Empirically Comparing the Quality of Automatically Generated and Manually Written Test Suites

Jeshua Kracht, Jacob Z. Petrovic,
Kristen R. Walcott-Justice
Department of Computer Science
University of Colorado at Colorado Springs
Email: {jkracht, jpetrovi, kjustice}@uccs.edu

René Just
Computer Science & Engineering
University of Washington
Email: rjust@cs.washington.edu

Gregory M. Kapfhammer
Department of Computer Science
Allegheny College
Email: gkapfham@allegheny.edu

***Abstract*—**

I. Introduction

In Software Engineering, testing the code is one of the most important parts of the development cycle. Often the quality of the tests dictate the quality of the software. (Need Reference) Test suites are created through a number of different methods. In most instances the test engineer will manually write the test cases. However, manual creation of test cases requires large amounts of time and effort. Although this gives developers the control over the quality of the tests, the cost of the time and effort required to produce these test suites can be too high to implement on established projects. In recent years, researchers have looked into a new way to reduce the cost and automate this process for developers. A new alternative has arisen as a potential solution to this issue – auto-generated test suites.

Manually generated test suites allow developers to spend more time and effort on improving the quality of a test to insure it captures the full depth of the code. With a focus on the quality of tests, the work needed to generate a high coverage with a large number of test cases becomes expensive. As programs become more complex, so do the test suites that test them. As the program complexity rises, manually written test cases can become expensive and labor intensive when applied to highly complex features . [2].

Additionally, there is no standard for writing test cases. This can lead to inconsistent and subjective methods of writing test cases. The cost of time and effort put into writing test cases leaves less resources for developers to invest into increasing the coverage of the test suite.

Auto-generated test suites have one goal in mind: reduce the cost of time and effort to write tests. The test suites themselves are generated from many different automated test suite generators [3], [10], [13]. These programs use a consistent standards to produce test cases, removing the subjectivity and styles one may find in manually generated tests. Automated test generation also reduces the time and effort of creating large amounts of test cases to increase the breadth of code coverage. This allows test suites to be built faster without the effort of a developer. However, with the developer removed from the test case generation, the quality may be lost in the creation of the test cases. If the quality of test cases are shallow, then certain special cases may be missed, despite a large code coverage.

Every developer's desire would be to have a faster, consistent, and cost effective way of testing their code. Most importantly though, an assurance that the tests cases still maintain a high quality received from manually written tests. The two choices developers have in testing is either manual or auto-generated test cases. When comparing automated and manual testing, automated tests are good at breadth but much less at depth [9].The question that engineers must ask then is whether or not the auto-generated test suites produce the same quality of test cases as those written manually by a test engineer.

The quality of a test suite can be measured by its depth and breadth of coverage. Although the breadth, measured in terms of coverage may be extensive, more intricate and complicated test cases may not be covered by many available programs. It becomes difficult to determine what metrics can be used to establish the true definition of "depth" and "breadth". Mutation scores are traditionally used to assess the quality of test suites by inserting artificial defects (mutations) into programs [6]. Based on the mutation score given by a mutation test suite, testers will determine the quality of the tests covering their code. However, when comparing automated and manually written tests, researchers need additional metrics to compare the two processes. For instance, what is the difference in creation time and execution time for an auto-generated versus a manually written test suite? If test creation takes less time, but produces poor quality test cases, then the cost of using these tools in terms of quality may outweigh the benefits received in time. If the automated generated tests has higher code coverage but the execution time is long, then the cost of time may be too high.

Developers must also consider code coverage as producing more tests may increase the overall code coverage. Yet there is a possibility that more tests may be produced, but the coverage is small because the tests are redundant. In evaluating the time to create tests, time to execute tests, code coverage, code complexity, and mutation score, we can compare the quality, time, and effort between manual and automated test suites.

This paper observes whether the selected automated tools maintain the needed quality and coverage of manually written test suites. First impressions of auto-generated tests may leave more to be desired in their simplistic and current underdeveloped state. However, there have been few tests to actually determine the usefulness of automated tools. This research attempts to answer questions lingering over the differences in modern automated tools and traditional manual testing.

Measuring with mutation score, code coverage, complexity, and time to generate the test suite, this paper addresses how auto-generated tests differ from manually generated test cases, and if relying on these tools can retain the standard of quality that manually written test suites produce. Furthermore, the paperl compares metrics acquired from auto-generated tests to manually written tests to determine if auto-generated tests provide any advantages over to manually written test suites.

The main contribution of this paper is to reveal the differences and techniques used among different automated test suites. With these results, this paper remarks the benefits and drawbacks may occur when relying on automated generated test suites. This paper evaluates Evosuite and CodePro, two free automated-test tools that will generate the JUnit test suites. In Section II, the paper first establishes the fundamentals of automated test generation, mutation analysis, and genetic algorithms. In Section III, the paper describes process for the evaluation each of these programs in comparison to manually written test suites, and in comparison together. The paper compares the test suites by mutation score, code coverage, complexity, and the time it takes to create the test suite. In Section IV, the paper compares other related work and research to this study. Finally, in Section V, the paper concludes the results and discusses future work possibilities.

