eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical Study

Gordon Fraser, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
Gordon.Fraser@sheffield.ac.uk
Matt Staats, SnT Centre for Security, Reliability and Trust, University of Luxembourg,
4 rue Alphonse Weicker
L-2721 Luxembourg, Luxembourg,
matthew.staats@uni.lu
Phil McMinn, Department of Computer Science, University of Sheffield,
Regent Court, 211 Portobello
S1 4DP, Sheffield, UK
p.mcminn@sheffield.ac.uk
Andrea Arcuri, Certus Software V&V Center at Simula Research Laboratory,
P.O. Box 134, Lysaker, Norway
arcuri@simula.no
Frank Padberg, Karlsruhe Institute of Technology,
Karlsruhe, Germany
frank.padberg@kit.edu

Work on automated test generation has produced several tools capable of generating test data which achieves high structural coverage over a program. In the absence of a specification, developers are expected to manually construct or verify the test oracle for each test input. Nevertheless, it is assumed that these generated tests ease the task of testing for the developer, as testing is reduced to checking the results of tests. While this assumption has persisted for decades, there has been no conclusive evidence to date confirming it. However, the limited adoption in industry indicates this assumption may not be correct, and calls into question the practical value of test generation tools. To investigate this issue, we performed two controlled experiments comparing a total of 97 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

General Terms: Algorithms, Experimentation, Reliability, Theory

---

1. INTRODUCTION

Controlled empirical studies involving human subjects are not common in software engineering. A recent survey by Sjoberg et al. [Sjoberg et al. 2005] showed that out of 5,453 analyzed software engineering articles, only 1.9% included a controlled study with human subjects. Buse et al., in their survey of 3,110 software engineering articles [Buse et al. 2011], highlight that specifically papers categorized as related to testing and debugging only rarely have user evaluations. Indeed, 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., [Fraser and Arcuri 2013; Harman and McMinn. 2010; McMinn 2004; Tillmann and de Halleux 2008; Tonella 2004]). When using white-box test generation, a developer does not need to 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 test generation 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 previously performed a controlled experiment involving 49 human subjects and three classes [Fraser et al. 2013]. This work extends this study with another, considerably larger controlled experiment including 48 human subjects and four classes. In each study, subjects were given one or two Java classes containing seeded faults and were asked to construct a JUnit test suite for each class either manually, or with the assistance of the automated white-box test generation tool EVOSUITE [Fraser and Arcuri 2013]. EVOSUITE automatically produces JUnit test suites that target branch coverage, and these unit tests contain assertions that reflect the current behaviour of the class [Fraser and Zeller 2012]. 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. In total across both studies, 145 combinations of subjects and Java classes were used.

Our studies yield 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 practically useful for testing.

The rest of this paper is organized as follows. Section 2 introduces the common setup shared by the two empirical studies. In Sections 3 and 4, we present the results of each study. The initial study is presented first in Section 3, and the results of the larger study second in Section 4. In Section 5, we discuss the "why" of these results—particularly those from the second study, as it presents a larger pool of data—presenting selected additional analysis as needed. Section 6 goes into the details of the background and exit surveys completed by the subjects . Section 7 discusses the implications of our results on future work. Our study is then put into context with related work in Section 8. Finally, Section 9 concludes the paper.

2. STUDY DESIGN

The purpose of these studies 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 studies were 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 studies around the following research questions (RQs):

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

$RQ1$ The structural code coverage achieved during testing?
$RQ2$ The ability of testers to detect faults in the class under test?
$RQ3$ The number of tests mismatching the intended behaviour of the class?
$RQ4$ The ability of produced test suites to detect regression faults?

The goal of the second study was to replicate the results found in the initial study, in order to improve our confidence in the results. Accordingly, the design and implementation of both studies are largely the same. Nevertheless, based on the results of the first study, we introduced small changes to improve the quality of the second study. Furthermore, to increase the number of data points available for analysis in the second study, participants were asked to perform both manual testing and tool-supported testing, and additional objects were used.

Below, we outline the study designs for both studies. In cases where the experimental design differs, we explicitly highlight such differences.

2.1. The Automated Testing Tool: EvoSuite

The automated testing tool used in our studies is EVOSUITE [Fraser and Arcuri 2013], 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 [Fraser and Arcuri 2013]. 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.
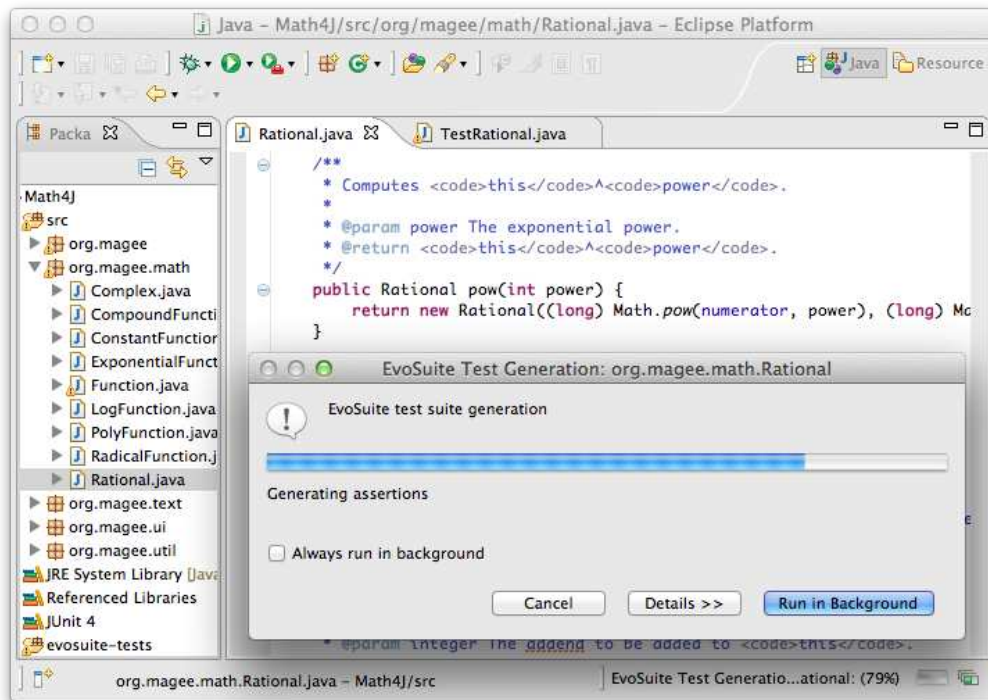
Fig. 1: The EVOSUITE Eclipse plugin, generating test cases for a class—as used by subjects in the study.

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 [Fraser and Zeller 2012]. The purpose of the assertions is to indicate to the testers what aspects of the program state are, for each test input, capable of detecting faults. They therefore act as guidelines for testers, from which correct assertions can (hopefully) be derived.

There exist several active paradigms for automatic test case generation. These approaches typically differ in how the test oracle is constructed, and include: automatically inferring program invariants [Staats et al. 2012b; Wei et al. 2011]; automatically inferring parametrized test input assertions [Tillmann and Halleux 2008]; automatically producing concrete test input assertions (as is done by EVOSUITE) [Fraser and Arcuri 2011; Staats et al. 2012a]. Currently, there is no scientific consensus which approach is preferable, and in practice all of these approaches appear to be used infrequently by industry.

From these approaches, we have selected EVOSUITE as a representative example of a modern approach for test case generation based on generating concrete test input assertions. Our results may not generalize to other paradigms of automated test generation (e.g., those based on parametrized test inputs or program invariants). Nevertheless, we note that all of these approaches require the user to correct generated test oracles, and we therefore believe that our results will generalize at least somewhat to other approaches.

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.

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 [Kitchenham et al. 2002; Seaman 1999]). 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. [Sjoberg et al. 2005] 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: Initial Study . We restricted our experiment to three Java classes to increase the likelihood of observing statistically significant effects in our initial study. 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 [Fraser and Arcuri 2012b] 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 be non-trivial, yet feasible to reasonably test within an hour. The classes should not require the subjects to learn and understand complicated algorithms. In particular, we considered classes with fewer than 50 lines of code or classes without conditional expressions as too easy. Furthermore, each class file should only involve one class definition (i.e., the class should not involve member classes).
(4) 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.
(5) The classes should represent 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 [Fraser and Arcuri 2013; Lakhotia et al. 2010], 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.

Table I: Study objects for first empirical study

"NCSS" refers to the number of non-commenting source statements 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 [Just et al. 2011].

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

Details of the classes used in the experiment can be found in Table I. 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 a parameter is mandatory, and a short accompanying descriptor. Finally, *Rational*, from the Math4J project (*http://math4j.sourceforge.net*), represents a rational number.

Andrews et al. [Andrews et al. 2005] argue that mutation faults can be representative of real faults, and that hand-seeded faults present validity issues for studies that investigate fault detection. For this reason, we applied mutation analysis to obtain a series of faults to use throughout our experiments; as have other authors, for example Do and Rothermel [Do and Rothermel 2006] in their study of test case prioritization for regression faults. Each Java class was injected with five faults prior to the experiment using the MAJOR [Just et al. 2011] mutation analysis tool. In order to ensure a range of different types of mutants, 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. See Appendix A for details of the actual faults.

2.2.2. Object Selection: Replication Study . As noted previously, for the second study participants were asked to conduct two study sessions, one with manual testing and one with tool-assisted testing. (This is described further in Section 2.2.4.) This increase in the number of study sessions allowed us to add another object, for a total of four. However, as the second study was designed primarily as a replication study, we elected to maintain largely the same set of objects.

We used four classes in the replicated experiment. We reused the Commons CLI *Option* and Commons Math *Fraction* classes, as is, from the first experiment. We also reused the XOM *DocType* class, with changes made to correct a classpath issue found in the initial study. (This issue is outlined later in Section 3.) Finally, we added a new class, which was selected using the same selection procedure as for the original experiment.

We selected *ArrayIntList* from the Commons Primitives open source library, as container classes are very commonly found in experimentation in the software engineering literature. The class represents a list implemented with an array, which unlike the

Table II: Study objects for replicated empirical study

"NCSS" refers to the number of non-commenting source statements 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 [Just et al. 2011].

Project	Class	NCSS	Methods	Branches	Mutants
Commons Primitives	*ArrayIntList*	65	12	28	112
XOM	*DocType*	296	26	242	186
Commons CLI	*Option*	155	42	96	140
Math4J	*Rational*	61	19	36	112

array list implementation in the Java standard library uses primitive `int` values rather than objects. We made one modification to this class: in Java, serialization behavior of classes can be modified by providing methods `readObject` and `writeObject`. However, these are private methods, and cannot be called directly; instead, one needs to pass the objects to an object stream where native code calls these methods. As this may not be standard knowledge and the methods would be trivially covered, we removed these two methods from the class. Mutants of this class were created using the same procedure as for the original set of classes (see Appendix A for details of the chosen mutants).

2.2.3. Subject Selection and Assignment: Initial Study. In our initial study, 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, so 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. Each subject was paid 15 GBP for their time and involvement.

2.2.4. Subject Selection and Assignment: Replication Study. To ensure that a sufficient number of data points were produced we arranged three sessions of the study, each with 16 different participants for a total of 48 participants. The participants were selected from the pool of students in the Department of Computer Science at the University of Sheffield, excluding all students that already participated in the first study. Students were invited by email, and invitations were given on a first-come-first-serve basis.

Selection and assignment of subjects was performed similarly in the second study, with one key difference. To further increase the number of data points collected in our second study, each participant performed two experiments—one with EVOSUITE and one without, on different classes—rather than just one, to produce 96 data points in this study (for a total of 145 data points across both studies). To account for the increased effort required from each participant, we doubled the remuneration per participant

to 30 GBP. This also had the benefit that the students' willingness to participate was increased from the first study.

As with the initial study, the classes and treatments were assigned to work places before the experiment, and participants were free to choose their computer. Experiments were started by the participants by clicking on desktop icons labeled "Tutorial", "Experiment 1", and "Experiment 2", and all printed experimental material was contained in unlabeled folders on these workplaces to prevent participants from choosing their laptop based on the treatment. We assigned treatments such that no two neighboring participants would be using the same treatment, i.e., every participant using EVOSUITE would have two neighbors doing manual testing. We also made sure that no two neighbors would be testing the same class at the same time. Furthermore, the assignment made sure that we would get the same number of data points for all classes/treatments.

To avoid learning effects we also carefully selected all variations of the order in which treatments are applied. Specifically, for each pair of classes X and Y, we aimed to have the same number of participants first testing class X with EVOSUITE followed by class Y with manual testing; class X with manual testing followed by class Y with EVOSUITE; class Y with EVOSUITE followed by class X with manual testing; and class Y with manual testing followed by class X with EVOSUITE. This assignment was done manually.

2.3. Experiment Process

In both studies, 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. In the initial study, the pack 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. In the second study, small changes reflecting differences in the study designs were made. For example, sheets describing both Java classes were used, the exit survey was updated to reflect that both treatments were done by each participant, etc.

Before commencing the experiment, each subject was required to fill in the questionnaire based on their background and programming experience, such that we would receive responses about the background knowledge that are not influenced by the following tutorial session.

In both studies, subjects were 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.

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). In the initial study, 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 replication of the experiment was performed in the "iLab" of the Information School at the University of Sheffield, a laboratory dedicated to usability testing and experiments. The lab offers 16 identical modern laptop computers running Windows 7. The setup of these machines was similar to the setup in the original experiment, although we used updated versions of software:

— Oracle Java SE Development Kit 7u45
— Eclipse Kepler (4.3.1)
— An updated version of the EVOSUITE Eclipse plugin

The version of EVOSUITE in the second study differed only in terms of bugfixes; no new features compared to the version of the initial study were activated.

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. Subjects were not given a code coverage tool to measure the coverage achieved, as we desired a natural code testing process. We felt the addition of code coverage tooling might encourage testers to achieve higher code coverage than they typically would during testing. Furthermore, not all subjects might be familiar with the use of code coverage tools, which would potentially bias the results further.

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.

When using EVOSUITE, subjects were asked to start by producing a test suite using the plugin, and to edit the test cases so 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 for later analysis (presented in the following sections).

In each study session, 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. In the second study, a short break was given between study sessions.

Including tutorial and break, the duration of the experiment was two hours for the initial study, and three hours for the second study. 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. Following the study, subjects were required to fill in an exit survey.

2.4. Analysis of Results

Each study session resulted in 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 [Just et al. 2011]. 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 must often 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. In the initial study, 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. Nevertheless, we acknowledge that our results may not perfectly generalize to scenarios where developers are also testers, and we hope to explore the impact of code ownership on testing in the future.

---

[1]Note that Cobertura only counts conditional statements for branch coverage, whereas the data given in Table I lists branches in the traditional sense, i.e., edges of the control flow graph. See [Li et al. 2013] for a discussion of the perks of bytecode-based coverage measurement.

Our study uses EVOSUITE for automatic test generation. It is possible that using different automatic test generation tools may yield different results. Nevertheless, EVO-SUITE 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* [Pacheco and Ernst 2007], *eToc* [Tonella 2004], *TestFul* [Baresi et al. 2010], *Java PathFinder* [Pasareanu and Rungta 2010], *Dsc* [Islam and Csallner 2010], *Pex* [Tillmann and de Halleux 2008], *JCrasher* [Csallner and Smaragdakis 2004], and others.

