Identifying Characteristics of Java Methods that May Influence Branch Coverage: An Exploratory Study on Open Source Projects

Camila Faria de Castro, Decio de Souza Oliveira Jr and Marcelo Medeiros Eler
EACH - Escola de Artes, Ciências e Humanidades
Universidade de São Paulo
São Paulo - SP - Brasil
{camila.faria.castro,decio.oliveira,marceloeler}@usp.br

***Abstract*—Software testing is an important activity to assure the quality of software. Testing techniques and criteria have been created over time to help testers to devise high quality test suites. However, duly and systematically testing a software to reach high coverage on criteria, such as branch coverage, requires much effort. In this context, identifying characteristics of a software that may influence branch coverage is important to create software easier to test since the beginning. Therefore, the main purpose of this paper is to present an investigation conducted by us to identify the differences between methods whose branches were fully covered and the methods that have been partially covered. This investigation has been conducted on 39 open source Java projects.**

***Index Terms*—software testing, testability, branch coverage, static analysis, symbolic execution.**

I. Introduction

Software testing is an important process to ensure that a software product will deliver the expected quality properties. It consists of executing a program under test with the aim of revealing failures [15]. It is unpractical or even impossible to test a software against all possible test case scenarios, therefore one can only prove the presence of defects, not their absence. In this context, testing techniques have been created to help testers to systematically devise and select test cases that are more likely to reveal a yet unrevealed failure. Such techniques also provide testers with useful criteria to evaluate the quality of their test suite.

Structural testing is a common technique whose main goal is to ensure that all structures of the program under test have been executed. Particularly, the all-branches criterion tries to assure that all control flow deviations within the program structure have been executed at least once. Although controversial, the coverage of structural testing criteria, such as branches and lines, for example, has been used as a metric to evaluate the overall quality of a test suite. The effort to achieve high coverage on structural testing criteria, however, may be huge or even prohibitive depending on its size and complexity.

In a context in which software complexity grows and vendors are more concerned about software quality due to an overcrowded and competitive market, reducing the effort and increasing the accuracy of the software testing activities may be crucial. Therefore, the main goal of this paper is to identify characteristics of a software that may influence testers on creating test cases to achieve high coverage of branches. Accordingly, we present an exploratory study that we have conducted over 39 open source Java projects from the perspective of the unit testing phase. We investigated the influence of structure, size and constraints on the branch coverage achieved by the test cases publicly available.

To achieve our goals we have compared characteristics of the methods for which the testers have reached full branch coverage with the methods whose branches have been only partially covered. Among other characteristics, we have taken into account the metrics used in a previous work to analyze which properties of a method could influence the effort of generating test data using symbolic execution [10]. These metrics are related to method's control flow structure (e.g. loops, nested loops, cyclomatic complexity), size (e.g. number of instructions, number of input arguments), and constraints (e.g. data types, number of elements, number of constraints, exceptions). Some of these metrics were collected directly from the Java bytecode, while others, such as constraint information, have been collected after symbolic execution. As we used test artifacts collected from open source repositories, we have considered only code coverage to draw our conclusions regarding the influence of the investigated factors. We have not considered, for example, the time spent to create each test suite, which is an important variable when it comes to test effort.

We believe that the results of this investigation may provide an insight into what makes a software easier to test, in a way that structural testing requirements can be accomplished and ideally without much effort. In spite of being a preliminary study, it statistically quantifies the relation of specific metrics to the obtainment of fully or partial code coverage in test cases, fomenting the discussion of the topic in the area and contributing with more detailed studies to be developed in the future.

The remainder of this paper is organized as follows. Sec-

978-1-5090-3339-3/16/$31.00 2016 IEEE

tion II presents the related work. Section III and Section IV shows the configuration of the study and the results we obtained, respectively. Section V discusses the results of our investigation, Section VI indicates final recommendations for software developers who aim to achieve deeper branch coverage based on our preliminary results, and, lastly, Section VII presents the concluding remarks and future directions.

II. Related Work

Several research papers have been published in the literature with respect to software testability and factors that may influence the effort to generate a complete test suite. Most of these work are related to the testability of object oriented software. Binder defines software testability as the relative ease and expense of revealing software faults [6]. He establishes that the effort to test object-oriented software is a result of several factors, such as characteristics of the representation, characteristics of the implementation, built-in test capabilities, the test support environment and the software development process, for instance.