II. Background

A. Automated Test Suite Generation

Software testing has evolved into a separate branch of software engineering, with a move towards a more involved test system involved in the development process [7]. With the increase cost of time and effort, the idea of Automated tests suites began in the 1990's when Test Driven Design (TDD) began to draw the attention of software developers. TDD development advocated for a the more involved form of testing, with a focus on creating unit tests for code during development to insure the code was always passing tests thereby improving the quality of the code [1]. Although reliability of the code increased, the cost of time and effort to manually write the test cases increased as the programs became more complex over time [2].

New tools arrived in he 2000's to mitigate this difficulty with the arrival of automated test generators. Unfortunately, these test suite generators were in an early and underdeveloped state upon their arrival, and were largely ignored by the software engineering industry. As time has passed, the automated test suite generators have matured much past their initial states.

Automated test suite generators are usually a plugin to an IDE or a command line tool. Tests are generated by evaluating the source code, and writing new test suite files that can be run to test the program. Tests can be generated deterministically or non-deterministically. CodePro for example, generates the same set of tests for programs, whereas Evosuite may generate a slightly different test suite if ran multiple times. The deterministic nature of test suites can make them reliable and consistent, whereas the non-deterministic generation will seek to cover more edge cases in code, which may pose NFE incomplete problems. In the case of Evosuite, tests are evolve using Mutation-based assertion, incorporating a genetic algorithm to dynamically generate tests [3].

B. Mutation Analysis

Mutation analysis is the process of evaluating the quality of test cases in a test suite. The objective is to identify test cases that either need to be added or updated to better test the code. This is done by injecting Artificial defects, or mutants, into the code [6].The test cases are then run against the fault-injected mutants. The mutants then expose errors in the test cases, and accordingly a mutation score is calculated. The mutation score represents the number of failed mutants to passing mutants. A higher mutation score is better because it indicates that the overall quality of the test cases are better.

This research utilizes the mutation analysis tool MAJOR to receive a mutation score from each of the test suites. MAJOR executes mutation analysis in the java compiler, generting mutants into JUnit test cases with a conditional approach [8]. An abstract syntax tree is generated to capture the original version of the program in the same basic block. MAJOR will execute the test suite, provide a mutation coverage, and then reorder the test cases based upon the execution time of test cases.

C. Genetic Algorithms

Genetic algorithms are a subset of Evolutionary Algorithms [11]. Based off a data set, or population, the algorithm will select the most fit element represented by a binary string. Genetic Algorithms modify the population overtime to create a more fit set of elements. As the population evolves, the fitness of the elements increase. The goal is to output the most fit element. In complex problems, this may take extended periods of time, as it is a nondeterministic algorithm. Limits on fitness are often set to allow the algorithms to finish within a shorter time.

This technique is used in test suite generation in Evosuite to evolve the test cases. First, Evosuite generates a random set of tests to set up for mutation analysis.Then the tool uses a cover criterion based on branch coverage to create a fitness function for each test case [4]. Finally, mutation analysis is used to evaluate the fitness of the test suite. The test suite dynamically evolves over time to improve the quality of the test.

III. Evaluation

A. Key Evaluation Goals

We are interested in the following research questions.

Q1. Do automated generated unit tests retain the same quality as manually written unit tests?

Q2. What is the cost of time and effort difference in using automated tests versus manually written tests?

Q3. Can developers learn to manually write JUnit tests better with the help of automated generated suites?

Q4. Do Mutation scores and Coverage correlate in any way?

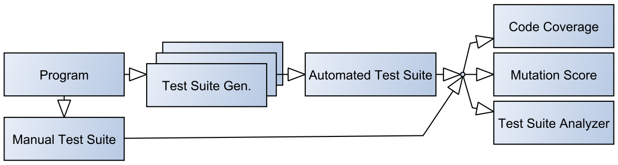

Fig. 1: Test and Mutation Process

B. Experiment Design

All experiments were performed on GNU/Linux workstations with kernel 3.2.0-44, a 2 GHzIntel Corporation Xeon E5/Core i7 processor and 15.6 GB of main memory. The unit tests were generated in the Java programming language, for both manual and automated tests. Figure 1 provides an overview of the test generation implementation with edges between interacting components. First we identified ten programs in the SF110 that have manually written tests. Table I provides a list of the selected SF110 programs with their respective lines of code and cyclomatic complexity. After identifying these programs, we use the automated test tools Evosuite and Codepro to generate test suites. We then took the manual and automated generated test suites and used several different tools to collect the metrics. MAJOR is used find the mutation score of the program, Jacoco is used to attain the branch coverage, and JavaNCSS measures the lines of code and the cyclic complexity of the programs. Time to create the tests was also recorded manually to compare Evosuite and Codepro. Time to create the manual tests was available with the SF110.