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 that 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. As participants had to fill in the background survey before the experiment, there is the threat that identifying demographic information prior to a task and conducting that task in the presence of other people can impact performance [Inzlicht and Ben-Zeev 2000]. However, as the tutorial we gave to participants would potentially influence the answers given we wanted to have unbiased responses to the questionnaire. 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 in the first experiment; in the replication each subject tested two different classes with different treatments to avoid or minimize possible learning effects.

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 [Andrews et al. 2005], it is possible the use of faults created by developers may yield different results.

Conclusion: We conducted our initial 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. The second study consisted of 48 subjects and four Java classes. 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: INITIAL STUDY

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 5.1.

Table III: Results of the first study

For each property, we report several statistics on the obtained results, like minimum values, median, average, standard deviation, maximum and kurtosis. For some properties (i.e., median and average), we calculated confidence intervals (CI) using bootstrapping at 95% significance level. For each property, we calculated the $\hat{A}_{12}$ effect size of EvoSuite (EvoS.) compared to Manual (Man.). We also report the p-values of a Wilcoxon-Mann-Whitney U-tests. Statistically significant effect sizes are shown in bold.

(a) *Option*

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	3	7.50	[3.00, 12.00]	8.50	[4.75, 11.75]	5.48	20	3.53	**0.97**	0.001
	Man.	0	1	[0.00, 2.00]	1.22	[0.33, 2.00]	1.30	4	3.28		
# of found bugs	EvoS.	0	0.00	[-1.00, 0.00]	0.38	[-0.12, 0.75]	0.74	2	3.86	0.31	0.165
	Man.	0	1	[1.00, 2.00]	0.89	[0.22, 1.44]	0.93	3	4.12		
Mutation score	EvoS.	44.44	51.69	[48.31, 54.37]	51.46	[49.13, 54.01]	3.78	55.66	2.45	0.65	0.321
	Man.	0.00	37.50	[12.93, 56.82]	37.82	[24.97, 51.23]	21.75	64.10	2.00		
% Statement coverage	EvoS.	86.36	91.82	[91.82, 95.45]	90.68	[89.43, 92.27]	2.21	92.73	2.79	**1.00**	0.001
	Man.	15.45	41.82	[31.82, 58.18]	37.98	[29.70, 46.36]	13.80	60.00	2.18		
% Branch coverage	EvoS.	80.00	85.71	[82.86, 88.57]	85.36	[83.57, 87.32]	2.93	88.57	2.40	**1.00**	0.001
	Man.	4.29	21.43	[12.86, 32.86]	20.95	[15.08, 27.14]	9.95	35.71	2.13		
% Method coverage	EvoS.	90.48	94.05	[92.86, 95.24]	93.75	[92.56, 94.94]	1.77	95.24	2.26	**1.00**	0.001
	Man.	9.52	40.48	[16.67, 57.14]	42.06	[29.37, 54.76]	20.62	73.81	2.03		
NCSS	EvoS.	369	418.50	[401.00, 457.50]	409.25	[390.38, 428.38]	29.08	444	1.46	**1.00**	< 0.001
	Man.	17	72	[30.00, 110.00]	67.56	[43.22, 90.11]	38.58	134	2.09		
# of tests	EvoS.	45	47.50	[47.00, 49.00]	47.12	[46.25, 48.00]	1.36	49	1.82	**1.00**	0.001
	Man.	4	8	[-13.00, 11.00]	14.56	[7.33, 20.89]	11.13	35	2.26		

(b) *Rational*

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	1	5	[2.00, 9.00]	5.14	[2.29, 7.71]	3.98	12	2.25	0.70	0.289
	Man.	0	2	[-4.00, 4.00]	2.80	[0.00, 5.20]	3.35	8	2.13		
# of found bugs	EvoS.	0	2	[0.00, 3.00]	2.29	[1.29, 3.29]	1.50	4	1.85	0.43	0.738
	Man.	0	3	[2.00, 6.00]	2.60	[1.40, 4.00]	1.67	4	2.13		
Mutation score	EvoS.	5.56	66.35	[53.06, 77.14]	60.62	[45.94, 80.72]	26.19	85.19	4.01	0.37	0.530
	Man.	64.29	75.49	[73.10, 86.69]	73.71	[70.40, 78.65]	5.47	77.88	2.89		
% Statement coverage	EvoS.	94.59	100.00	[100.00, 105.41]	98.07	[96.53, 100.00]	2.57	100.00	1.55	0.79	0.109
	Man.	72.97	97.30	[97.30, 121.62]	91.35	[85.41, 101.08]	10.54	97.30	2.99		
% Branch coverage	EvoS.	80.00	90.00	[90.00, 95.00]	87.14	[85.00, 90.00]	3.93	90.00	2.36	0.64	0.431
	Man.	70.00	85.00	[80.00, 100.00]	83.00	[77.00, 90.00]	8.37	90.00	2.13		
% Method coverage	EvoS.	94.74	100.00	[100.00, 105.26]	98.50	[96.99, 100.75]	2.57	100.00	1.90	**0.91**	0.014
	Man.	57.89	94.74	[94.74, 131.58]	85.26	[75.79, 100.00]	15.96	94.74	2.82		
NCSS	EvoS.	68	105.00	[96.00, 117.00]	109.75	[84.50, 128.25]	34.33	187	4.55	0.52	0.948
	Man.	63	103.50	[72.50, 137.50]	102.50	[79.50, 124.17]	31.13	147	1.79		
# of tests	EvoS.	11	16.50	[12.00, 19.00]	18.25	[13.25, 22.00]	6.96	34	4.49	0.39	0.517
	Man.	14	18.50	[13.00, 22.50]	19.00	[15.83, 22.17]	4.34	25	1.65		

(c) *DocType*

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	0	2.00	[0.00, 2.00]	2.50	[1.50, 3.50]	1.51	5	2.61	**0.83**	0.026
	Man.	0	0.00	[-1.00, 0.00]	0.75	[-0.25, 1.38]	1.39	4	4.97		
# of found bugs	EvoS.	1	1.00	[1.00, 1.00]	1.00	[1.00, 1.00]	0.00	1	NaN	0.50	< 0.001
	Man.	0	1.00	[0.00, 2.00]	1.00	[0.50, 1.50]	0.76	2	2.00		
Mutation score	EvoS.	30.00	45.31	[36.66, 52.16]	46.65	[38.46, 53.81]	11.96	70.13	3.11	0.22	0.065
	Man.	41.67	55.95	[44.68, 63.66]	56.50	[50.86, 62.37]	8.87	67.69	2.17		
% Statement coverage	EvoS.	21.93	32.09	[25.13, 36.90]	32.75	[27.94, 37.30]	7.15	44.39	2.18	0.37	0.399
	Man.	25.13	37.97	[35.83, 48.66]	36.16	[31.68, 40.91]	7.07	46.52	2.17		
% Branch coverage	EvoS.	6.92	12.69	[3.85, 18.46]	14.90	[9.90, 19.52]	7.47	27.69	1.99	**0.19**	0.040
	Man.	16.15	21.92	[15.38, 26.15]	23.94	[18.37, 28.27]	7.82	40.77	3.76		
% Method coverage	EvoS.	57.69	76.92	[73.08, 92.21]	75.00	[68.27, 82.21]	11.07	92.31	2.33	0.70	0.178
	Man.	57.69	73.08	[69.23, 88.27]	70.19	[65.87, 75.96]	7.90	76.92	2.24		
NCSS	EvoS.	73	101	[82.00, 114.00]	103.71	[88.71, 117.86]	21.40	136	1.98	0.60	0.562
	Man.	40	81.50	[25.00, 117.50]	88.00	[59.38, 116.25]	43.63	147	1.45		
# of tests	EvoS.	12	17	[13.00, 20.00]	17.57	[14.71, 20.43]	4.16	23	1.51	0.62	0.449
	Man.	9	13.50	[2.00, 16.00]	16.12	[11.38, 20.25]	6.90	28	2.16		

(a) Statement Coverage

(b) Branch Coverage

(c) Method Coverage

(d) # of Tests Failing on Correct System

(e) % Generated Mutants Killed

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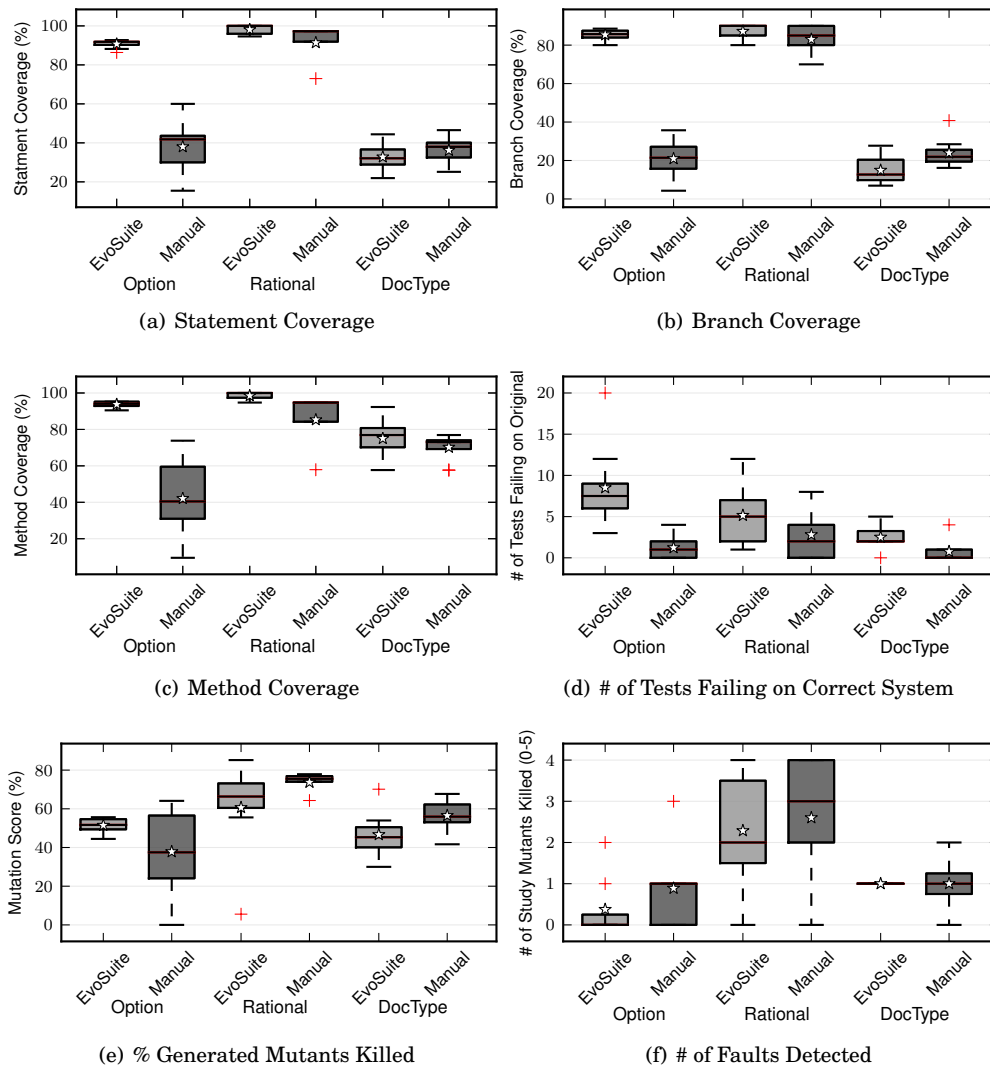
Fig. 2: Test suite properties, comparing EVOSUITE against manual testing (boxes spans from 1st to 3rd quartile, middle lines mark the median, whiskers extend up to $1.5\times$ the inter-quartile range, while plus symbols represent outliers and stars signify the mean).

The results for our initial study are summarized in the form of boxplots in Figure 2, and detailed statistical analysis is presented in Table III(a) for *Option*, Table III(b) for *Rational*, and finally Table III(c) for *DocType*. The Mann-Whitney U-test was used to check for statistical difference 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. 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 [Arcuri and Briand 2014].

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 automatically.