Voas and Miller [18, 19] suggest that software testability decreases when an information computed by the program is not returned as an output. Baudry [5] measured the test effort considering object oriented designs such as UML diagrams. Briand et. al [7] analyzed the effect of contracts (precondition and post condition of operations and invariants) on testability. Zhou et. al [20] investigated five property dimensions of a class that may impact testability, including size, cohesion, coupling, inheritance, and complexity. Many authors have focused on studying the impact of the CK metrics [8] on software testability [1, 3, 4, 13, 17].

Most of related work on testability focus on characteristics of the process, of the abstractions and models, and in the object oriented characteristics. They are also concerned with the most general concept of testability, related to the control and observability issues. Few approaches have investigated characteristics of the implementation that may lead to high or low coverage of some specific criterion such as branch coverage. Moreover, as far as we know, there is not any approach that have investigated the impact of the characteristics of a method regarding the characteristics of its constraints and using symbolic execution.

III. Study Setup

This section describes the configuration of the study we undertook to identify characteristics of a method that may influence full or partial branch coverage. First, we present the research questions of this investigation. Following, we discuss the granularity of our analysis, i.e., what artifacts we have considered during our study. Next, we reason about the criterion we used to differentiate the two groups of methods we compare in our study. Then, we discuss how we have chosen the open source projects we have analyzed. Finally, we show the metrics we have chosen to compare the two groups of methods and how we have collected their data.

A. Research questions

We have conducted our investigation based on three research questions:

- RQ1 - What is the influence of control-flow structure in branch coverage?
- RQ2 - What is the influence of constraints in branch coverage?
- RQ3 - What is the influence of the method size in branch coverage?

B. Scoping and granularity

Several artifacts produced during a software development process may impact the testability of a software. In this study, however, we want to investigate which characteristics at code level may lead to full or partial branch coverage. Particularly, we want to understand the impact of each investigated characteristic with respect to the unit testing phase. For this particular investigation, we have decided to adopt the method as the smallest unit. Therefore, we only resort to metrics related to the method itself without considering the characteristics of the class it belongs to.

C. Testability criteria

The effort required to test software is extremely related to its testability. Testability is an important quality indicator since its measurement leads to the prospect of facilitating and improving a test process. It has been been defined as the degree to which a system or component facilitates the establishment of test criteria and the performance of tests to determine whether those criteria have been met [12]. According to [16], a relevant measure to assess the testability is how many test cases are needed to form a complete test suite, which unambiguously determine whether the software is correct or not according to some specification. A common approach to evaluate whether the test suite is complete is resorting to testing criteria such as the all-branches criterion[2].

In general, it is not possible to prove that a program is free of defects, hence it is not possible to establish for sure whether a test suite is complete or not. Given this limitation, we have decided to resort to a common criterion used by testers to evaluate the quality and completeness of their test suites: code coverage. In the context of this paper, we consider a complete test suite one that can achieve 100% of branch coverage on its target.

D. Sample selection

In hopes of selecting an unbiased sample of Java programs, we have selected a third party benchmark named SF110[1], which is an evolution of the SF100 benchmark [11]. SF110 is made up of a collection of 110 open source Java projects that differ considerably in size, complexity, and application domains.

From the 110 projects of SF110, we have selected only those projects whose test cases were available and written in

[1] http://www.evosuite.org/experimental-data/sf110/

the JUnit[2] format because we have automated the execution of the test cases and the measurement of the code coverage through the Apache Ant[3] tasks and Jacoco[4] tool, respectively. During our analysis, we have excluded 50 projects because no test cases were available, 14 projects because the test cases were not written in the JUnit format, and 7 projects due to compilation and library issues. Therefore, after applying all criteria, only 39 out of the 110 projects remained. These 39 projects have 16,125 classes and 114,718 methods altogether.

The main goal of this exploratory study is to identify which characteristics of methods may impact the effort of testing activities and, in particular, achieving full or partial branch coverage. There are methods, however, which poses no challenge to testers. Such methods are those whose bodies are empty or have only one execution path, i.e., there is no constraint to be analyzed due to absence of branches. Methods that have only one path are often associated with attribute accessors (getters and setters) and constructors, meaning that any input will result in a complete execution of the method [10].

A constraint is simply a logical relation among several elements in which their possible values is restricted. The elements of a constraint may have different data types (e.g. integer, float, object) and categories (e.g. variable, method call, constant). A constraint is typically defined by instructions such as if, swich, while, for, and so forth. However, in this study, we also consider implicit constraints that arise from exception handling environments. For instance, many static analysis tools would consider the excerpt of code `try doSomething(); catch(ExceptionType et)` as constraint free, but in our approach we consider that the following constraint is present: `if (ExceptionType condition)`. This means that `(ExceptionType condition)` must be replaced by a condition in order to raise or avoid the target exception.