Because Evosuite uses a genetic algorithm to generate unit tests, we generated the tests for Evosuite ten times for each program. With multiple datasets, we can more accurate results with the standard deviation between mutation score, coverage, and time. Codepro uses a deterministc algorithm to generate the tests, so multiple test suite generations were not needed. With results gathered from Evosuite, Codepro, and the manually written tests, we then compared the scores based upon complexity, time to generate test suite, mutation score, lines of code, and branch coverage.

For the graphs, we attempted to find the best fit line to accurately represent to results. It is difficult to find a truly best fit line that gives an overall statistical overlook of the data. This is due to the varying nature of both Evosuite and the manually written tests. Evosuite uses a genetic algorithm, which results in varied results for each iteration. This is why as mentioned before, we run ten generations of the test suites and calculate an average for the standard deviation. Manually written test suites are written by different developers. These varying factors result in low R2 scores, which correlate to how well the best fit line actually represents the data. However, CodePro generated a mutation score of 0 in some cases. We removed these results for CodePro results because it skewed the data in a way that was not graphically indicative of the trends we discovered with CodePro.

TABLE I: Benchmark Programs and their Properties

Program	LOC	Cyclomatic Complexity
Netweaver	17953	2.82
Inspirento	1769	1.76
Jsecurity	9470	2.05
Saxpath	1441	2.10
Jni-inchi	783	2.05
Xisemele	1399	1.29
Diebierse	1539	1.74
Lagoon	6060	3.52
Lavalamp	1039	1.50
Jnfe	1294	1.38

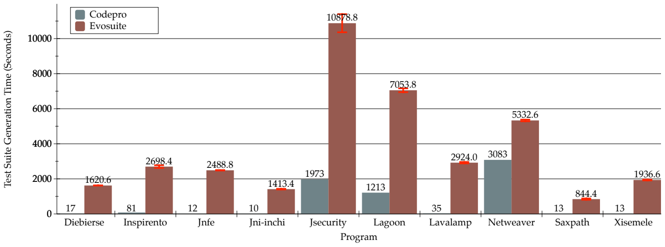

Fig. 2: Time to Generate Test Suite

C. Experiments and Results

In the experiments, we first evaluated the time to create each test suite in comparison between Evosuite and CodePro. As shown in Figure 2, CodePro generated test suites much more quickly than Evosuite did. Although Evosuite consistently takes longer to generate tests regardless of number lines of code and the complexity of the test suite. However, when comparing the complexity of the source code to the time to generate and the complexity of the test suites themselves, there is a clear trade off between time and complexity.

In Figure 3, we list the complexities of the source code of the program, and the generated test suites from manual, Evosuite, and Codepro. Note that more complex tests can be more difficult to maintain. The Evosuite test suites are generally more complex than the other two, whereas CodePro is consistently at a lower complexity level. Due to increased complexity of the test suites, it can take more time create and run them [12]. As evidenced by Figure 4, this indeed seems to be the case. In this data set, it would seem that the more complex the source code is, the more complex the tests are. Furthermore, the more complex the tests are, the longer the test generation takes to complete. This Figure displays the complexity of the test suite versus the time it took to generate the test suite.

CodePro maintains a low complexity level, but in doing so, this may lead to an inability to address complex code that requires deeper test cases. Although there is a performance boost in the speed to generate the test cases, one must also further examine the quality of the tests that are generated. Evosuite is more complex than the manually generated tests, which may mean that Evosuite cover deeper cases. If the depth

