Calcuting Code Coverage Without Running the Tests: A Heuristic Based on Static Code Analysis

Mauricio F. Aniche, Gustavo A. Oliva, Marco A. Gerosa
Institute of Mathematics and Statistics
University of São Paulo
{aniche,goliva,gerosa}@ime.usp.br

ABSTRACT

bla bla

1. INTRODUCTION

One advantage of studies that make use of mining software repository techniques is that they can leverage a large quantity of projects and data. This huge amount of information allows researchers to validate hypotheses with more property.

Researchers can extract data from many different sources, such as source code repositories (a.k.a. version control systems), issue trackers, mailing lists, and so on. Most of these repositories store textual information, which is easy to extract but hard to process. In the case of source code, a particular problem arises: some metrics require the code to be compiled first. And compiling code is a complicated task. Each project has its own build process and required libraries. Furthermore, it is common that many of these libraries are not available by the time researchers obtain the source code.

An example of code metric that usually requires compiled code is code coverage. This metric calculates the percentage of production code that is covered by a set of unit tests. Popular tools, such as Emma[1] and Cobertura[2], instrument the code, execute the unit test suite, and then extract the information. As said before, when dealing with many different repositories, compiling and executing unit tests may not be feasible.

To sort this problem out, we conceived, implemented, and evaluated a heuristic to calculate code coverage without compiling the code first (i.e., without even running unit tests). In Section 2, we introduce the concept of code coverage. In Section 3, we present our heuristic. In Section 4, we describe the experiment design. In Section 5, we present the results we obtained and discuss them. In Section 6, we present the threats that may have influenced the validity of our study. In Section 7, we show related studies and compare them to our approach. Finally, in Section 8, we state our conclusions and plans for future work.

We found out that, although our tool implementation is not fully completed yet, the proposed heuristic on how to calculate code coverage using static analysis seems valid. The maximum error on the projects evaluated was around 11%.

[1]http://emma.sourceforge.net/
[2]http://cobertura.github.io/cobertura/

2. CODE COVERAGE

Code coverage measures the quantity of production code that is being tested by the test suite. When a unit test exercises a piece of code, developers say that the piece of code is covered. Many studies argue that code coverage is an important metric when dealing with software maintenance [5] [2] [4].

The common output of the tools is a percentage, which is calculated by the quantity of production code lines that are covered by at least one unit test divided by the total number of production code lines.

There are many different ways to calculate code coverage. The tool can check the number of lines (as mentioned before) that were executed, the number of instructions, the number of conditions, or even the number of executed branches.

As mentioned before, Emma and Cobertura are popular tools. They both use dynamic analysis to calculate the metric. They instrument the code, execute the unit test suite, and then discover which instructions were executed.

Besides taking a long time to calculate (as they need to execute the entire test suite), they require compiled code. As said before, when analysing a large quantity of repositories, compiling the code may not be an option. Researchers need an alternate way to calculate that.

3. THE PROPOSED HEURISTIC

Our heuristic is based on the idea that if a method has N different branches, then it should have at least N different unit tests. Then, with these two numbers (branches and number of unit tests), we calculate the ration between both. If a class contains more tests than it needs to, we round it to 100%. In Table 1, we show a few examples of combinations and the calculated coverage.

Table 1: Examples of unit tests, branches and the calculated coverage

# of Tests	# of Branches	Coverage in %
10	10	100%
20	10	100%
10	0	0%
5	10	50%
7	10	70%

As we wanted to avoid dynamic analysis, the only option was to apply static analysis to the source code. The first step was to estimate the number of branches a production class has. To do that, we decided to use McCabe's cyclomatic complexity [3]. McCabe's number basically shows the different number of execution paths per method. The listing 1 below exemplifies it. In that case, McCabe's number would be 2. As we are calculating code coverage in class method, we calculate the class' cyclomatic complexity by summing up McCabe's number of all methods.

```
public void doSomething(int a) {
  int total = 0;
  if(a>10) total += 1;
  return total;
}
```

Listing 1: McCabe's number to this method is 2.