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, EVO-SUITE 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:

```
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.

3.1. 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 criterion 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 for coverage in Table III(a), 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).

For *Option*, where the difference in coverage is largest, this increase comes at the price of an increased number of tests: EVOSUITE produces 47.12 tests on average,

whereas manual testing results in only 14.56 tests on average. For *Rational* and *DocType*, however, the numbers of tests generate manually and automatically are quite comparable: For *Rational* manual testing leads to slightly more tests (18.25 generated by EVOSUITE vs. 19.00 written manually), even though EVOSUITE's tests have higher coverage on average. For *DocType*EVOSUITE produces slightly more tests with slightly lower coverage. Thus, although more tests generally means higher coverage, this is not the only deciding factor.

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.*

3.2. 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
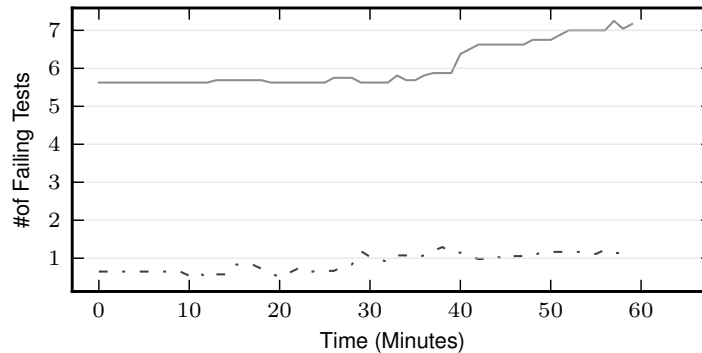
Fig. 3: Average number of test cases failing on the original, correct version of the class. EVOSUITE users are shown with dark gray solid lines, manual testers with light gray dashed lines.

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 5.2.

3.3. RQ3: Tests Mismatching the Intended Program Behavior

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.

Naturally, we expect several failing tests in the initial test suite when using EVO-SUITE: 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.*

3.4. 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.

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 [Namin and J.H.Andrews 2009].

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 [Namin and J.H.Andrews 2009]. 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. RESULTS: REPLICATION STUDY

Recall from Section 3 that our second study differs only in two main respects: the systems used in the study and the subjects. Therefore, we wish to not only answer our research questions, but also highlight differences in the results with our previous study's results. Such differences—or lack of differences—indicate how well our initial results generalize to both other systems, i.e., those not used in the initial work, and to the larger body of testers.

As before, we examine our results in the context of each research question, and apply the same analyses to answer each question. Our analyses are done using the final test suites produced by the testing processes, with analyses examining intermediate test suites shown and discussed in Section 5.1 (jointly with the analyses for the initial study). In Figure 4, we summarize the results as box plots, and in Tables IV(a)-IV(a) we list the outcome of statistical tests. Two statistical tests were used: the Mann-Whitney U-test was used to check for statistical differences between manually and tool-assisted generated test suites, and the Vargha-Delaney $\hat{A}_{12}$ statistic was used to compute the standardized effect sizes.

In contrast to our previous study, no data was discarded; all subjects wrote test suites and no modifications to the classes' interfaces were performed. Also recall that *DocType*'s EVOSUITE configuration in this study was corrected, and does not contain the classpath issue found in the initial study. Thus the results when using EVOSUITE for *DocType* represent a testing scenario not explored in the previous study.

4.1. RQ1: Structural Code Coverage Achieved

As seen in Figure 4, without fail the use of EVOSUITE results in equal or better structural coverage for all combinations of classes and structural coverage criteria. This is perhaps most pronounced for method coverage, where we can see relative improvements in average coverage of 17.7% to 117.2% when using EVOSUITE generated test suites over manually created test suites. Indeed, for method coverage, the lowest average coverage for EVOSUITE generated test suites is 95% (for *Option*). Similar results are found for statement and branch coverage, with relative improvements when using EVOSUITE generated test suites of 6.6%-156.3% and 1.5%-329.2% for statement and branch coverage, respectively.

Examining Tables IV(a)-IV(a) (rows for percentage "%" of statement, branch and method coverage), we see these results reflected in the statistical analysis: standardized effect sizes are consistently high, typically higher than 0.8, with *p*-values typically below the traditional statistical significance mark of 0.05. The effect is strongest for method coverage, with statistically significant results for every class; however, only in one case — branch coverage over *Rational*, with a *p*-value of 0.84 — there is no strong effect on coverage found with respect to generation strategy.

In all cases except *Rational*, the number of tests generated automatically is again higher than that written manually. For *Rational*, the number of automatically generated tests is again lower (18.58 on average) than the number of manually written tests (23.42 on average). Thus, although within a short time an automated tool generates more tests

Table IV: Results of replicated empirical study.

(Refer to Table III for descriptions of column headings)

(a) *Option*.

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	2	8.50	[4.50, 9.00]	10.50	[7.17, 13.42]	5.89	25	4.25	**0.97**	< 0.001
	Man.	0	1.50	[0.50, 2.50]	1.58	[0.83, 2.25]	1.31	4	2.06		
# of found bugs	EvoS.	0	0.00	[-1.00, 0.00]	0.33	[0.083, 0.58]	0.49	1	1.50	0.40	0.368
	Man.	0	0.50	[0.00, 1.00]	0.58	[0.25, 0.92]	0.67	2	2.42		
Mutation score	EvoS.	14.68	42.35	[36.40, 48.89]	41.27	[35.20, 47.98]	11.73	61.67	3.69	0.56	0.644
	Man.	8.57	35.47	[14.09, 47.74]	37.85	[28.30, 48.10]	18.17	59.62	1.63		
% Statement coverage	EvoS.	83.64	92.73	[90.45, 95.00]	92.05	[90.23, 94.09]	3.60	95.45	3.40	**1.00**	< 0.001
	Man.	18.18	38.18	[35.00, 45.00]	35.91	[31.89, 40.53]	8.19	45.45	2.87		
% Branch coverage	EvoS.	61.43	83.57	[80.71, 87.14]	82.26	[78.69, 86.90]	7.67	91.43	5.74	**1.00**	< 0.001
	Man.	7.14	20.71	[16.43, 29.29]	19.17	[14.64, 23.57]	8.25	32.86	1.98		
% Method coverage	EvoS.	83.33	97.62	[97.62, 101.19]	95.24	[93.06, 98.02]	4.54	100.00	4.98	**1.00**	< 0.001
	Man.	16.67	46.43	[39.29, 57.14]	43.85	[37.10, 51.19]	12.82	59.52	2.59		
NCSS	EvoS.	171	244.00	[224.50, 252.00]	244.25	[229.83, 261.00]	29.12	281	4.42	**1.00**	< 0.001
	Man.	15	76.50	[48.50, 104.00]	75.75	[56.67, 95.42]	35.67	126	1.85		
# of tests	EvoS.	28	36.50	[33.50, 38.00]	36.33	[33.92, 39.00]	4.68	44	2.81	**0.97**	< 0.001
	Man.	2	19.00	[12.00, 26.50]	18.50	[13.67, 23.67]	9.33	31	1.80		

(b) *Rational*.

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	1	4.50	[2.00, 6.00]	5.08	[3.17, 6.83]	3.37	13	3.60	0.35	0.234
	Man.	0	9.50	[2.50, 17.00]	9.50	[5.42, 13.58]	7.56	19	1.39		
# of found bugs	EvoS.	0	2.00	[1.00, 3.50]	1.92	[1.08, 2.75]	1.51	4	1.62	0.65	0.223
	Man.	0	1.00	[0.00, 2.00]	1.17	[0.42, 1.83]	1.34	4	2.67		
Mutation score	EvoS.	15.74	52.32	[31.96, 63.44]	53.70	[42.66, 65.60]	21.30	82.41	2.11	0.61	0.371
	Man.	0.00	49.95	[30.87, 87.84]	41.98	[24.63, 59.37]	31.91	81.48	1.38		
% Statement coverage	EvoS.	91.89	97.30	[94.59, 97.30]	97.75	[96.62, 99.10]	2.26	100.00	4.89	0.73	0.050
	Man.	75.68	95.95	[94.59, 106.76]	91.67	[87.39, 96.62]	8.66	100.00	2.17		
% Branch coverage	EvoS.	75.00	85.00	[85.00, 90.00]	83.33	[80.83, 85.83]	4.92	90.00	2.34	0.53	0.835
	Man.	70.00	82.50	[75.00, 87.50]	82.08	[78.33, 86.25]	7.53	90.00	1.89		
% Method coverage	EvoS.	89.47	100.00	[100.00, 100.00]	99.12	[98.25, 100.88]	3.04	100.00	10.09	**0.84**	0.002
	Man.	52.63	92.11	[86.84, 113.16]	84.21	[75.44, 94.30]	17.24	100.00	2.13		
NCSS	EvoS.	64	90.50	[73.50, 100.00]	94.08	[82.67, 105.00]	20.72	138	2.77	0.44	0.665
	Man.	63	94.00	[75.50, 105.00]	100.67	[85.42, 114.25]	27.04	166	3.93		
# of tests	EvoS.	11	19.00	[16.50, 22.50]	18.58	[16.17, 21.17]	4.64	26	2.18	0.27	0.064
	Man.	17	22.00	[17.00, 25.00]	23.42	[20.42, 26.25]	5.42	35	2.61		

(c) *DocType*.

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	1	3.00	[2.00, 4.00]	3.17	[2.33, 3.92]	1.47	6	2.28	0.69	0.113
	Man.	0	2.00	[1.00, 3.00]	2.08	[1.17, 2.92]	1.56	5	2.15		
# of found bugs	EvoS.	0	1.00	[1.00, 1.50]	0.83	[0.50, 1.17]	0.58	2	2.95	0.47	0.823
	Man.	0	1.00	[0.50, 2.00]	1.00	[0.42, 1.50]	0.95	3	2.64		
Mutation score	EvoS.	0.00	65.50	[60.84, 72.67]	60.62	[51.95, 73.18]	20.49	79.67	7.72	0.70	0.101
	Man.	39.06	54.87	[45.10, 60.10]	56.51	[49.91, 62.56]	11.67	78.50	2.33		
% Statement coverage	EvoS.	71.66	89.04	[87.17, 95.99]	86.36	[82.40, 90.95]	7.84	95.19	2.37	**1.00**	< 0.001
	Man.	28.88	38.24	[35.03, 41.98]	39.35	[34.09, 43.45]	8.73	63.10	5.76		
% Branch coverage	EvoS.	59.23	84.62	[81.54, 92.31]	81.41	[76.35, 87.50]	10.38	92.31	2.93	**1.00**	< 0.001
	Man.	18.46	23.46	[20.00, 25.77]	26.54	[19.68, 31.09]	10.87	58.46	7.48		
% Method coverage	EvoS.	84.62	100.00	[100.00, 101.92]	98.08	[96.15, 100.96]	4.49	100.00	8.04	**0.99**	< 0.001
	Man.	61.54	80.77	[80.77, 88.46]	78.85	[74.36, 83.33]	8.44	92.31	3.10		
NCSS	EvoS.	132	172.50	[161.50, 188.00]	170.33	[159.67, 181.67]	20.29	201	2.34	**0.99**	< 0.001
	Man.	50	83.50	[73.50, 103.00]	82.67	[68.50, 95.42]	24.59	139	3.52		
# of tests	EvoS.	29	35.50	[31.50, 38.00]	36.33	[33.67, 39.00]	4.96	45	2.22	**0.99**	< 0.001
	Man.	9	18.00	[13.50, 20.50]	19.00	[15.33, 22.33]	6.44	32	2.78		

Table IV: Results of replicated empirical study (continued).

(Refer to Table III for descriptions of column headings)

(a) *ArrayIntList*.

Variable	Method	Min	Median	CI	Avg.	CI	SD	Max	Kurt.	$\hat{A}_{12}$	p-value
# failing tests on original	EvoS.	1	2.00	[0.50, 3.00]	2.50	[1.58, 3.25]	1.57	6	2.98	0.64	0.237
	Man.	0	1.50	[0.50, 2.50]	1.75	[0.83, 2.58]	1.60	5	2.54		
# of found bugs	EvoS.	0	0.00	[-1.50, 0.00]	0.75	[0.083, 1.33]	1.22	3	2.56	0.64	0.122
	Man.	0	0.00	[0.00, 0.00]	0.083	[-0.083, 0.17]	0.29	1	10.09		
Mutation score	EvoS.	10.47	30.20	[24.94, 43.55]	26.56	[21.33, 32.22]	10.02	37.21	1.40	0.40	0.436
	Man.	5.56	33.06	[25.45, 44.57]	31.10	[24.12, 38.73]	13.46	51.40	2.34		
% Statement coverage	EvoS.	71.43	76.79	[65.18, 78.57]	80.21	[76.04, 84.08]	7.59	92.86	1.74	0.73	0.056
	Man.	17.86	71.43	[60.71, 95.54]	63.99	[52.23, 76.93]	22.98	94.64	2.35		
% Branch coverage	EvoS.	66.67	77.78	[72.22, 80.56]	78.70	[75.00, 82.41]	6.63	88.89	2.41	**0.85**	0.003
	Man.	5.56	50.00	[33.33, 66.67]	50.46	[37.50, 63.43]	23.62	88.89	2.49		
% Method coverage	EvoS.	91.67	100.00	[100.00, 108.33]	96.53	[94.44, 98.61]	4.29	100.00	1.11	**0.91**	0.001
	Man.	25.00	83.33	[79.17, 91.67]	77.78	[69.44, 88.89]	18.91	100.00	6.27		
NCSS	EvoS.	75	87.50	[62.00, 91.50]	97.08	[86.83, 106.83]	18.51	133	2.06	**0.77**	0.026
	Man.	21	57.00	[22.50, 88.00]	60.42	[40.59, 79.50]	36.17	121	1.71		
# of tests	EvoS.	14	16.00	[13.50, 17.00]	16.50	[15.42, 17.58]	2.02	20	1.91	0.71	0.087
	Man.	1	9.50	[1.00, 14.00]	11.83	[7.00, 16.25]	8.55	28	2.15		

than a human tester, it seems that, if given sufficient time, manual testing will converge at a number of tests no less than automated test generation. In fact, considering that EVOSUITE not only optimises for code coverage, but as a secondary criterion also optimises the test suite size, it is likely that manual testing will in practice result in more tests for the same level of coverage.

These results match our expectations — depending on the structural coverage criteria considered, achieving high coverage is, at worst, made no more difficult by the introduction of tools for automatic test generation. Indeed, in every case except branch coverage over the *Rational* class, the use of EVOSUITE results in improvements in coverage, up to 329.2%, with high statistical confidence. We therefore conclude that, as in our previous study, generated test suites do achieve higher structural coverage than those constructed manually.

> **RQ1:** *The replication confirms that automatically generated test suites achieve higher coverage than manually created test suites.*

4.2. RQ2: Faults Detected

Of the four classes used in our study, test suites generated using EVOSUITE outperform those manually constructed for only two classes, *Rational* and *ArrayIntList*, by 64.3% and 800% on average, respectively (Figure 4(f)). In both cases this represents an improvement of roughly one fault detected more on average. Test suites constructed for *Option* and *DocType* are slightly more effective when manually constructed, by 75% and 20%, respectively. As shown in Tables IV(a)-IV(a) however, the differences in fault detection are not strong enough within any class to be considered statistically significant. Even for *ArrayIntList*, the class for which the difference in faults detected is most visible, the standardized effect size is only 0.64 and the *p*-value is 0.122, short of the traditional $\alpha = 0.05$ mark.

These results are comparable to those in Section 3.2 with regards to the effectiveness of EVOSUITE derived test suites. As before, in most cases both sets of test suites are typically of roughly equal fault detection power. Indeed, even for *DocType*, which was

(a) Statement Coverage

(b) Branch Coverage

(c) Method Coverage

(d) # of Tests Failing on Correct System

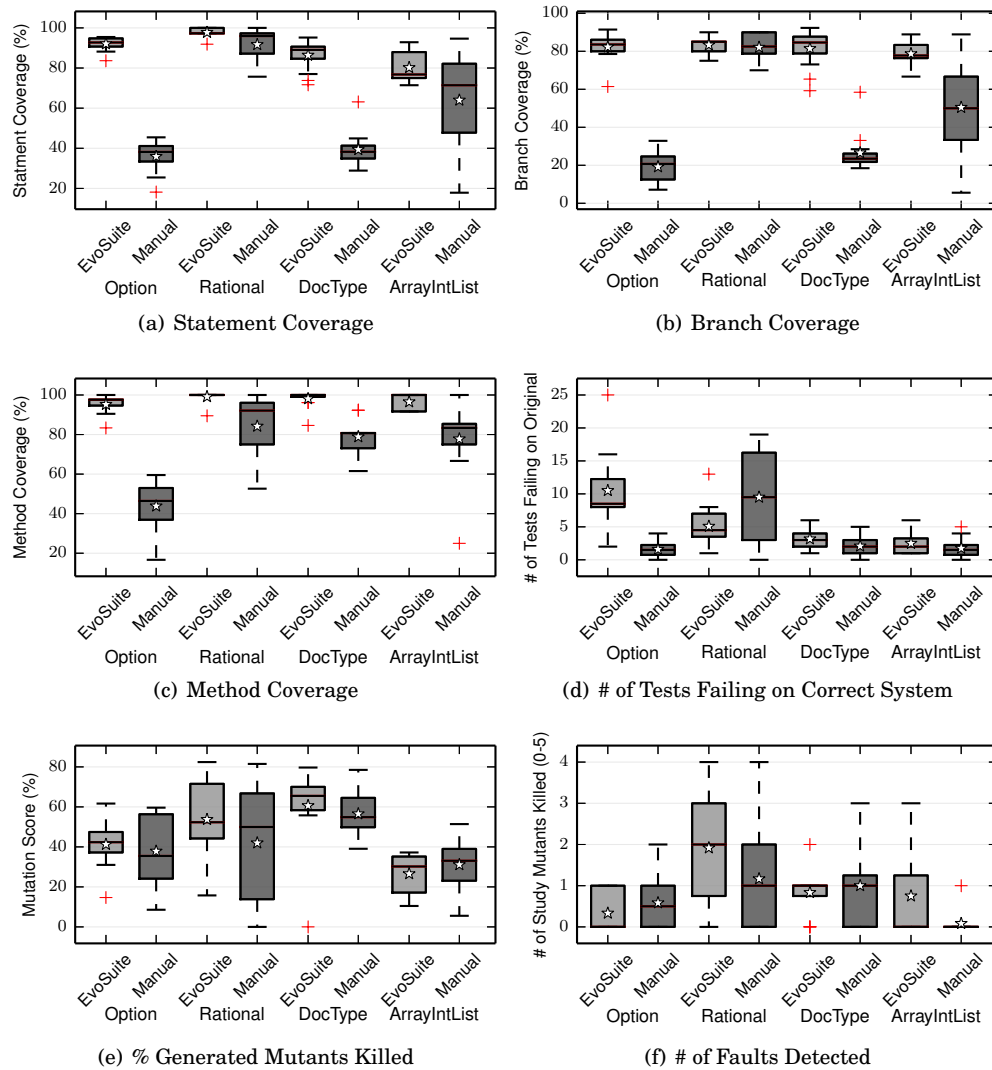(e) % Generated Mutants Killed

(f) # of Faults Detected

Fig. 4: 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).

misconfigured in the previous study, both sets of subjects again detected on average roughly one fault. (Though new to this study, some variation in fault detection was observed for those subjects using EVOSUITE.) On the whole, automatic test generation failed to improve the ability of subjects to detect more faults in the given classes.

Despite the lack of a statistically significant effect, the results indicate that EVOSUITE is most useful when applied to *ArrayIntList*, for reasons that were not immediately obvious to us. Examining the version of the class of *ArrayIntList* used in our study, we found that the five faults added to the class resulted in a clearly incorrect class. For
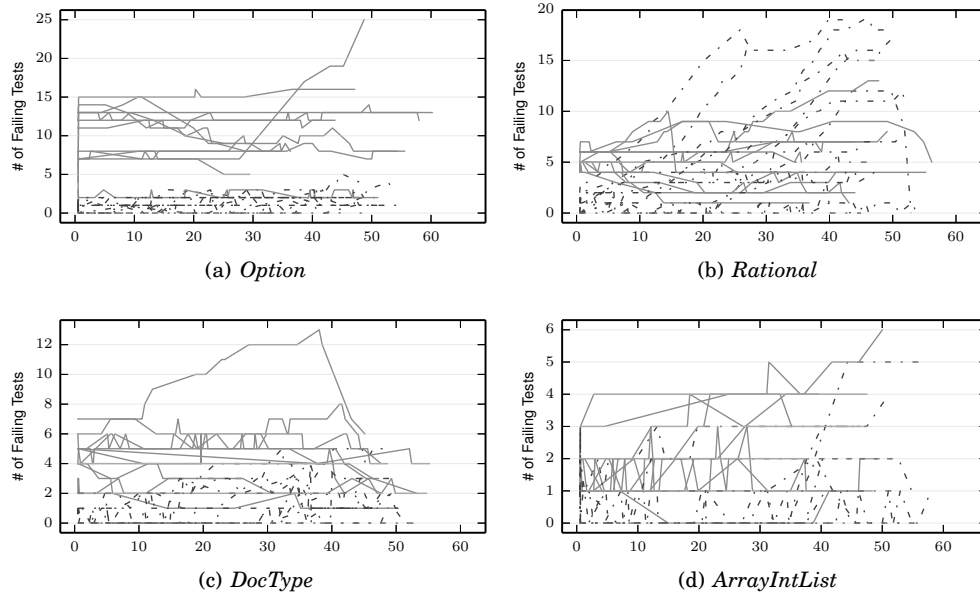
Fig. 5: Average number of test cases failing on the original, correct version of the class. EVOSUITE users are shown with dark gray solid lines, manual testers with light gray dashed lines.

example, attempting to add an element to the list immediately causes an exception to be thrown. Our initial expectation was this class would be easier to test relative to the other classes used, as the class exhibits the seeded faults with even basic testing. However, given that most subjects could not detect any faults in the class, it appears that in practice the class's very incorrect behavior makes traditional manual unit testing difficult. In contrast, EVOSUITE, generates high coverage test suites over *ArrayIntList* (on average, 78.7% branch coverage as compared to 50.5% for manually constructed suites) despite the system's faulty behavior, resulting in stronger test suites. A possible conjecture is that manual testers struggled to find ways to make use of this broken class, whereas EVOSUITE has no expectations based on past experience of similar data structures and simply generates test inputs covering the code, possibly in ways not conceived by manual testers.

