/Users/Jesh/Documents/School/Summer 2015/Independent Study/Output/Major/bpmail/codepro.csv

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

Jeshua S. Kracht
Department of Computer Science
University of Colorado, Colorado Springs
Email: jkracht@uccs.edu

Jacob Z. Petrovic
Department of Computer Science
University of Colorado, Colorado Springs
Email: jpetrovi@uccs.edu

Kristen R. Walcott-Justice
Department of Computer Science
University of Colorado, Colorado Springs
Email: kjustice@uccs.edu

***Abstract*— The creation, execution, and maintenance of tests are some of the most expensive tasks in software development. To help reduce the cost, automated test generation tools can be used to assist and guide developers in creating test cases. Yet, the tests that automated tools produce range from simple skeletons to fully executable test suites, hence their complexity and quality vary.**

This paper compares the complexity and quality of test suites created by sophisticated automated test generation tools to that of developer-written test suites. The empirical study in this paper examines ten real-world programs with existing test suites and applies two state-of-the-art automated test generation tools. The study measures the resulting test suite quality in terms of code coverage and fault-finding capability. On average, manual tests covered 31.5% of the branches while the automated tools covered 31.8% of the branches. In terms of mutation score, the tests generated by automated tools had an average mutation score of 39.8% compared to the average mutation score of 42.1% for manually written tests. Even though automatically created tests often contain more lines of source code than those written by developers, this paper's empirical results reveal that test generation tools can provide value by creating high quality test suites while reducing the cost and effort needed for testing.

I. Introduction

Since the release of low quality software has serious economic and social consequences [**?**], software testing is commonly used throughout the software development process to identify defects and establish confidence in program correctness. Kochhar et al.'s recent investigation of over 20,000 GitHub projects revealed that over 61% of the projects include test cases [**?**]. Even though testing is both valuable and prevalent, the creation, execution, and maintenance of tests is one of the most expensive aspects of software development — often comprising more than 25% of the total development costs [**?**].

Traditionally, test cases are manually written by software developers and/or members of a quality assurance team. Manual creation of a high quality test cases requires human effort in terms of thought and time. This effort can be prohibitive in large projects, which often necessitate the most testing [**?**]. As an alternative to manually writing test cases, many automatic test case generation tools (e.g., [**?**], [**?**], [**?**]) are now available to help developers test their software. However, the quality of the test cases generated by these tools varies [**?**], [**?**], [**?**], and it is unclear how the test suites of automatic test generation tools compare to those that are manually developed.

When given sufficient time and resources, developers can often manually implement high-quality test cases that exercises the majority of the application's features and covers its source code. Yet, the focus of the test cases may vary depending on the software developer, the goals of the project, and/or the standards of the company or open-source community [**?**]. Some developers may write test cases with the goal of increasing the coverage of the code, particularly in terms of statements or branches. Others may focus test cases on code that is likely to fail, most commonly executed, or "important" according to other standards [**?**]. In any case, as the complexity and number of features of a program increases, manually writing test cases becomes expensive [**?**].

Although there are tutorials that explain how to write JUnit test suites (e.g., [**?**]), there is no well-established standard for writing test cases, thus making the testing process even more challenging for developers. Instead of, or in addition to, manually implementing test cases without rigorous guidelines, developers may employ automatic test case generation tools that could reduce the time and cost associated with testing while also improving code coverage. Many automatic test suite generators currently exist. Some tools, like the MoreUnit plugin for Eclipse [**?**], generate test case stubs for the method that is currently highlighted by the cursor. Alternatively, tools such as CodePro [**?**], EvoSuite [**?**], JCrasher [**?**], Palus [**?**], Randoop [**?**], and TestEra [**?**] can generate complete test suites that need little to no modification prior to execution.

Automatic test case generation tools use both deterministic (e.g., hard-coded rules and regular expressions) and learning-based (e.g., randomized or evolutionary) algorithms to produce test cases based on particular strategies, thereby avoiding the subjectivity and wide variation in styles commonly found in manually generated tests. Test suite generators also have the potential to significantly reduce the amount of human time and effort required of the developer to create the test suite.

While goals, company policy, and/or community standards may influence the quality of manually implemented test suites, automatically generated ones are similarly affected by both the choice of the tool and the configuration of the tool's parameters. Test suite quality is frequently measured based on the amount of code covered and on the fault-finding ability of the test suite. Code coverage is a structure-based criterion that requires the exercising of certain control structures and variables within the program [**?**]. A fault-based test adequacy criterion, on the other hand, attempts to ensure that the program does not contain the types of faults that developers inadvertently introduce into software systems [**?**].

A common type of test quality evaluation, in both industrial and academic work, leverages structurally-based criteria that require the execution of a statement or branch in a program [**?**]. Alternatively, mutation testing is a fault-based technique that

measures the fault-finding effectiveness of test suites on the basis of induced faults [**?**], [**?**]. Originally proposed by DeMillo et al. [**?**], mutation testing is a well-known technique that evaluates the quality of tests by seeding faults into the program under test; each altered version containing a seeded fault is called a mutant. A test is said to kill a mutant if the output of that test varies when executed against the mutant instead of the original program; the mutation score is the ratio of killed mutants to generated mutants. Mutants of the original program are obtained by applying mutation operators. For example, a conditional statement `if (a < b)` results in multiple mutants by replacing the relational operator < with valid alternatives such as <= or `!=`. Prior studies have used mutation adequacy to experimentally gauge the effectiveness of different testing strategies [**?**], [**?**], [**?**], [**?**].

When confronted with the wide variety of non-standardized manual testing strategies and complex automated test generation tools, real-world software developers face the challenge of defining a strategy for developing quality software. Through an empirical study of automatically and manually generated test suites for ten real-world open-source programs (e.g., Netweaver and Jsecurity), this paper characterizes and compares the test suites, providing useful insights for both researchers and practitioners. With a focus on the time required to automatically create the test suite and the statement, branch, and mutation scores of the tests, the experiments compare the manually created tests with those that are automatically generated by two tools, CodePro and EVOSUITE.