The next step was to count the number of unit tests that invoke a specific production method. To do that, in a statically way, we count all the invocations that happen in the unit tests. As an example, in Listing 2, we consider the invocation to *calculateTaxes()* as a unit test to it. We then repeat this procedure to all unit tests.

```
class InvoiceTest {
  @Test
  public void shouldCalculateTaxes() {
    Invoice inv = new Invoice("Customer",
        5000.0);

    double tax = inv.calculateTaxes();
    double taxAgain = inv.calculateTaxes();

    assertEquals(5000 * 0.06, tax);
    assertEquals(5000 * 0.06, taxAgain);
  }
}
```

Listing 2: An example of a unit test

There are a few exception cases that are worth mentioning. As we are interested in the number of distinct unit tests per production method, even if a test invokes the same method twice, it counts as one. In the example above, *calculateTaxes()* is invoked twice, but we only mark that it is being tested by *shouldCalculateTaxes()*.

Also, if a production method makes use of many other methods in its implementation, we mark all of these methods as tested by the current unit test. In Listing 3, we show an example of the implementation of *calculateTaxes()*. In this case, we mark methods *calculateTaxes()*,*taxA()*, and *taxB()* as tested by the method *shouldCalculateTaxes()*, as they are all invoked by that unit test.

```
public double calculateTaxes() {
  double taxA = taxA();
  double taxB = taxB();

  return taxA + taxB;
}
```

Listing 3: Internal implementation of calculateTaxes()

We then calculate the ratio between the number of tests and McCabe's number. As all metrics are calculated in method level, and we are interested in code coverage at class level, we decided to sum up all McCabe's number and all unit tests that test a method in that class. In the Figure 1, we show the formula. Again, suppose that the class *Invoice* has McCabe's number equals to 10, and there are 6 unit tests that invoke methods that belong to it. In this case, **6 / 10 = 0.6**, which means **60%** of code coverage.

$$\frac{\sum(Numberoftestspermethod)}{\sum(McCabe'snumberpermethod)}$$

Figure 1: Formula used to calculate code coverage in our heuristic

The tool that analyses the source code and calculates all of it is freely available on Github[3]. The all data used in this study is also freely available[4].

4. EXPERIMENT DESIGN

To validate the heuristic, we decided to compare the code coverage calculated by it with the ones calculated by a dynamic analysis tool. We chose Emma as we had a previous experience with it. We had 3 projects as subjects, all of them developed by the same company, in Java. Two of them are web applications and one of them is a console application. In Table 2, we describe the size of each one.

Table 2: Quantity of classes and methods per project

	Classes	Production Methods	Unit Tests
Gnarus	769	1823	826
MetricMiner	225	1009	341
Tubaina	261	372	298

With both code coverage numbers (Emma's and ours), we then compared both to see how different they were. We used histograms and descriptive statistics. In Section below, we present the results found.

5. FINDINGS AND DISCUSSION

To do the analysis, we decided to use the number calculated by our heuristic minus the number calculated by Emma. It means that every time we find a zero, it means that both numbers were equal, and the heuristic was perfect. A negative number indicates that the heuristic calculated a smaller number than the real; a positive number indicates that the heuristic calculated a higher number.

In Figure 2, we show the histogram of the difference between Emma and our approach to the MetricMiner project. In the X axis, we have the difference, and in the Y axis, we have the frequence. By looking to this Figure, one can notice that most of the data are between 0 and 0.2. The median of the distribution is 0, and the mean is -0.1226. The first quartile is 0, and the third quartile is also 0.

[3]https://github.com/mauricioaniche/gelato2
[4]https://github.com/mauricioaniche/msr2014

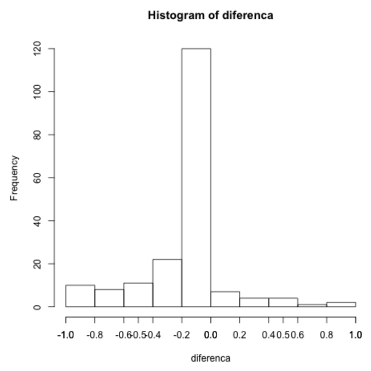

Figure 2: Our approach compared to Emma's in the MetricMiner

In Figure 3, from Tubaina project, the data is between 0 and -0.2. However, there are a few classes that were between -0.5 and -1.0. The median is -0.1603, and the mean is -0.3074. The first quartile is -0.6, and the third quartile is 0.