> **RQ2:** *The replication provides no new evidence that using automatically generated test suites would lead to detection of more faults.*

4.3. RQ3: Tests Mismatching the Intended Program Behavior

For three of the four classes, the number of tests failing on the original class in the final test suite is higher for subjects using EVOSUITE than those manually constructing test suites, by 42.9%-563% on average (Figure 4(d)). Each test which fails on the original class represents a failure to capture the class's intended behavior. Thus, as before, we can see that manual testing seems to have an edge with respect to producing tests which, while not necessarily effective, do not erroneously flag faults during testing. As noted previously, we expect that the initial test suites produced by EVOSUITE will

contain several tests which fail on the correct system, as such suites are designed to pass on the original system. We nevertheless expected failing tests to be corrected as the test suite evolves, an expectation unmet in our initial study.

However, in contrast to the results from our initial study, for two of these classes — *DocType* and *ArrayIntList*— the difference is quite small, with the averages for both within one failing test, and p-values are relatively high, over 0.1. Additionally, for one class, *Rational*, testers manually testing the system typically produced test suites with roughly twice as many tests which fail against the original class (9.5 tests versus 5.1 tests), though the results yield a non-statistically significant p-value, 0.234.

Thus we can see that, in this respect, the results from our previous study are not supported by these new results. While the initial test suites generated by EVOSUITE do indeed frequently fail on the original class, it seems that (1) several subjects using EVOSUITE do manage to lower the number of failures, and (2) several subjects manually constructing test suites construct many tests which fail over the original system. As a result, at the end of the study the average number of failing tests in each test suite is comparable for *DocType* and *ArrayIntList*, and actually higher for *Rational*.

We illustrate this behavior in Figure 5, plotting the number of failing tests for each tester over time. As shown here, we can see that in most cases, subjects using EVOSUITE start with a number of failing tests and achieve varying levels of success in correcting them. However, we see that the final number of failing tests is typically comparable to the starting number of failing tests, though naturally some subjects are better or worse than others. In contrast, subjects manually testing the system naturally begin with no failing tests, but typically produce a number of failing tests, such that after one hour the number of failing tests is comparable for both sets of subjects. The chief exceptions to this are the *Option* class and (arguably) the *Rational* class. For *Option*, we can see, unlike the other classes, the initial number of failing tests starts quite high when using EVOSUITE— 5-15, typically, as compared to no more than 8 for the other classes. Furthermore, the number of failing tests either remains almost constant or drops some during testing. Thus while subjects testing *Option* manually do create erroneous tests (up to five), it is intuitively harder for them to create as many failing tests as EVOSUITE produces, as compared to when testing other classes.

We were surprised by the results for the *Rational* class: here, unlike the other classes in this study and all the classes in the previous study — including *Rational*— subjects not using EVOSUITE actually produced, on average, four more failing test than those subjects using EVOSUITE. It is unclear why most subjects manually testing struggled relative to EVOSUITE-enabled subjects for this specific class, particularly since this was not observed in the previous study. Additionally, we can see in Figure 5 that some subjects performing manual testing over *Rational* did produce test suites comparable to EVOSUITE-derived test suites. This variation in subjects performing manual testing results in the high p-value of 0.234, and the relatively low standardized effect size of 0.35 (Table IV(b)). Given the results from the previous study, we cannot conclude whether there is any real trend here, or rather noise in any of the two studies.

In any case, these results temper the conclusions made in Section 3.3 — while in some cases the use of automatic test generation can lead to an increase in tests which fail to capture intended class behavior, in other cases automatic test generation can result in a manageable level of failing tests.

> **RQ3:** *In the replication automatically generated tests did not always have a negative effect on the ability to capture intended class behaviour.*

4.4. RQ4: Regression Fault Detection

We estimate the regression fault detection ability by examining the mutation score achieved by the final test suite. As before, the results are inconclusive: for three of the four classes used, the average mutant score is higher when using EVOSUITE, 7.3%-27.9%, while for *ArrayIntList* the average fault detection rate is higher for manually constructed test suites (17.1% greater). For each class tested, the difference in average mutation score is not particularly high, no greater than 27.9%; typically, the variation between subjects within a testing strategy is greater than the difference in average mutation score. This is reflected in the high *p*-values and low standardized effect scores seen for mutation score in Tables IV(a)-IV(a): no *p*-value less than 0.101 and no standardized effect score greater than 0.7 was observed (both for *DocType*). Figure 4(e) also illustrates this behavior, with differences in median and average fault finding overshadowed by large, overlapping boxplots. Of particular note are results for *Option*— in contrast to the previous study, improvements in mutation generation are not clearly seen when moving from manual to EVOSUITE-driven testing. Note that statistically significant differences in mutation score observed for *DocType* in the previous study were not observed in this study, indicating that the configuration error in the previous study was the likely cause.

Like in the previous study, large, statistically significant increases in structural coverage were observed for most classes. Nevertheless, these gains are reflected only weakly by the mutation scores observed here. For example, test suites for *Option* and *DocType* have much higher branch coverage when they are generated through EVOSUITE; the improvements in mutation score are neither statistically significant (minimum *p*-value of 0.101) nor are the improvements relatively high (maximum 9%). To some extent this difference can be attributed to the nature of mutation analysis: Mutation typically leads to large numbers of trivial or similar mutants, such that one would not expect an increase proportional to the coverage increase. Nevertheless, these results indicate that while automatic test generation may offer improvements in regression testing, these gains appear to be limited.

In the Section 3.4, we provided two hypotheses concerning this unexpectedly weak relationship between coverage and mutation scores. First, we suggested that the higher number of failing tests when using EVOSUITE, which increase the coverage of the system without increasing the mutation score, may be to blame. Second, we proposed that the assertions generated by EVOSUITE are weaker than those written by manual testers. Given the results from Section 4.3, it appears that the first hypothesis relating to failing tests is not well supported. As discussed, only in the case of *Option* do tests generated with EVOSUITE contain a significantly higher number of failing tests. Test suites for the other three classes exhibit comparable levels of failing tests, yet — despite the higher structural coverage for suites generated with EVOSUITE— there is no significant difference in mutation score. The second hypothesis relating to weak generated assertions thus seems more plausible.

> **RQ4:** *The replication shows gains in mutation scores, but suggests that assertion generation needs to be improved.*

5. DISCUSSION

Our results for both studies indicate that while the use of automated test generation tools consistently improves structural coverage over manual testing, this is not reflected in 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

(a) Branch Coverage

(b) Fault Finding

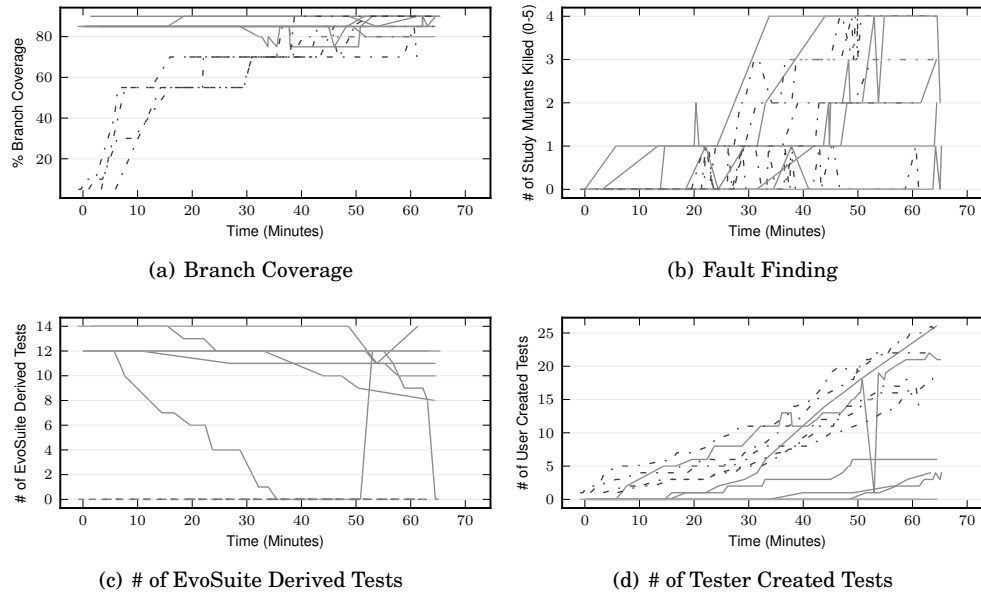(c) # of EvoSuite Derived Tests

(d) # of Tester Created Tests

Fig. 6: Test suite evolution for *Rational*, first study. EVOSUITE users shown in light gray solid lines, manual testers dark gray dashed lines.

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 Section 3 and 4, we further explore how the use of automated test generation impacts testing effectiveness and discuss implications for future work in automated test generation.

5.1. Evolution of a Test Suite

As shown previously, subjects kept using those test suites produced by EVOSUITE at the beginning of testing, even when it gives tests which were largely the same and unhelpful. In this latter case, the final, resulting test suites have 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 can better serve testers.

Towards this, in Figures 6 and 7, we illustrate how test suites change over time for the *Rational* and *DocType* classes in the first study. (Other classes in both studies exhibit behavior similar to *Rational* here; *DocType* in the first study is a special counterpoint.) These figures illustrate the branch coverage achieved, the number of EVOSUITE-derived tests, the number of user-created tests, and the number of study faults detected. Each line represents a single subject's test suite over time; dark gray dashed lines represent

(a) Branch Coverage

(b) Fault Finding

(c) Number of EvoSuite Derived Tests

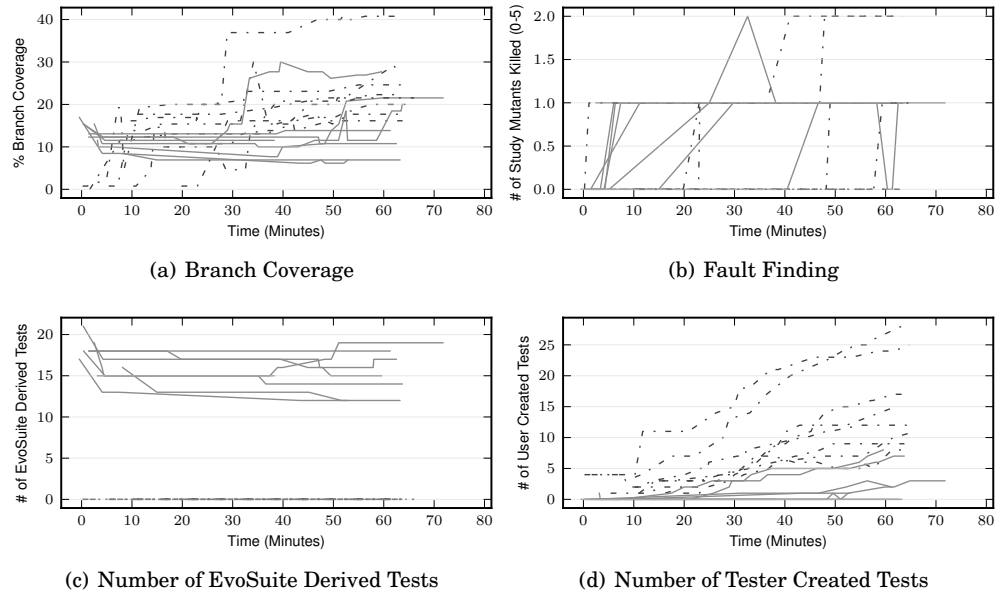(d) Number of Tester Created Tests

Fig. 7: Test suite evolution for *DocType*, first study. EVOSUITE users shown in light gray solid lines, manual testers dark gray dashed lines.

manually constructed test suites, while lighter solid lines represent EVOSUITE derived test suites.

First, consider the data related to *Rational* (Figure 6). 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, subjects working without EVOSUITE have no need to understand or modify any generated tests. These testers exhibit an immediate, linear increase in the number of tests created. The resulting rapid increase in branch coverage approaches—but does not quite achieve—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 (see Table V). For EVOSUITE derived test suites, we found that the size of the final (i.e., manually processed) test suite 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 unmodified EVOSUITE derived tests has a moderate negative correlation (-0.50), highlighting the need to evolve the test suite. Branch coverage, surprisingly, has

---

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

a weak relationship with fault detection (-0.21). With manual testing, however, both branch coverage and test suite size have a strong relationship with bug detection (0.79), as previous work suggests [Namin and J.H.Andrews 2009].

Now consider the data related to *DocType* from the first experiment (Figure 7). 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 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).

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 difficulty of testing *DocType*. Note that, as shown in Figure 2, the mutation score for manual test suites in the first study 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 begin to repair them, and 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 [Fraser and Arcuri 2012b; Lakhotia et al. 2010]. If testers struggle to improve and correct poor generated test suites, the use of automatic generation tools may be a drag on the testing process.

5.2. Influence of Assertions

In Sections 3.2 and 4.2, we noted that EVOSUITE generated tests would have been 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 V and Figure 8.

We can see from Figure 8 that 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* (for both studies), 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. However, in some cases the difference is not present, notably *DocType* and *ArrayIntList* in the second study, where no significant difference in the number of assertions generated was found.

Table V: Correlation of Test Suite Properties with Fault Detection,
Using Subjects' Final Test Suites

"MA" is the mean number of assertions, "NS" is the number of statements, "NT" is the number of tests. "GMK" is the number of generated mutants that were killed, "SFD" is the number of study faults detected. Correlations statistically significant at $\alpha = 0.05$ are formatted with bold fonts.

|	EVOSUITE *Testing*		*Manual Testing*	
	SFD	**GMK**	**SFD**	**GMK**
	Study #1			
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**
Option **NT**	0.0	**-0.05**	0.0	0.31
Rational **NT**	0.54	0.72	**0.97**	0.7
DocType **NT**	0.0	0.74	0.15	0.28
	Study #2			
Option **MA**	**0.50**	-0.10	0.02	**-0.15**
Rational **MA**	**-0.52**	**-0.61**	**0.46**	**0.51**
DocType **MA**	**-0.42**	0.10	0.02	**-0.22**
ArrayIntList **MA**	**0.42**	**0.21**	0.05	**0.28**
Option **NT**	**-0.25**	**0.62**	**0.71**	**0.83**
Rational **NT**	0.14	**0.37**	0.54	**0.57**
DocType **NT**	0.10	**0.74**	-0.4	0.19
ArrayIntList **NT**	0.26	0.27	-0.31	0.23

(a) Study #1   (b) Study #2
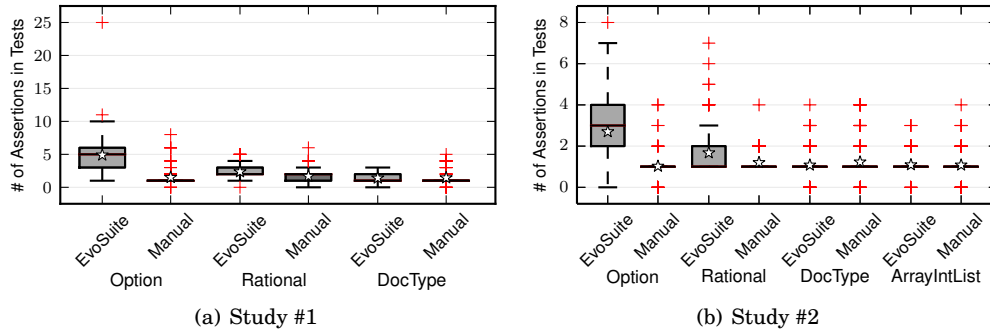
Fig. 8: Number of assertions per test (each box spans from 1st to 3rd quartile, middle line marks the median, whiskers extend up to $1.5\times$ the inter-quartile range, while plus symbols represent outliers and stars signify the mean).

Furthermore, from Table V we see that when manually testing *Rational* (in either study) the mean number of assertions per test suite has a positive (albeit low/moderate) correlation with mutation score and bug detection (0.30/0.46 and 0.35/0.51 respectively). However, when using EVOSUITE to test *Rational*, the mean number of assertions has a moderate, negative correlation with effectiveness (-0.65/-0.52 and -0.73/-0.61). 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 indicates 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. However, this behavior appears limited to *Rational*; results for other classes vary depending on the study and are often not statistically significant.