Although automatic test generation requires far less human time and effort than manual test generation, this time savings may not be worthwhile if the quality of the resulting test suites is poor in terms of coverage or fault-finding capability. Alternatively, automatically generated tests may not be as useful as manually implemented ones if they are too complex or difficult to maintain. Subsequently, this paper additionally examines the complexity and size of the original program and both automatically and manually generated test. Finally, we point out the practical benefits and challenges associated with using automatically generated tests instead of manually created ones.

In summary, this paper's main contributions are:

- An examination of the techniques used in sophisticated automatic test case generation tools (Section **??**);
- An empirical analysis of existing manually written test suites for open-source applications (Section **??**);
- An empirical analysis of automatically generated test suites for open-source applications (Section **??**);
- A comparison of test suites that are both manually written and automatically generated (Section **??**);
- A discussion of the benefits and drawbacks of using automatic test case generation tools (Section **??**).

II. TEST CASE GENERATION TECHNIQUES

This section discusses the processes of writing test cases manually and using automatic test case generation tools. We also describe CodePro and EVOSUITE, the automatic test case generation tools that are empirically studied in this paper.

A. Manually Written Tests

Test suites are most often written manually, either by the developers themselves or through a quality assurance team. While companies may have their own standards and goals that are followed when writing test cases—such as high levels of statement or branch coverage (e.g. [**?**], [**?**])—no well-established patterns exist to help standardize test writing practice throughout the software development industry. Thus, the methods and styles of writing individual tests, fulfillment of coverage and fault-finding goals, and the ordering of test suites are often left to industry requirements or personal preference.

B. Automatically Generated Tests

Due to the high cost and inconsistencies introduced when developing test suites by hand, automatic test suite generation research is on the rise. In the past, the writing of test cases was left as an afterthought, and their implementation was the responsibility of a separate quality assurance team rather than the developer. This led to a disconnect between the code and the tests. However, in recent years, there has been a move towards a more involved test development system in tandem with the development process [**?**]. This movement includes a focus on creating unit tests for code as it is developed, ensuring that code always passes tests, thereby improving the quality of the code [**?**]. Although this improvement in test creation processes successfully improved the reliability of the code, the cost of human time and effort needed to manually write high quality tests increased as programs became more complex [**?**].

While many different techniques have been used to automatically generate tests, they can be divided into two key categories: Deterministic and Learning-Based.

1) Deterministic: Deterministic automatic test case generators normally analyze method parameters and basic paths to create unit tests. The simplest of these tools statically analyze the basic source code paths alone and create skeletons of needed tests. For example, JUnitDoclet [**?**] uses Javadoc to parse the source code of the application classes. From the collected information, JUnitDoclet writes test cases and test suites where there is a test suite for each Java package, a test case for each public, non-abstract class, and a skeleton test method for each public method.

While these test skeletons are helpful, more sophisticated tools have been developed that create more complete tests by taking the method parameters into consideration. CoView [**?**], for instance, is a commercial Eclipse plug-in tool that analyzes Java source code and calculates the number of data-driven and cyclomatic paths in a method. Each path is one that should be verified via a unit test. CoView then analyzes existing JUnit tests to determine which paths are and are not being tested. This determination is made with instrumented bytecode that calculates path and branch coverage. CoView then creates the missing JUnit tests for the developer. The developer will have to modify parts of the tests, such as the assertions, but the tool helps the developer by identifying the minimum number of tests that should be created given parameter options and paths.

Other tools are capable of generating fully executable tests that require no modification: for instance, this paper considers CodePro [**?**], an Eclipse plug-in tool with many powerful code analysis features and metrics. Given an input class, the tool creates a corresponding test class complete with multiple test methods for each input class method. The tool analyzes each method and input argument with the goal of generating tests that exercise each line of code using a combination of both static analysis and by dynamically executing the code to be tested. [**?**]. CodePro was a Jolt Award finalist and has been

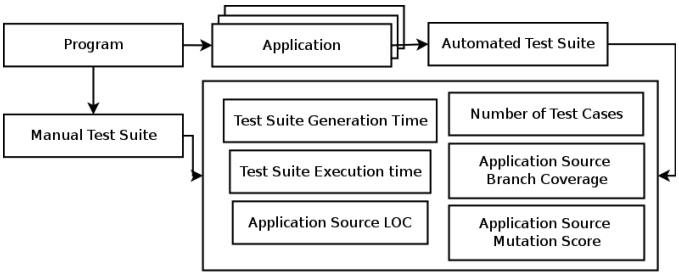

Fig. 1: Evaluation Process

studied in terms of the types of tests it can write in comparison to other tools [**?**]. Yet, to the best of our knowledge, no work has compared the overall quality of the test cases it creates.

2) Learning-Based: Another set of automatic test case generation tools use learning algorithms to improve the overall quality of the generated test suites. The two top-ranked tools in this area are Randoop and EVOSUITE [**?**]. Using feedback-directed random test generation, Randoop automatically creates tests for Java classes in JUnit format [**?**]. This technique randomly generates sequences of methods and constructor invocations for the classes under test and uses the sequences to create tests. Randoop then executes the sequences it creates and uses the results of the execution to create more assertions, attempting to avoid redundant and illegal inputs while guiding towards generation of tests that lead to new object states.

EVOSUITE [**?**], which is used in this paper, ranked first in SBST 2013 Tool Competition [**?**] and similarly uses a learning algorithm to generate a full, executable test suite. The tool's evolutionary search approach evolves whole test suites with respect to both coverage and mutation scores. Optimization with respect to a coverage criterion rather than individual coverage goals helps the algorithm to not be adversely influenced by difficulty of infeasibility of individual coverage goals. Repeated mutation testing is used to produce a reduced set of assertions that maximizes the number of seeded defects in a class that are revealed by the generated test cases.

