Does Automated White-Box Test Generation *Really* Help Software Testers?

Gordon Fraser[1] Matt Staats[2] Phil McMinn[1] Andrea Arcuri[3] Frank Padberg[4]

[1]Department of Computer Science, University of Sheffield, UK

[2]Division of Web Science and Technology, KAIST, South Korea

[3]Simula Research Laboratory, Norway

[4]Karlsruhe Institute of Technology, Karlsruhe, Germany

ABSTRACT

Automated test generation techniques can efficiently produce test data that systematically cover structural aspects of a program. In the absence of a specification, a common assumption is that these tests relieve a developer of most of the work, as the act of testing is reduced to checking the results of the tests. Although this assumption has persisted for decades, there has been no conclusive evidence to date confirming it. However, the fact that the approach has only seen a limited uptake in industry suggests the contrary, and calls into question its practical usefulness. To investigate this issue, we performed a controlled experiment comparing a total of 49 subjects split between writing tests manually and writing tests with the aid of an automated unit test generation tool, EVOSUITE. We found that, on one hand, tool support leads to clear improvements in commonly applied quality metrics such as code coverage (up to 300% increase). However, on the other hand, there was no measurable improvement in the number of bugs actually found by developers. Our results not only cast some doubt on how the research community evaluates test generation tools, but also point to improvements and future work necessary before automated test generation tools will be widely adopted by practitioners.

Categories and Subject Descriptors. D.2.5 [Software Engineering]: Testing and Debugging – *Testing Tools*;

General Terms. Algorithms, Experimentation, Reliability, Theory

Keywords. Unit testing, automated test generation, branch coverage, empirical software engineering

1. INTRODUCTION

Controlled empirical studies involving human subjects are not common in software engineering. A recent survey by Sjoberg et al. [28] showed that out of 5,453 analyzed software engineering articles, only 1.9% included a controlled study with human subjects. For software testing, several novel techniques and tools have been developed to automate and solve different kinds of problems and tasks—however, they have, in general, only been evaluated using surrogate measures (e.g., code coverage), and not with human testers—leaving unanswered the more directly relevant question: *Does technique X really help software testers?*

This paper addresses this question in the context of automated white-box test generation, a research area that has received much attention of late (e.g., [8, 12, 18, 31, 32]). When using white-box test generation, a developer need not manually write the entire test suite, and can instead automatically generate a set of test inputs that systematically exercise a program (for example, by covering all branches), and only need check that the outputs for the test inputs match those expected. Although the benefits for the developer seem obvious, there is little evidence that it is effective for practical software development. Manual testing is still dominant in industry, and research tools are commonly evaluated in terms of code coverage achieved and other automatically measurable metrics that can be applied without the involvement of actual end-users.

In order to determine if automated test generation is really helpful for software testing in a scenario without automated oracles, we performed a controlled experiment involving 49 human subjects. Subjects were given one of three Java classes containing seeded faults and were asked to construct a JUnit test suite either manually, or with the assistance of the automated white-box test generation tool EVOSUITE [8]. EVOSUITE automatically produces JUnit test suites that target branch coverage, and these unit tests contain assertions that reflect the current behaviour of the class [10]. Consequently, if the current behaviour is faulty, the assertions reflecting the incorrect behaviour must be corrected. The performance of the subjects was measured in terms of coverage, seeded faults found, mutation score, and erroneous tests produced.

Our study yields three key results:

1. The experiment results confirm that tools for automated test generation are effective at what they are designed to do—producing test suites with high code coverage—when compared with those constructed by humans.
2. The study does *not* confirm that using automated tools designed for high coverage actually helps in finding faults. In our experiments, subjects using EVOSUITE found the same number of faults as manual testers, and during subsequent mutation analysis, test suites did not always have higher mutation scores.
3. Investigating how test suites evolve over the course of a testing session revealed that there is a need to re-think test generation tools: developers seem to spend most of their time analyzing what the tool produces. If the tool produces a poor initial test suite, this is clearly detrimental for testing.

These results, as well as qualitative feedback from the study participants, point out important issues that need to be addressed in order to produce tools that make automated test generation without specifications practicably useful for testing.

ISSTA'13, July 15–20, 2013, Lugano, Switzerland
ACM 978-1-4503-2159-4/13/07
http://dx.doi.org/10.1145/2483760.2483774

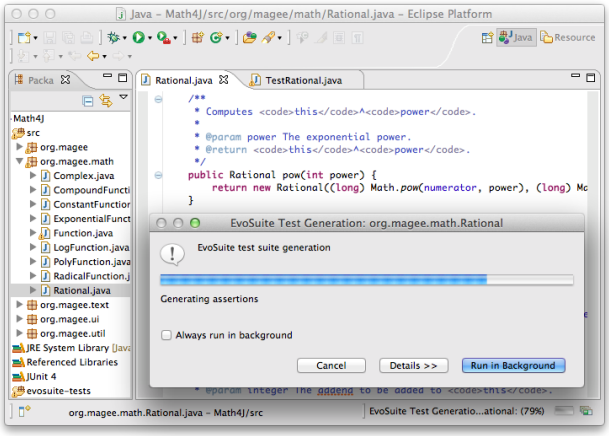


Figure 1: The EVOSUITE Eclipse plugin, generating test cases for a class—as used by subjects in the study.

2. EXPERIMENTAL METHODOLOGY

The purpose of this study was to investigate how the use of an automatic test generation tool, when used by testers, impacts the testing process compared to traditional manual testing. Our study was designed around a testing scenario in which a Java class has been developed and a test suite needs to be constructed, both to reveal faults in the newly created class and for later use in regression testing. We therefore designed our study around the following research questions (RQs):

How does the use of an automated testing tool impact . . .

*RQ*1 The structural code coverage achieved during testing?

*RQ*2 The ability of testers to detect faults in the class under test?

*RQ*3 The number of tests mismatching the specified behaviour?

*RQ*4 The ability of produced test suites to detect regression faults?

2.1 The Automated Testing Tool: EvoSuite

The automated testing tool used in our study is EVOSUITE [8], which automatically produces JUnit test suites for a given Java class. As input it requires the Java bytecode of the class under test, along with its dependencies. EVOSUITE supports different coverage criteria, where the default criterion is branch coverage over the Java bytecode. Internally, it uses a genetic algorithm to evolve candidate test suites according to the chosen coverage criterion, using a fitness function [8]. When EVOSUITE has achieved 100% coverage or hits another stopping condition (e.g., a timeout), the best individual in the search population is post-processed to (1) reduce the size of the test suite while maintaining coverage achieved and (2) add JUnit assertions to the test cases. As EVOSUITE assumes no specification, these assertions reflect the *observed* behaviour rather than the *intended* behaviour. The selection is based on mutation analysis, and the assertions of the final test suite are minimized with respect to the set of mutants they can expose [10].

For large scale experimentation, EVOSUITE can be used as a command-line tool. However, for our experiment, the Eclipse plugin was used (shown in Figure 1), with which the user can produce a test suite for a class by right-clicking its name in the project explorer. The Eclipse plugin usually only exposes two of EVOSUITE's many available configuration properties. However, in our experiments these were fixed: the time for test generation was set to one minute (this does not include time spent for minimization or assertion generation), and assertion generation was enabled.

Table 1: Study objects

"LOC" refers to the number of non-commenting source code lines reported by JavaNCSS (*http://www.kclee.de/clemens/java/javancss*), "Branches" is the number of branches reported by EVOSUITE, while "Mutants" is the number of mutants created by the MAJOR tool [14].

