Concordia University

Engineering and Computer Science

Project Proposal

SOEN-6611 : SOFTWARE MEASUREMENTS

TEAM K - May 22, 2019

Submitted to :

Jinqiu Yang

jinqiuy@encs.concordia.ca

Team Information

NAME	STUDENT ID	E-MAIL
ANAND KACHA	40047673	a_kacha@encs.concordia.ca
SHAREEN ALI	40075802	a_hareen@encs.concordia.ca
MANPREET SINGH	40083737	si_preet@encs.concordia.ca
SHIVANI JINDAL	40071296	shivani1995jindal@gmail.com
JASMEET KAUR	40088712	k_asmeet@encs.concordia.ca

Selected Metrics

- Metric 1 (Test coverage metric): Statement Coverage
- Metric 2 (Test coverage metric): Branch Coverage
- Metric 3 (Test suit effectiveness): Mutation Score
- Metric 4 (Complexity Metric): McCabe Complexity
- Metric 5 (Software Maintenance Metric): Adaptive Maintenance Effort Model
- Metric 6 (Software Quality Metric): Post-release defect density

Test Coverage Metric

Code coverage metrics, largely derived from metrics used in software testing, identify which structure classes in the hardware description language (HDL) code to exercise during simulation. Hence, also called Test coverage metrics.
More sophisticated code-based metrics used in hardware verification are branch, statement, and path coverage. These metrics involve the control flow through the HDL code during simulation and are best described by the control flow graph (CFG) corresponding to that code. Control statements (the statements from which control can jump to one of several places) constitute the branching points in the CFG—for example, if, case, or while statements.

Metric 1 : Statement Coverage

Statement coverage is a white box testing technique, which involves the execution of all the statements at least once in the source code. It is a metric, which is used to calculate and measure the number of statements in the source code which have been executed. Using this technique we can check what the source code is expected to do and what it should not. It can also be used to check the quality of the code and the flow of different paths in the program. It is also known as Segment Coverage or Line Coverage.

Statement coverage = (Number of statements executed/Total number of statements) *100%

Pros Of Statement Coverage :
- Confirms Code Quality
- Checks The Flow Of Statement

Cons Of Statement Coverage :
- Covers True Conditions Only
- Cannot Check If Loop Reaches Termination

Metric 2 : Branch Coverage

Branch coverage is a testing method, which aims to ensure that each one of the possible branch from each decision point is executed at least once and thereby ensuring that all reachable code is executed. That is, every branch taken each way, true and false. It is also known as All-edge coverage or Decision coverage.

Branch Testing = (Number of decisions outcomes tested / Total Number of decision Outcomes) x 100 %

Pros Of Branch Coverage :
- To validate that all the branches in the code are reached.
- To ensure that no branches lead to any abnormality of the program's operation.
- It eliminate problems that occur with statement coverage testing.

Cons Of Statement Coverage :
- This metric ignores branches within boolean expressions which occur due to short-circuit operators.

Metric 3 : Mutation Score

Software testing is an important part of the software engineering lifecycle process.It plays a pivotal role in the quality assurance of the software product.During software maintenance, testing is a crucial activity to ensure the quality of program code as it evolves over time. With the increasing size and complexity of software, adequate software testing has become increasingly important.In order to measure the effectiveness, the quality of test suites developed is measured. Good software testing have software tests than detect and find faults, if a test cases is unable to detect fault it will be hard to build a good software. This is done by Mutation Testing.
It's a kind of Whitebox-testing used to develop the hidden test cases by modifying code and check if it could detect the error probability. Furthermore, changes in code statements are

made at the lower level so that the meaning of the code remains neutral. The changed statements are the called the mutant and the score derived is known as the mutation score. Thus, to know the failed mutant codes, the statement should be vigorous.

Mutation Score can be defined as the number of Mutants that were killed divided by the total number of Mutants multiplied by 100.

Pros of Test Suite Effectiveness :
- The entire source code is covered.
- Program mutants are thoroughly tested.
- The test reveals all the ambiguities in the source code.

Cons of Test Suite Effectiveness :
- In the mutation testing since the program's mutants should be separately generated, it becomes extremely costly and time consuming process.
- It is not automated.
- It is not applicable to blackbox testing as the source code is not changed.