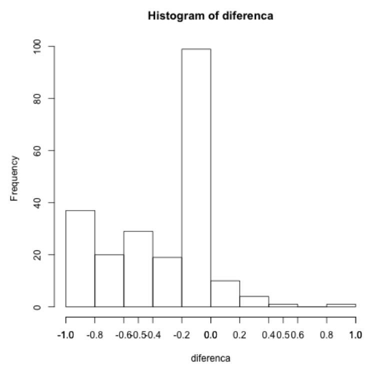

Figure 3: Our approach compared to Emma's in the Tubaina

In Figure 4, from Gnarus project, we can see that the data is very distributed between frequencies. The median is -0.3852, the mean is -0.3272. The first quartile is -0.5, and the third quartile is 0.

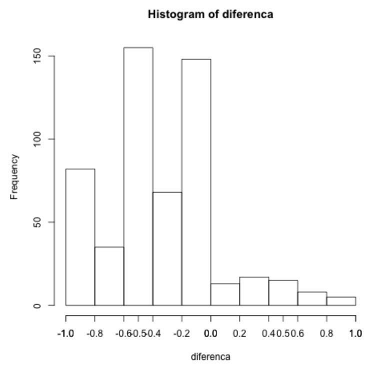

Figure 4: Our approach compared to Emma's in the Gnarus

By looking to the numbers above, we notice that the heuristic was good in the first project, in which the median was 0, regular in the second one, in which the median was -0.16. However, in the last project, the heuristic did not go well, as the median was -0.38.

Based only on these three projects, it is hard to affirm that the heuristic is valid. However, we decided to investigate closely on why some classes had such a bad performance. We found out that our compiler was not interpreting a few expressions. In Listing 4, we show an example of a misinterpreted test. In that case, the current implementation of the tool does not identify the invocation to *doX()* and *doY()* as both invocations come from a generic list.

```
List<SomeClass> list = obj.getList();
assertEquals(1, list.get(0).doX());
assertEquals(1, list.get(1).doY());
```

Listing 4: The tool does not identify invocations from a list

5.1 Isolating the heuristic

As we identified that the implementation contains problems that are confusing the number found, we decided to try to isolate the heuristic from the implementation. To do that, we decided to create an aspect to the code to make it print all methods that are invoked by a unit test. After executing the unit tests, we had an output like the one in Listing 5, which indicates the name of the test and the methods that were invoked.

```
test com.pack1.pack2.UnitTest.testA
com.pack1.ProductionClass.methodA
com.pack1.ProductionClass.methodB
com.pack1.ProductionClass.methodC
test com.pack1.pack2.UnitTest.testB
com.pack1.ProductionClass.methodD
com.pack1.ProductionClass.methodE
```

Listing 5: The output produced by the aspect

With this information in hand, we then had the real number of unit tests per production method. We then calculated the ratio, using McCabe's number. In Figures 5, 6, and 7, we show the histogram for MetricMiner, Tubaina, and Gnarus, respectively.

In MetricMiner, we can notice that most of the data are between -0.2 and 0.2. The median of the distribution is 0, and the mean is 0.1149. The first quartile is 0, and the third quartile is 0.2414. In Tubaina project, the data is between -1 and 0. The median is 0, and the mean is 0.0790. The first quartile is 0, and the third quartile is 0.1667. In Gnarus project, we can see that most of data are between -2 and 0. The median is 0, the mean is -0.0152. The first quartile is -0.7273, and the third quartile is 0.

Based on the numbers we obtained, we believe that the heuristic has an acceptable performance. As the maximum mean error was 0.1149 found in these 3 projects, which means 11%, we do not think that it will be a problem when using it in large software repositories.

6. THREATS TO VALIDITY

There are a few threats to validity to this research. The first one is that our implementation still does not fully work: implementing a parser is not an easy task, and one should spend a lot of effort to completely interpret a source code. However, as we showed, the heuristic itself seems valid.