After a further analysis of the 114,718 methods of the 39 analyzed projects, we have excluded 82,351 due o the absence of constraints. According to our initial plan, the remaining 32,367 should be split in two groups: Full Branch Coverage and Partial Branch Coverage. The Full Branch Coverage were supposed to contain the methods whose branches were 100% covered after executing the test cases provided by the developers, while the Partial Branch Coverage were suppose to contain the methods whose branches were covered only partially. In that sense, the distinct characteristics of the two groups would give an insight, at least in this context, which characteristics may have influenced those methods to be fully our partially covered.

The limitation of using a third party benchmark, however, is that there is no guarantee that the testers have put a lot of effort into the testing activities to achieve high coverage on some testing criteria such as all branches, for example. In that sense, one does not know whether a method has only 40% of its branch covered due to lack of effort or due to its intrinsic characteristics that may difficult its test and consequently achieving high coverage. Similarly, we cannot know, for example, whether a method whose branches were 85% covered had not been completely covered due to its characteristics, infeasible paths or even because the goal of the tester was not reaching 100% of code coverage, but a certain minimum level.

Given these limitations, comparing the characteristics of methods that reached 100% of branch coverage with those whose branches were not completely covered would not give us real insights on which factors may lead to full or partial branch coverage. Therefore, we decided to put into the Partial Branch Coverage only methods in which more than 50% of the branches were executed. Again, there is no guarantee that the testers have put a lot of effort to test these methods and their characteristics may be the reason why they were not fully covered. Nevertheless, we can assume that at least some effort was required to achieve more than 50% of branch coverage and it is a way of mitigating the limitations of using a third party benchmark.

At the end of the sample selection process, 2,469 methods had 100% of their branches covered and therefore they were put into the Full Branch Coverage group, while 2,114 methods had more than 50% and less than 100% of their branches covered and consequently they were put into the Partial Branch Coverage group.

E. Metrics selection

In previous work, Eler et. al [10] have investigated the characteristics of Java programs that may influence test data generation based on symbolic execution and from the context of unit testing. Broadly, they have collected metrics to understand in which frequency and/or the distribution the commonly reported issues of that context appear, namely path explosion, constraint complexity, dependency and exception-dependant paths. All metrics have been collected considering the method as the smallest unit.

In this context, we assumed that the same issues that may hamper test data generation based on symbolic execution might also hinder testers when manually creating test cases to meet testing criteria, such as branch coverage, for instance. In fact, testers sometimes have to mentally and symbolically execute a method under test to discover which test inputs would cause a particular path to be traversed. Therefore, we decided to use some of the metrics used in that specific investigation.

The metrics we analyze in our exploratory study are related to its control-flow structure, its constraints, and its size. When it comes to the control-flow structure metrics, we calculate the cyclomatic complexity, the number of loops and the number of nested loops. Such metrics are important to characterize how many independent paths a method present and what is the potential for path explosion, i.e., increasing the number of paths to handle such that it is too difficult to handle. The constraints of a method (or predicates) may be simple or complex. Constraints that contain only integers and simple

[2]www.junit.org
[3]http://ant.apache.org/
[4]www.eclemma.org/jacoco/

variables (e.g. `(x==5)`), for example, are easier to handle than constraints that include method calls and complex types such as objects and arrays (e.g. `(m.foo()==array[x])`). Therefore, we collect the number of constraint elements of each data type (e.g. integer, float) and category (e.g. method call, variable). Regarding the method size, we collect the number of instructions, the number of constraints and the number of input arguments.

F. Data extraction

In previous work, a tool called CP4SE[5] (Constraint Profiling for Symbolic Execution) have been developed to perform static analysis of Java Bytecode [9, 10]. This tool has been adapted over time to collect several static metrics of Java programs, including the metrics defined in Section III-E.

It is important to mention that the metrics related to the control-flow structure and the size of the method were collected from the method as it is. However, the metrics related to the constraints (types and category of the elements) were collected after symbolically executing each method. We decided to collect these metrics after performing symbolic execution because we claim that it represents a more realistic complexity analysis of the constraints of a method. In fact, when a tester is trying to figure out which test input would cause the execution of a particular path or branch, he or she has to mentally and symbolically execute the code at some level.