III. EMPIRICAL EVALUATION

Given the many different techniques for generating test suites, the primary goal of this paper's empirical study is to compare the quality and complexity of the resulting test suites. We implemented the empirical evaluation approach as shown in Figure **??**. As can be seen in the figure, existing programs are fed into automatic test suite generators to create executable test suites. These test suites are then compared to the programs' associated, manually written test suites based on six metrics.

The goals of the experiments are as follows:

- Determine the time of automated test case generation along with the size and time of execution of the test suites
- Compare the test suites, automatically generated and manually written, to the case studies' source code
- Analyze the quality of these test suites based on branch coverage and fault-based mutation scores

A. Experiment Design and Metrics

All experiments were performed on GNU/Linux workstations with kernel 3.2.0-44, a 2 GHz Intel Corporation Xeon E5/Core i7 processor and 15.6 GB of main memory.

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
Jdbacl	13296	2.50
Geo-Google	3941	1.18
Bpmail	1252	1.58
Schemaspy	7987	3.09
Hft-Bomberman	6474	1.87

Case Study Applications:

Ten programs were identified from the SF110 code suite [**?**]. The case study applications were selected due to their size, the existence of associated manually developed JUnit test cases, and their use in tuning EVOSUITE parameters for mutation and test generation, one of our test suite generation tools. Table **??** provides a list of the selected SF110 programs with their respective lines of code (LOC) and average cyclomatic complexity per method. LOC and cyclomatic complexity were measured using JavaNCSS [**?**].

Netweaver is the largest program under consideration with nearly 18K lines of code. Netweaver has an average Cyclomatic Complexity (CC) of 2.82 across all methods, which implies that for a specific method M, 1) CC_M is an upper bound for the number of test cases that are necessary to achieve a complete branch coverage within the method M, and 2) CC_M is a lower bound for the number of paths through the control flow graph. Assuming each test case takes one path, the number of cases needed to achieve path coverage is equal to the number of paths that can actually be taken, ignoring infeasible paths. The smallest program, Jni-inchi has 783 lines of code with an average cyclomatic complexity of 2.05.

After the case study applications were identified and analyzed, automated test tools EVOSUITE and CodePro were used to generate test suites [**?**], [**?**]. As EVOSUITE is non-deterministic and learning-based, ten sets of tests were generated for evaluation, and the standard deviation is given across the ten test generations for all EVOSUITE related results. EVOSUITE was configured using its default values [**?**].

Evaluation Metrics:

The manually written test suites and automatically generated test suites are compared based upon the time to generate test suites, the number of test cases generated, the time to execute generated tests, lines of code in the benchmark application, complexity of the benchmark application, branch coverage of generated suites, and the mutation score of generated suites. To perform these evaluations, three tools are used.

All tests are written or generated in JUnit form. The time to generate test cases, number of test cases generated, and the time to execute the test suite are measured using the JUnit tool. We also measure the non-commented LOC from the source code of the benchmark applications using JavaNCSS [**?**].

Following the automatic generation of test cases, Jacoco [**?**] is used to calculate branch coverage of the tests. Jacoco calculates branch coverage by instrumenting all branches at the

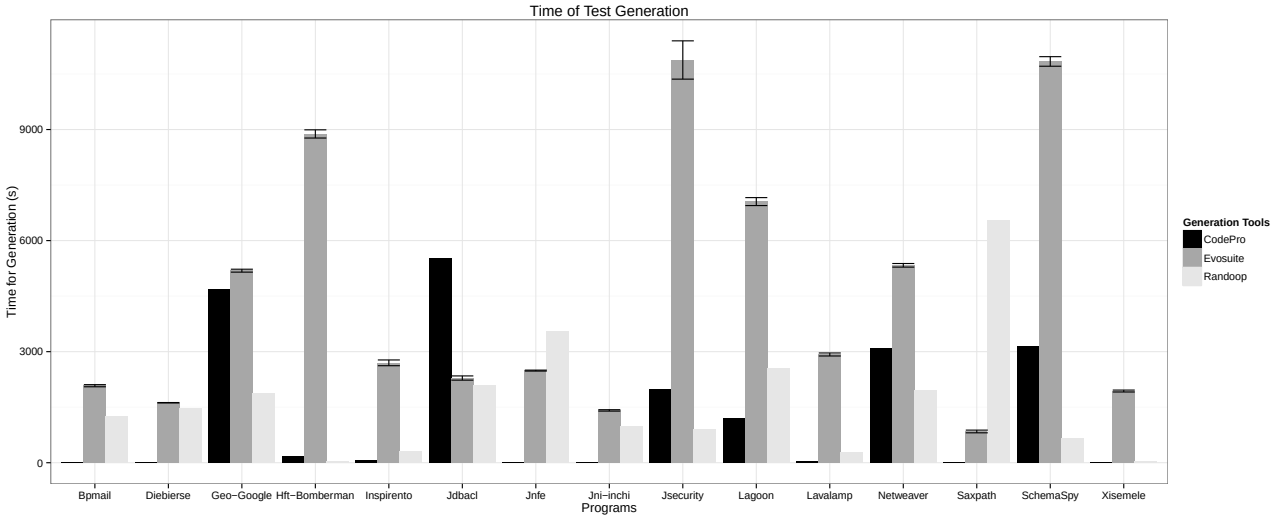

Fig. 2: Time to Generate Test Suites.