We only evaluated three projects that belong to the same company. It means that these projects follow basically the

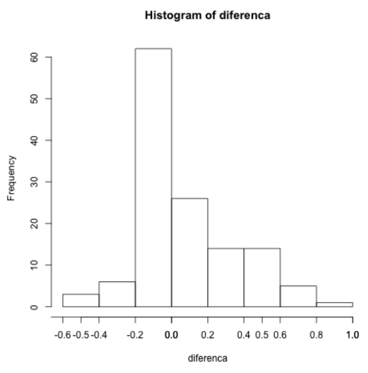

Figure 5: Our second implementation compared to Emma's in the MetricMiner

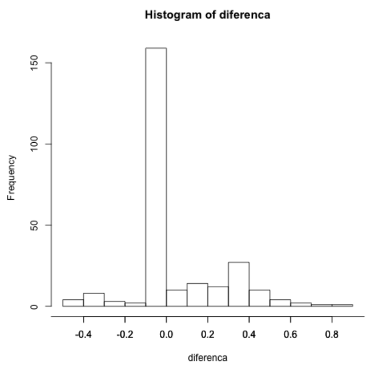

Figure 6: Our second implementation compared to Emma's in the Tubaina

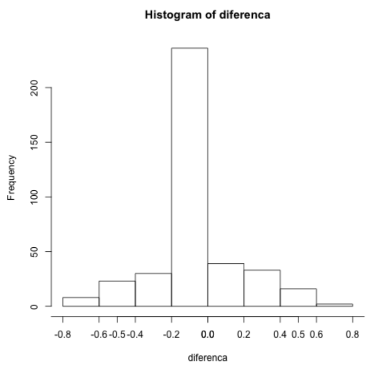

Figure 7: Our second implementation compared to Emma's in the Gnarus

same structure, development and testing rules. That may bias the results. In a future work, we should run the experiment in many different projects.

7. RELATED WORK

Many studies discuss the importance of code coverage when analysing code quality. Sebastian et al [5] says that many software development practices and tools are based on this number. However, he argues that developers usually calculate it for a single version of the system, and perform analysis on future versions without recalculating the numbers. He shows that even relatively small modifications on the source code can affect the code coverage, and the impact of the change on the metric is hard to predict.

To the best of our knowledge, there is only one study that discusses the calculation of code coverage through static analysis. Tiago and Visser [1] proposes a technique that uses slicing of static call graphs to estimate the dynamic test coverage. In their approach, they define method coverage as the ratio between covered and defined methods per class. They showed that, at system level, the approach proved to be satisfactory. In package and class level, the difference between the real and the calculated coverage was small in most cases.

This approach is slightly different from ours. We use McCabe's number to estimate the minimum number of tests a class should have. We also are more interested on calculating code coverage in class level, while they are more worried to do it in system level.

Also, creating a call graph is not simple. That takes more effort than just counting the invoked methods, as we suggested.

8. CONCLUSION AND FUTURE WORK

Code coverage is an important metric to analyse software evolution. However, it is not easy to be calculated, as most tools require compiled code. In this study, we discuss an heuristic to calculate it using static analysis. Apparently, our heuristic can be used in mining software repositories studies, as the maximum error was around 11%.

A future work would be to run the study on more projects from different companies and domains.

9. ACKNOWLEDGMENTS

We thank Caelum Ensino and Inovação for allowing us to run the study in their environment, as well as supporting the development of the tool.

10. REFERENCES

[1] T. L. Alves and J. Visser. Static estimation of test coverage. *IEEE International Working Conference on Source Code Analysis and Manipulation*, 2009.

[2] e. a. F. Del Frate, P. Garg. On the correlation between code coverage and software reliability. *Sixth International Symposium on Software Reliability Engineering*, 1995.

[3] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[4] W. E. W. Mei-Hwa Chen, M. R. Lyu. An empirical study of the correlation between code coverage and reliability estimation. *Proceedings of the 3rd International Software Metrics Symposium*, 1996.

[5] G. R. Sebastian Elbaum, David Gable. The impact of software evolution on code coverage information. *IEEE International Conference on Software Maintenance*, 2001.