Project	Class	LOC	Methods	Branches	Mutants
XOM	*DocType*	296	26	242	186
Commons CLI	*Option*	155	42	96	140
Math4J	*Rational*	61	19	36	112

2.2 Study Subject and Object Selection

Running an empirical study involving human subjects leads to several challenges and possible pitfalls. Guidelines exist in the literature to help researchers to carry out such type of studies (e.g., see [15,27]). A common problem with controlled empirical studies is that, due to their cost and complexity, they are often limited in size. This reduces the power of the statistical analyses. For example, the studies surveyed by Sjoberg et al. [28] involved between 4 and 266 participants (49 on average). Of these participants, 87% were students. Furthermore, in 75% of the cases, the applications used in the experiments were constructed for the sole purpose of running those experiments.

2.2.1 Object Selection

As the number of subjects available to us was limited, we restricted our experiment to three Java classes to increase the likelihood of observing statistically significant effects. The classes were chosen manually, based on the following criteria:

1. EVOSUITE should be able to generate test suites with high coverage, as addressing cases where test generation struggles [7] is an ongoing research area. This excludes all classes with I/O dependencies and classes using Java Generics.
2. The source code documentation needs to be sufficient to serve as a specification. In particular, we required JavaDoc comments for all methods of the class.
3. The classes should not require the subjects to learn and understand complicated algorithms.
4. The classes should be non-trivial, yet feasible to reasonably test within an hour. In particular, we considered classes with fewer than 100 lines of code or classes without conditional expressions as too easy.
5. The classes should be understandable without extensively examining other classes in the same library. Notably, there should be neither many dependencies nor complex inheritance hierarchies.
6. Each class file should only involve one class definition (i.e., the class should not involve member classes).
7. The classes should represent three different types of applications and testing scenarios. In particular, we aimed to include one numeric class and one class dependent on string inputs.

We investigated the libraries used in our earlier experiments [8, 17], and identified a set of 25 candidate classes largely matching our criteria from the NanoXML, Commons CLI, Commons Math, Commons Collections, java.util, JDom, Joda Time and XOM libraries. We then identified several candidate classes of appropriate difficulty by first writing test suites for them ourselves, and ran a pilot study with volunteer subjects (who were not included later in the main experiment) on Commons CLI *Option*, Commons Math *Fraction*, and XOM *Attribute*. Seeing that even seasoned programmers required significantly longer than an hour for *Fraction* and *Attribute*, we replaced these classes with the similar but simpler *Rational* from the Math4J library and *DocType* from XOM.

Details of the classes used in the experiment can be found in Table 1. XOM is a tree-based API for processing XML documents (*http://www.xom.nu*), with *DocType* representing an XML document type declaration, which appears at the header of an XML file (e.g., "`<!DOCTYPE html>`"), potentially giving further details regarding a DTD. *Option* is part of the Apache Commons CLI API (*http://commons.apache.org/cli*) for parsing common line options passed to programs. The class represents a single command-line option (e.g., "`-a`", "`--all`", "`-param <value>`", etc.), including its short name, long name, whether the option is mandatory, and a short accompanying descriptor. Finally, *Rational*, from the Math4J project (*http://math4j.sourceforge.net*), represents a rational number.

Each Java class was injected with five faults prior to the experiment using the MAJOR [14] mutation analysis tool. We used the following procedure to select injected faults: For each of the classes, we used EVOSUITE to produce 1,000 random test cases with assertions. We then calculated the number of test cases killing each mutant produced by MAJOR (i.e., an assertion generated by EVOSUITE fails), thus estimating the difficulty of killing the mutant. Next, we partitioned all killed mutants into five equally sized buckets of increasing difficulty of being killed. From each of these buckets, we randomly selected one mutant, while prohibiting the selection of multiple mutants in the same method. All five selected mutants were applied to the class, producing the faulty version given to the subjects.

2.2.2 Subject Selection and Assignment

Email invitations to participate were sent to industrial contacts, as well as students and post-doctoral research assistants in the Department of Computer Science at the University of Sheffield. Due to physical laboratory space restrictions, only the first 50 people who responded were allowed to participate. One person failed to attend the experiment, leaving a total of 49, of which five were industrial practitioners and 44 from the Computer Science department. Of the five industrial developers, one was a Java programmer while the other four were web developers from a local software company. Of the 44 subjects from the Computer Science department, two were post-doctoral research assistants, eight were PhD students and the rest were second year or higher undergraduate students. Each subject had prior experience with Java and testing using JUnit (or similar, i.e., xUnit for a different programming language).

Before starting the experiment, we produced a fixed assignment of subject ID to class and technique, so that we had roughly the same number of subjects for each class and technique pairing. We assigned successive subject IDs to the computers in the lab, such that any two neighbouring subjects would be working on different classes with different techniques. Subjects freely chose their computers before any details of the study were revealed.

2.3 Experiment Process

Each subject received an experiment pack, consisting of their subject ID, a statement of consent, a background questionnaire, instructions to launch the experiment, and an exit survey. The pack also contained a sheet describing their target Java class and whether the subject was asked to test manually or using EVOSUITE. For those testing with EVOSUITE, the pack included further instructions on launching the EVOSUITE plugin.

Before commencing the experiment, each subject was required to fill in the questionnaire based on their background and programming experience. Subjects were then presented with a short tutorial of approximately 15 minutes, which provided a refresher of JUnit annotation syntax, along with the different assertion types available, and their various parameters. The tutorial further included screencasts demonstrating the use of Eclipse and the EVOSUITE tool. The slides of the presentation were made available as individual crib sheets for reference during the study.

Following the tutorial, subjects were given a short warm-up exercise to reacquaint themselves with Eclipse and JUnit, and to become familiar with the EVOSUITE plugin. The exercise consisted of an artificial ATM example class, including an intentional bug highlighted with code comments. Subjects were asked to write and run JUnit test cases, and to produce test suites with the EVOSUITE plugin. During this exercise, we interacted with the subjects to ensure that everybody had sufficient understanding of the involved tools and techniques.

After a short break the study commenced. To initiate the experiment, each subject entered their subject ID on a web page, which displayed a customized command to be copied to a terminal to automatically set up the experiment infrastructure (this process was also used for the tutorial example). The experimental infrastructure consisted of:

- Sun JDK 1.6.0-32
- Eclipse Indigo (3.7.2)
- The EVOSUITE Eclipse plugin
- An Eclipse workspace consisting of only the target project, with the class under test opened in the editor. The workspace for subjects performing manual testing was opened with an empty skeleton test suite for the target class.

All subjects used machines of roughly the same hardware configuration, booting Ubuntu Linux. As such, the technical setting for each individual subject was identical.

The stated goal was to test the target class as thoroughly as possible using the time available, referring to its project JavaDoc documentation for a description of its intended behaviour. We did not reveal the number of faults in a class to the subjects, but instructed subjects that test cases which reveal a fault in the class under test should fail. Subjects were told not to fix any of the code, unless the changes were trivial and eased the discovery of further faults.

Subjects of the EVOSUITE group were asked to start by producing a test suite using the plugin, and to edit the test cases such that they would fail if they revealed a fault on the class. They were also instructed to delete tests they did not understand or like, and to add new tests as they saw fit. As EVOSUITE uses a randomized search, each subject using it began with a different starting test suite. Furthermore, subjects working with EVOSUITE had to spend some of their time waiting for its results.

We modified Eclipse so that each time the test suite was run (initiated by a button-click), a copy of the test suite was saved locally and to a central storage point on the network for later analysis (presented in the following sections).

The subjects were given one hour to complete the assignment, and we asked them to remain seated even if they finished their task before the time limit. To be considered "finished", we required them to be certain that their test cases would a) cover all the code and b) reveal all faults. All subjects continued to refine their test suite until within 10 minutes of the end of study, as evidenced by the recorded test suite executions.

Including tutorial and break, the duration of the experiment was two hours. The task was completed under "exam conditions", i.e., subjects were not allowed to communicate with others, or consult with other sources to avoid introducing biases into the experimental findings. Each subject was paid 15 GBP for their time and involvement, and was asked to fill in an exit questionnaire before leaving.