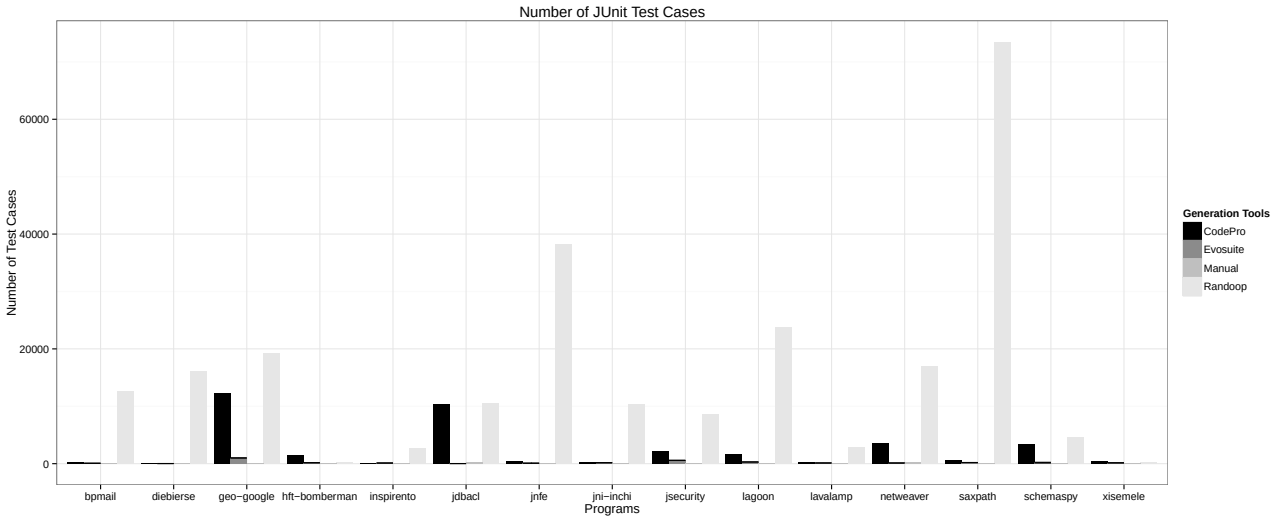

Fig. 3: Number of Test Cases per Test Suite.

byte code level through ASM, an all purpose Java bytecode manipulation and analysis framework. We also use MAJOR [**?**] to calculate fault-based mutation scores given the case study and associated tests. MAJOR is a Java compiler-integrated mutator that serves as a mutation analysis back-end for JUnit tests. It provides a domain specific language to configure the mutation process, although we used its default values for our experiments.

B. Experiments and Results

Experiments were run to compare how test suites are generated using automated tools, the differences between the resulting test suites in terms of size and complexity, and the over all quality of the generated test suites.

1) Generating Test Suites : In the first set of experiments, we tracked the time required to generate the test suites, the number of test cases generated, and the execution time of the resulting test suite. Figure **??** displays the time required to generate each test suite using the two automatic test case generation tools, CodePro and EVOSUITE. In the case of Netweaver, CodePro completed test suite generation in approximately 51 minutes whereas EVOSUITE required 89 minutes. This was a small difference of 1.7% compared to Xisemele, for which CodePro generated a test suite in just 13 seconds compared to 32 minutes from EVOSUITE. The time needed by EVOSUITE was 148% greater in this case. On average, EVOSUITE took 78% more time in test generation compared to CodePro.

Next, the number of test cases generated per method was analyzed. Figure **??** shows the number of tests generated for the ten case study applications, CodePro produced an average of 5% more test cases than EVOSUITE and 16.4 % more than were written manually. EVOSUITE produced, on average, 4% more than were created manually.

The time to execute the generated test suites was also evaluated. Figure **??** reveals the execution time of manual,

EVOSUITE, and CodePro test suites, with test execution times ranging from between 2.1 to 25.5 seconds. In Figure **??**, the Netweaver test suite contains the most test cases. However, the Netweaver CodePro test suite does not take the longest to execute. Rather, the second largest test suite, Jsecurity by EVOSUITE, does. This may be the result of skeleton-like test cases that CodePro produces. CodePro's skeleton-like test cases provide comments for developers to find where to write test cases, forming a hybrid approach to the automated and manual world of testing.

2) Comparing Generated Tests to Case Studies: The LOC for the generated test suites and the application source code were also compared. Figure **??** shows that CodePro generates more LOC than either manual or EVOSUITE test suites. As shown in Figure **??**, CodePro generated the most tests out of the automated test generators. In a comparison between the two graphs, a close correlation can be seen between the LOC and the number of tests. For example, Netweaver contains 17953 LOC, and CodePro produces 3513 test cases for the application. Likewise, the other applications and their tests suites share this same trend. As the size of the application increases, the number of tests will increase as well. In proportion to the original lines of code, the number of test cases are greatest with CodePro, then EVOSUITE, and then manual.

3) Quality: Manual versus Generated: In this study, both branch coverage and mutation scores were measured for each of the generation test suites. The branch coverage for manually written test suites varied greatly, but was the only method able to attain a score over 70% with Lavalamp and Xisemele. Both applications contained the least number of test cases with manual as evidenced by Figure **??**. Overall CodePro had higher branch coverage scores than EVOSUITE or manual. This could indicate a direct relationship between the number of tests and the branch coverage.

The relationship between the number of tests and the branch coverage for each generated test suite was furthermore examined. In comparing Figure **??** and the branch coverage, manual test suites increased in branch coverage as the number of test cases increased. When evaluating EVOSUITE, the trend line has a low R^2 value at 0.184, but begins in a similar trend to manual and CodePro, but actually drops in branch coverage for the two larger test suites. The branch coverage in comparison with the number of tests indicates that CodePro branch coverage drops as the number of generated test cases increases. Larger, more complex applications may require more tests to be written to increase the branch coverage.

Our next experiment evaluates the relationship between LOC and branch coverage for each of the generated test suites. EVOSUITE, CodePro, and manual test suites demonstrated a similar trend in the decrease of Branch Coverage as the LOC of the application code increased. Only CodePro displays a larger increase in branch coverage by at least 27% more for the largest program, Netweaver. Despite generating more tests in proportion to the application source code size, CodePro did not meet the quality of the tests generated by EVOSUITE or the manually written test suites.

