The correlation coefficients among all the measures analyzed in the benchmark 100%CP

	HD	MC	LOCE	N	DD	DLOCE	BCE	LOC	SLOC	TNDj	TNCj	TNE	TNI	TND	CpD	HL	HV	HVL	HLV	HE	HT	HB	ES	GA	RND
HD	-	0.796	0.786	-0.108	0.052	-0.035	0.285	0.932	0.853	0.742	0.731	0.644	0.639	0.799	0.454	0.870	0.842	0.864	1.0	0.920	0.920	0.864	0.070	-0.101	0.077
MC	0.796	-	0.965	0.266	0.519	0.408	0.025	0.805	0.962	0.925	0.934	0.829	0.811	0.985	0.524	0.976	0.969	0.977	-0.796	0.954	0.954	0.977	-0.150	-0.226	-0.074
LOCE	0.786	0.965	-	0.344	0.515	0.474	-0.038	0.796	0.974	0.884	0.882	0.822	0.789	0.976	0.501	0.945	0.938	0.945	-0.786	0.921	0.921	0.945	-0.186	-0.251	-0.133
N	-0.108	0.266	0.344	-	0.765	0.877	-0.540	-0.207	0.180	0.235	0.240	0.311	0.234	0.276	0.136	0.138	0.127	0.139	0.108	0.089	0.089	0.139	-0.543	-0.381	-0.434
DD	0.052	0.519	0.515	0.765	-	0.912	-0.377	-0.043	0.405	0.449	0.489	0.485	0.437	0.538	0.283	0.368	0.367	0.372	-0.052	0.302	0.302	0.372	-0.439	-0.304	-0.311
DLOCE	-0.035	0.408	0.474	0.877	0.912	-	-0.485	-0.132	0.336	0.352	0.380	0.410	0.353	0.418	0.217	0.270	0.258	0.271	0.035	0.208	0.208	0.271	-0.504	-0.345	-0.397
BCE	0.285	0.025	-0.038	-0.540	-0.377	-0.485	-	0.307	0.081	0.065	0.008	-0.124	0.009	0.017	0.078	0.121	0.129	0.120	-0.285	0.159	0.159	0.120	0.510	0.375	0.534
LOC	0.932	0.805	0.796	-0.207	-0.043	-0.132	0.307	-	0.879	0.753	0.730	0.634	0.646	0.810	0.419	0.891	0.892	0.890	-0.932	0.910	0.910	0.890	0.136	-0.053	0.120
SLOC	0.853	0.962	0.974	0.180	0.405	0.336	0.081	0.879	-	0.884	0.878	0.794	0.778	0.973	0.492	0.975	0.970	0.975	-0.853	0.960	0.960	0.975	-0.091	-0.194	-0.050
TNDj	0.742	0.925	0.884	0.235	0.449	0.352	0.065	0.753	0.884	-	0.773	0.813	0.719	0.897	0.515	0.919	0.908	0.919	-0.742	0.900	0.900	0.919	-0.119	-0.175	-0.036
TNCj	0.731	0.934	0.882	0.240	0.489	0.380	0.008	0.730	0.878	0.773	-	0.734	0.806	0.905	0.497	0.913	0.901	0.913	-0.731	0.895	0.895	0.913	-0.158	-0.235	-0.072
TNE	0.644	0.829	0.822	0.311	0.485	0.410	-0.124	0.634	0.794	0.813	0.734	-	0.618	0.822	0.435	0.798	0.785	0.797	-0.644	0.779	0.779	0.797	-0.272	-0.279	-0.207
TNI	0.639	0.811	0.789	0.234	0.437	0.353	0.009	0.646	0.778	0.719	0.806	0.618	-	0.799	0.439	0.794	0.791	0.795	-0.639	0.774	0.774	0.795	-0.121	-0.201	-0.095
TND	0.799	0.985	0.976	0.276	0.538	0.418	0.017	0.810	0.973	0.897	0.905	0.822	0.799	-	0.503	0.961	0.959	0.962	-0.799	0.935	0.935	0.962	-0.147	-0.226	-0.082
CpD	0.454	0.524	0.501	0.136	0.283	0.217	0.078	0.419	0.492	0.515	0.497	0.435	0.439	0.503	-	0.524	0.518	0.523	-0.454	0.514	0.514	0.523	-0.089	-0.132	0.035
HL	0.870	0.976	0.945	0.138	0.368	0.270	0.121	0.891	0.975	0.919	0.913	0.798	0.794	0.961	0.524	-	0.991	1.0	-0.870	0.989	0.989	1.0	-0.071	-0.180	-0.012
HV	0.842	0.969	0.938	0.127	0.367	0.258	0.129	0.892	0.970	0.908	0.901	0.785	0.791	0.959	0.518	0.991	-	0.994	-0.842	0.971	0.971	0.994	-0.061	-0.172	-0.003