Listing 1 shows an example of a method before and after symbolic execution. Notice that the constraint `(ration<0.75)` may look extremely simple at first, but in fact it depends on method calls and two multiplications. In the code before symbolic execution, the constraint (Line 5) that may lead the method to returning false or true has two elements: a float type variable and a constant. The constraint after symbolic execution is presented as a comment (Line 5). It has seven elements: two integer variables (x and y), one object used twice (wd), two method calls (getHeight() and getWidth()), and a constant (0.75).

Listing 1: A simple method before symbolic execution

```
1  public boolean m1(Window wd, int x, int y){
2    int h = wd.getH();
3    int w = wd.getW();
4    int sq = x * y;
5    int sqwd = h*w;
6    float ratio = (float) sq/sqwd;
7    if (ratio<0.75)
8    //((x * y)/(wd.getH()*wd.getW())<0.75)
9       return true;
10   return false;
11 }
```

The CP4SE tool writes all metrics collected on the analyzed projects to a .CSV file. Each line contains the information of a particular method. In this study, we crossed the information provided by the Jacoco tool regarding the code coverage with the files generated by CP4SE in order to generate one file for each group of methods considered in this paper: FULL, whose methods had 100% of their branches covered; and PARTIAL, whose methods had more than 50% and less than 100% of their branches exercised. The data of these two groups of methods can be found here: http://each.uspnet.usp.br/marceloeler/sccc16/.

[5]https://github.com/marceloeler/CP4SE

G. Threads to the validity

This results of this study may be influenced by two main factors: the subject programs and the tool that extracts the metrics we have analyzed. The subject programs are open source software and we have no information on how their test cases have been desgined by the developers and testers. We are not sure that their intention was to reach high coverage in a structural testing criterion such as all-branches, for example. Consequently, the decision to analyze only methods which more than 50% of the branches have been executed may influence the results.

Moreover, the way CP4SE symbolically executes the subject programs and extracts the metrics may also influence the results of this study. CP4SE does not perform intraprocedural integration, which means that the analysis is based on the unit testing perspective. This tool also considers only one iteration on each loop of the program, even when several loops may be required to cover a specific branch. For more information on how CP4SE extract metrics from a Java program we refer to the work of Eler et al. [9, 10].

IV. Study Results

According to our sampling process, 2,469 methods had 100% of their branches covered, hence they were labeled as FULL, while 2,114 methods had more than 50% and less than 100% of their branches executed, therefore labeled as PARTIAL. From now on, we refer to the full covered group of methods as FULL and the partially covered group of methods as PARTIAL. It is important to mention that most methods in PARTIAL had from around 70 to 85% of their branches covered, which may imply that this set of methods had been target of careful test tasks. The following subsections show the comparisons between these two groups regarding control-flow structures, constraints, and size.

A. Control-flow structure

When it comes to the control-flow structure metrics, we calculate the number of loops and nested loops, and also the cyclomatic complexity of each method. The cyclomatic complexity is a quantitative measure of the number of linearly independent paths through a program's source code. The more paths to handle, the more difficult it is to completely test a method. The number of loops and nested loops is also a significant factor to investigate since their presence may imply much more execution paths to handle before covering all branches of a method under test.

Figure 1 shows the boxplot comparing the cyclomatic complexity of the methods in PARTIAL and FULL. The cyclomatic complexity of the methods in PARTIAL are much larger

than the cyclomatic complexity of the methods in FULL. Table I shows the distribution of methods by cyclomatic complexity. One of the most remarkable differences is that only 3% of the methods in PARTIAL have 2 linearly independent paths, while 52% of the methods in FULL have the same characteristic. Another striking difference is that 25% of the methods of PARTIAL have more than 10 linearly independent paths, while only 0.3% of the methods in FULL have this same number of paths.

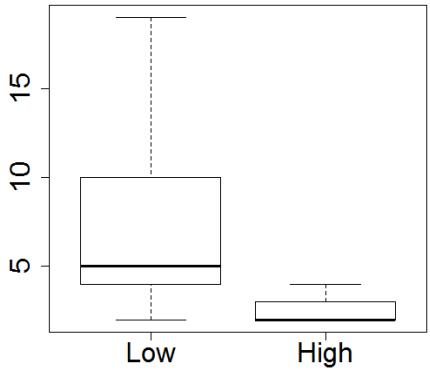

Fig. 1: Boxplot with the cyclomatic complexity of groups PARTIAL and FULL

TABLE I: Number of method for each amount of loop

Cyclomatic Complexity	PARTIAL	FULL
2	61 (3%)	1299 (52.5%)
3	401 (19%)	577 (23%)
4	355 (16%)	227 (9%)
5	250 (12%)	133 (5%)
6	176 (8%)	70 (3%)
7	146 (7%)	40 (1.5%)
8	98 (4.5%)	23 (1%)
9	93 (4.5%)	23 (1%)
>=10	534 (25%)	77 (0.3%)