For the mutation score, EVOSUITE attained the highest scores for five of ten applications. However, Diebierse scores ranged dramatically between 18% and 40%. CodePro resulted in the worst mutation scores, acquiring a 0% mutation score for lagoon, Saxpath, and Xisemele. This is due to the skeleton-like tests that CodePro often creates. While these tests can lead

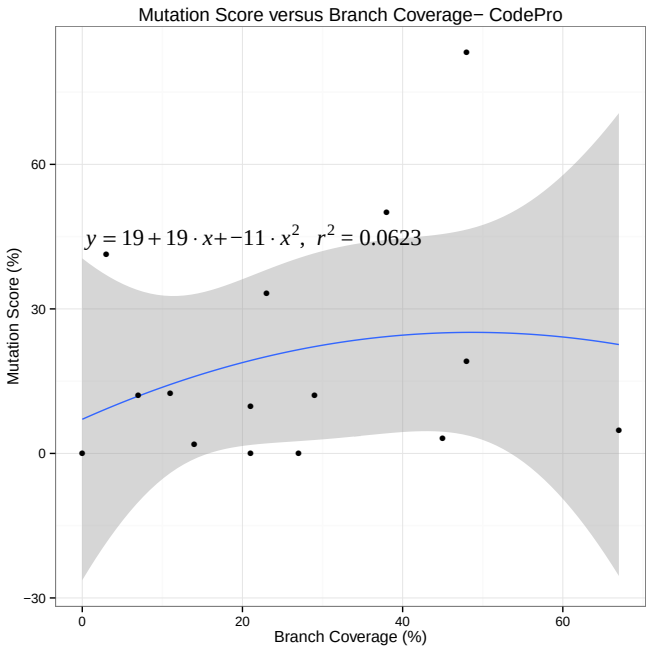

Fig. 6: Mutation Score compared to Branch Coverage for test suites generated by CodePro.

to the execution of branching behavior due to the tool's goal of generating test cases that exercise each line of code, the oracles, when they exist, are often poorer than EVOSUITE's intelligent oracles and thus miss capturing faults. EVOSUITE, on the other hand, was able to generate test suites averaging between 33.7% and 77.1% for these applications. Manual tests remained between 18.3% and 56.5% mutation scores for the applications, while CodePro's mutation scores ranged between 0% and 41.3%.

Figures **??**, **??**, and **??** compare the branch and mutation score for manually generated test suites, EVOSUITE's-generated suites, and CodePro-generated suites, respectively. In general, the mutation score increased as the branch coverage increased. Figure **??** displays a much lower R^2 value at 0.137, but the data indicates overall that the mutation score neither increases nor decreases with higher branch coverage. This can be explained because the mutation scores were all much lower than manual and EVOSUITE mutation scores. With exception to the outlier of a 40% mutation score, the trend would otherwise indicate that the mutation score increases as the branch coverage increases.

For each comparison, the Kendall τ coefficient and Pearson's product-moment correlation were calculated. While there is a possibility of rank ties when calculating Kendall's τ values, none were identified in this work. In comparing the coverage and mutation scores for CodePro, Kendall τ's coefficient is -0.0698. The Pearson's test gives a correlation of -0.275. For EVOSUITE, the Kendall τ's coefficient is 0.111. The Pearson's test gives an overall correlation value of 0.212. Manually written tests earn a Kendall τ value of 0.135. The Pearson's test reveals an overall correlation of 0.155.

The Kendall τ measurement of correlation, calculated in R, falls between -1 and 1, representing a strong negative and strong positive association, respectively, and 0 showing no

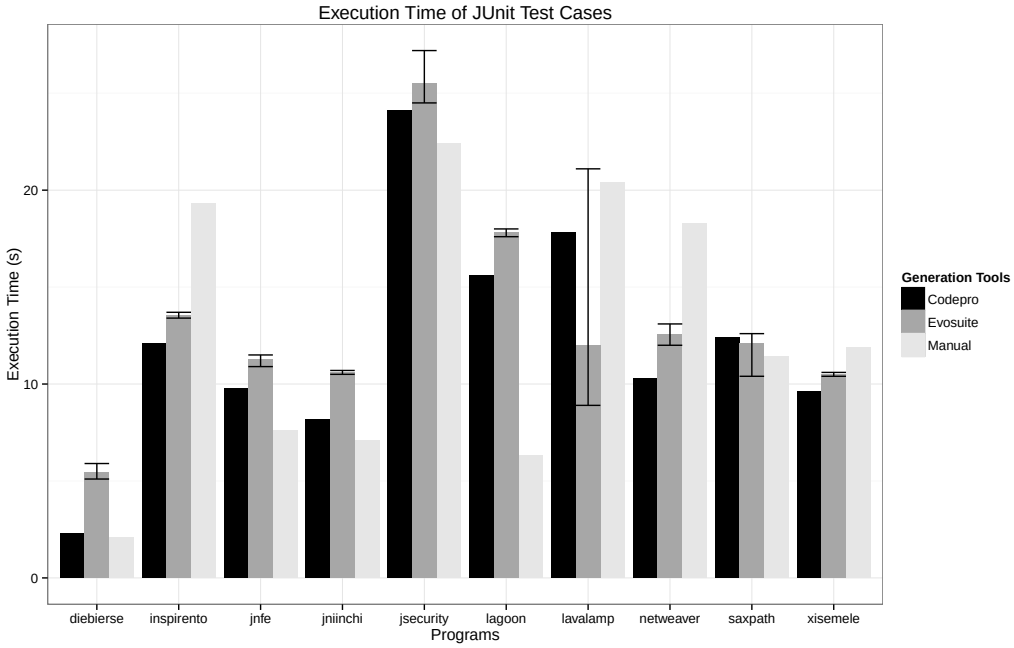

Fig. 4: Execution Time of all Test Suites.