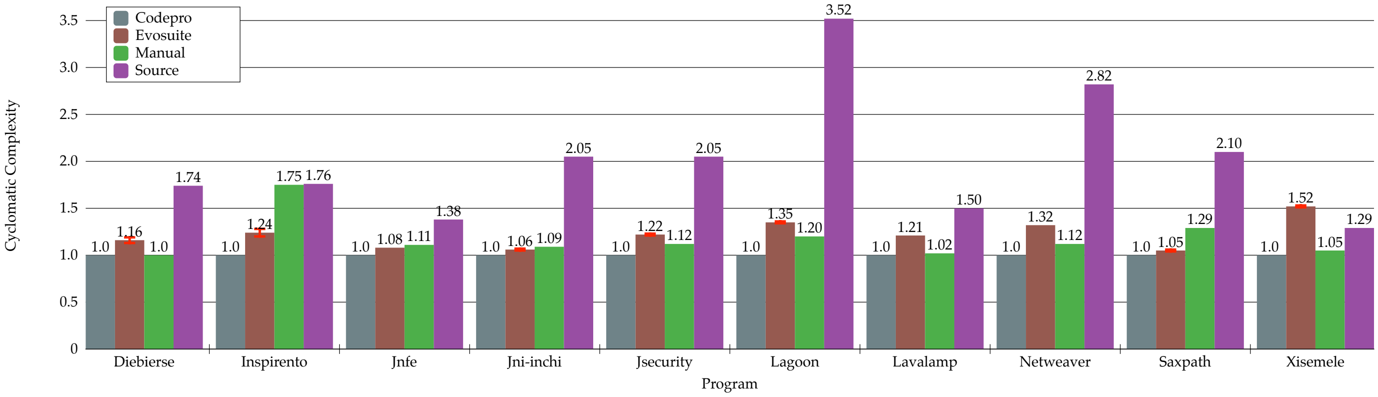

Fig. 3: Cyclomatic Complexity of Test Suites and Source Code

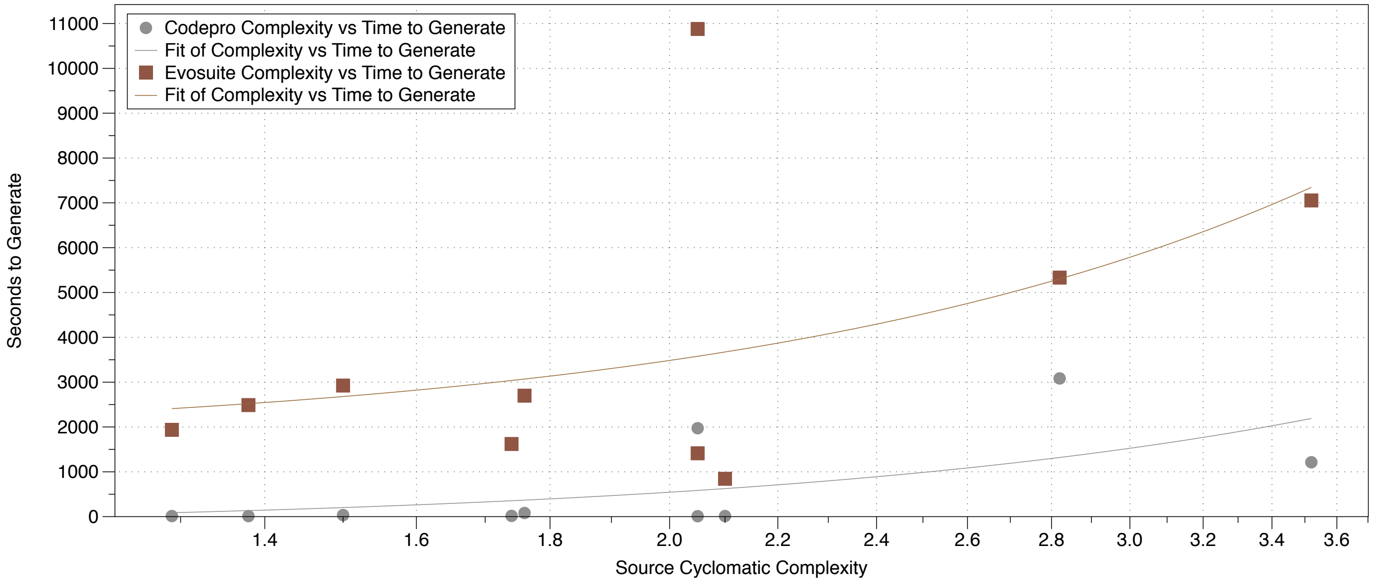

Fig. 4: Time to Generate vs. Test Suite Cyclomatic Complexity

of the tests are deeper in Evosuite, than one would expect the mutation scores to increase because the quality of tests would be higher.

Using mutation scores as a measurement for quality of the tests, we then observe the overall mutation scores given to manual, Evosuite, and CodePro test suites in Figure 5. Manual tests generate higher mutation scores in four of the programs, whereas Evosuite also receives higher mutations scores for half of the programs. CodePro receives the highest mutation score for one of the programs by a small margin, but for the rest of the test suites, the scores are the lowest by a large gap. Only in Diebierse did Evosuite vary greatly in the scores received. This indicates that although the average mutation score for Evosuite is within a more unpredictable range of results in one case, on average Evosuite can match the mutation score and quality of the manually written tests.

To further examine the cause of this, Figure 7describes the relationships between the mutation score of the test suites and the complexities of the source code. Notice that the manually written tests retain the highest mutation score overall at first. As the complexity of the program increases over time, the the mutation score begins to drop. Around a cyclomatic complexity of 2.3, we find that Evosuite's retains a higher mutation score than manually written tests. CodePro overall receives abysmal mutation scores, and as the cyclomatic complexity of the program increases, the mutation score drops to 0.

This graph is an eye opener to the reality of capturing deeper test cases. Up until a certain point human beings can write deeper test cases to more complex functions. Once the programs become complex, it may be too difficult for a person to cover complex parts of code that require deeper test cases. Evosuite may be at an advantage with its learning algorithm that modifies itself over time to improve the mutation score. With Evosuite seeming to edge out the other two methods in terms of depth, one must also examine the breadth each of the tools can cover as well.