2.4 Analysis of Results

Each subject in our study produced a sequence of test suites, with each new test suite saved whenever the subject executed it via Eclipse. These sequences are used to conduct our analysis, with the final test suite produced by each subject being of particular interest. For each test suite produced, we computed several metrics, specifically: statement, branch, and method coverage (using Cobertura[1]); the number of tests which fail on the original, correct system; the number of faults detected; the mutation score; and number of (non-assertion) statements and assertions present in each test.

These statistics form the base from which subsequent statistical analysis is done and our research questions are addressed. Statistical analysis was performed using the *scipy* and *numpy* Python frameworks and the R statistical toolset.

To determine which of the five individual study faults were detected by the test suites, for each class, each corresponding fault was used to a create a separate version of that class. This results in a total of six versions of each class for the analysis (five incorrect and one correct). In the subsequent analysis we refer to the correct version as the *original* version of the class. We then determined, for each faulty version of a class, if there exists a test which passes on the correct class, but fails on the faulty version.

The mutation score was computed by running the test suite using the MAJOR mutation framework, which automates the construction of many single-fault mutants and computes the resulting mutation score, i.e., the percentage of mutants detected by the test suites [14]. Tests which fail on the correct system are ignored and do not count towards the mutation score. (This was facilitated by modifications to MAJOR performed by the tool's author.) Note that these failing tests are still included when calculating coverage.

Finally, the number of statements and assertions was computed using automation constructed on top of the Eclipse Java compiler.

2.5 Threats to Validity

External: Many of our subjects are strictly students, and do not have professional development experience. However, analysis of the results indicated subjects did not appear to vary in effectiveness according to programmer experience or student/professional status. Furthermore, we see no reason why automatic test generation should be useful only to developers with many years of experience.

The classes used in our study were not developed by the subjects and may have been unfamiliar. However, in practice developers often must test the code of others, and as previously discussed, the classes chosen were deemed simple enough to be understood and tested within the time allotted through pilot studies. This is confirmed in the survey, where we asked subjects if they felt they had been given enough time for the experiment. Only three subjects strongly disagreed about having enough time, whereas 33 subjects stated to have had enough time; there was no significant difference between EVOSUITE users and manual testers on this question, indicating there was sufficient time for both groups. Additionally, the classes selected were relatively simple Java classes. It is possible more complex classes may yield different results. As no previous human studies have been done in this area, we believe beginning with small scale studies (and using the results to expand to larger studies) is prudent.

Our study uses EVOSUITE for automatic test generation. It is possible using different automatic test generation tools may yield different results. Nevertheless, EVOSUITE is a modern test generation tool, and its output (both in format and structural test coverage achieved) is similar to the output produced by other modern test generation tools, such as *Randoop* [21], *eToc* [32], *TestFul* [3], *Java PathFinder* [23], *Dsc* [13], *Pex* [31], *JCrasher* [4], and others.

[1]Note that Cobertura only counts conditional statements for branch coverage, whereas the data given in Table 1 lists branches in the traditional sense, i.e., edges of the control flow graph.

Table 2: Effect sizes for EVOSUITE results compared with manual testing for the three study classes

When p-values are lower than 0.05, the effect size $\hat{A}_{12}$ is highlighted in bold.

Variable	Option		Rational		DocType	
	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value	$\hat{A}_{12}$	p-value
# failing tests on original	**0.97**	0.001	0.70	0.289	**0.83**	0.026
# of detected faults	0.31	0.165	0.43	0.738	0.50	-
Mutation score	0.65	0.321	0.37	0.530	0.22	0.065
% Statement coverage	**1.00**	0.001	0.79	0.109	0.37	0.399
% Branch coverage	**1.00**	0.001	0.64	0.431	**0.19**	0.040
% Method coverage	**1.00**	0.001	**0.91**	0.014	0.70	0.178

Internal: Extensive automation is used to prepare the study and process the results, including automatic mutation tools, tools for automatically determining the faults detected over time, tools measuring the coverage achieved by each test suite, etc. It is possible faults in this automation could lead to incorrect conclusions.

To avoid a bias in the assignment of subjects to objects we used a randomized assignment. Subjects without sufficient knowledge of Java and JUnit may affect the results; to avoid this problem we only accepted subjects with past experience (e.g., all undergraduates at the University of Sheffield learn about JUnit and Java in the first year), as confirmed by the background questionnaire, and we provided the tutorial before the experiment. In addition, the background questionnaire included a quiz question showing five JUnit assertions, asking for each whether it would evaluate to true or to false. On average, 79% of the answers were correct, which strengthens our belief that the existing knowledge was sufficient for the experiment. A further threat to internal validity may result if the experiment objectives were unclear to subjects; to counter this threat we thoroughly revised all our material, tested it on a pilot study, and interacted with the subjects during the tutorial exercise to ensure they understood the objectives. As each subject only tested one class with one technique, there are no learning effects that would influence our results.

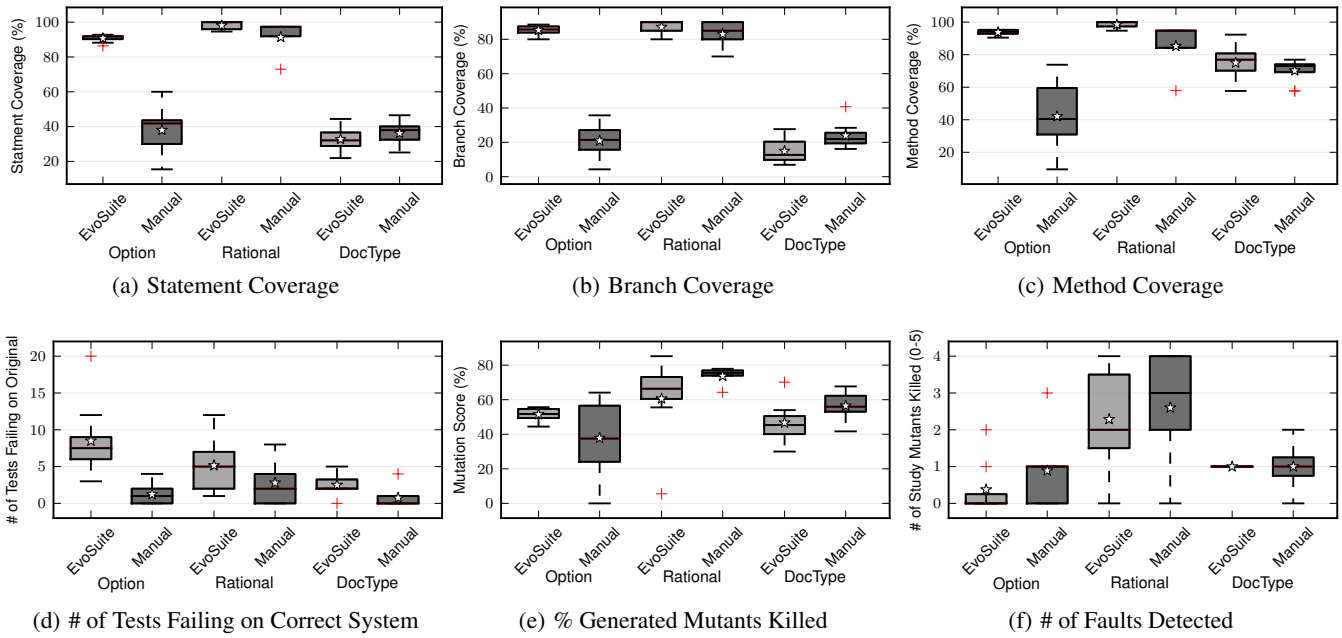
Construct: We used automatically seeded faults to measure the fault detection ability of constructed test suites. While evidence supports that the detection of the class and distribution of faults used correlates with the detection of real world faults, it is possible the use of faults created by developers may yield different results.