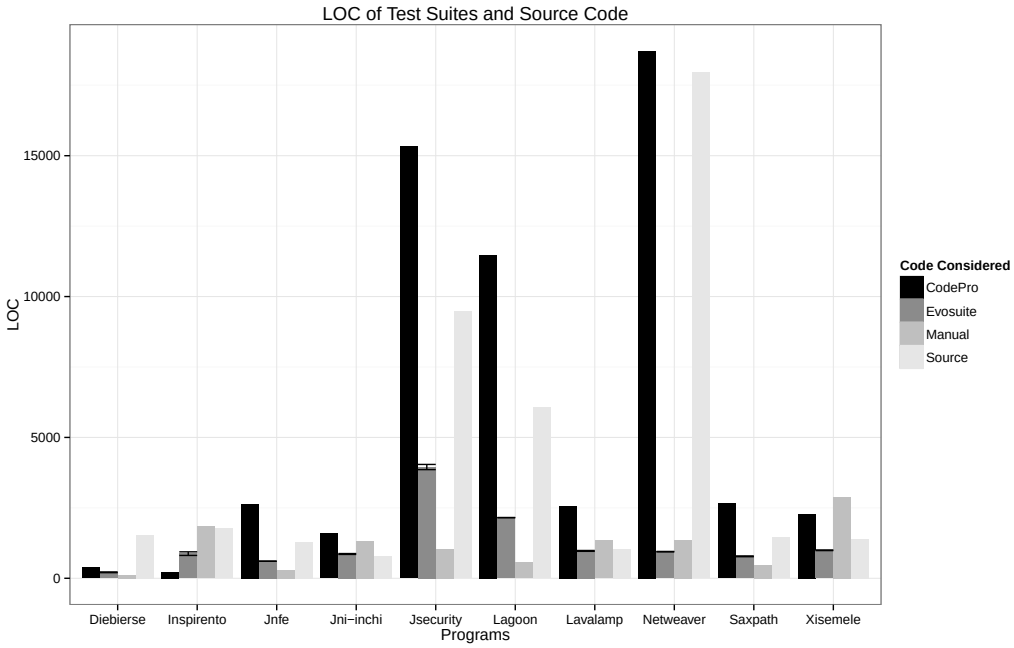

Fig. 5: Non-commented lines of code for automatically generated tests and manual tests compared to case study source code.

correlation. We use an accepted interpretation of τ, whereby the qualitative terms "small", "medium" and "large" correspond to 0.1, 0.3 and 0.5 [**?**]. For CodePro, there is essentially no correlation between branch coverage and mutation scores. EVOSUITE demonstrates a small correlation, and manually written tests have a slightly larger, but still small, correlation.

The Pearson's product-moment correlation is similarly between -1 and 1. A value of 1 implies that a linear equation describes the relationship between X and Y perfectly, with all data points lying on a line for which Y increases as X increases. A value of -1 implies that all data points lie on a line for which Y decreases as X increases. A value of 0 implies that there is no linear correlation between the variables. The results between the Kendall τ and Pearson's correlations agree for all three sets of data. While correlations do exist between the branch coverage and mutation scores for

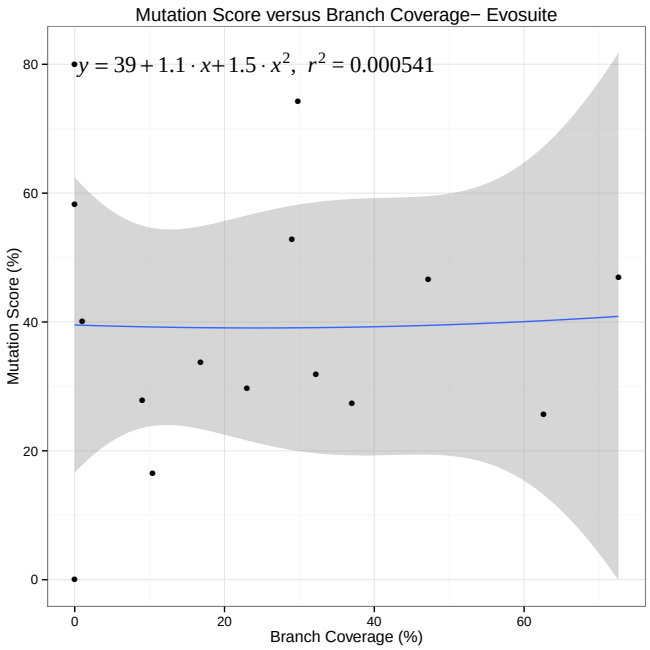

Fig. 7: Mutation Score compared to Branch Coverage for test suites generated by EVOSUITE.

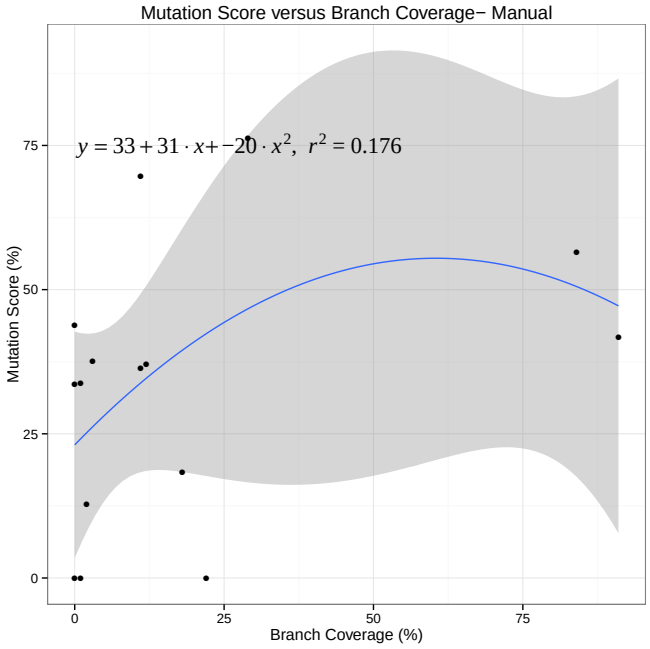

Fig. 8: Mutation Score compared to Branch Coverage for test suites created manually.