Metric 4 : McCabe Complexity

Cyclomatic Complexity is a mathematical technique to modularize the program. Thomas McCabe suggested that every possible control flow graph has a finite number of independent paths. He derived the following formula to calculate the cyclomatic complexity of the software,
CC = E - N + 2P or CC = D + 1

Where E is the number of edges in the graph, N is the number of nodes in the graph, P is the number of connected components in the graph, D is the control predicates in the graph.

Pros of Cyclomatic Complexity :
- It calculates the total possible linearly independent paths executed for a component.
- The number of independent paths can be used to determine the number of test cases needed to cover the component.

Cons of Cyclomatic Complexity :
- It considers all the control predicates as same level of complexity. E.g. for loop and if statement can have different levels of complexity.
- It doesn’t take the nesting into account when calculating from the independent paths.
- It does not consider the flow of data for the particular component.

Metric 5 : Adaptive Maintenance Effort Model

It is a metric derived using the regression model on the number of lines in the code that changed.
In software maintenance, when the software has to undergo the change in the system, the team often underestimates the time and effort required to make the changes. This also leads to a lack of validation, acceptance, planning, estimating, and maintenance in the software maintenance cycle. To compute this metric, we derive the estimation model using regression analysis on the data that we collected. Then we use the leave one out approach to validate the model.
For the calculations, we can treat the change in the LOC as a parameter to the regression model. The DLOC we can determine from the project's version control system based on the diff in versions of the file. We then can determine the data points from the available data. Once we determine the DLOC, we can move forward to calculate the maintenance effort using the following formula:
E = -40 + 6.56DLOC

Pros of Adaptive Maintenance Effort Model :

- The metric is calculated using an adaptive method. It can also be used to predict the parameters changing.
- With the regression model, we can determine that relationship between the variables used and effort.

Cons of Adaptive Maintenance Effort Model :

- In the predictive models we generally split the training/testing data, the output depends on the distribution of the data as well. For different distribution of data, we might experience the change in results.

Metric 6 : Post Release Defect Density

Defect is what is the deviation observed from an expected behaviour. Once the software is released to the customer and then if defects are encountered, is referred as post-release defects. Defect density is the total count of defects found per 1000 lines of source code in a software after software has been released. Defect density is used as the indicator for the Product Quality.

Let DD be Defect Density being calculated by dividing defect count as Dc by size of software as SS.
Defect Density(DD) = Defect Count(Dc)/ Size of the Software(SS)

When defect density is evaluated to be zero, it is the indicator of the best product quality delivered to the customer.

Pros of Post Release Defect Density :

- The post-release Defect Density is used to track the quality of the product in order to improve it and helps in defect reduction.
- The metric is to map against software component size with respect to defects being detected after software is released.
- This metric help to identify defect prone/high risk components, allowing the concentration of limited resources into areas with the highest potential return on the investment.

Cons of Post Release Defect Density :

- It does not follow uniform distribution, as there's high number of defect is specific area of code due to ineffective testing.[2]

Correlation Analysis

Coverage Metric and Test-Suite Effectiveness

As per the results of paper "Coverage Is Not Strongly Correlated with Test Suite Effectiveness", , there is a moderate to high correlation between the effectiveness and the coverage of a test suite when the influence of suite size is ignored.
The correlation between coverage and effectiveness drops when suite size is fixed. After this drop, the correlation typically ranges from low to moderate, meaning it is not generally safe to assume that effectiveness is correlated with coverage. The correlation is stronger when the non-normalized effectiveness measurement is used. Additionally, the type of coverage used had little influence on the strength of the relationship.

Coverage Metric and Software Quality Metric

Statement Coverage confirms the quality of the code by ensuring that every statement is executed at least once. It also checks each line of the code to ensure that each statement actually performs the task that it should ideally be performing. If the statement does not execute as per requirement, then it can be fixed to prevent future trouble.