Conclusion: We conducted our study using 49 subjects and three Java classes. Thus for each combination of testing approach (EVOSUITE and manual) and Java class, six to nine subjects performed the study. This is a relatively small number of subjects, but yields sufficient statistical power to show an effect between testing approaches. Furthermore, the total number of test suites created over the course of the study is quite high (over 1000), easily sufficient for analysis examining the correlations between test suite characteristics and fault detection effectiveness.

3. RESULTS

In answering our original research questions, we use only the final test suite produced by each subject, as this represents the end product of both the manual and tool-assisted testing processes. We explore how the use of automated test generation impacts the evolution of the test suite in Section 4.1.

The results are summarized in the form of boxplots in Figure 2, with the outcome of statistical tests summarized in Table 2. The Mann-Whitney U-test was used to check for statistical difference



(a) Statement Coverage (b) Branch Coverage (c) Method Coverage

(d) # of Tests Failing on Correct System (e) % Generated Mutants Killed (f) # of Faults Detected

Figure 2: Test suite properties, comparing EVOSUITE against manual testing (boxes spans from 1st to 3rd quartile, middle lines mark the median, whiskers extend up to $1.5\times$ the inter-quartile range, while plus symbols represent outliers and stars signify the mean).

among the stochastic rankings of these two groups for each variable, and the Vargha-Delaney $\hat{A}_{12}$ statistic was used to calculate standardized effect sizes. We also computed common statistics, such as minimum, maximum, mean, median, standard deviation, and kurtosis. For space reasons we do not exhaustively list these values, but instead refer to them in our discussion as needed. There is disagreement in the literature concerning which statistical tests should be used and how to interpret their results; in this paper, we follow the guidelines presented by Arcuri and Briand [2].

The data for three subjects was discarded, as one subject produced no test suites (for *DocType*), and two subjects ignored study instructions and modified the class interface (for *Rational*). As these modifications were not captured by the EVOSUITE plugin, we could not analyze the resulting test suite.

After conducting the study, the analysis indicated unexpected results for *DocType*. Although per our class selection criterion, we expected EVOSUITE would achieve high coverage with *DocType*, the coverage values achieved by participants using EVOSUITE were very low (as seen in Figure 2), no greater than 44%. Upon investigation, we identified a configuration problem in EVOSUITE related to how the Eclipse plugin constructed the classpath when launching EVOSUITE during test generation; specifically, EVOSUITE could not load the *nu.xom.Verifier* class. As it is possible to instantiate *DocType* even without *Verifier*, EVOSUITE silently ignored this problem. However, many of the methods of *DocType* indirectly depend on *Verifier*, and calling such a method leads to a *NoClassDefFoundError*. Consequently, EVOSUITE produced many test cases for *DocType* similar to the following:

```
public void test0() {
  String string0 = ...;
  DocType docType0 = ...;
  try {
    docType0.setInternalDTDSubset(string0);
    fail("Expecting exception: NoClassDefFoundError");
  } catch(NoClassDefFoundError e) {
    // Could not initialize class nu.xom.Verifier
  }
```

This explains the simultaneous high method coverage and low statement/branch coverage achieved over this class. This configuration problem only affected EVOSUITE, not Eclipse itself, and consequently these test cases would fail when executed in Eclipse, as the *NoClassDefFoundError* would not occur.

Interestingly, none of the subjects noted this issue, and the experiment was conducted as planned. The results on *DocType* therefore do not represent the standard behaviour of EVOSUITE. However, they do represent an interesting case: what happens if the test generation tool produces bad or misleading test cases? We therefore decided to keep the data set.

RQ1: Structural Code Coverage Achieved

As seen in Figure 2(a–c), for both *Option* and *Rational*, the use of EVOSUITE improves code coverage for every structural coverage criteria used. The relative increases in median coverage range from 9.4% in the case of branch coverage for *Rational*, to 300%—a threefold increase—for branch coverage for *Option*. Indeed, the improvement in coverage when testing *Option* with the aid of EVOSUITE is particularly substantial: the minimum coverage achieved with EVOSUITE derived test suites is 80.0% and 90.48% for branch and method coverage, while the maximum coverage achieved by any subject working without EVOSUITE is 35.71% and 73.81%, indicating nearly all the coverage achieved during testing is likely due to automatically generated tests.

Considering the standardized effect sizes and the corresponding p-values in Table 2, results for *Option* are as strong as possible ($\hat{A}_{12} = 1$ and p-value close to 0.0). For *Rational*, there are strong effect sizes (from 0.69 to 0.94), but sample sizes were not large enough to obtain high confidence in statistical difference for branch coverage (p-value equal to 0.247).

The results for *Option* and *Rational* matched our expectations: the development of automatic test generation tools has long focused on achieving high structural coverage, and the high coverage achieved here mirrors results found in previous work on a number of similar tools. For *DocType*, however, the use of EVOSUITE results in considerably lower branch coverage, with a relative change

in the median branch coverage of -42.12% (though method coverage tends to be slightly higher). As discussed above, this is due to a configuration error; given that EVOSUITE typically achieves around 80% branch coverage within one minute over *DocType*, we expect that the behavior observed over the *Rational* and *Option* classes would also apply on *DocType* under normal circumstances.

Nevertheless, we conclude that in scenarios suited to automated test generation, generated test suites do achieve higher structural coverage than those created by manual testers.

RQ1: *Automatically generated test suites achieve higher structural coverage than manually created test suites.*

RQ2: Faults Detected

For all three classes, there is no case in which the ability of subjects to detect faults improves by using EVOSUITE, and in fact detection often slightly decreases. For example, from Figure 2(f) we see *Option* shows a slight benefit when using manual testing, with average fault detection of 0.89 compared to 0.38 (effect size 0.31). For *Rational* the data show that manually created test suites detect 2.33 faults versus the 2.12 detected with test suites derived with EVOSUITE (effect size 0.46). However, test suites created for *DocType* find on average the exact same number of faults, 1.0 (effect size 0.5). In no case are the differences in fault detection statistically significant at $\alpha = 0.05$, as the lowest *p*-value is equal to 0.165 (for *Option*). A larger sample size (i.e., more subjects) would be needed to obtain more confidence to claim that EVOSUITE actually is worse than manual testing.

RQ2: *Using automatically generated test suites does not lead to detection of more faults.*

Of the results found, this is perhaps the most surprising. Automated test generation tools generate large numbers of tests, freeing testers from this laborious process, but also forcing them to examine each test for correctness. Our expectation was that either testers would be overwhelmed by the need to examine and verify the correctness of each test, and thus be largely unable to make the necessary changes to the test to detect faults, *or*, that testers would be relatively effective in this task, and, free from the need to create their own test inputs, could detect faults more efficiently.

To determine if this behavior stems from the generation of poor tests suites by EVOSUITE, we examined how many faults subjects *could* have found using generated tests given the right changes to the test assertions. To estimate this, we looked at the initial test suites generated by EVOSUITE. We assume that a test suite can potentially reveal a fault if there exists a test which passes on the class with the fault, and fails on the original, correct class (i.e., there is a test where an assertion, if corrected, would reveal the fault). On average, test suites for *Option* could reveal 3.0 faults, test suites for *Rational* 2.86 faults, and test suites for *DocType* 2.6. Consequently, it would have been possible for subjects using EVOSUITE to find more faults than manual testers if they had identified and fixed all incorrect assertions. We take a closer look at the influence of assertions and how they vary when using EVOSUITE in Section 4.2.