In Figure 6 the relationship between lines of code of the test suite and and mutation score gives an initial impression that

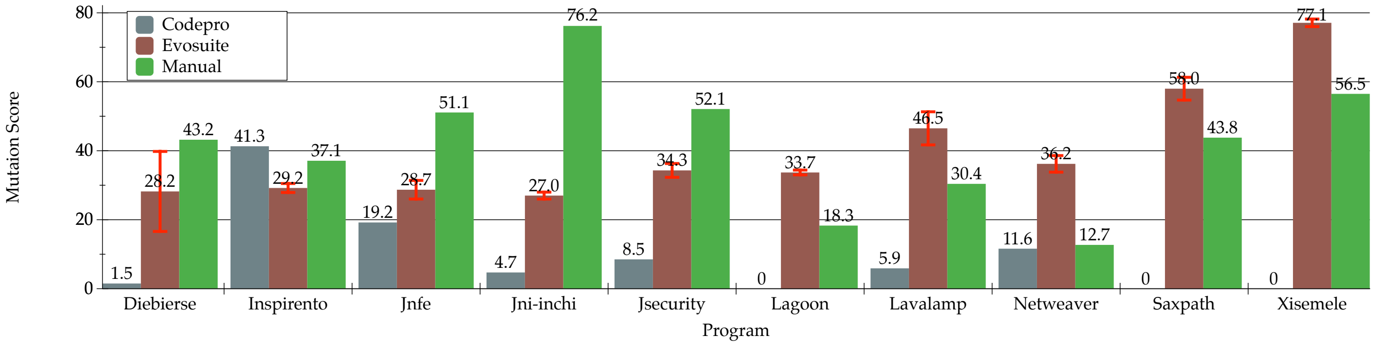

Fig. 5: Mutation Score

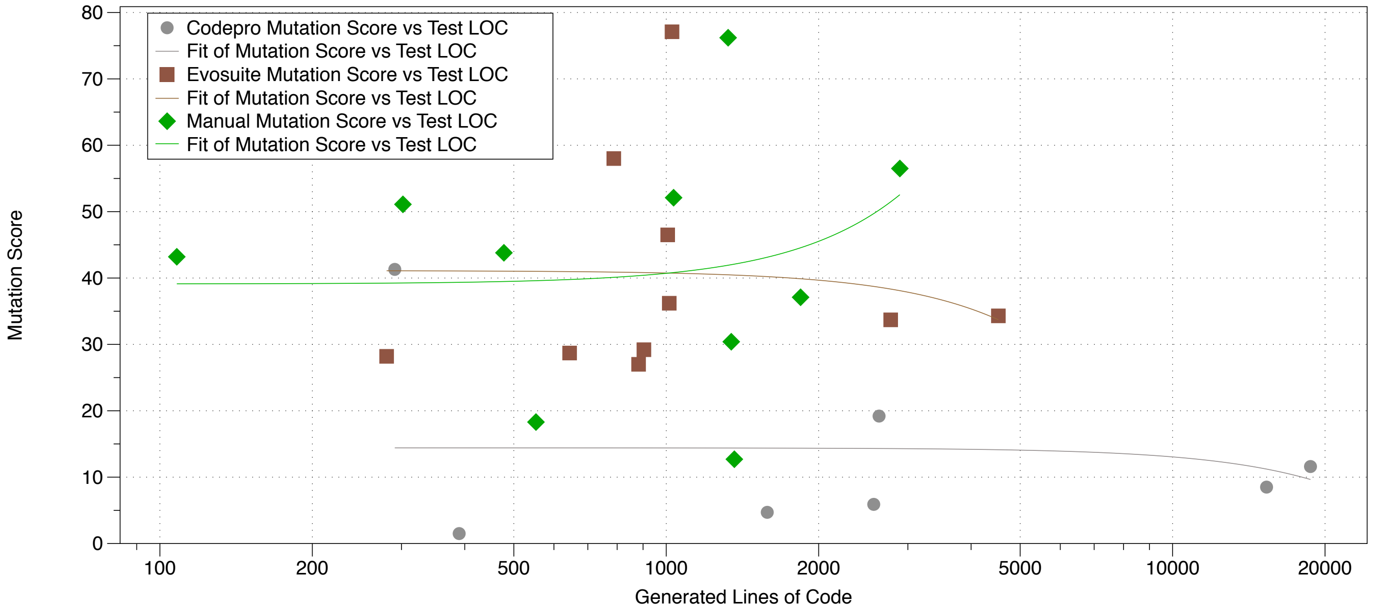

Fig. 6: LOC vs. Mutation Score

automated test suites lack the efficiency of a manually written test suite. CodePro not only results in the lowest mutation scores, but also creates the largest test suites. In the research, we found many of the test cases generated by CodePro were empty and contained comments for developers to put tests into templates. This reveals a different function that CodePro, unlike Evosuite, may be able to provide to developers.