```
@Test
public void testAddAtEmptyValidIndex() {
  ArrayIntList list = new ArrayIntList();
  list.add(0,10);
  assertEquals(10, list.get(0));
}
```

Fig. 9: Assertions on getters with parameters are only found in manually written tests.

```
@Test
public void test12() {
  String rootElementName = "abc";
  String publicID = "123";
  String systemID = "abc123";
  DocType d = new DocType(rootElementName, publicID, systemID);
  DocType d2 = new DocType(d);
  assertEquals(d.getSystemID(),d2.getSystemID());
}
```

Fig. 10: Assertion using two getters.

Manual inspection of the test cases revealed several patterns of assertions present in manually written test cases, but not produced by EVOSUITE. In general, EVOSUITE produces its assertion from traces of return values, of results of object comparisons, and of results of calls to inspector values on the class under test and its mutants. Assertions are added where the traces produced on mutants differ from the traces of the un-mutated class under test, and these assertions follow a fixed set of patterns.

As an example of an assertion pattern that is not captured by EVOSUITE's process, Figure 9 shows a manually written test case for the *ArrayIntList* class, with an assertion using a call to the get method. Although EVOSUITE uses purity analysis to identify inspector methods, assertion generation only considers methods without parameters as inspectors (e.g., size(), isEmpty()). Thus, to produce an assertion like in Figure 9, EVOSUITE would first need to add the call to get with a parameter of 0 to the test case, and could only then add an assertion on the return value of that call. However, branch coverage does not provide any guidance towards this specific call, and so EVOSUITE is unlikely to produce this specific assertion.

Another related pattern found in manually written tests but not in EVOSUITE's tests is the use of more than one call to an inspector method in the same assertion. For example, Figure 10 shows a manually written test case performing a comparison of two *DocType* instances in terms of their getters. For EVOSUITE to reproduce this assertion, both calls would need to be contained in the test case, such that an assertion comparing the two return values could be generated. However, the optimization for branch coverage offers no incentive to do so: After the first call to getSystemID() the method is fully covered, and calling it again represents no coverage improvement.

Figure 11 shows an interesting case demonstrating the limits of automated assertion generation: The *Rational* class implements no equals method, which in Java means that it inherits the equals method from Object, and this version of equals only returns true when the two compared references point to the identical instance. That means that to compare whether two instances of *Rational* represent the same rational number, one needs to do this by calling inspector methods (e.g., doubleValue()) or by accessing the public fields numerator and denominator. EVOSUITE currently only compares objects by calling equals, it does not compare the public members individually. This severely

```java
public boolean equals(Rational r1, Rational r2) {
  if(r1.numerator == r2.numerator && r1.denominator == r2.denominator) {
    return true;
  } else {
    return false;
  }
}

@Test
public void testPow(){
  Rational r = new Rational(3,2);
  Rational pow = r.pow(2);
  Rational expected = new Rational(9,4);
  assertTrue(equals(pow,expected));
}
```

Fig. 11: The class *Rational* implements no custom `equals` method, making comparisons and assertion generation more difficult. Manual testers were able to circumvent this problem by defining a custom helper method they used in the tests.

```java
@Test
public void testGcf() {
  assertEquals(1,Rational.gcf(1, 1));
  assertEquals(2,Rational.gcf(6, 4));
  assertEquals(3,Rational.gcf(33, 102));
  assertEquals(11,Rational.gcf(33, 88));
  assertEquals(5, Rational.gcf(10, -15));
}
```

Fig. 12: Manually written "shotgun" test; if the method under test is easy to call and takes numeric parameters, manual testers play with variations even if it does not contribute to coverage.

restricts the possible assertions EVOSUITE can generate for *Rational*. Some of the manual testers were able to conveniently circumvent this problem by providing a custom `equals` method, as shown in Figure 11. Even though subjects using EVOSUITE could in theory add such a method manually as well, they had no option of making EVOSUITE use it.

Although manually written tests have fewer assertions on average, there are some exceptions. In particular, methods that can be called without needing to instantiate and configure many complex parameter objects are sometimes called repeatedly in a "shotgun" style as shown in Figure 12. EVOSUITE will not produce such tests, as it would aim to cover each branch in the method `gcf` independently, and once it is covered there will be no further tests for the same behaviour.

These differences in the generated assertions compared to manually written assertions may also make identifying the correct value more difficult. For example, Figure 13(b) shows a manually written test case where a specific string is stored in variable `rootElementName`, then used in a setter (`setRootElementName`), and then used again in the assertion to check whether the getter returns the correct value. It is easy to see that the value returned by this getter should be the same as the one for the setter. Figure 13(a) shows a test case for the same method (`setRootElementName`) generated by EVOSUITE. First, we notice that EVOSUITE uses the method `toString` rather than `getRootElementName` in the assertion. From the point of view of EVOSUITE this method is preferable as it detects the same mutants as `getRootElementName` (the root element

```
@Test
public void test11() throws Throwable {
  DocType docType0 = new DocType("MGRFCXLD", "MGRFCXLD", "MGRFCXLD");
  docType0.setRootElementName("MGRFCXLD");
  assertEquals("[nu.xom.DocType: MGRFCXLD]", docType0.toString());
}
```

(a) Generated test for method setRootElementName

```
@Test
public void testSetRootElementName0() {
  DocType docType = new DocType(testPublicID, testSystemID);
  docType.setRootElementName(testRootElemName);
  assertEquals(testRootElemName, docType.getRootElementName());
}
```

(b) Manually written test for method setRootElementName:
Strings are stored in a fixture.

Fig. 13: Example test cases illustrating the effects of randomly generated strings.

name is contained in the result of `toString`, yet `toString` will also detect other mutants that `getRootElementName` will not detect). However, from the point of view of understanding the test case, `toString` seems like the wrong choice. Second, the assertion generated by EVOSUITE uses a hard coded string, rather than refering to a variable. This means that to correct the assertion, the user first needs to make the connection between this concrete string and earlier inputs.

Figure 13 reveals another problem with automatically generated tests: EVOSUITE uses the string `MGRFCXLD` as input, and deciding on the correctness given an assertion on such a randomly generated string is potentially more difficult than using a "realistic" string, as used in manually written tests. This is a well known problem, and there are experimental techniques to improve the "readability" of generated strings (e.g., [Afshan et al. 2013]). However, EVOSUITE currently does not implement these, and thus strings are based on either random characters, or constants found in the bytecode of the class under test. The classes *DocType* and *Option* both are dependent on string inputs, and so this problem holds in both cases. Interestingly, however, these strings have only little influence on the control flow, such that the use of unrealistic strings only seems to be a problem for determining correctness of assertions, not for covering code. In the few cases where there are some constraints on strings, EVOSUITE was able to evolve the strings accordingly.

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*, a relatively smaller class, did subjects using EVOSUITE feel testing was easier than subjects manually testing. (*ArrayIntList* is comparably small, but as noted in Section 4 this class was challenging to test for other reasons.) Considering that subjects were more effective at fixing assertions for *Rational* than other classes, it seems that the more difficult a 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.

6. BACKGROUND AND EXIT QUESTIONNAIRES

Before starting the experiments, each participant received a background questionnaire to fill in, in order to collect information that might explain our results. At the end of the experiment, participants had to complete an exit questionnaire, in which they had the chance to provide feedback on their experiences. In this section, we analyze their responses.

6.1. Background Questionnaire

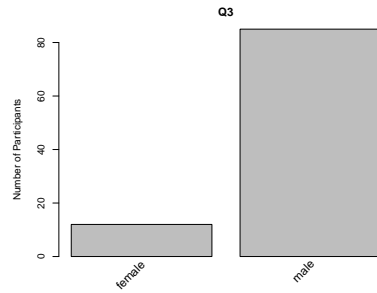The background questionnaire consisted of the following 15 questions:

Q1: Your name?
Q2: What is your birth year?
Q3: Gender?
Q4: What is your current education level?
Q5: What is your course?
Q6: Do you have industrial work experience in a Computer Science related field?
Q7: How much programming experience do you have?
Q8: How much programming experience do you have *in Java*?
Q9: How many programming languages do you know?
Q10: Have you used JUnit (or similar testing frameworks) before?
Q11: Have you used Eclipse before?
Q12: Do you have any prior experience with automated test generation?
Q13: How well do you understand the concept and usage of code coverage?
Q14: How often do you write unit tests when programming?
Q15: For each JUnit assertion, state whether you think it would evaluate to true or false:

```
(1) assertEquals(20, 4*5)
(2) assertTrue(new ArrayList<String>().isEmpty())
(3) assertNull(null)
(4) assertEquals(2.0, Math.sqrt(4.0), 0.1)
(5) assertSame(new Object(), new Object())
```

Figure 14 plots the demographics of participants on the basis of responses to initial questions in the background questionnaire, and Table VI shows the raw data. Most participants were undergraduate students in the Department of Computer Science at the University of Sheffield (as seen in parts (a) and (b) of the figure), a predominantly male environment.

Although some participants were drawn from industry for the first iteration of the study, participants in general had little industrial experience (part (c)), yet were generally well-trained for the task that was asked of them. Most had at least two, if not several years of programming experience (parts (d) and (e)), and all but one had experience in Java, while part (f) shows a general competence in several languages. Table VII shows that only a small minority had never used Eclipse (question 11), many had prior experience with automation in testing (question 12). Although answers were mixed with respect to the level of understanding of code coverage (question 13), a negative response here did not represent a barrier to participating in the experiment, since the onus was on fault finding rather than coverage. Answers were similarly mixed regarding the *habit* of writing unit tests in practice (question 14), as answers to the next question—question 15—showed a good level of understanding of JUnit assertions. Considering the influence of the answers to question 13 and 14 on the achieved branch coverage, Table VII shows less variations for users of EVOSUITE compared to manual testers. This seems to suggest that automated test generation tools

Table VI: Data for background questionnaire, questions 3–9. For each question and "Method" (EVOSUITE or Manual), "Size" is the no. of participants giving a particular response ("Group"). "Branch", "Mutation", "Failing" and "Faults" are median values for branch coverage, mutation score, failing tests on original and no. of faults found respectively. Confidence intervals ("CI") for median values are shown using bootstrapping at 95% confidence level.

| Question	Method	Group	Size	Branch	CI	Mutation	CI	Failing	CI	Faults	CI
Q3	EVOSUITE	female	4	86.78	[83.57, 98.57]	18.92	[-11.65, 32.30]	6.0	[0.00, 9.00]	0.5	[-1.00, 1.00]
		male	31	83.33	[80.95, 88.10]	51.48	[47.41, 56.37]	4.0	[1.00, 5.00]	1.0	[1.00, 2.00]
	Manual	female	7	25.71	[-28.57, 31.43]	50.00	[32.79, 75.93]	1.0	[-2.00, 2.00]	1.0	[0.00, 2.00]
		male	51	33.07	[16.15, 41.54]	51.47	[45.05, 61.27]	1.0	[0.00, 1.00]	1.0	[1.00, 2.00]
Q4	EVOSUITE	2nd year undergraduate	17	83.07	[80.44, 144.62]	42.59	[33.70, 55.19]	4.0	[3.00, 5.00]	0.0	[-1.00, 0.00]
		3rd year undergraduate	6	79.03	[68.79, 117.36]	56.96	[40.04, 71.45]	3.0	[-2.00, 4.50]	1.0	[-0.50, 2.00]
		4th year undergraduate	0	-	-	-	-	-	-	-	-
		industrial partner	0	-	-	-	-	-	-	-	-
		master's student	6	87.50	[84.29, 119.01]	61.11	[47.34, 72.28]	6.5	[5.00, 11.50]	1.0	[-1.00, 1.50]
		phd student	5	82.85	[77.14, 154.18]	44.44	[33.65, 73.15]	7.0	[-6.00, 11.00]	1.0	[0.00, 2.00]
		post doc	1	87.14	-	51.88	-	8.0	-	0.0	-
	Manual	2nd year undergraduate	25	33.07	[16.15, 41.54]	51.40	[45.36, 67.92]	1.0	[0.00, 2.00]	0.0	[-1.00, 0.00]
		3rd year undergraduate	10	23.84	[-6.54, 29.12]	58.09	[47.35, 74.53]	1.5	[-1.00, 2.00]	1.0	[0.00, 1.00]
		4th year undergraduate	4	44.23	[8.46, 70.00]	40.36	[16.26, 65.10]	3.5	[-11.00, 7.00]	0.5	[0.00, 1.00]
		industrial partner	1	90.00	-	77.88	-	0.0	-	4.0	-
		master's student	13	35.71	[-13.57, 49.89]	55.22	[47.11, 77.66]	1.0	[0.00, 2.00]	1.0	[1.00, 2.00]
		phd student	4	31.09	[-21.14, 52.20]	38.62	[16.64, 59.06]	1.0	[-2.00, 2.00]	1.0	[0.00, 2.00]
		post doc	1	85.00	-	73.95	-	8.0	-	4.0	-
Q5	EVOSUITE	?	1	87.14	-	51.88	-	8.0	-	0.0	-
		AI and computer science	4	89.28	[87.14, 158.57]	52.42	[49.19, 99.29]	8.0	[4.00, 14.00]	0.5	[0.00, 1.00]
		computer science	21	83.07	[81.15, 93.08]	49.00	[42.44, 57.38]	4.0	[2.00, 5.00]	1.0	[1.00, 2.00]
		enterprise computing	1	90.00	-	82.40	-	2.0	-	3.0	-
		itmb	1	75.00	-	22.11	-	5.0	-	1.0	-
		software engineering	7	85.00	[84.29, 108.57]	49.50	[32.66, 56.42]	5.0	[2.00, 9.00]	1.0	[-1.00, 2.00]
	Manual	?	1	85.00	-	73.95	-	8.0	-	4.0	-
		AI and computer science	2	28.97	[24.62, 33.33]	34.47	[14.29, 54.67]	1.0	[0.00, 2.00]	0.5	[0.00, 1.00]
		computer science	37	27.14	[15.40, 31.98]	51.47	[45.37, 61.27]	1.0	[-1.00, 1.00]	1.0	[1.00, 2.00]
		enterprise computing	1	80.00	-	64.47	-	0.0	-	1.0	-
		itmb	1	33.33	-	18.51	-	4.0	-	1.0	-
		software engineering	16	51.19	[17.38, 79.30]	46.22	[28.17, 59.67]	1.0	[-0.97, 1.50]	1.0	[0.00, 2.00]
Q6	EVOSUITE	none	20	83.09	[80.48, 92.15]	50.19	[45.54, 57.21]	4.0	[2.00, 5.50]	1.0	[1.00, 2.00]
		1 to 3 months	7	85.00	[81.43, 148.46]	49.35	[43.15, 63.52]	7.0	[6.00, 11.00]	0.0	[-1.00, 0.00]
		4 to 6 months	2	90.00	[90.00, 90.00]	75.92	[66.67, 85.19]	4.5	[3.00, 6.00]	3.0	[2.00, 4.00]
		7 to 12 months	2	75.71	[61.43, 90.00]	56.70	[31.00, 82.41]	5.0	[2.00, 8.00]	2.0	[1.00, 3.00]
		1 to 2 years	1	80.00	-	44.44	-	20.0	-	1.0	-
		3 to 5 years	3	83.07	[81.15, 138.46]	15.74	[-38.65, 31.48]	5.0	[3.00, 8.00]	0.0	[-1.00, 0.00]
		6 to 10 years	0	-	-	-	-	-	-	-	-
	Manual	none	34	29.23	[-8.21, 33.85]	47.88	[39.25, 58.44]	1.0	[-1.00, 2.00]	1.0	[1.00, 2.00]
		1 to 3 months	7	21.42	[18.57, 25.16]	50.00	[43.33, 69.05]	2.0	[1.00, 4.00]	1.0	[1.00, 2.00]
		4 to 6 months	7	58.46	[31.92, 91.54]	42.45	[21.57, 50.02]	3.0	[-2.00, 5.00]	1.0	[1.00, 2.00]
		7 to 12 months	3	33.07	[-23.85, 33.30]	64.38	[47.29, 69.15]	1.0	[0.00, 2.00]	1.0	[-2.00, 2.00]
		1 to 2 years	5	33.33	[25.90, 55.24]	57.89	[51.69, 101.50]	1.0	[0.00, 2.00]	0.0	[-2.00, 0.00]
		3 to 5 years	1	90.00	-	77.88	-	0.0	-	4.0	-
		6 to 10 years	1	85.00	-	73.95	-	8.0	-	4.0	-
Q7	EVOSUITE	1 year or less	1	85.00	-	35.18	-	13.0	-	0.0	-
		2 years	3	85.00	[80.00, 96.92]	58.27	[51.16, 110.99]	4.0	[-4.00, 7.00]	0.0	[-2.00, 0.00]
		3 years	11	83.33	[76.67, 105.24]	50.87	[42.89, 79.64]	4.0	[1.00, 5.00]	1.0	[1.00, 2.00]
		4 years	7	84.28	[82.86, 90.00]	49.00	[42.44, 53.56]	7.0	[6.00, 12.00]	1.0	[1.00, 2.00]
		5 to 10 years	12	82.96	[77.36, 148.24]	49.42	[37.69, 57.25]	4.0	[1.00, 6.00]	1.0	[0.50, 2.00]
		more than 10 years	1	87.14	-	51.88	-	8.0	-	0.0	-
	Manual	1 year or less	5	33.33	[8.21, 33.81]	37.33	[-3.84, 69.11]	2.0	[0.00, 4.00]	1.0	[-1.00, 2.00]
		2 years	5	70.00	[51.11, 115.38]	54.66	[33.84, 92.67]	4.0	[-3.00, 8.00]	0.0	[-2.00, 0.00]
		3 years	8	23.68	[-42.64, 28.90]	53.64	[30.36, 68.22]	1.5	[-1.00, 2.00]	1.0	[-1.00, 2.00]
		4 years	15	27.14	[4.29, 34.29]	41.66	[21.26, 59.26]	1.0	[-1.00, 2.00]	0.0	[-1.00, 0.00]
		5 to 10 years	20	27.08	[-19.16, 32.69]	52.61	[41.51, 62.00]	1.0	[0.00, 2.00]	1.0	[0.50, 1.50]
		more than 10 years	5	35.71	[-13.57, 61.43]	30.35	[-19.47, 46.43]	2.0	[0.00, 4.00]	1.0	[-1.00, 2.00]
Q8	EVOSUITE	1 year or less	12	84.03	[82.36, 94.04]	43.75	[31.14, 68.57]	5.0	[3.00, 6.50]	0.0	[-1.00, 0.00]
		2 years	7	21.53	[-41.92, 31.54]	43.75	[38.15, 49.04]	4.0	[1.00, 6.00]	1.0	[1.00, 2.00]
		3 years	10	84.16	[78.33, 114.49]	54.81	[30.00, 68.69]	3.5	[-1.00, 5.00]	1.0	[-1.00, 1.50]
		4 years	3	88.57	[87.14, 98.57]	49.50	[43.45, 52.41]	3.0	[-2.00, 4.00]	1.0	[0.00, 2.00]
		5 to 10 years	2	86.42	[82.86, 90.00]	60.95	[55.24, 66.67]	9.0	[6.00, 12.00]	1.0	[0.00, 2.00]
		none	1	87.14	-	51.88	-	8.0	-	0.0	-
	Manual	1 year or less	18	33.33	[-1.67, 42.05]	50.73	[44.49, 82.95]	2.0	[1.00, 4.00]	0.5	[0.00, 1.00]
		2 years	12	36.92	[9.62, 51.98]	54.64	[43.26, 72.88]	2.0	[0.00, 3.50]	0.5	[-1.00, 1.00]
		3 years	18	26.42	[-5.00, 34.29]	55.94	[48.80, 73.02]	1.0	[0.50, 2.00]	1.0	[0.50, 1.50]
		4 years	7	21.42	[-37.14, 24.40]	47.54	[27.87, 64.13]	3.0	[0.00, 6.00]	1.0	[1.00, 1.00]
		5 to 10 years	3	83.33	[81.67, 100.00]	32.78	[-8.38, 35.87]	1.0	[-6.00, 1.00]	0.0	[-4.00, 0.00]
		none	0	-	-	-	-	-	-	-	-
Q9	EVOSUITE	1 to 2	3	75.00	[70.00, 128.46]	49.35	[32.36, 76.59]	5.0	[5.00, 7.00]	1.0	[-1.00, 1.00]
		3 to 5	18	84.64	[83.57, 115.44]	47.93	[41.91, 54.77]	5.0	[3.00, 7.50]	1.0	[1.00, 2.00]
		5 to 10	14	85.35	[81.43, 97.64]	53.56	[48.26, 66.50]	4.0	[0.00, 5.50]	0.5	[0.00, 1.00]
	Manual	1 to 2	1	4.28	-	0.00	-	0.0	-	0.0	-
		3 to 5	32	33.33	[-3.33, 40.79]	51.43	[43.26, 66.76]	1.0	[-0.50, 1.00]	1.0	[1.00, 2.00]
		5 to 10	25	25.71	[10.66, 30.00]	55.22	[49.84, 68.78]	2.0	[1.00, 3.00]	1.0	[1.00, 1.00]

(a) Gender
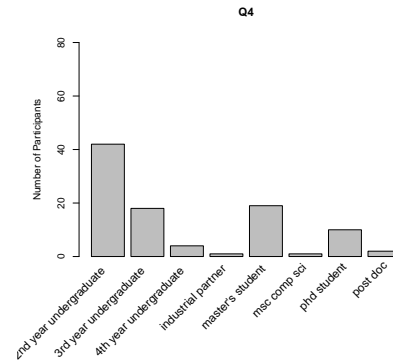
(b) Education Level
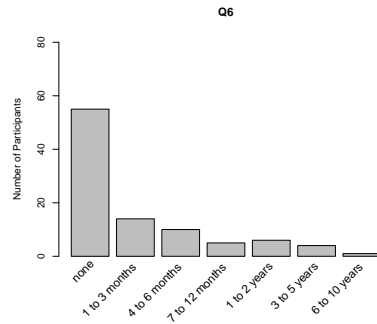
(c) Industrial Experience
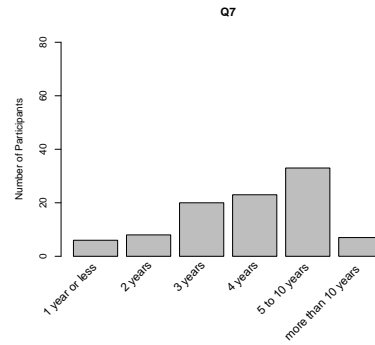
(d) General Programming Experience
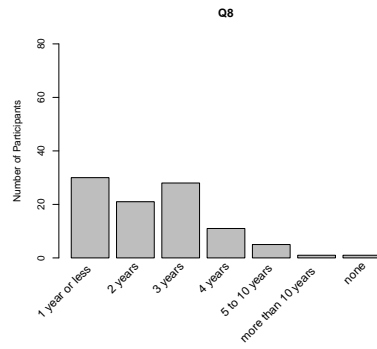
(e) Java Experience
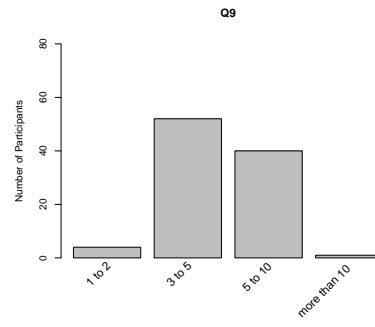
(f) Languages Known

Fig. 14: Demographics of the study participants, as collected from the background questionnaire. Each bar represents the number of participants for each given category.

Table VII: Data for background questionnaire, questions from Q10 to Q15. For Q15, we group the participants by the number of correct answers. For each question and "Method" (EVOSUITE or Manual), "Size" is the no. of participants giving a particular response ("Group"). "Branch", "Mutation", "Failing" and "Faults" are median values for branch coverage, mutation score, failing tests on original and no. of faults found respectively. Confidence intervals ("CI") for median values are shown using bootstrapping at 95% confidence level.

Question	Method	Group	Size	Branch	CI	Mutation	CI	Failing	CI	Faults	CI
Q10	EVOSUITE	no	2	45.76	[11.54, 80.00]	44.09	[43.75, 44.44]	12.0	[4.00, 20.00]	1.0	[1.00, 1.00]
		yes	33	85.00	[84.29, 90.00]	50.87	[46.52, 55.15]	5.0	[3.00, 7.00]	1.0	[1.00, 2.00]
	Manual	no	1	35.71	-	30.35	-	2.0	-	0.0	-
		yes	57	30.00	[19.23, 35.38]	51.47	[45.37, 60.49]	1.0	[0.00, 1.00]	1.0	[1.00, 2.00]
Q11	EVOSUITE	no	2	86.07	[85.00, 87.14]	58.63	[51.89, 65.38]	4.5	[1.00, 8.00]	1.0	[0.00, 2.00]
		yes	33	83.33	[80.95, 91.67]	49.35	[44.25, 54.95]	5.0	[3.00, 7.00]	1.0	[1.00, 2.00]
	Manual	no	2	24.72	[22.31, 27.14]	45.88	[24.07, 67.69]	0.5	[0.00, 1.00]	1.0	[0.00, 2.00]
		yes	56	32.96	[16.32, 40.93]	51.43	[45.36, 60.81]	1.0	[0.00, 1.00]	1.0	[1.00, 2.00]
Q12	EVOSUITE	no	24	85.00	[82.86, 90.00]	51.68	[45.10, 58.93]	5.0	[3.00, 6.00]	1.0	[1.00, 2.00]
		yes	11	82.85	[80.00, 151.87]	49.00	[44.03, 59.54]	3.0	[-6.00, 4.00]	1.0	[1.00, 2.00]
	Manual	no	43	28.57	[18.25, 32.86]	51.40	[46.14, 61.14]	1.0	[0.00, 2.00]	1.0	[1.00, 2.00]
		yes	15	33.33	[-16.67, 41.28]	57.44	[50.79, 96.38]	2.0	[0.00, 3.00]	1.0	[0.00, 2.00]
Q13	EVOSUITE	?	1	6.92	-	46.87	-	0.0	-	1.0	-
		not at all	3	85.71	[84.29, 92.86]	51.48	[51.08, 56.37]	3.0	[-2.00, 4.00]	0.0	[0, 0]
		very poorly	1	83.07	-	0.00	-	5.0	-	0.0	-
		somewhat poorly	9	85.71	[81.43, 98.35]	54.45	[50.05, 91.47]	4.0	[-4.00, 4.00]	0.0	[-1.00, 0.00]
		well	17	80.00	[75.00, 138.46]	49.00	[42.44, 62.81]	5.0	[2.00, 8.00]	1.0	[1.00, 1.00]
		very well	4	85.71	[81.43, 143.74]	60.95	[51.77, 72.40]	4.5	[-3.00, 7.00]	1.5	[1.00, 3.00]
	Manual	?	1	30.00	-	37.50	-	1.0	-	3.0	-
		not at all	4	52.77	[25.56, 88.41]	47.00	[18.51, 56.67]	1.5	[-1.00, 3.00]	0.5	[-1.00, 1.00]
		very poorly	5	33.07	[-23.85, 39.01]	48.23	[32.00, 96.47]	1.0	[-17.00, 2.00]	0.0	[-1.00, 0.00]
		somewhat poorly	16	33.09	[24.52, 41.90]	46.56	[36.47, 74.62]	1.5	[0.00, 2.50]	1.0	[1.00, 2.00]
		well	26	25.54	[-15.57, 29.56]	57.04	[50.76, 67.87]	1.0	[-1.00, 2.00]	1.0	[1.00, 1.50]
		very well	6	52.38	[17.26, 90.48]	47.52	[19.61, 71.11]	1.5	[-3.00, 2.50]	1.0	[-1.50, 2.00]
Q14	EVOSUITE	never	2	79.28	[78.57, 80.00]	45.52	[44.44, 46.60]	11.0	[2.00, 20.00]	0.5	[0.00, 1.00]
		rarely	11	85.71	[82.86, 88.57]	50.87	[46.52, 52.00]	5.0	[2.00, 7.00]	0.0	[-1.00, 0.00]
		occasionally	15	83.33	[79.52, 93.59]	49.00	[31.65, 68.00]	5.0	[3.00, 8.00]	1.0	[0.00, 2.00]
		often	5	90.00	[87.69, 173.08]	55.55	[28.70, 80.11]	4.0	[0.00, 8.00]	1.0	[-1.00, 2.00]
		always	2	56.34	[27.69, 85.00]	42.93	[15.74, 70.13]	4.5	[2.00, 7.00]	0.5	[0.00, 1.00]
	Manual	never	3	40.76	[1.54, 71.54]	60.60	[45.72, 103.03]	4.0	[4.00, 7.00]	2.0	[2.00, 3.00]
		rarely	20	26.59	[-20.15, 34.73]	53.06	[48.47, 64.08]	1.0	[0.00, 2.00]	1.0	[1.00, 2.00]
		occasionally	21	27.14	[20.95, 30.00]	47.54	[37.51, 65.38]	1.0	[-1.00, 1.00]	1.0	[1.00, 2.00]
		often	11	58.46	[28.03, 91.21]	62.06	[50.18, 86.80]	2.0	[-1.00, 4.00]	1.0	[-1.00, 2.00]
		always	3	33.33	[30.95, 48.10]	30.35	[-3.39, 55.16]	2.0	[2.00, 3.00]	0.0	[-1.00, 0.00]
Q15	EVOSUITE	2	3	80.00	[71.43, 153.08]	49.50	[32.66, 52.13]	3.0	[1.00, 6.00]	2.0	[1.00, 3.00]
		3	2	48.46	[6.92, 90.00]	17.77	[5.56, 30.00]	7.0	[2.00, 12.00]	0.5	[0.00, 1.00]
		4	16	84.03	[82.36, 94.86]	50.11	[44.99, 61.77]	6.5	[3.00, 10.00]	1.0	[1.00, 2.00]
		5	14	84.64	[80.71, 96.21]	53.17	[39.68, 62.59]	4.0	[1.50, 5.50]	0.0	[-1.00, 0.00]
	Manual	2	1	24.28	-	55.81	-	3.0	-	1.0	-
		3	12	23.46	[13.25, 26.98]	44.95	[30.15, 53.20]	0.5	[-1.50, 1.00]	1.0	[0.50, 2.00]
		4	27	33.33	[-3.33, 41.28]	56.52	[50.97, 80.26]	2.0	[0.00, 3.00]	1.0	[1.00, 2.00]
		5	18	34.52	[-5.95, 47.62]	50.70	[39.43, 63.13]	1.0	[0.00, 2.00]	1.0	[1.00, 2.00]

can level the playing field with respect to different levels of expertise. An interesting observation is that for question 14 the users of EVOSUITE who claim to always write unit tests when programming achieved lower coverage than the other groups (partially even significantly so). This may be an effect of the difficulty of changing habits — users who do not write tests so often do not need to change their habits as much to use automated test generation tools.