HVL	0.864	0.977	0.945	0.139	0.372	0.271	0.120	0.890	0.975	0.919	0.913	0.797	0.795	0.962	0.523	1.0	0.994	-	-0.864	0.987	0.987	1.0	-0.072	-0.181	-0.011
HLV	-1.0	-0.796	-0.786	0.108	-0.052	0.035	-0.285	-0.932	-0.853	-0.742	-0.731	-0.644	-0.639	-0.799	-0.454	-0.870	-0.842	-0.864	-	-0.920	-0.920	-0.864	-0.070	0.101	-0.077
HE	0.920	0.954	0.921	0.089	0.302	0.208	0.159	0.910	0.960	0.900	0.895	0.779	0.774	0.935	0.514	0.989	0.971	0.987	-0.920	-	1.0	0.987	-0.046	-0.168	0.006
HT	0.920	0.954	0.921	0.089	0.302	0.208	0.159	0.910	0.960	0.900	0.895	0.779	0.774	0.935	0.514	0.989	0.971	0.987	-0.920	1.0	-	0.987	-0.046	-0.168	0.006
HB	0.864	0.977	0.945	0.139	0.372	0.271	0.120	0.890	0.975	0.919	0.913	0.797	0.795	0.962	0.523	1.0	0.994	1.0	-0.864	0.987	0.987	-	-0.072	-0.181	-0.011
ES	0.070	-0.150	-0.186	-0.543	-0.439	-0.504	0.510	0.136	-0.091	-0.119	-0.158	-0.272	-0.121	-0.147	-0.089	-0.071	-0.061	-0.072	-0.070	-0.046	-0.046	-0.072	-	0.365	0.445
GA	-0.101	-0.226	-0.251	-0.381	-0.304	-0.345	0.375	-0.053	-0.194	-0.175	-0.235	-0.279	-0.201	-0.226	-0.132	-0.180	-0.172	-0.181	0.101	-0.168	-0.168	-0.181	0.365	-	0.403
RND	0.077	-0.074	-0.133	-0.434	-0.311	-0.397	0.534	0.120	-0.050	-0.036	-0.072	-0.207	-0.095	-0.082	0.035	-0.012	-0.003	-0.011	-0.077	0.006	0.006	-0.011	0.445	0.403	-

Table A.11: The correlation coefficients among all the measures analyzed in the benchmark ¬100%CP

	HD	MC	LOCE	N	DD	DLOCE	BCE	LOC	SLOC	TNDj	TNCj	TNE	TNI	TND	CpD	HL	HV	HVL	HLV	HE	HT	HB	ES	GA	RND
HD	-	0.698	0.359	-0.062	0.023	0.014	0.051	0.664	0.648	0.653	0.651	0.557	0.569	0.463	0.441	0.764	0.576	0.747	-1.0	0.872	0.872	0.747	0.069	0.067	0.079
MC	0.698	-	0.571	0.257	0.432	0.351	-0.142	0.472	0.667	0.936	0.937	0.803	0.827	0.718	0.671	0.782	0.762	0.786	-0.698	0.803	0.803	0.786	-0.177	-0.168	-0.173
LOCE	0.359	0.571	-	0.692	0.590	0.833	-0.461	0.414	0.717	0.435	0.432	0.479	0.485	0.814	0.086	0.564	0.503	0.560	-0.359	0.524	0.524	0.560	-0.461	-0.452	-0.476
N	-0.062	0.257	0.692	-	0.708	0.870	-0.575	-0.160	0.190	0.163	0.161	0.229	0.220	0.502	-0.031	0.020	0.009	0.019	0.062	-0.007	-0.007	0.019	-0.563	-0.554	-0.589
DD	0.023	0.432	0.590	0.708	-	0.774	-0.426	-0.178	0.280	0.306	0.304	0.385	0.372	0.723	0.026	0.089	0.056	0.087	-0.023	0.070	0.070	0.087	-0.476	-0.473	-0.497
DLOCE	0.014	0.351	0.833	0.870	0.774	-	-0.556	-0.113	0.284	0.247	0.243	0.308	0.291	0.593	0.013	0.096	0.076	0.095	-0.014	0.073	0.073	0.095	-0.577	-0.564	-0.602
BCE	0.051	-0.142	-0.461	-0.575	-0.426	-0.556	-	0.075	-0.143	-0.078	-0.079	-0.200	-0.138	-0.318	0.080	-0.021	-0.006	-0.020	-0.051	0.001	0.001	-0.020	0.714	0.698	0.732
LOC	0.664	0.472	0.414	-0.160	-0.178	-0.113	0.075	-	0.857	0.398	0.397	0.386	0.406	0.494	0.144	0.906	0.821	0.901	-0.664	0.874	0.874	0.901	0.102	0.099	0.116