Although at first glance one may assume that CodePro lacks any usefulness because the quality of tests are low, one could also say that the templates provided by CodePro may give developers the flexibility of modifying and improving the test suite. Evosuite tests are much more complex, and could be conceivably difficult to maintain if the developers wished to evolve the test suite on their own. The trade off between automated and manually written tests could be an argument of modifiability in this case. Although Evosuite scores close to manually written tests when it comes to quality, how would one go about improving the test suites to meet that level depth that manual tests cover in larger programs? The tests would need to be modifiable from a human perspective, which in this case, could be an issue with tests suites like Evosuite.

Depth is important when evaluating the quality of tests, but breadth is an area one must consider when quantifying quality as well, as a test suite that deeply covers just one function will result in little breadth. In Figure 8,there is an extreme diversity of branch coverage amongst the three evaluated test suites. Overall, CodePro actually scores higher in branch coverage for four of the programs that were tested. Evosuite and the manually generated tests are tied for three programs in scoring the highest branch coverage. For the automated test generators, the branch coverages are closer on average, but manually written test suite scores seem to vary greatly. Again, this is probably due to the inconsistencies in different developers writing their test suites. For example, the developers on Lavalamp could have focused on a TDD philosophy in development, which would predictably yield higher test coverage. However, the developers for Netweaver may have only been concerned with testing one part of their code, and thus branch coverage remained low. In fact, unit tests for only complex parts of the code are common in development, and thus this chart alone

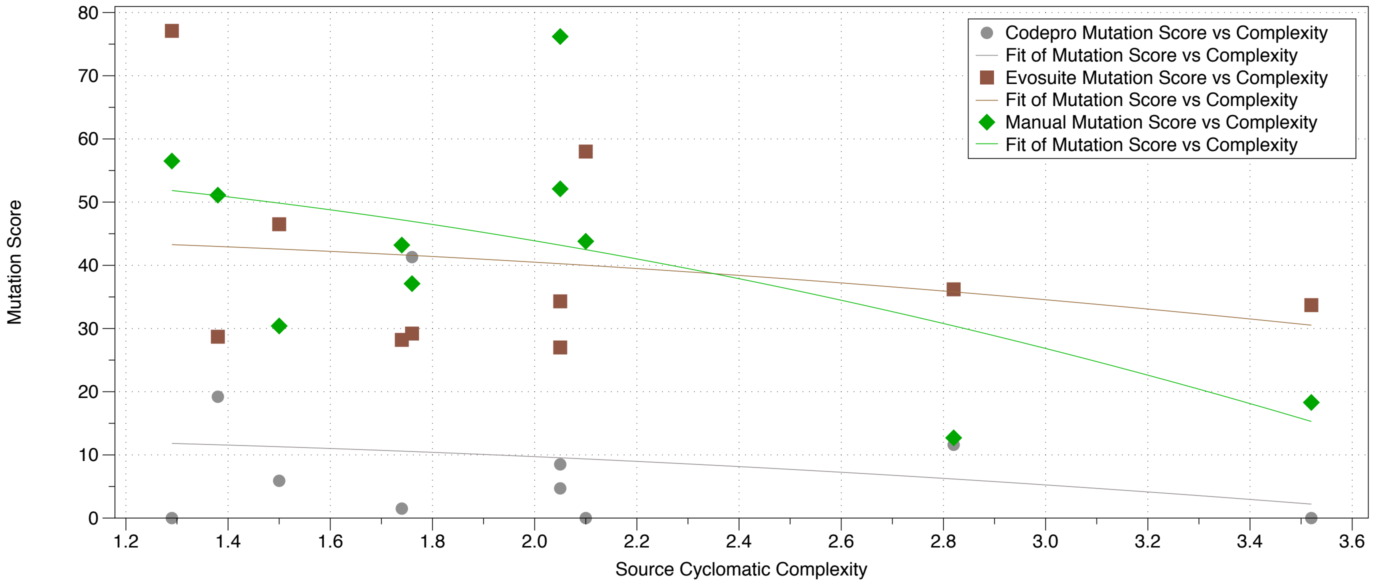

Fig. 7: Mutation Score vs. Complexity

only reveals the inconsistent nature of how developers write unit tests. One must look further at the mutation the branch coverage to reach a more definitive

Finally, we examined the center piece of the results, in comparing the evaluation of mutation scores and branch coverage. In Figure 9, we compare the mutation score (depth) to the branch coverage (breadth) of the test suites. CodePro falls short in the mutation scores, but does have higher branch coverage coverage scores. As the branch coverage increases however, the mutation score in CodePro dips drastically. Understanding that the mutation score was never good to begin with in CodePro, this leads us back to the discussion in perhaps using CodePro as template tool rather than a upfront ready-to-go test suite. Although the mutation scores are low, the coverage is high, providing a large amount of tests for developers to look at and fill over time.