Question 15 presented five JUnit assertions and asked participants whether they would pass or not. Almost all scored 3/5 or more, with the majority scoring 4 or 5 out of 5. This again shows that participants were well-trained for the task in hand. Interestingly, none of the categories of responses to the different questions showed any form of correlation with different performance indicators obtained during the experiment. Median values for branch coverage, mutation score, failing tests on the

original artifact and faults found are shown in Table VII, with confidence intervals, for the answers to questions 10–15. Confidence intervals demonstrate a high degree of overlap between responses, and in many cases the numbers of participants giving a particular response are too small to draw any real conclusions.

6.2. Exit Questionnaire

At the end of the experiment, each participant received an exit questionnaire to complete. The questionnaire in the replicated study (presented below) is slightly different from that used in the original study. In the original study, participants either used EVOSUITE to generate test cases, or worked manually. Hence they were only asked questions relevant to the activity they undertook during the experiment. In the second iteration of the experiment, subjects participated in both activities, and so were presented with both sets of questions—i.e., those focussed on EVOSUITE and those centering on writing test cases manually. Finally, participants in the second experiment were asked a new question—whether they found working with EVOSUITE easier, or whether writing test cases manually was in fact easier, i.e., without the assistance of EVOSUITE.

The questions were as follows:

Q1: For each of the following questions about the part of the experiment where you *manually created tests*, please specify whether you: fully agree, partially agree, partially disagree, or fully disagree.
— (A) I had enough time to finish my task.
— (B) It was easy to test the given class.
— (C) I have produced a good test suite.
— (D) I am certain I have found all bugs.
— (E) The class under test was easy to understand.

Q2: For each of the following questions about *manually creating tests*, please specify whether you: fully agree, partially agree, partially disagree, or fully disagree.
— (A) Writing assertions is more difficult than creating the test input sequence.
— (B) A test generator would be useful for covering trivial code such as getters/setters.
— (C) A test generator would be useful for covering complex code.
— (D) Manual testing works great, there is no need for test generation tools.
— (E) Before writing tests manually, an automatic test generator should be applied.
— (F) Automated test generation should be applied after manual testing is finished.

Q3: What do you find most difficult about *manually writing unit tests*?

Q4: What features should an automated unit test generator have so that you would consider using it?

Q5: For each of the following questions about the part of the experiment where you *used EvoSuite to automatically create tests*, please specify whether you: fully agree, partially agree, partially disagree, or fully disagree.
— (A) I had enough time to finish my task.
— (B) It was easy to test the given class.
— (C) I have produced a good test suite.
— (D) I am certain I have found all bugs.
— (E) The class under test was easy to understand.

Q6: For each of the following questions about *automatic test generation with EvoSuite*, please specify whether you: fully agree, partially agree, partially disagree, or fully disagree.
— (A) It is easier to test with generated tests than having to manually write tests.
— (B) It is easier to confirm suggested assertions than to manually add them to generated tests.

— (C) Generated unit tests are difficult to read and understand.
— (D) Generated unit tests are too long to understand.
— (E) Generated unit tests are too short to exercise useful behaviour.
— (F) Automated test generation does not provide enough tests.
— (G) Automatically generated unit tests only exercise the "easy" parts of the program.
— (H) Automated test generation selects the right assertions.
— (I) Automated test generation selects too many assertions.
— (J) Adding assertions to generated unit tests is prohibitively difficult.
— (K) Performance is a prime concern when using automated test generation.
— (L) Difficulty in understanding generated unit tests depends on the complexity of the tested class, not the actual tests.

Q7: In what respect would the test generator need to be improved such that you would consider using it in practice? For each of the following aspects, please rank whether they are: irrelevant, nice to have or important.
— (A) Performance.
— (B) Coverage.
— (C) Readability.
— (D) Better assertions.
— (E) Complex behaviour.
— (F) Documentation / comments.
— (G) User interface.
— (H) More tests.

Q8: Do you have any suggestions on improving the readability of generated unit tests?

Q9: Do you have any suggestions on making unit test generation more usable?

Q10: Which of the two parts of today's experiment did you feel was easier?

Tables VIII, IX, and X show the results to the background questions; open text questions are discussed in Section 6.2.1. Figures 15 and 16 summarize the proportions of different responses to questions from the exit survey with stacked bar charts. The results demonstrate opinions that, on balance, were favorable towards EVOSUITE (and automation in general). However, these opinions did not always match actual reported experiences. For example, when quizzed on their experience when testing a particular class, responses to the statement "It was easy to test the given class" and "The class under test was easy to understand" were more in more favorable agreement for experiments where manual testing had actually been used rather than EVOSUITE (answers to part (b) and (e) respectively of questions 1 and 5 shown in Figure 15). Responses to the latter statement may allude to the possibility that it was easier to understand the class having been forced to write tests for it by hand.

However, when manual testers were quizzed specifically on their opinions regarding manual testing versus automation, they displayed strong support for the potential role of automation (question 2(d), Figure 16). Figure 16 further shows that manual testers in general preferred the idea of using automatic test case generation for "trivial" code (getters and setters) as opposed to more complex code (questions 2(b) and 2(c)), and generally participants preferred the idea of using automated test case creation before tackling the problem by hand, as opposed to the other way around (questions 2(e) and 2(f)).

When participants were quizzed specifically about EVOSUITE, subjects were of the opinion that writing tests was easier with the tool than without, and that having assertions generated automatically was also of benefit (questions 6(a) and (b), Figure 16). When responding to questions regarding specific aspects of EVOSUITE, the survey

A:38

Table VIII: Raw data for exit questionnaire, questions Q1 and Q5. For each question, we report how many participants answered to those questions, and how many agree and disagree.

Question	Size	Agree		Disagree	
		Fully	Partially	Partially	Fully
(Q.1A) Manual: I had enough time to finish my task.	72	21	21	19	11
(Q.5A) EvoSuite: I had enough time to finish my task.	73	20	30	17	6
(Q.1B) Manual: It was easy to test the given class.	72	14	39	17	2
(Q.5B) EvoSuite: It was easy to test the given class.	73	5	31	31	6
(Q.1C) Manual: I have produced a good test suite.	72	6	34	28	4
(Q.5C) EvoSuite: I have produced a good test suite.	72	5	28	33	6
(Q.1D) Manual: I am certain I have found all bugs.	72	2	20	30	20
(Q.5D) EvoSuite: I am certain I have found all bugs.	73	1	18	29	25
(Q.1E) Manual: The class under test was easy to understand.	72	21	31	17	3
(Q.5E) EvoSuite: The class under test was easy to understand.	73	8	27	26	12

Table IX: Raw data for exit questionnaire, questions Q2 and Q6. For each question, we report how many participants answered to those questions, and how many agree and disagree.

Question	Size	Agree		Disagree	
		Fully	Partially	Partially	Fully
(Q.2A) Manual: Writing assertions is more difficult than creating the test input sequence.	71	5	26	31	9
(Q.2B) Manual: A test generator would be useful for covering trivial code such as getters/setters	72	55	11	4	2
(Q.2C) Manual: A test generator would be useful for covering complex code	72	10	20	36	6
(Q.2D) Manual: Manual testing works great, there is no need for test generation tools.	72	0	14	41	17
(Q.2E) Manual: Before writing tests manually, an automatic test generator should be applied.	71	19	36	14	2
(Q.2F) Manual: Automated test generation should be applied after manual testing is finished.	71	5	17	32	17
(Q.6A) EvoSuite: It is easier to test with generated tests than having to manually write tests.	72	21	27	19	5
(Q.6B) EvoSuite: It is easier to confirm suggested assertions than to manually add them to generated tests.	72	21	33	16	2
(Q.6C) EvoSuite: Generated unit tests are difficult to read and understand.	73	4	30	31	8
(Q.6D) EvoSuite: Generated unit tests are too long to understand.	72	0	9	44	19
(Q.6E) EvoSuite: Generated unit tests are too short to exercise useful behaviour.	72	0	9	42	21
(Q.6F) EvoSuite: Automated test generation does not provide enough tests.	71	15	26	19	11
(Q.6G) EvoSuite: Automatically generated unit tests only exercise the "easy" parts of the program.	70	7	26	32	5
(Q.6H) EvoSuite: Automated test generation selects the right assertions.	73	1	40	27	5
(Q.6I) EvoSuite: Automated test generation selects too many assertions.	72	9	26	28	9
(Q.6J) EvoSuite: Adding assertions to generated unit tests is prohibitively difficult.	71	3	11	34	23
(Q.6K) EvoSuite: Performance is a prime concern when using automated test generation.	73	7	26	26	14
(Q.6L) EvoSuite: Difficulty in understanding generated unit tests depends on the complexity of the tested class, not the actual tests.	72	25	30	16	1

Table X: Raw data for exit questionnaire, question Q7. For each property, we report how many participants gave an opinion, and how they ranked the importance of each of those properties.

Question	Size	Irrelevant	Nice To Have	Important
Q. 7A: Performance	72	15	35	22
Q. 7B: Coverage	72	3	22	47
Q. 7C: Readability	71	3	20	48
Q. 7D: Better assertions	72	4	33	35
Q. 7E: Complex behaviour	72	14	39	19
Q. 7F: Documentation / comments	72	10	36	26
Q. 7G: User interface	72	29	31	12
Q. 7H: More tests	72	10	40	22
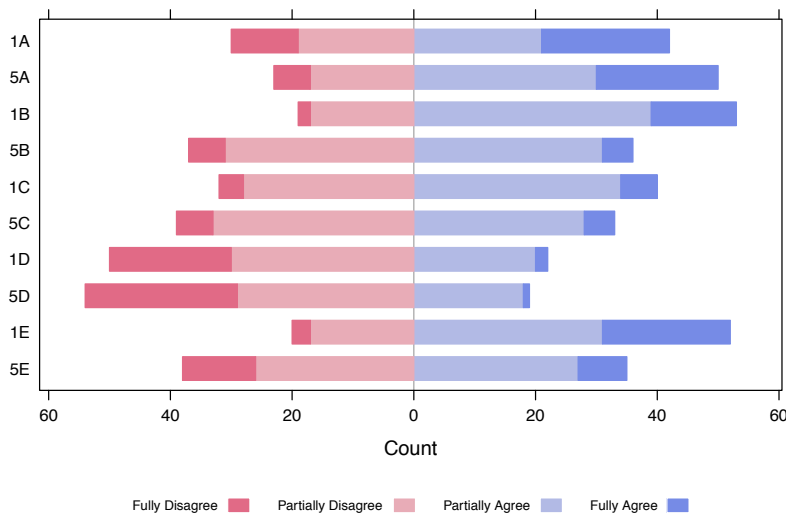
Fig. 15: Stacked bar charts for exit questionnaire, questions Q1 and Q5.

results found no general agreement that EVOSUITE presented obstacles to testing in the form of tests that were too complex, long or short (questions 6(c–k)). Participants generally agreed that understanding generated tests depended on the class being tested, not the actual tests themselves (question 6(l)). Participants clearly felt there were several avenues for improving the tool for its use in practice, however, as seen by the responses to question 7 of the exit survey, shown in Figure 17. Coverage and improved readability of test cases received the highest proportion of votes for importance of improvement while only refinements to the GUI were largely skewed towards "irrelevant" when participants were asked about the features that were relevant to its future improvement.

Participants that performed *both* manual testing and EVOSUITE were asked which of their classes they found easier to test. The results are reported in Table XI, with a majority responding that the class they tested using EVOSUITE was easier.

6.2.1. User Suggestions. As part of the background questionnaire, subjects were given the chance to provide feedback and suggestions on improving EVOSUITE and automated test generation.
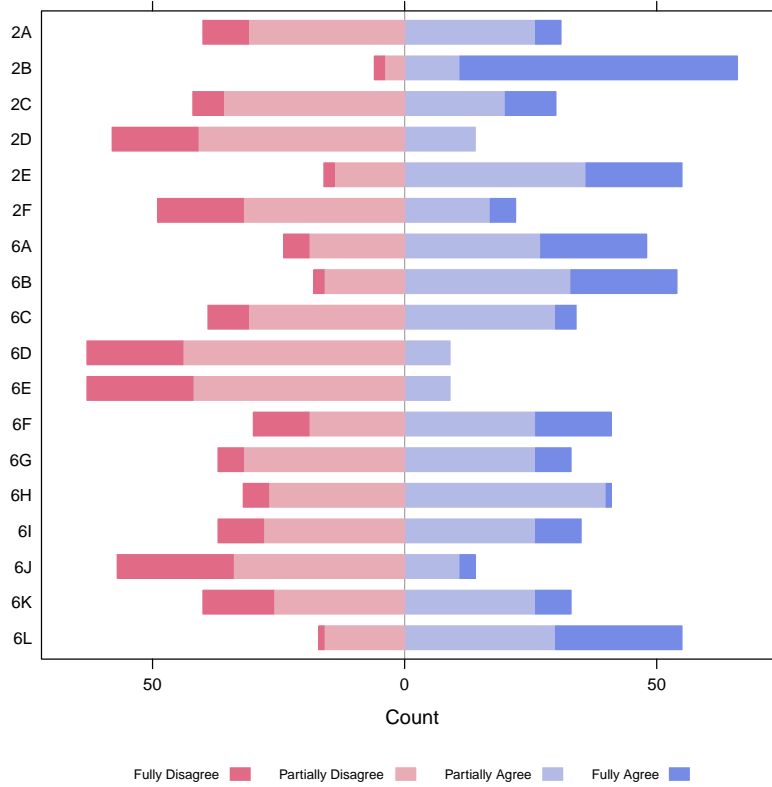
Fig. 16: Stacked bar charts for exit questionnaire, questions Q2 and Q6.
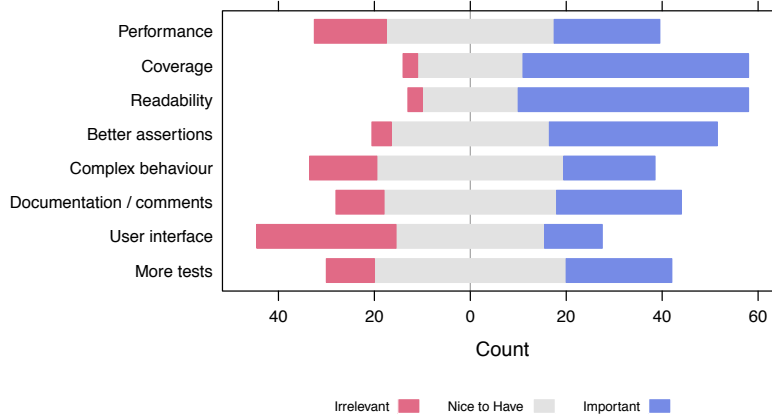

Fig. 17: Stacked bar charts for exit questionnaire, question Q7.

Table XI: Data for exit questionnaire, question Q10, which applied only to the replicated study. Participants are divided on whether they first use EVOSUITE (and then Manual), or the other way round.

Order	Size	EvoSuite is Easier	Manual Is Easier
EvoSuite and then Manual	22	12	10
Manual and then EvoSuite	24	19	5

In the first experiment, 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.

Comments fed back by participants of the second study mirrored those of the first. Several subjects commented on the tedious and time consuming nature of testing, and the desire for an automated tool like EVOSUITE to automate repetitive and trivial aspects of testing. Some participants suggested the ability to be able to select which methods to automatically test and which to leave for manual testing, and one suggested a facility for checking boundary input values thoroughly. An overwhelmingly frequent suggestion made by participants was the addition of code comments or JavaDocs around the tests explaining what the tests were doing (23 individual comments). Another was to assign more meaningful, readable identifiers (18 comments). A few participants commented on a general need to improve test case readability and in one case, string values. Related to this, one subject suggested a graphical component to explain what the automatically generated tests were doing.