RQ3: Tests Mismatching the Specification

For all three classes, the number of tests failing on the original version (i.e., the version of the class without the seeded faults) is larger when EVOSUITE is used (cf. Figure 2(d)). Each failing test represents a misunderstanding in how the class should operate, manifested as an incorrect assertion. For *Option*, the number increases from 1.22 on average for manually written tests to 8.5 for participants using EVOSUITE; for *Rational* the number increases from 2.8 to 5.14; and for *DocType* the number increases from 0.75 to 2.5. The increase is statistically significant for *Option* ($p = 0.001$) and *DocType* ($p = 0.026$), whereas *Rational* ($p = 0.289$) would need a larger sample size to achieve significance.

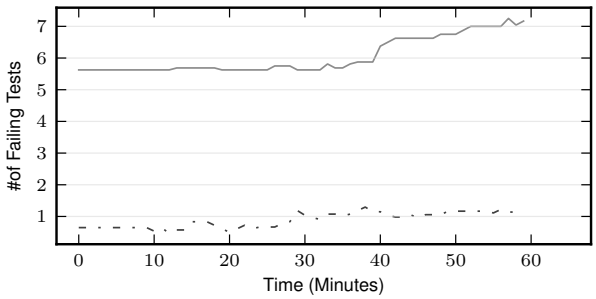


Figure 3: Average number of test cases failing on the original, correct version of the class. EVOSUITE users are shown with dark gray solid lines, manual testers light gray dashed lines.

Naturally, we expect several failing tests in the initial test suite when using EVOSUITE: assertions produced by EVOSUITE reflect the behaviour of the class they are generated from, which in the study is faulty. Over time, as the test suite evolves, we expected these failing tests to be corrected. However, Figure 3 shows that this is not the case; the number of incorrect tests remains fairly constant for both EVOSUITE users and manual testers, and even slightly increases for EVOSUITE users (due to *Option*).

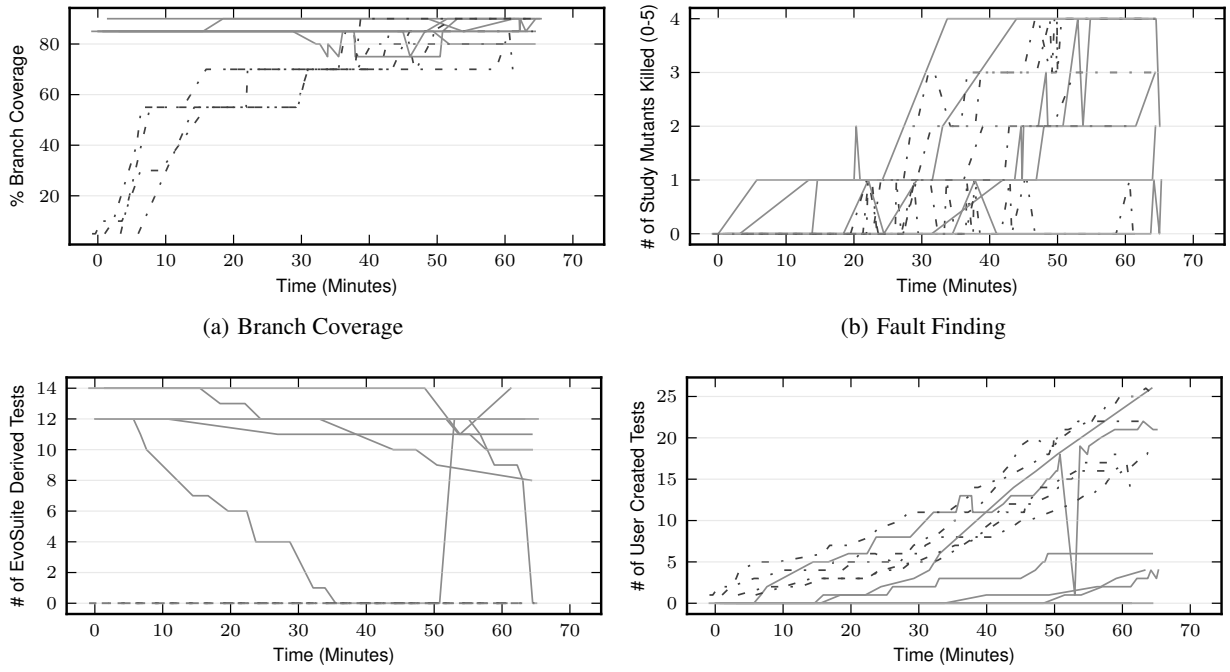
The persistent number of failing tests may occur because testers struggle to understand the generated tests, or because in general testers struggle to correct failing tests, and the generation process merely exacerbates this. In any case, the existence of failing tests represents a potential drain on time, as these tests may fail to catch faults in the program or may signal the existence of a fault where none exists, both undesirable outcomes.

RQ3: *Automatically generated test cases have a negative effect on the ability to capture intended class behaviour.*

RQ4: Regression Fault Detection

We estimate the ability of subjects' test suites to find regression faults by examining the mutation score achieved. In contrast to the results for *RQ2*, the results for *RQ4* are mixed: the ability of subjects to construct test suites capable of detecting faults later introduced in the class under test is impacted by the use of EVOSUITE, but only for one class. As shown in Figure 2(e), when performing regression testing over *Option*, test suites derived from EVOSUITE detect, on average, 51.46% of mutants as compared to 37.82% detected by manual testing alone. This indicates that the much higher structural coverage achieved over this class, while apparently not beneficial at detecting existing faults in it, nevertheless does help improve the ability to detect mutants later introduced.

However, for the other two classes, *Rational* and *DocType*, test suites constructed with EVOSUITE seem to perform worse. For *Rational*, manually created test suites killed on average 72.92% of generated mutants, a rather consistent improvement over the 60.56% of mutants found by the EVOSUITE derived test suites. For *DocType*, 56.50% and 46.65% of mutants were killed by manually created and EVOSUITE generated test suites, respectively. In both cases, however, the most effective test suite was created by a subject using EVOSUITE (note the outliers in Figure 2(e)). Only for *DocType* there is enough evidence to claim results can be statistically significant (*p*-value equal to 0.065), though this is influenced by the configuration problem discussed earlier.



(a) Branch Coverage

(b) Fault Finding

(c) # of EvoSuite Derived Tests

(d) # of Tester Created Tests

Figure 4: Test suite evolution for *Rational*. EVOSUITE users shown in light gray solid lines, manual testers dark gray dashed lines.

We hypothesize that to some extent this difference among classes is due to the difference in method coverage achieved over *Option*: as noted previously, we selected independent faults to be part of each class, and some methods do not contain faults. During mutation testing, these methods are mutated, and the tests generated by EVOSUITE targeting these methods—which are correct, as the methods themselves are correct—will detect these mutants. As manually created test suites may not cover these methods, they cannot detect these mutants. In contrast, for both *Rational* and *DocType*, test suites manually created or derived using EVOSUITE achieved similar levels of method coverage, and this behavior is thus absent. Our results for *RQ4* thus reflect previous empirical studies relating structural coverage and mutation scores—higher structural coverage roughly corresponds to higher levels of mutation detection ability [20].

On the whole, these results indicate that the use of automatic test generation tools may offer improvements in regression testing in scenarios where testers struggle to manually generate sufficient tests to cover a system. However, the relationship between coverage and mutation score is clearly not as strong as found in previous studies (where correlations above 0.9 were common) highlighting the impact of the tester in constructing effective regression test suites [20]. For example, although on *Rational* the coverage values are higher for EVOSUITE, its mutation score is lower.

We provide two possible conjectures why manual testers achieved higher mutation scores with similar coverage. First, consider again the results from RQ2 and RQ3: users of EVOSUITE produced more tests failing on the original version of a class than manual testers. Although a failing test still contributes towards code coverage, it cannot detect any mutants by definition (mutation analysis assumes that the tests pass on the original version of the class). Consequently, the mutation score is based on only the passing subset of the produced test cases. It is therefore likely that if users had managed to correct more assertions, the mutation score of the EVOSUITE tests would have been significantly higher.