A correlation between the number of faults detected in a version and the coverage of the program constructs of the version is that the better a program is covered during testing, the more faults will be detected, unseemly bugs have been found.In the effort of using testing coverage measures to derive high quality test data, it is implicitly assumed that a good test has a high data flow coverage score, namely, software with higher reliability. Requirements traceability gives you a reliable way to build good test cases. So, let's extend that for a minute

to think about why that is. If you only write test cases that can trace back to a requirement, you're effectively eliminating any redundant, unnecessary test cases. This improves the efficiency of your team's testing efforts. Example, if you turn that around, what you'd get is this: if you ensure 100% of your requirements are covered by test cases, then you have all the test cases you need. Interesting, right? Then again, how do you ensure you have 100% accurate requirements? What if some requirements are incorrect, or missed? Well, as they say, "A high-quality product built on bad requirements is a poor quality product."

The focus then shifts to ensuring you have high-quality user stories that cover all functional and non-functional requirements. A cursory search on the internet will tell you what you need to ensure all your requirements are captured adequately and effectively.

Maintenance Effort and Post Release Defect Density

The more changes we make in the source code, the probability of increasing the defect density increases. In other words, the more efforts we make for the maintenance (in terms of change in lines of code and effort), the more number of defects are likely to increase.

Test Suite Effectiveness and Post Release Defect Density

The more effective test suite is, the more chances of detecting the defects at an earlier stage, that will decrease the post release defect density.

Complexity and Test Suit Effectiveness

The more effective test suit is, the less complex the source code becomes.

Related Work

Metric 1 and 2 : Test Coverage Metrics

Branch Coverage and Statement Coverage

- According to paper, "Coverage Is Not Strongly Correlated with Test Suite Effectiveness" who tests on Java projects there is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, they found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Their results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.
- According to paper, "Coverage Metrics for Functional Validation of Hardware Designs". Each coverage metric has an underlying error, or fault, model. In some instances, this connection is obvious, such as with toggle-coverage or stuck-at faults, assertions, and

assertion violations. In other cases, the error model is not made explicit, and only a loose connection exists between the metric and the errors it is intended to catch. For instance, a coverage metric requiring that certain instruction sequences be exercised may be intended to catch all erroneous behavior of a pipeline's interlock logic. Some coverage metrics are defined directly by reference to an error model. These fault based metrics mimic mutation coverage for software testing and manufacturing testing of hardware. Fault coverage, therefore, is the percentage of faults that the test suite would have detected.

- For most reasonable definitions of coverage metrics, the effort invested in deriving the metrics and measuring them pays off in the form of better error detection. The value of increased confidence in the design's correctness almost always outweighs the overhead of measuring coverage. Because this confidence depends on the connection between bug classes and coverage metrics.
- According to paper, "Achieving Software Quality with Testing Coverage Measures: Metrics, Tools, and Application". In the effort of using testing coverage measures to derive high quality test data, it is implicitly assumed that a good test has a high data flow coverage score. This hypothesis requires that we show that good data flow testing implies good software, namely, software with higher reliability. One would hope, for example, that code tested to 85% c-uses coverage would exhibit a lower field failure rate than similar code tested to 20% c-uses coverage. The establishment of a correlation between good data flow testing and a low rate of field faults (or that there is none) is the ultimate and critical test of the usefulness of data flow coverage testing.

Metric 3 : Mutation Score

According to paper, "Coverage Is Not Strongly Correlated with Test Suite Effectiveness"
The coverage of a test suite is often used as a proxy for its ability to detect faults. However, previous studies that investigated the correlation between code coverage and test suite effectiveness have failed to reach a consensus about the nature and strength of the relationship between these test suite characteristics. Moreover, many of the studies were done with small or synthetic programs, making it unclear whether their results generalize to larger programs, and some of the studies did not account for the confounding influence of test suite size.

There is a low to moderate correlation between coverage and effectiveness when the number of test cases in the suite is controlled for. In addition, we found that stronger forms of coverage do not provide greater insight into the effectiveness of the suite. Our results suggest that coverage, while useful for identifying under-tested parts of a program, should not be used as a quality target because it is not a good indicator of test suite effectiveness.

- Created faulty versions of one or more programs by manually seeding faults, reintroducing previously fixed faults, or using a mutation tool.

- Created a large number of test suites by selecting from a pool of available test cases, either randomly or according to some algorithm, until the suite reached either a pre-specified size or a pre-specified coverage level.
- Measured the coverage of each suite in one or more ways, if suite size was fixed; measured the suite's size if its coverage was fixed.