TABLE II: Comparison between PARTIAL and FULL considering the number of loops

	Loops		Nested Loops	
Number	**PARTIAL**	**FULL**	**PARTIAL**	**FULL**
0	956 (45%)	1364 (55%)	1697 (80.5%)	2263 (92%)
1	700 (33%)	889 (36%)	242 (11.5%)	142 (5.5%)
2	242 (11.5%)	145 (6%)	80 (4%)	38 (1.5%)
3	93 (4.5%)	37 (1.5%)	41 (2%)	17 (0.6%)
>=4	123 (6%)	34 (1.5%)	54 (2.5%)	9 (0.3%)

Concerning the number of loops, most methods of both PARTIAL and FULL have at least one loop structure. However, only around 20% of the methods in PARTIAL have nested loops, while 8% of the methods in FULL present this same characteristic. Table II shows a comparison between the number of methods for each number of loops and nested loops, respectively.

B. Constraints

The effort to test a particular method may be impacted by the characteristics of its constraints. First, we measured the number of constraints of each method, considering the mean number among its paths. Then, we analyzed the data types (e.g. integer, float) and the categories (e.g. variable, method call) of the constraint elements.

Figure 2 shows a boxplot that compares the mean number of constraints. This boxplot clearly shows that the methods of the PARTIAL group has more constraints to be solved than the methods of FULL. Table III shows the distribution of methods by each number of constraints. Notice that 80% of the methods in FULL has up to 2 constraints, while only around 10% has more than 3 constraints. This is in line with the fact that the methods in FULL have less execution paths to handle than the methods in PARTIAL.

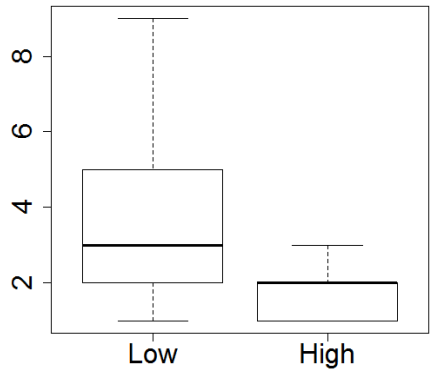

Fig. 2: Boxplot with the number of constraints

TABLE III: Frequency of methods by number of constraints

Number constraints	PARTIAL	FULL
1	52 (2.5%)	924 (37.5%)
2	677 (32%)	1070 (43.5%)
3	409 (19%)	250 (10%)
4	326 (15.5%)	124 (5%)
5	175 (8%)	44 (2%)
6	149 (7%)	30 (1.5%)
>=7	326 (15.5%)	27 (1%)

The number of constraints may not imply a method too difficult to test, as long as the constraints are simple to solve. That is why we compare the two groups with respect to the number of elements of their constraints. Figure 3 shows a boxplot with this comparison. Notice that the number of elements in PARTIAL is higher than the number of elements in FULL.

Beyond the number of constraints and constraint elements, we have investigated the complexity of the constraints by looking at the data types and the category of their elements. We found out that the two most popular types are the integer (2032 methods in PARTIAL and 1975 in FULL) and object (1898 in PARTIAL and 1897 in FULL). The null (1019 in PARTIAL

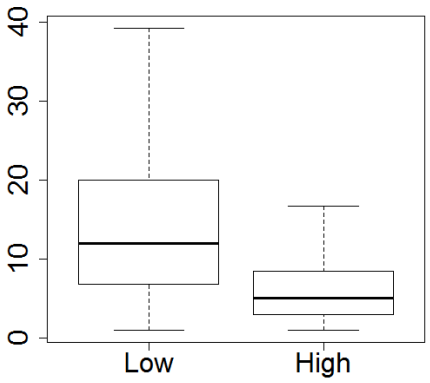

Fig. 3: Boxplot with the number of constraint elements

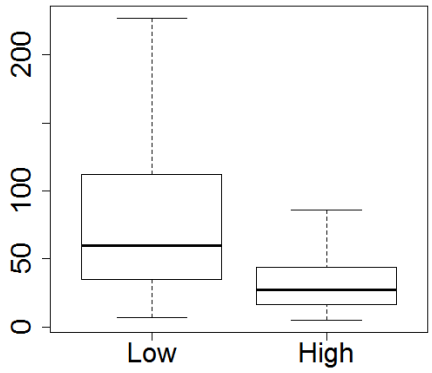

Fig. 4: Boxplot with the number of bytecode instructions

and 624 in FULL) and array types (696 in PARTIAL and 498 in FULL) are also common, but float and string types are not. Notice that proportion of methods that present the same data types are also quite similar in both groups.