More interesting are the similar trends between manual and Evosuite generated test suites. Initially Evosuite begins a bit lower in mutation score and branch coverage, but as the branch coverage increases, it would seem that Evosuite surpasses the Manual tests in quality. Overall, the quality of manual tests seems to rise above Evosuite ever so slightly. However, the trends seems to indicate that Evosuite will generate tests that will cover more code with better quality than manually written test suites, although both systems increase in quality as the test branch increases.

D. Threats to Validity

Measurements in quality of software tests is a subjective measurement. Although mutation score is one way to measure the quality of a test suite, ignoring the human elements of tests are nigh impossible. Developers may need to view tests in order to diagnose faults in the code, and if a developer cannot understand the test they are reading, then the cost of time and effort could be increased on the human part.

The statistical analysis also may be a threat to validity, as the inconsistency in how both Evosuite and manually written tests are created. Also, CodePro generated some test suites with a mutation score of 0, which could mislead one to believing that CodePro has no use for creating quality tests. For this reason we removed these results to give a better impression of the trend CodePro test suites.

IV. Related Work

There are few studies comparing the quality of manual and automated test suites. In one study researchers found that using automated test generation tools did not improve the ability to test the software [5]. However, the study does not measure the quality of manual and automated generated test suites directly. Furthermore, only Evosuite is used to evaluate the mutations and generate the tests. This paper expands upon the question of the usefulness of automated generated test suites, and if certain systems in generating test suites work better than others.

V. Conclusions and Future Work

Developers have struggled to find a standard of measuring the quality of tests. Mutation testing and seeding faults has proven as one effective way to identify lower quality tests, and many developers consider high branch coverage a sufficient mark of quality for their test suites. The dilemma becomes a cost trade off between depth and breadth, two measurements that this paper uses to address quality. If mutation analysis results represent the depth of test suites, and branch coverage represents the breadth of test suites, the results of this research identifies trends between the automated tools and developers who write tests. Mutation scores and branch coverage correlate in a positive fashion between the two.

Complexity of the program appeared to also influence the mutation score and time to generate the test suite. In the case of the time needed to generate the test suite, a tool utilizing