Second, it is possible that the assertions generated by EVOSUITE are weaker than those written by manual testers. Both conjectures imply that assertion generation is in strong need of further research.

RQ4: *Automated test generation does not guarantee higher mutation scores.*

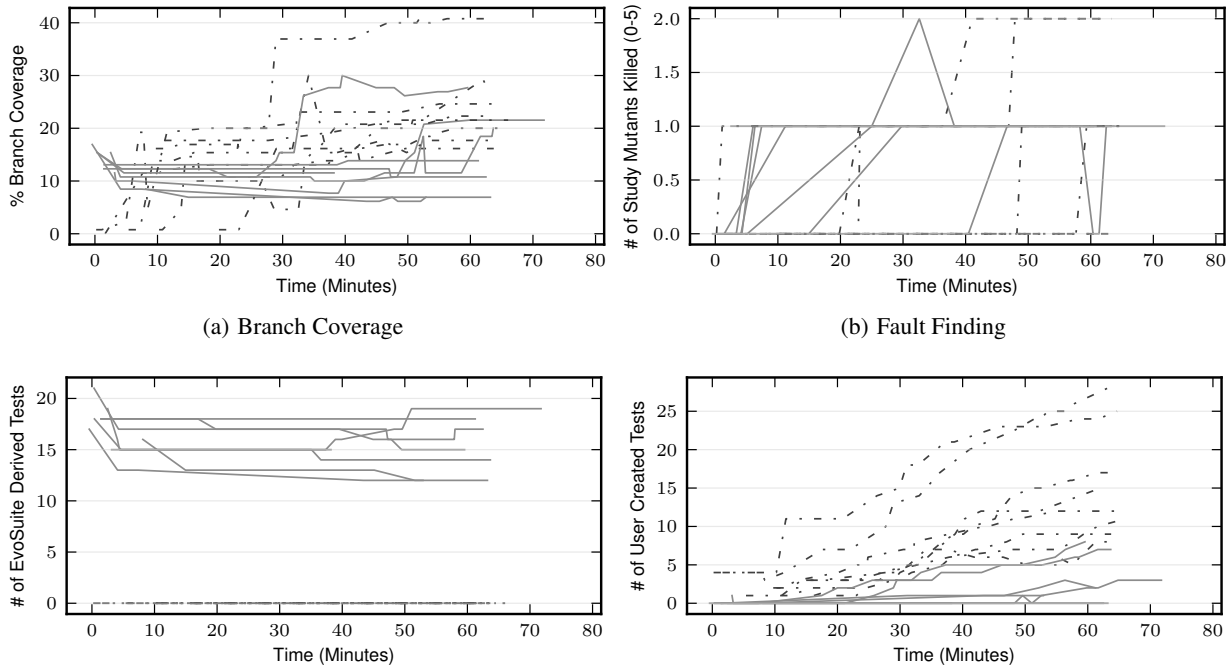
4. DISCUSSION

Our results indicate that while the use of automated test generation tools can improve structural coverage over manual testing, this does not appear to improve the ability of testers to detect current or future regression faults. These results highlight the need to improve automated test generation tools to be capable of achieving not just higher structural coverage, but also better fault finding when actually *used* by testers. To accomplish this, we require a better understanding of how the testing process is influenced by the use of automated testing tools. Based on the observations made in the previous section, we further explore how the use of automated test generation impacts testing effectiveness and discuss implications for future work in automated test generation.

4.1 Evolution of a Test Suite

As shown previously, even when given tests which were largely the same and unhelpful, subjects largely used the test suite produced by EVOSUITE, resulting in test suites with considerably less coverage (though surprisingly, similar fault detection effectiveness) than manually produced test suites.

This highlights that beginning the testing process with an automated testing tool is not a simple boost, a pure benefit with no downsides. Instead, the use of these tools results in the creation of a different starting point for testing from that of traditional manual testing, one which changes the tasks the tester must perform and thus influences how the test suite is developed. Understanding the differences in how a test suite evolves during testing with respect to the starting point may suggest how automated testing tools



(a) Branch Coverage

(b) Fault Finding

(c) Number of EvoSuite Derived Tests

(d) Number of Tester Created Tests

Figure 5: Test suite evolution for *DocType*. EVOSUITE users shown in light gray solid lines, manual testers dark gray dashed lines.

can better serve testers. Towards this, in Figures 4 and 5, we illustrate how test suites change over time for the *Rational* and *DocType* classes, illustrating the branch coverage achieved, the number of EVOSUITE-derived tests, the number of user-created tests, and the number of study faults detected. (*Option* displays behavior similar to *Rational*.) Each line represents a single subject's test suite over time; dark gray dashed lines represent manually constructed test suites, while lighter solid lines represent EVOSUITE derived test suites.

First, consider the data related to *Rational* (Figure 4). EVOSUITE assisted subjects begin with a test suite achieving high coverage, which they then must begin to manually verify and understand. We see after 20–30 minutes these testers begin testing in earnest—having spent roughly three minutes per generated test understanding their starting point—with resulting fluctuations in branch coverage, a decrease in EVOSUITE derived tests[2] (though most of these tests are retained) and a corresponding uptick in the number of added tests to around five, though two testers create even more tests than some of those manually testing the system.

In contrast, unassisted testers have no need to understand or modify any generated tests. These subjects exhibit an immediate, linear increase in the number of tests created, with a rapid increase in branch coverage approaching (but not quite achieving) that of EVOSUITE derived test suites within 40 minutes. Thus while in the end, all subjects produced final test suites performing well in terms of coverage and fault detection, the path to these effective test suites varies considerably depending on the process used.

To quantify this dichotomy, we computed the Spearman correlation of the number of user and EVOSUITE created tests against fault detection. For EVOSUITE derived test suites, we found that test suite size has a moderate, positive correlation with fault detection (0.45). However, the number of user created tests has a stronger relationship with fault detection (0.72), while the number of EVOSUITE derived tests has a moderate negative correlation (-0.50), highlighting the need to evolve the test suite. Branch coverage, surprisingly, has a weak relationship with fault detection (-0.21). When manual testing, however, both branch coverage and test suite size have a strong relationship with bug detection (0.79), as previous work suggests [20].

Now consider the data related to *DocType* (Figure 5). Despite the issues with this class discussed in Section 3, the behavior here is similar. Subjects using EVOSUITE tend to delete tests much earlier than manual testers for *Rational*, dropping 5-10 tests within the first 10 minutes, but are on the whole unwilling to scrap the entire test suite. Again, after about 30–40 minutes, changes in the number of EVOSUITE derived tests, the number of user created tests, and branch coverage occur, though as with *Rational*, for most subjects the coverage increases only slightly (or decreases) and few new tests are created. Manual testers again start immediately, and after 30 minutes they have created 5–15 tests, achieving greater coverage than most subjects using EVOSUITE achieve during the entire experiment (though all subjects struggle to achieve high coverage). Spearman correlations (not shown) again reflect this, albeit not as strongly.

The problem here is that, unlike *Rational*, the starting point offered by EVOSUITE is terrible in terms of both coverage and test quality, but users nevertheless appear to invest the majority of their effort into making use of this test suite. While both groups of subjects ultimately achieve similar levels of fault finding (one fault detected), we believe this is more a consequence of the relative difficultly of testing *DocType*, and note as shown in Figure 2 that the mutation score for manual test suites is roughly 10% greater than those for EVOSUITE derived test suites.

Based on this, we can clearly see that when using automatic test generation tools, the starting point given to testers in the form of the test suite generated is where they tend to stay. Even very bad test suites require time for testers to understand before they can

[2]We use the method names of tests originally produced by EVOSUITE to determine whether a test case has been added or deleted.