Metric 4 : McCabe Complexity

The complexity of the software can be described with two different approaches, a quantitative approach, and the structural approach. The quantitative metrics include the complexity which considers the source lines of code. It is the oldest measure of software that is used. Using different methods on top of this metric, we can also estimate the efforts in hours.

Another method for measuring complexity is Halstead's metric. Maurice Halstead introduced this measure with the use of a number of distinct operators and operands as well as the total number of operators and operands. Using different methods on top of this metric as well, we can estimate the efforts and other dimensions of software development.

The cyclomatic complexity is the commonly used metric in software engineering. It is based on the control flow structure of the program. Each procedure's statements are considered as a graph. And the complexity is measured by linearly independent paths inside the graph.

An improvement of this complexity has also been proposed by Ayman Madi, Oussama Kassem Zein and Seifedine Kadry using Total Cyclomatic Complexity and Coupled Cyclomatic Complexity. They calculate the intra-modular complexity which involves the entire target of measuring complexity. It is generally applicable in the design phase.

Metric 5 : Adaptive Maintenance Effort Model

There has been a significant number of effort estimation models in practice. The commonly referred is the Constructive Cost Model, which supports the cost estimation, effort, and timeline for the project. Some model suggests the use of estimated cost or/and the algorithmic cost and/or the function points as we covered in the lecture. The relevant papers related to such work cited in the references.

With the growing era of Machine Learning, there are recently surfaced models that could be used to measure the efforts in Software Maintenance. Gray and MacDonell used ML-based algorithms such as Artificial Neural Network, Regression Analysis, and a measurement model based on function point, and also the fuzzy logic. With all their analysis, they concluded that the model that uses fuzzy logic proved to perform well with various inputs,

but it gave a bit poor accuracy compared to the Artificial Neural Network. And the research has shown that the stepwise regression with bi-directional elimination method provides better accuracy than the linear regression.

Metric 6 : Post Release Defect Density

According to the research work, "Estimate the post-release Defect Density based on the Test Level Quality", the author had a focus on today's software industry, that concentrates on enhancing the quality of a software product, which can be done during testing and development phase by improving the code quality. But to measure the effectiveness of code, post release defect density is considered an important metric that can be calculated by author "as the number of detected defects per 1000 lines of source code." Then it was concluded that the post release defect density of a software is decreased to half of its value, when test level quality is increased from lower level. Thus they follow an inverse relation and hence, defect density is evaluated as an important factor in determining intrinsic product quality.

Open Source Systems :

1. Groovy
    - Lines of Code : 356k
    - Version : 2.5.7
    - Issue Tracking System : https://issues.apache.org/jira/projects/GROOVY/issues/GROOVY-9076?filter=allopenissues
    - Repository: https://gitbox.apache.org/repos/asf?p=groovy.git;a=summary
2. Openfire
    - Lines of Code : 162k
    - Version : 4.3.2
    - Issue Tracking System : https://issues.igniterealtime.org/projects/OF/issues/OF-1594?filter=allopenissues
    - Repository: https://github.com/igniterealtime/Openfire
3. Spring Framework
    - Lines of Code : 1.3M
    - Version : 5.1.7
    - Issue Tracking System : https://github.com/spring-projects/spring-framework/issues
    - Repository: https://github.com/spring-projects/spring-framework
4. Apache Bookkeeper
    - Lines of Code : 131k
    - Version : 4.9.2
    - Issue Tracking System : https://issues.apache.org/jira/projects/BOOKKEEPER/issues/BOOKKEEPER-807?filter=allopenissues

- Repository: https://github.com/apache/bookkeeper

5. Angular
   - Lines of Code : 908k
   - Version : 8.0.0
   - Issue Tracking System : https://github.com/angular/angular/issues
   - Repository: https://github.com/angular/angular

Resource Planning

Team Member	Responsibility
Anand Kacha	Metric 5 (Software Maintenance Metric)
Shareen Ali	Metric 4 (Complexity Metric)
Shivani Jindal	Metric 1 and 2 (Test coverage metric)
Manpreet Singh	Metric 3 (Test suit effectiveness)
Jasmeet Kaur	Metric 6 (Software Quality Metric)