Considering the category types, we found out that, in PARTIAL, 2,080 (98%) methods have constants, 2,032 (96%) methods have variables, 1,912 (90%) methods have method calls, and 317 (15%) methods have exception dependant paths, i.e., methods that are executed only when a particular exception is raised. In FULL, 2,314 (94%) methods have constants, 2,287 (93%) methods have variables, 1,642 (66%) methods have method calls, and 198 (8%) methods have exception dependant paths. The distribution of constants and variables are about the same, but the number of elements that are method calls or exception-related constraints is higher in PARTIAL.

C. Size

Figure 4 shows a boxplot comparing the PARTIAL and FULL group with respect to the number of Bytecode instructions of each method. It is clearly shown that the methods of the PARTIAL group has much more Bytecode instructions than the methods in the FULL group. Most of methods of PARTIAL has from nearly 40 to 100 Bytecode instructions while most methods of FULL has from around 20 to 40 Bytecode instructions.

Table IV show the distribution of methods by each number of input arguments. Notice that the frequency of methods in each group with the same amount of input arguments are almost the same.

TABLE IV: Number of method arguments for each amount of loop

Number of input arguments	PARTIAL	FULL
0	900 (42%)	1071 (43%)
1	731 (34.5%)	860 (35%)
2	257 (12%)	326 (13%)
>=3	226 (10.5%)	212 (8.5%)

V. Discussion

In this subsection we provide a discussion on the results we have obtained in our investigation. The discussion will be around the three main characteristics we have considered: control-flow structure, constraints, and size.

RQ1 - What is the influence of control-flow structure in branch coverage? When it comes to the control flow structure of a method, it is clear that the most important characteristic that may influence full or partial branch coverage is the cyclomatic complexity. The difference between the two groups of methods is an evidence that methods should have limited number of linearly independent paths. In the context of this investigation, methods with cyclomatic complexity up to 4 have more chance to be fully covered than methods with complexity ciclomatic over 4. When it comes to the number of loops, the methods in FULL have less loops and nested loops than the methods in PARTIAL, but it is not an outstanding difference.

RQ2 - What is the influence of constraints in branch coverage? The most outstanding difference in this type of analysis is the number of constraints and the number of constraint elements in each method. It is in line with the fact that the methods in PARTIAL have more bytecode instructions and execution paths than methods in FULL. We expected that the complexity of the constraints, regarding their category or data types, would emerge in our investigation, but they appear to be very similar when we consider how much each element contribute to the structure of the constraints.

RQ3 - What is the influence of the method size in branch coverage? The number of byte code instructions seems to be a factor that makes difference when testing a method. It is natural that the more code written, the more complex the method is, hence it is more difficult to test. That is why long methods are considered code smells. We expected the number of arguments of a method would have some impact on our analysis, but both groups are about the same when it comes to this characteristic.

It seems that, apart from the cyclomatic complexity and the

TABLE V: Correlation between cyclomatic complexity, number of constraints, number of constraint elements and number of instructions

	PARTIAL					FULL			
Factor	**CC**	**Cnstr**	**Elem**	**Instr**		**CC**	**Cnstr**	**Elem**	**Instr**
CC	1.0000000	0.7522520	0.5654953	0.6274813		1.0000000	0.7709065	0.5230226	0.4878781
Cnstr	0.7522520	1.0000000	0.7070565	0.6307769		0.7709065	1.0000000	0.6843779	0.5446451
Elem	0.5654953	0.7070565	1.0000000	0.5853424		0.5230226	0.6843779	1.0000000	0.5030967
Instr	0.6274813	0.6307769	0.5853424	1.0000000		0.4878781	0.5446451	0.5030967	1.0000000

number of constraints and constraint elements, the methods of the two groups share many similarities. In fact, the number of constraints may be directly impacted by the cyclomatic complexity, once the more number of execution paths, the more number constraints, and probably the more number of constraint elements. To check whether these factors are inline, we have calculated the correlation between them, which is shown in TableV.