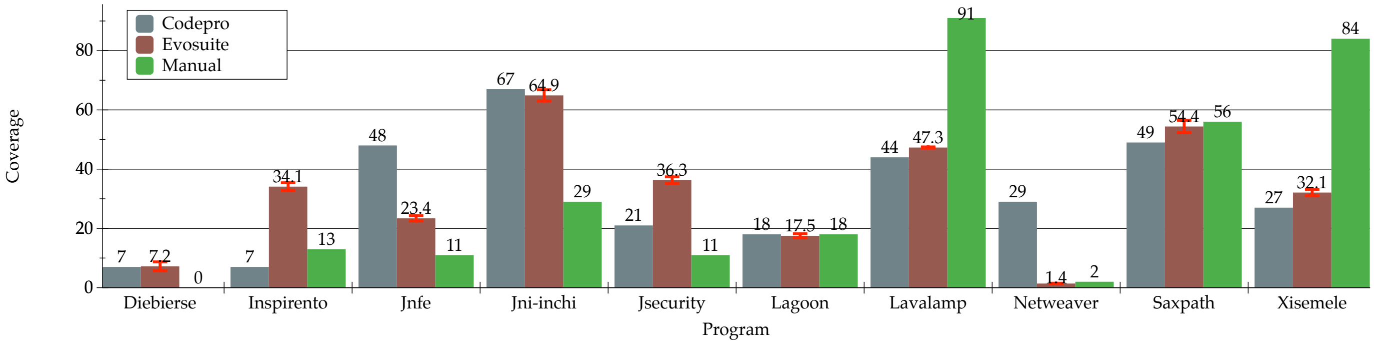

Fig. 8: Coverage

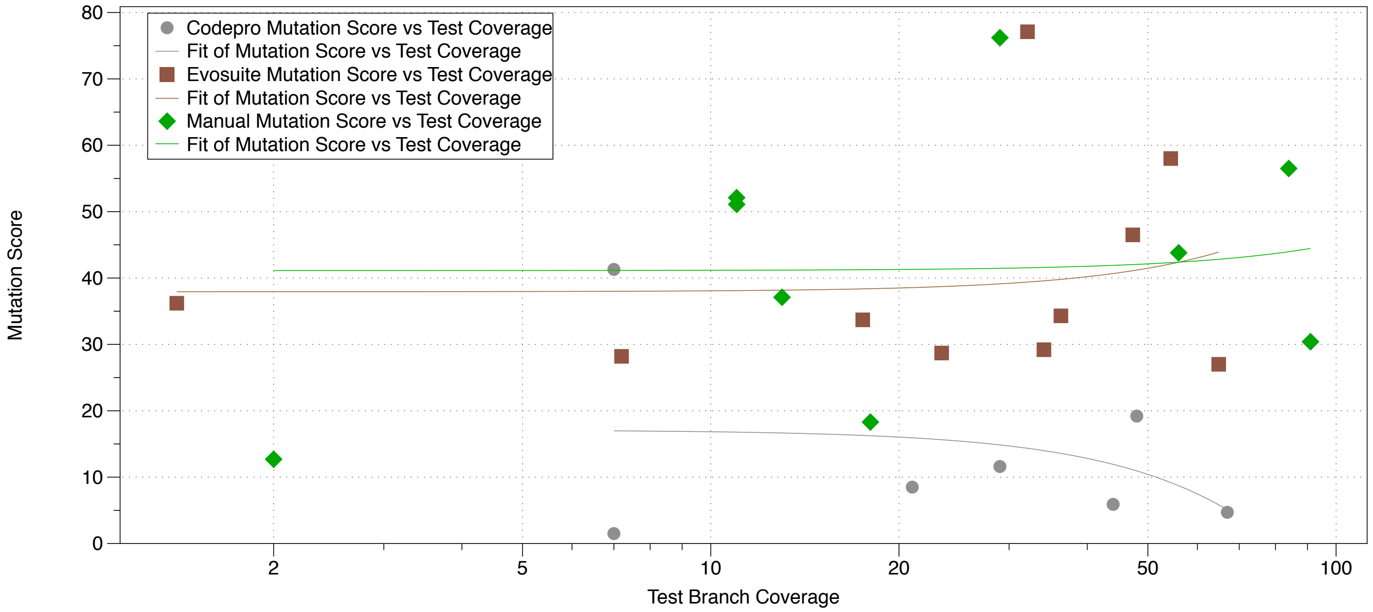

Fig. 9: Coverage vs. Mutation

a genetic algorithm took much longer than a deterministic algorithm to generate test suite. However, the cost of time could be outweighed by the better quality received in the mutation scores. The genetic algorithm sustained a higher mutation score than even the manually written tests when the programs were larger. It may be that automated test tools cover larger programs better than manual tests do.

This paper also addresses new questions. First, what is the purpose of auto generating test suites? CodePro garnered poor scores for both mutation and branch coverage results. Although the quality suffered, the research revealed that the CodePro tests were modifiable. Perhaps some tools could be used assist developers in developing a test suite with a template, placing a focus on modifable tests rather than a complete test suite to match the quality of manually written tests.

Secondly, what is the future of genetic algorithm based tools such as Evosuite? The tests may not be as modifiable as CodePro's, but the mutation scores in some cases matched and exceeded the mutation scores of manually written tests. If the cost of time and effort given from running Evosuite is less and will generate almost the same results for a large program, perhaps developers ought to consider using a tool like Evosuite.

Finally, the paper identifies the inconsistencies between manually generated tests. There is no way one can say with certainty that the goal of each of the developers for each of the projects was to achieve a high mutation score or high branch coverage. Perhaps developers only care to test one function of the code in depth, or rather they would seek to achieve a high branch coverage. Either way, the intentions of achieving the quality standards outlined in this paper may not be ideal for every developer.

In order to reproduce this study and further research in software testing, please reference our material in the github repository:

www.github.com

References

[1] G. Canfora, A. Cimitile, F. Garcia, M. Piattini, and C. A. Visaggio. Evaluating advantages of test driven development: A controlled experiment with professionals. In *Proceedings of the 2006 ACM/IEEE International Symposium on Empirical Software Engineering*, ISESE '06, pages 364–371, New York, NY, USA, 2006. ACM.

[2] J. M. Clarke. Automated test generation from a behavioral model. Citeseer, 1998.

[3] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, New York, NY, USA, 2011. ACM.

[4] G. Fraser and A. Arcuri. Whole test suite generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, 2013.

[5] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. Does automated white-box test generation really help software testers? In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 291–301, New York, NY, USA, 2013. ACM.

[6] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 147–158, New York, NY, USA, 2010. ACM.

[7] D. Gelperin and B. Hetzel. The growth of software testing. *Commun. ACM*, 31(6):687–695, June 1988.

[8] R. Just, F. Schweiggert, and G. M. Kapfhammer. Major: An efficient and extensible tool for mutation analysis in a java compiler. In *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 612–615, Washington, DC, USA, 2011. IEEE Computer Society.

[9] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: The autotest experience. In *System Sciences, 2007. HICSS 2007. 40th Annual Hawaii International Conference on*, pages 261a–261a, Jan 2007.

[10] D. Marinov and S. Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[11] H. M. Pandey, A. Dixit, and D. Mehrotra. Genetic algorithms: Concepts, issues and a case study of grammar induction. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, pages 263–271, New York, NY, USA, 2012. ACM.

[12] M. B. G. M. K. Sara Alspaugh, Kristen R. Walcott and M. L. Soffa. Efficient time-aware prioritization with knapsack solvers. In *WEASEL-Tech07: Proceedings of the ASE Workshop on Empirical Assessment of Software Engineering Languages and Technologies*, Atlanta, Georgia, November 2007.

[13] S. Zhang. Palus: A hybrid automated test generation tool for java. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 1182–1184, New York, NY, USA, 2011. ACM.