SLOC	0.648	0.667	0.717	0.190	0.280	0.284	-0.143	0.857	-	0.533	0.532	0.549	0.572	0.834	0.152	0.916	0.813	0.910	-0.648	0.875	0.875	0.910	-0.137	-0.137	-0.137
TNDj	0.653	0.936	0.435	0.163	0.306	0.247	-0.078	0.398	0.533	-	0.849	0.753	0.781	0.555	0.747	0.702	0.697	0.707	-0.653	0.731	0.731	0.707	-0.110	-0.101	-0.102
TNCj	0.651	0.937	0.432	0.161	0.304	0.243	-0.079	0.397	0.532	0.849	-	0.753	0.771	0.551	0.746	0.702	0.697	0.707	-0.651	0.731	0.731	0.707	-0.116	-0.107	-0.111
TNE	0.557	0.803	0.479	0.229	0.385	0.308	-0.200	0.386	0.549	0.753	0.753	-	0.623	0.600	0.544	0.633	0.619	0.636	-0.557	0.646	0.646	0.636	-0.278	-0.270	-0.270
TNI	0.569	0.827	0.485	0.220	0.372	0.291	-0.138	0.406	0.572	0.781	0.771	0.623	-	0.619	0.559	0.658	0.645	0.662	-0.569	0.671	0.671	0.662	-0.207	-0.198	-0.204
TND	0.463	0.718	0.814	0.502	0.723	0.593	-0.318	0.494	0.834	0.555	0.551	0.600	0.619	-	0.132	0.688	0.605	0.683	-0.463	0.648	0.648	0.683	-0.338	-0.336	-0.348
CpD	0.441	0.671	0.086	-0.031	0.026	0.013	0.080	0.144	0.152	0.747	0.746	0.544	0.559	0.132	-	0.394	0.436	0.402	-0.441	0.437	0.437	0.402	0.026	0.026	0.031
HL	0.764	0.782	0.564	0.020	0.089	0.096	-0.021	0.906	0.916	0.702	0.702	0.633	0.658	0.688	0.394	-	0.932	0.999	-0.764	0.980	0.980	0.999	-0.021	-0.018	-0.010
HV	0.576	0.762	0.503	0.009	0.056	0.076	-0.006	0.821	0.813	0.697	0.697	0.619	0.645	0.605	0.436	0.932	-	0.946	-0.576	0.874	0.874	0.946	-0.040	-0.030	-0.022
HVL	0.747	0.786	0.560	0.019	0.087	0.095	-0.020	0.901	0.910	0.707	0.707	0.636	0.662	0.683	0.402	0.999	0.946	-	-0.747	0.974	0.974	1.0	-0.023	-0.020	-0.011
HLV	-1.0	-0.698	-0.359	0.062	-0.023	-0.014	-0.051	-0.664	-0.648	-0.653	-0.651	-0.557	-0.569	-0.463	-0.441	-0.764	-0.576	-0.747	-	-0.872	-0.872	-0.747	-0.069	-0.067	-0.079
HE	0.872	0.803	0.524	-0.007	0.070	0.073	0.001	0.874	0.875	0.731	0.731	0.646	0.671	0.648	0.437	0.980	0.874	0.974	-0.872	-	1.0	0.974	0.004	0.005	0.016
HT	0.872	0.803	0.524	-0.007	0.070	0.073	0.001	0.874	0.875	0.731	0.731	0.646	0.671	0.648	0.437	0.980	0.874	0.974	-0.872	1.0	-	0.974	0.004	0.005	0.016
HB	0.747	0.786	0.560	0.019	0.087	0.095	-0.020	0.901	0.910	0.707	0.707	0.636	0.662	0.683	0.402	0.999	0.946	1.0	-0.747	0.974	0.974	1.0	-0.023	-0.020	-0.011
ES	0.069	-0.177	-0.461	-0.563	-0.476	-0.577	0.714	0.102	-0.137	-0.110	-0.116	-0.278	-0.207	-0.338	0.026	-0.021	-0.040	-0.023	-0.069	0.004	0.004	-0.023	-	0.954	0.940
GA	0.067	-0.168	-0.452	-0.554	-0.473	-0.564	0.698	0.099	-0.137	-0.101	-0.107	-0.270	-0.198	-0.336	0.026	-0.018	-0.030	-0.020	-0.067	0.005	0.005	-0.020	0.954	-	0.950
RND	0.079	-0.173	-0.476	-0.589	-0.497	-0.602	0.732	0.116	-0.137	-0.102	-0.111	-0.270	-0.204	-0.348	0.031	-0.010	-0.022	-0.011	-0.079	0.016	0.016	-0.011	0.940	0.950	-