This table shows that there is a positive correlation between the factors we have investigated. There is a positive and linear relationship between the cyclomatic complexity (CC) and the number of constraints in both groups. There is a also a positive and linear relationship between the number of constraints with the number of constraint elements in PARTIAL, but moderate in FULL. The relationship between the number of instructions and other factors are moderate in both groups.

VI. Recommendations

Based on the discussed results of this study, we are now able to recommend some best practices to be followed by software developers who aim to facilitate the process of software testing in the future.

It was demonstrated that methods with limited number of linearly independent paths and with smaller size are more likely to be fully covered, because in these cases the methods are less complex. Therefore, developers are advised to reduce the number of instructions and the cyclomatic complexity in methods in development. One way of doing this is suggested by McCabe in its original applications [14]: limiting the complexity of methods through the establishment of a maximum permitted value for cyclomatic complexity. When such value is reached for a method, it could be divided into smaller methods, aiming to reduce each one's complexity and size, resulting in an easier process of testing.

Moreover, the number of loops and nested loops were smaller in fully covered methods. Even if the identified difference is not outstanding, it is suggested for software developers to reduce its number in their implementations, once they increase a method's complexity for being, in a general case, a decision point.

VII. Concluding Remarks

In this paper we have presented an exploratory study aiming at identifying characteristics that may influence testers on reaching high coverage levels on structural testing criteria such as all-branches. We have explored 39 open source projects to compare the characteristics of methods whose branches were 100% covered with the methods whose branches were more than 50% and less than 100% covered.

In conclusion, we found out that there are some factors that have significant influence on the coverage results: it was demonstrated that cyclomatic complexity, size, number of constraints and number of constraint elements are strongly correlated to the branch coverage. The correlation between the studied characteristics shows that the cyclomatic complexity have high influence on branch coverage and it also has a positive influence on the number of constraints, which influences branch coverage as well. The number of constraints has a positive relationship with the number of constraint elements, but the cyclomatic complexity has only a moderate relationship with the number of constraint elements and the number of instructions.

We believe that this preliminary study is useful to provide some insight into how characteristics of software may influence its resultant coverage. Although the obtained results could be easily derived by experienced developers, it is essential to verify such concepts numerically, making use of statistical models to systematically prove correlations of certain metrics in the branch coverage.

Apart from the factors that clearly influence branch coverage, the complexity that emerge from the data type and the category type of the constraint elements is worthy investigating. The number of elements is moderately influenced by the cyclomatic complexity, but it would be interesting to perform further investigations to discover independent factors that influence branch coverage coming from the structure of the constraints, either by their data or category type, or complexity of the expressions, dependency of other constraints, and many other related factors. This study also motivates further and future investigations to identify which are the dependent and independent factors that causes the software to be partially covered. In future investigations we intend to compare methods from FULL and PARTIAL groups that share the same size and constraint complexity to discover whether there are more factors that can lead to high or low branch coverage.

Acknowledgments.: Marcelo M. Eler is partially financially supported by FAPESP (Sao Paulo Research Foundation, Brazil), grant 2014/08713-9. Camila F. Castro and Decio S. Oliveira Jr are partially financially supported by the Tutorial Education Program of the Brazilian Education Ministry (PET/MEC).

References

[1] R. S. Abdullah and M. H. Khan. Testability estimation of object oriented design:a revisit. *International Jounal of Advanced Research in Computer and Communication Engineering*, pages 3086–3090, numpages = 5,, Aug. 2013. ISSN 2319-5940.

[2] R. Bache and M. Mullerburg. Measures of testability as a basis for quality assurance. *Softw. Eng. J.*, 5(2):86–92, Apr. 1990. ISSN 0268-6961. doi: 10.1049/sej.1990.0011. URL http://dx.doi.org/10.1049/sej.1990.0011.

[3] L. Badri and F. Tour. An empirical analysis of lack of cohesion metrics for predictiong testability of classes. Jan 2011.

[4] M. Badri, F. Toure, and L. Lamontagne. Predicting unit testing effort levels of classes: An exploratory study based on multinomial logistic regression modeling. *Procedia Computer Science*, 62:529 – 538, 2015. ISSN 1877-0509. doi: http://dx.doi.org/10.1016/j.procs.2015.08.528. URL http://www.sciencedirect.com/science/article/pii/S1877050915026630. Proceedings of the 2015 International Conference on Soft Computing and Software Engineering (SCSE'15).