Table 3: Correlation of Mean Number of Assertions with Fault Detection, Using Subjects' Final Test Suites

"MA" is the mean number of assertions, "GMK" is the number of generated mutants that were killed, "SFD" is the number of study faults detected. Correlations not statistically significant at $\alpha = 0.05$ are denoted with an asterisk.

	EvoSuite *Testing*		*Manual Testing*	
	SFD	**GMK**	**SFD**	**GMK**
Option **MA**	-0.06*	0.19	0.26	0.10*
Rational **MA**	-0.65	-0.73	0.30	0.35
DocType **MA**	-0.12*	0.05*	0.17	0.17

begin to repair them, and following this they are loath to replace them completely. This stickiness, which naturally does not exist during manual test suite construction, is cause for concern. Strong evidence exists that, in many, perhaps most cases, automatic test generation performs poorly in terms of coverage [7, 17]. If testers struggle to improve and correct poor generated test suites, the use automatic generation tools may be a drag on the testing process.

4.2 Influence of Assertions

In Section 3, we noted that EvoSuite generated tests capable of detecting faults, if subjects could understand and correct the assertions. One possible explanation why subjects failed to correct these assertions is a problem in the generated test cases. According to the exit survey, subjects were happy about the length and readability of the test cases, and they agreed that confirming the correctness of an assertion is easier than writing an assertion for a generated test case. However, while they thought that EvoSuite chose good assertions, they often stated that it chose *too many* assertions.

To understand how EvoSuite generated assertions differed from those of manual testers, we examined the number of assertions per test for both sets of subjects, and computed the Spearman correlation of the number of assertions constructed and the fault finding effectiveness. These results are listed in Table 3 and Figure 6.

We can see from Figure 6 subjects performing manual testing do in fact tend to construct fewer assertions, producing an average of 1.44 to 1.68 assertions per test, versus the 1.41 to 4.9 assertions per test present in test suites derived from EvoSuite. In the case of *Option* and *Rational*, testers examining EvoSuite tests must inspect up to 2–3 times more generated assertions than they typically chose to construct, a potentially large increase in effort.

Furthermore, from Table 3 we see that when manually testing *Rational* (the only case in which the number of assertions constructed by manual testers exhibits variation) the mean number of assertions per test suite has a positive (albeit low/moderate) correlation with mutation score and bug detection (0.30 and 0.35). However, when using EvoSuite to test *Rational*, the mean number of assertions has a moderate, negative correlation with effectiveness (-0.65 and -0.73). Thus as EvoSuite derived test suites evolve to become more effective, extraneous tests/assertions are dropped and replaced with tests using a smaller number of assertions. This hints that it may be possible to replace the relatively large number of generated assertions with a smaller number of more targeted assertions, with no decrease in fault detection effectiveness.

Note another possibility exists concerning why subjects failed to correct assertions: a problem in understanding the class under test and its specification. Indeed, the subjects using EvoSuite consistently felt that the difficulty in understanding generated unit tests depended more on the complexity of the class, not the actual tests. Only for *Rational*, the smallest class, did subjects using EvoSuite feel testing was easier than subjects manually testing. Considering that subjects were more effective at fixing assertions for *Rational* than other classes, it seems that the more difficult a

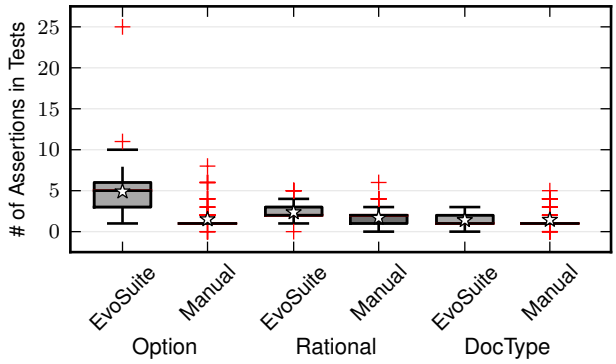


Figure 6: Number of assertions per test (each box spans from 1st to 3rd quartile, middle line marks the median, whiskers extend up to $1.5\times$ the inter-quartile range, while plus symbols represent outliers and stars signify the mean).

class is to understand, the more difficult it becomes to successfully apply an automated test generation tool. Developing methods of selecting assertions to mitigate this issue seems key to producing generated test suites which testers feel comfortable using.

4.3 Implications for Future Work

4.3.1 *Earn the Trust of Testers*

As discussed in the previous two subsections, even given terrible generated tests, most testers still invest a significant amount of effort in understanding and attempting to repair the tests. In short, testers trust these tools, much as they trust their compiler. However, as evidenced by *DocType* and previous empirical studies, this trust is misplaced. These tools have been developed to provide a test suite using a *best-effort* approach, and thus will always present the user with a test suite—even when the quality of the test suite (in terms of coverage, etc.) is quite poor. Over time, testing tools which require significant effort from testers but which only sometimes repay that effort seem likely to be discarded, as testers learn to not trust the tool and invest their time elsewhere (in this case, manual test development).

We therefore believe developers of automatic test generation tools must develop strong methods of culling their test suites, presenting only tests and groups of tests which are very likely to repay the effort required to understand them. Thus we need not only the ability to generate test suites, but also to assess our confidence in them before presenting them to testers. In particular, methods of determining when the tool is performing well or poorly appear to be needed. If EvoSuite was capable of, for example, distinguishing the quality of the test suites generated for *Option* and *DocType*—either by coverage, or mutation score, or by detecting configuration errors—testers could be spared useless test suites while still gaining the benefits of EvoSuite when those benefits are present. Given previous results indicating that situations similar to *DocType* occur frequently in practice [7], such a method seems a necessary step towards creating trustworthy test generation tools.

Naturally, part of improving our ability to generate tests which testers trust is adapting our test generation techniques to create more understandable tests. This need has been established in the testing research literature (e.g., [1, 9, 10, 24]) and underscored by our study's results; however, to the best of our knowledge no human subject study has been conducted to determine what factors impact human understanding of tests. In lieu of this, researchers have developed ad-hoc metrics for quantifying user understanding,

in particular assuming that test length and the number methods invoked correspond to user understanding. While these metrics appear to be intuitively sensible, given the importance of accurately predicting test understanding, human subject studies examining the relationship between these proposed factors and test understanding are warranted.

4.3.2 High Test Coverage is Not the Goal

Much of the work in test case generation focuses on improving structural test coverage under the assumption that improving coverage is the key to improving test generation effectiveness. While this is inevitably partly true—we cannot detect faults in unexecuted code—our results indicate that at best a moderate relationship between test coverage and fault detection is present (correlation generally less than 0.6). From this we infer two implications. First this underscores the need for continual user studies, since we have no effective proxy measure for determining if a tool will be effective when used by users. Mechanically evaluating automatic test generation tools, without user studies, increases the likelihood of forming misleading or incorrect conclusions.

Second, this highlights the need for carefully considering how assertions are generated during test generation. We must execute incorrect code in order to detect faults, but without an effective test oracle, detection remains unlikely. Indeed, previous work has demonstrated that careful consideration of how the test oracle is selected can increase the detection of faults [29], and based on our results we believe additional work focused on improving test oracle selection is needed.

4.3.3 User Suggestions

As part of the survey, subjects were given the chance to provide feedback and suggestions on improving EvoSuite and automated test generation. Several subjects not using EvoSuite commented that they would like to use an automated tool, in particular to test getters/setters and other trivial pieces of code. As one subject put it, test generation tools would be great to take over the "boring" parts of testing.

Subjects using EvoSuite listed a number of concrete suggestions on how to improve unit test generation. The suggestion given most frequently (by seven subjects) was that automatically generated test cases need a short comment explaining what they do. An additional frequent suggestion was to reduce the number of assertions per test. In particular, if one test has several assertions it might even be better to split it up into several tests with fewer assertions.