EVOSUITE and manually written tests, they are small. The correlation between the branch coverage and mutation scores of CodePro are slightly negative.

For full result analyses, including full Kendall τ analyses and Pearson correlation summaries—in addition to more results and graphs from our evaluations related to LOC of the generated test suites, cyclomatic complexity of the test suites, and cyclomatic complexity of the application source code—please refer to: http://cs.uccs.edu/~kjustice/QSIC2014/.

C. Threats to Validity

There are several threats to the validity of this work. First, EVOSUITE was used to generate test suites using its default configuration values. While tuning can have impact on the performance of a search algorithm, in the context of test data generation, it is difficult to find good settings that significantly outperform the "default" values suggested in the literature [**?**]. Thus, the default values were used. Presented results in this paper represent five test suites generated by EVOSUITE. Although more test suites were generated for a subset of the applications, the average and standard deviation remained the same given more executions of EVOSUITE, and thus, five generated test suites can be viewed as sufficient.

Second, the determination of the quality of software tests can be considered a subjective measurement. Although mutation score and coverage are two ways to measure test suite quality, that does not consider the readability of the test cases. If the developers who need to view tests in order to diagnose defects cannot understand what the tests do, then the human time and effort could be substantially increased.

Third, the tools used for coverage and mutation analysis also lead to a potential internal threat to validity. Jacoco was used for all coverage analysis, and MAJOR was used for mutation analysis. Jacoco, a well-established coverage monitor for Java programs, is based on Emma, another standard tool for analyzing Java programs [**?**]. MAJOR [**?**] is an experimentally verified (e.g., [**?**]) and well tested (e.g., [**?**]) mutation testing tool integrated into the Java compiler. However, other options such as PIT [**?**] could be used in future comparisons.

IV. DISCUSSION

In comparing LOC and number of test cases, we observe that higher proportions of test cases compared to original application code does not necessarily imply that either mutation score or branch coverage will be high. Especially in the case of CodePro, in which many tests are produced in a skeleton-like fashion, the mutation scores and branch coverage are much lower than EVOSUITE and manual tests. However, as indicated earlier, the skeleton approach to generation gives developers a template to easily modify and implement. Automated test suite generation can aide the developer in covering a large portion of the application, allowing developers to reduce the time and effort spent in identifying source code to be covered by tests and then enhancing the tests later. Although this feature could be desirable for developers who want help in achieving high coverage, it will still require more manual effort than fully automated test generation tools such as EVOSUITE.

As seen from the results, learning-based test suite generation tools are more helpful in creating executable test cases that do not need modification prior to use. These tests attain coverage and mutation scores that are only slightly lower than manually written test suites. On average, the branch coverage scores are comparable, and mutation scores are only about 1% less than manual tests. However, the modifiability and readability of the resulting tests varies. For example, CodePro tends to create tests that build empty object-oriented tests that are executable, but require the developer to generate complete test oracles. Thus, CodePro's mutation scores especially are lower than those written by a human or through a learning-based tool.

EVOSUITE, on the other hand, creates more complete tests focusing on exceptions, and parameter paths. As EVOSUITE takes potential faults into consideration as it "grows" tests, its mutation scores are higher. Manual tests are generally written with the functionality of the program itself in mind. While all three processes have their benefits, the learning capability of tools like EVOSUITE to achieve high-quality results in little time gives a strong initial set of tests that can then be broadened as developers enhance their knowledge of the code under test.

V. RELATED WORK

Since this paper focuses on empirically comparing manually implemented and automatically generated test suites, it is most directly related to Bacchelli et al.'s evaluation of the effectiveness of manual and automated test data generation [**?**]. However, there are several distinctions between our paper and this prior work. While the experiments in this paper focus on the test suites for ten case study applications, Bacchelli et al. only consider select classes that implement data structures like `LRUHashtable`. In contrast to the use of manual tests that were implemented by real-world open-source software developers, the manually-coded test cases in Bacchelli et al.'s were created by the authors themselves. Moreover, Bacchelli et al. use Randoop [**?**] and JCrasher [**?**], instead of picking EVOSUITE, the current state-of-the-art test data generation tool [**?**]. Finally, it is important to note that even though Bacchelli et al. report coverage and mutation scores, they neither investigate the correlation between coverage and mutation nor perform a comprehensive statistical analysis of the results. In addition, it is possible to make similar comparisons between our paper and the experimental work of Assylbekov et al. [**?**].

The design of the our paper's experiments is informed, in part, by the experimental design and results presented by Inozemtseva and Holmes [**?**]. Like our paper, this prior work also uses real-world programs to empirically investigate the relationship between code coverage and mutation score. However, the primary intent of our work is different than that of Inozemtseva and Holmes: while they investigate the effectiveness correlation for the test suites that come with programs, we consider both manually and automatically generated tests. Moreover, it is important to note that while their paper uses PIT [**?**] for fault seeding, our experiments use MAJOR—the only mutation testing tool whose mutants are currently known to be statistically similar to real-world faults [**?**].