[5] B. Baudry and Y. L. Traon. Measuring design testability of a uml class diagram. *Inf. Softw. Technol.*, 47(13):859–879, Oct. 2005. ISSN 0950-5849. doi: 10.1016/j.infsof.2005.01.006. URL http://dx.doi.org/10.1016/j.infsof.2005.01.006.

[6] R. V. Binder. Design for testability in object-oriented systems. *Commun. ACM*, 37(9):87–101, Sept. 1994. ISSN 0001-0782. doi: 10.1145/182987.184077. URL http://doi.acm.org/10.1145/182987.184077.

[7] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to improve the testability of object-oriented code. *Softw. Pract. Exper.*, 33(7):637–672, June 2003. ISSN 0038-0644. doi: 10.1002/spe.520. URL http://dx.doi.org/10.1002/spe.520.

[8] S. Chidamber and C. Kemerer. A metrics suite for object oriented design. *Software Engineering, IEEE Transactions on*, 20(6):476–493, Jun 1994. ISSN 0098-5589. doi: 10.1109/32.295895.

[9] M. M. Eler, A. T. Endo, and V. H. S. Durelli. Quantifying the characteristics of java programs that may influence symbolic execution from a test data generation perspective. In *Proceedings of the 2014 IEEE 38th Annual Computer Software and Applications Conference*, COMPSAC '14, pages 181–190, Washington, DC, USA, 2014. IEEE Computer Society. ISBN 978-1-4799-3575-8. doi: 10.1109/COMPSAC.2014.26. URL http://dx.doi.org/10.1109/COMPSAC.2014.26.

[10] M. M. Eler, A. T. Endo, and V. H. Durelli. An empirical study to quantify the characteristics of java programs that may influence symbolic execution from a unit testing perspective. *Journal of Systems and Software*, 121:281 – 297, 2016. ISSN 0164-1212. doi: http://dx.doi.org/10.1016/j.jss.2016.03.020. URL http://www.sciencedirect.com/science/article/pii/S0164121216000868.

[11] G. Fraser and A. Arcuri. Sound Empirical Evidence in Software Testing. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 178–188, 2012.

[12] ISO. *International standard ISO/IEC 9126. information technology:Software product evaluation: Quality characteristics and quidelines for their use*. ISO, 1991.

[13] A. Kout, F. Toure, and M. Badri. An empirical analysis of a testability model for object-oriented programs. *SIGSOFT Softw. Eng. Notes*, 36(4):1–5, Aug. 2011. ISSN 0163-5948. doi: 10.1145/1988997.1989020. URL http://doi.acm.org/10.1145/1988997.1989020.

[14] T. McCabe, U. S. N. B. of Standards, and M. . Associates. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*. NBS special publication. U.S. Department of Commerce, National Bureau of Standards, 1982. URL https://books.google.co.uk/books?id=AzduMAEACAAJ.

[15] G. J. Myers. *The Art of Software Testing*. Wiley, New York, 1979.

[16] I. Rodrguez, L. Llana, and P. Rabanal. A general testability theory: Classes, properties, complexity, and testing reductions. *IEEE Transactions on Software Engineering*, 40(9):862–894, Sept 2014. ISSN 0098-5589. doi: 10.1109/TSE.2014.2331690.

[17] A. Tahir, S. G. MacDonell, and J. Buchan. Understanding class-level testability through dynamic analysis. pages 1–10, April 2014.

[18] J. M. Voas and K. W. Miller. Semantic metrics for software testability. *J. Syst. Softw.*, 20(3):207–216, Mar. 1993. ISSN 0164-1212. doi: 10.1016/0164-1212(93)90064-5. URL http://dx.doi.org/10.1016/0164-1212(93)90064-5.

[19] J. M. Voas and K. W. Miller. Software testability: The new verification. *IEEE Softw.*, 12(3):17–28, May 1995. ISSN 0740-7459. doi: 10.1109/52.382180. URL http://dx.doi.org/10.1109/52.382180.

[20] Y. Zhou, H. Leung, Q. Song, J. Zhao, H. Lu, L. Chen, and B. Xu. An in-depth investigation into the relationships between structural metrics and unit testability in object-oriented systems. *Science China Information Sciences*, 55(12):2800–2815, 2012. ISSN 1869-1919. doi: 10.1007/s11432-012-4745-x. URL http://dx.doi.org/10.1007/s11432-012-4745-x.