An interesting suggestion was to prioritize test cases by their importance to avoid the problem of "1000 tests without structure", although of course the question what makes a test case important is not easy to answer. Finally, subjects using EvoSuite were generally happy about the readability, and several subjects explicitly commended it. However, there were a couple of useful suggestions on how to improve the tests, in particular by changing variable and method naming as well as value choices to something closer to what manual testers would choose.

5. RELATED WORK

Although controlled human studies are not common in software engineering, there has been some recent work evaluating techniques with users (e.g., [26]), and not always with positive results. Parnin and Orso [22] conducted a study to determine if debugging techniques based on statement ranking help to locate bugs quicker. They found that significant improvements in speed were "limited to more experienced developers and simpler code". Staats et al. [30] conducted a study to investigate if dynamic invariant generation tools are helpful, e.g. in creating automated oracles. Thirty subjects were asked to classify automatically generated invariants as correct or incorrect. Unfortunately, subjects "misclassified $9.1 - 39.8\%$ of correct invariants and $26.1 - 58.6\%$ of incorrect invariants", "calling into question the ability of users to effectively use generated invariants".

One study addressing the question of how automated testing tools compare to manual testing was carried out by Ramler et al. [25], involving 48 subjects. The fault detection of manually written test cases was compared with randomly-generated test cases using Randoop. Fault detection rates were found to be similar, although the techniques revealed different kinds of faults.

The EvoSuite tool is based on the use of metaheuristic search algorithms [6], e.g., genetic algorithms, which have achieved several "human competitive" results (e.g., for genetic programming [16]). While the application of search-based algorithms in software engineering (SBSE) has been increasing in the last few years [11], there have only been a few studies involving human subjects and SBSE. Pastore et al. [24] used crowdsourcing to verify the correctness of assertions generated with EvoSuite against JavaDoc documentation, but unlike our study the source code of the tested classes was not shown to participants. In this study, the human participants (crowd-workers) performed well at this task, but only as long as the documentation and tests were both readable and understandable. Afshan et al. [1] also used crowdsourcing to evaluate the readability of test cases involving string inputs produced with another search-based testing tool, IGUANA. Finally, Souza et al. [5] compared the solutions generated with search-based techniques against those constructed by humans, concluding that SBSE was capable of generating solutions of higher quality and, of course, in less time.

6. CONCLUSIONS

Beginning with earliest attempts at automated test data generation in the 70s, the assumption has been even partial automation of the testing process yields a net benefit. Initially, test data generation explicitly assumed that test data would be analyzed manually [19], although later work distinguishes between the availability of an automated test oracle and manual test oracles [33]. Successive work on white-box test generation has frequently ignored the question of using the test data after generation, and focused on the technically challenging aspects of test generation.

Our study reveals that merely automatically generating a set of test cases, even high coverage test cases, does not necessarily improve our ability to test software. This result can be seen as a call to arms to the software testing research community. It is time for software testing research to consider the follow-up problem to white-box test data generation: once we have generated our test data, how should the developer *use* it?

In order to facilitate reproduction of our study and future studies in software testing, we provide all experimental material of this study as well as EvoSuite on our Web site:

`http://www.evosuite.org/study`

7. ACKNOWLEDGEMENTS

We thank René Just for his help in modifying the MAJOR mutation system. This work is supported by a Google Focused Research Award on "Test Amplification"; the World Class University program under the National Research Foundation of Korea (Project No: R31-30007); EPSRC grant EP/I010386/1, "RE-COST: REducing the Cost of Oracles in Software Testing"; and the Norwegian Research Council.

8. REFERENCES

[1] S. Afshan, P. McMinn, and M. Stevenson. Evolving readable string test inputs using a natural language model to reduce human oracle cost. In *Int. Conference on Software Testing, Verification and Validation (ICST)*, 2013. (To appear).

[2] A. Arcuri and L. Briand. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*. (To appear).

[3] L. Baresi, P. L. Lanzi, and M. Miraz. Testful: an evolutionary test approach for Java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 185–194, 2010.

[4] C. Csallner and Y. Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[5] J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. The human competitiveness of search based software engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*, pages 143–152, 2010.

[6] G. Fraser and A. Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *ACM Symposium on the Foundations of Software Engineering (FSE)*, pages 416–419, 2011.

[7] G. Fraser and A. Arcuri. Sound empirical evidence in software testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 178–188, 2012.

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

[9] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, pages 80–89. IEEE Computer Society, 2011.

[10] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering*, 28(2):278–292, 2012.

[11] M. Harman, S. Mansouri, and Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys (CSUR)*, 45(1):11, 2012.

[12] M. Harman and P. McMinn. A theoretical and empirical study of search based testing: Local, global and hybrid search. *IEEE Transactions on Software Engineering*, 36(2):226–247, 2010.

[13] M. Islam and C. Csallner. Dsc+mock: A test case + mock class generator in support of coding against interfaces. In *International Workshop on Dynamic Analysis (WODA)*, pages 26–31, 2010.

[14] R. Just, F. Schweiggert, and G. Kapfhammer. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *International Conference on Automated Software Engineering (ASE)*, pages 612–615, 2011.

[15] B. Kitchenham, S. Pfleeger, L. Pickard, P. Jones, D. Hoaglin, K. El Emam, and J. Rosenberg. Preliminary guidelines for empirical research in software engineering. *IEEE Transactions on Software Engineering*, 28(8):721–734, 2002.

[16] J. Koza. Human-competitive results produced by genetic programming. *Genetic Programming and Evolvable Machines*, 11(3):251–284, 2010.

[17] K. Lakhotia, P. McMinn, and M. Harman. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *Journal of Systems and Software*, 83(12):2379–2391, 2010.

[18] P. McMinn. Search-based software test data generation: A survey. *Software Testing, Verification and Reliability*, 14(2):105–156, 2004.

[19] E. F. Miller, Jr. and R. A. Melton. Automated generation of testcase datasets. In *International Conference on Reliable Software*, pages 51–58. ACM, 1975.

[20] A. Namin and J.H.Andrews. The influence of size and coverage on test suite effectiveness. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2009.

[21] C. Pacheco and M. D. Ernst. Randoop: feedback-directed random testing for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 815–816. ACM, 2007.

[22] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 199–209, 2011.

[23] C. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, volume 10, pages 179–180, 2010.

[24] F. Pastore, L. Mariani, and G. Fraser. Crowdoracles: Can the crowd solve the oracle problem? In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2013. (To appear).

[25] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 286–293. IEEE, 2012.

[26] G. Sautter, K. Böhm, F. Padberg, and W. Tichy. Empirical evaluation of semi-automated XML annotation of text documents with the GoldenGATE editor. *Research and Advanced Techn. for Digital Libraries*, pages 357–367, 2007.

[27] C. Seaman. Qualitative methods in empirical studies of software engineering. *IEEE Transactions on Software Engineering*, 25(4):557–572, 1999.

[28] D. Sjoberg, J. Hannay, O. Hansen, V. By Kampenes, A. Karahasanovic, N. Liborg, and A. C Rekdal. A survey of controlled experiments in software engineering. *IEEE Transactions on Software Engineering*, 31(9):733–753, 2005.

[29] M. Staats, G. Gay, and M. Heimdahl. Automated oracle creation support, or: how I learned to stop worrying about fault propagation and love mutation testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*, pages 870–880, 2012.

[30] M. Staats, S. Hong, M. Kim, and G. Rothermel. Understanding user understanding: determining correctness of generated program invariants. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 188–198. ACM, 2012.

[31] N. Tillmann and N. J. de Halleux. Pex — white box test generation for .NET. In *International Conference on Tests And Proofs (TAP)*, pages 134–253, 2008.

[32] P. Tonella. Evolutionary testing of classes. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*, pages 119–128, 2004.

[33] J. Wegener, A. Baresel, and H. Sthamer. Evolutionary test environment for automatic structural testing. *Information and Software Technology*, 43(14):841–854, 2001.