Our paper's experiments are also partially influenced by the design and results reported on by Gopinath et al. [**?**]. This paper is similar to ours because it also investigates the correlation between a test suite's coverage and mutation score. In addition, Gopinath et al. use both manually and automatically generated test suites for more programs than we do; yet, since our experimentation framework is easy to apply to new programs and our presented results demonstrate promise, we will scale our study to Gopinath et al.'s level in future work. Moreover, even though EVOSUITE automatically generates better test suites than Randoop [**?**]—thus motivating its use in our experiments—Gopinath et al. use Randoop to create test suites. In addition, while Gopinath et al. employ PIT to seed faults into their case study applications, we decided to use MAJOR since recent results indicate that this tool's faults are a valid substitute for real faults [**?**].

Ultimately, the results from Inozemtseva and Holmes [**?**] and Gopinath et al. [**?**], in conjunction with those in this paper, present a complementary understanding of the effectiveness of automatically and manually generated test suites. It is also important to remark that, while Just et al. [**?**] are the first to establish a statistical correlation between a test suite's mutation score and its effectiveness at detecting real-world faults, the purpose of that work is not to develop a full-featured understanding of the quality characteristics of automatically and manually generated test suites—the aim of our paper.

It is important to note that all of the aforementioned related work focuses of the empirical and technical aspects of test suite effectiveness that are not related to human-centric issues. In part, our paper was motivated to investigate the complexity and understandability of automatically and manually generated test suites by Fraser et al.'s empirical results revealing that automatically generated tests do not always help software testers find more defects in a program [**?**]. While Fraser et al. also consider coverage and mutation scores for automatically and manually generated test suites, they only use one automated test suite generator, EVOSUITE, while we additionally consider tests created with a deterministic tool called CodePro.

In contrast to our focus on manually and automatically generated test suites for real-world open-source applications, other related work has considered coverage and mutation scores for test suites written by students. For instance, Aaltonen et al. observe that automated grading programs may reward students for high-coverage test suites that actually have poor defect-revealing potential [**?**], concluding that a combination of coverage and mutation scores may be the best way to give students accurate feedback on the quality of their test cases. In addition, Shams and Edwards experimentally observe that mutation scores are lower than coverage values for student-implemented test suites [**?**]—a result that corresponds to what we found for open-source programs.

There are many past empirical studies that compare different coverage criteria. For instance, Frankl et al. experimentally compare the all-uses dataflow criterion to mutation adequacy, finding that, although mutation testing is more expensive that dataflow testing, neither approach was obviously better than the other [**?**]. As an additional example, Gligoric et al. also use mutation testing to empirically compare the effectiveness of non-adequate test suites that aim to fulfill different adequacy criteria, revealing that branch coverage and intra-procedural acyclic path coverage are the best [**?**]. Namin and Andrews report on an experimental investigation of test suite effectiveness, concluding that only non-linear relationships exist between test suite size, code coverage, and test effectiveness [**?**], thus motivating our search for non-linear fits in our own analyses. Finally, Li et al. compared four test adequacy criteria, finding that mutation testing is the best at finding seeded faults and minimizing the number of test required to ensure adequacy [**?**].

In addition to the aforementioned study by Just et al. [**?**], there have been several papers that examine the role that mutation testing should play in the experimental evaluation of testing strategies. Like Just et al., Andrews et al. also suggest that mutation analysis can be used to empirically compare testing methods [**?**]. In addition, Do and Rothermel report that mutation analysis can suitably support the study of regression test suite prioritization techniques [**?**]. There has also been much work in the design, implementation, and evaluation of mutation testing tools, with MuJava [**?**], Javalanche [**?**], PIT [**?**], and MAJOR [**?**] representing those most commonly

used in practical and experimental settings. A comparison of mutation testing tools by Delahaye and du Bousquet suggests that MAJOR is ideally suited for empirical studies [**?**], thus motivating our incorporation of this tool into our experiments.

VI. Conclusions and Future Work

The creation, execution, and maintenance of tests is one of the most expensive aspects of developing software, but tools such as CodePro and EvoSuite can aide developers in reducing the cost of this process. This paper examines the tradeoffs of using automatically and manually generated tests. The time for test suite generation, number of tests produced, time of execution of the resulting test suites, and quality of the test suites are evaluated. Quality is measured by the branch coverage and mutation scores of the resulting test suites.

The results indicate that automated test case generation tools can quickly generate more tests than are provided in manual test suites. While CodePro creates an average of 5% more test cases than EvoSuite and 16.4% more than were written manually and EvoSuite produced, on average, 4% more than were created manually, CodePro's test quality was low when both branch coverage and mutation score were considered. There was little correlation between branch coverage and mutation score for these test suites overall. EvoSuite's test suites, however, were positively correlated in quality between branch coverage and mutation score with an R^2 value of 0.4. Manually generated tests demonstrated an even stronger correlation between branch coverage and mutation score with an R^2 value of 0.64.

While some manually written tests were of higher quality overall in terms of branch coverage and mutation scores, the results indicate that more sophisticated, learning-based automated test generation tools such as EvoSuite can be used to produce test suites of similar quality on average compared to manual tests. On average, manual tests covered 31.5% of the branches while EvoSuite covered 31.86% of the branches. In terms of mutation score, EvoSuite's generated tests had an average mutation score of 39.89% compared to the average mutation score of 42.14% for manually written tests. Given the time reduction of using an automated tool compared to hand writing tests, these results are significant and encourage the use of automated tools for test production.

In future work, we will increase the number and types of case study applications under consideration. There are also a number of other tools including Randoop, T2, and DSC that we would like to include in our calculations. These three automated test case generation tools have proven competitive in producing high quality tests. Finally, we will consider other quality analysis measurements such as statement coverage and modified condition/decision coverage and tools such as PIT to back up our results and perform further analysis.

For full result analyses, additional data visualizations, statistical correlation summaries, and instructions for replicating our empirical study please refer to our web page: http://cs.uccs.edu/~kjustice/QSIC2014/.

VII. Acknowledgements

We thank René Just for his comments on earlier versions of this paper and for his support with the MAJOR system. We thank Gregory Kapfhammer for his insight and comments on earlier versions of this paper.