References

[1] Laura Inozemtseva and Reid Holmes, School of Computer Science University of Waterloo Waterloo, ON, Canada. Coverage Is Not Strongly Correlated with Test Suite Effectiveness http://www.linozemtseva.com/research/2014/icse/coverage/coverage_paper.pdf

[2] Serdar Tasiran, Compaq Systems Research Center Kurt Keutzer University of California, Berkeley. Coverage Metrics for Functional Validation of Hardware Designs https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=936247&tag=1

[3] Qiushuang Zhang and Ian G. Harris, Department of Electrical and Computer Engineering University of Massachusetts Amherst. A Data Flow Fault Coverage Metric For Validation of Behavioral HDL Descriptions https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=896500&tag=1

[4] Statement Coverage, Released on May 21, 2016 http://www.professionalqa.com/statement-coverage

[5] J. R. Horgan, Saul London, Michael R. Lyu Bellcore. Achieving Software Quality with Testing Coverage Measures: Metrics, Tools, and Applications https://www.researchgate.net/profile/Michael_Lyu/publication/267376138_Achieving_Software_Quality_with_Testing_Coverage_Measures_Metrics_Tools_and_Applications/links/5461d7e30cf27487b4530cb6/Achieving-Software-Quality-with-Testing-Coverage-Measures-Metrics-Tools-and-Applications.pdf

[6] Statement Coverage in Testing, Released in 2014 https://www.zyxware.com/articles/4161/what-is-statement-coverage-in-testing

[7] Ulf Eriksson, DevOps. Measuring Code Quality With Test Coverage Metrics. Released on 31 may, 2017 https://dzone.com/articles/measuring-code-quality-with-test-coverage-metrics

[8] What is Branch Coverage or Decision Coverage? Its advantages and disadvantages http://tryqa.com/what-is-decision-coverage-its-advantages-and-disadvantages/

[9] Laura Inozemtseva and Reid Holmes, School of Computer Science University of Waterloo Waterloo, ON, Canada. Coverage Is Not Strongly Correlated with Test Suite Effectiveness http://www.linozemtseva.com/research/2014/icse/coverage/coverage_paper.pdf

[10] Thomas J McCabe. A Complexity Measure. I", IEEE Transactions on Software Engineering.

[11] M. Halstead, "Elements of Software Science", North Holland,

[12] Ayman Madi, Oussama Kassem Zein and Seifedine Kadry. "On the Improvement of Cyclomatic Complexity Metric". International Journal of Software Engineering and Its Applications

[13] Jane Huffman Hayes, Sandip C Patel, Liming Zhaom, "A Metrics-Based Software Maintenance Effort Model". Eighth European Conference on Software Maintenance and Reengineering, 2004. CSMR 2004. Proceedings.
[14] Boehm, B., Horowitz, E., Madachy, R Reifer, D., Clark, B.K., Steece, B., Brown, A.W., Chulani, S., and Abts, C. Software Cost Estimation with Cocomo II, Prentice-Hall 2000.

[15] Albrecht, A. J. Measuring Application Development Productivity. Proceedings SHARE/GUIDE IBM Applications Development Symposium, Monterey, CA., Oct 14-17, 1979.

[16]Mukhopadhyay, T., and Kekre, S. Software Effort Models for Early Estimation of Process Control Applications. IEEE Transactions on Software Engineering, Volume 18, Number 10, 1992, 915-924.

[17] Gray, A., and MacDonell, S.G. Applications of fuzzy logic to software metric models for development effort estimation. Proceedings of the 1997 Annual Meeting of the North American Fuzzy Information Processing Society - NAFIPS. Syracuse NY, IEEE (1997) 394-399.

[18] Mendes, E., and Mosley, N. Comparing effort prediction models for web design and authoring using boxplots. Proceedings of the 24th Australasian conference on Computer science, Gold Coast, Queensland, Australia, 2001. ACM International Conference Proceeding Series, 125-133.

[19] Vinke, L. (2011). Estimate the post-release Defect Density based on the Test Level Quality. Retrieved February 09, 2019, from https://research.infosupport.com/wp-content/uploads/2017/08/MasterThesis-LammertVinke-Final.pdf

[20] Chapter 4, Software Quality Metrics Overview, https://pdfs.semanticscholar.org/15b8/87c06021c21821807e614541b8691abd0af9.pdf