7. IMPLICATIONS FOR FUTURE WORK

7.1. Test Generation Must Move Beyond Best-Effort

As discussed in Section 5, even given terrible generated tests, most testers still invest a significant amount of effort in understanding and attempting to repair the tests. Even if testers recognize that a test generation tool will sometimes produce very poor tests, they must invest some effort toward recognizing the generated tests should be discarded. This is a natural reaction, as there is typically an expectation with widely-used software development tools, such as compilers, debugging frameworks, IDEs, etc., that information presented to the developer is accurate.

In contrast, however, these tools have been developed to employ *best-effort* approaches, 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). For example, the use of EVOSUITE with *DocType* in the initial study essentially wastes the time of the developers; it seems likely that those testers will be far less likely to use EVOSUITE after experiencing such a poor return on time invested.

We therefore believe developers of automatic test generation tools must develop trustworthy test generation tools. Such tools should feature 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 *DocType* in the initial study (when EVOSUITE was misconfigured) and the much better test suites generated in the second study—either by coverage, or mutation score, or some other method—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 misconfigured *DocType* occur frequently in practice [Fraser and Arcuri 2012b], 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., [Fraser and Zeller 2011; Pastore et al. 2013; Afshan et al. 2013; Fraser and Zeller 2012]) and are 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.

7.2. 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 [Staats et al. 2012a], and based on our results we believe additional work focused on improving test oracle selection is needed.

7.3. Linking Automated Test Generation with Manual Test Generation

The scenario of our experiment was that a class is fully implemented and needs to be tested. Without specification, this process cannot be fully automated, which means that

the result of the automated test generator need to be manually processed, and the test suite potentially needs to be complemented by manual test cases. Our instructions to the participants were to start this by invoking EVOSUITE, and taking the resulting test suite as a starting point for the manual work. The exit questionnaire suggests that our participants agreed with this, and would generally prefer to start testing with automatically generated tests. However, as discussed in subsection 5.1 this approach is not without problems if the tool provides a starting test suite of poor quality.

An alternative way to integrate automated test generation into the testing process would be to start testing manually, and then to complement the manual tests with automatically generated tests. In fact this may happen implicitly, if developers follow a test-driven development approach and start by writing unit tests rather than code. Doing manual testing first, one could initially focus on the more "interesting" scenarios and, as also suggested by the exit questionnaire, defer testing of simpler code (e.g., getters and setters) to the automated testing tool for later. A particular advantage of this alternative would be that automated test generation could leverage information from the manually written tests. On one hand, this could lead to improved effectiveness and efficiency of the test generator (e.g., [Fraser and Arcuri 2012a; Yoo and Harman 2012]. On the other hand, these manually written tests could provide realistic input values or suitable assertions to improve readability of the generated tests, although we are aware of no existing technique that would exploit this information.

However, conversely manual testing can also benefit from automatic test generation. If the user is not familiar with an API, then automatically generated tests can provide possible usage examples and scenarios. Furthermore, trivial test obligations might only be recognised as such once automated tests have been generated, and the alternative approach would also raise the question when to stop testing manually and use the automated test generator. Indeed, the answer may be that the two approaches are best used together iteratively, alternating between manual and automatic test generation.

8. RELATED WORK

Although controlled human studies are not common in software engineering, there has been some recent work evaluating techniques with users (e.g., [Sautter et al. 2007]), and not always with positive results. Parnin and Orso [Parnin and Orso 2011] 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. [Staats et al. 2012b] 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. [Ramler et al. 2012], 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 [Fraser and Arcuri 2011], e.g., genetic algorithms, which have achieved several "human competitive" results (e.g., for genetic programming [Koza 2010]). While the application of search-based algorithms in software engineering (SBSE) has been increasing in the last few years [Harman et al. 2012], there have only been a few studies involving human subjects and SBSE. Pastore et al. [Pastore et al. 2013] 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. [Afshan et al. 2013] 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. [de Souza et al. 2010] 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.

This work is an extension of a previously published paper [Fraser et al. 2013]. It extends the previous work chiefly through the replication study presented in Section 4. This replication study adds 48 additional subjects, corrects the configuration issue found when using EVOSUITE to generate tests over *DocType*, and adds an additional Java class for study, *ArrayIntList*. Through this replication study, we have altered some conclusions — for example, finding that the use of EVOSUITE does not always lead to an increase in the number of failing tests in generated test suites — and have increased our confidence in many of our previous findings. Additionally, analyses which had to be cut or trimmed for space reasons in the previous work, for example many statistical analyses, have been re-included in this work.

9. CONCLUSIONS

Beginning with earliest attempts at automated test data generation in the 70s, the assumption has been that even partial automation of the testing process yields a net benefit. Initially, test data generation explicitly assumed that test data would be analyzed manually [Miller and Melton 1975], although later work distinguishes between the availability of an automated test oracle and manual test oracles [Wegener et al. 2001]. 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.

In this work, we have conducted two studies examining the core assumption behind automatic test generation — that by generating high coverage test data, we aid testers in constructing test suites capable of detecting faults. Our studies illustrate that this assumption does not appear to be true; merely achieving high coverage 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? Our results highlight several possible avenues to explore, in particular: providing a better indication of what each generated test input does; more careful selection of the assertions which form the foundation of the test oracles; and stronger, perhaps non-coverage based methods of reducing test suites.

Besides these important areas of future research on improving test generation, there is also the need to conduct further empirical studies to understand the effects and the problems of using automated test generation better, such as:

— **What is the influence of code ownership?** In our experiments, subjects were asked to test code they had not seen before. Thus, they spent a good amount of the time trying to understand what the existing code does. The effects of using automated unit test generation may be different if applied by the *developers* who write the code.
— **What is the influence of the complexity of the code under test?** In our experiments, the classes under test are all relatively easy by necessity, such that they can be understood and tested within an hour. If a class under test is more complex, will that lead to more benefits of using automated test generation, or do the problems

of automatically generated tests (e.g., tests without understandable purpose) become more inhibitive leading to overall worse results?

— **How does automated test generation influence maintenance?** Once generated, the unit tests need to be maintained together with the software they are testing. As automatically generated tests are sometimes less readable than manually written ones and have less clear purposes, what is the influence of automatic test generation throughout software maintenance?

— **How does the presence of partial oracles (e.g., assertions) change results?** The existence of powerful test generation tools may offer incentive to add more assertions or code contracts during development, potentially with profound effects on testing and quality.

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

10. ACKNOWLEDGEMENTS

REFERENCES

S. Afshan, P. McMinn, and M. Stevenson. 2013. 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)*. (To appear).

J. H. Andrews, L. C. Briand, and Y. Labiche. 2005. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th international conference on Software engineering (ICSE '05)*. ACM, New York, NY, USA, 402–411.

A. Arcuri and L. Briand. 2014. A Hitchhiker's Guide to Statistical Tests for Assessing Randomized Algorithms in Software Engineering. *Software Testing, Verification and Reliability* 24, 3 (2014), 219–250.

L. Baresi, P. L. Lanzi, and M. Miraz. 2010. TestFul: an Evolutionary Test Approach for Java. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. 185–194.

Raymond PL Buse, Caitlin Sadowski, and Westley Weimer. 2011. Benefits and barriers of user evaluation in software engineering research. In *ACM SIGPLAN Notices*, Vol. 46. ACM, 643–656.

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

J. T. de Souza, C. L. Maia, F. G. de Freitas, and D. P. Coutinho. 2010. The Human Competitiveness of Search Based Software Engineering. In *International Symposium on Search Based Software Engineering (SSBSE)*. 143–152.

Hyunsook Do and Gregg Rothermel. 2006. On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

G. Fraser and A. Arcuri. 2011. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software.. In *ACM Symposium on the Foundations of Software Engineering (FSE)*. 416–419.

G. Fraser and A. Arcuri. 2012a. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*.

G. Fraser and A. Arcuri. 2012b. Sound Empirical Evidence in Software Testing. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. 178–188.

G. Fraser and A. Arcuri. 2013. Whole Test Suite Generation. *IEEE Transactions on Software Engineering* 39, 2 (2013), 276–291.

G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg. 2013. Does Automated White-Box Test Generation Really Help Software Testers?. In *ACM International Symposium on Software Testing and Analysis (ISSTA)*.

G. Fraser and A. Zeller. 2011. Exploiting Common Object Usage in Test Case Generation. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE Computer Society, 80–89.

G. Fraser and A. Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Transactions on Software Engineering* 28, 2 (2012), 278–292.

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

M. Harman and P. McMinn. 2010. A Theoretical and Empirical Study of Search Based Testing: Local, Global and Hybrid Search. *IEEE Transactions on Software Engineering* 36, 2 (2010), 226–247.

Michael Inzlicht and Talia Ben-Zeev. 2000. A threatening intellectual environment: Why females are susceptible to experiencing problem-solving deficits in the presence of males. *Psychological Science* 11, 5 (2000), 365–371.

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

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

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

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

K. Lakhotia, P. McMinn, and M. Harman. 2010. An Empirical Investigation Into Branch Coverage for C Programs Using CUTE and AUSTIN. *Journal of Systems and Software* 83, 12 (2010), 2379–2391.

N. Li, X. Meng, J. Offutt, and L. Deng. 2013. Is bytecode instrumentation as good as source code instrumentation: An empirical study with industrial tools (Experience Report). In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*. IEEE, 380–389.

P. McMinn. 2004. Search-based Software Test Data Generation: A Survey. *Software Testing, Verification and Reliability* 14, 2 (2004), 105–156.

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

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

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

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

C.S. Pasareanu and N. Rungta. 2010. Symbolic PathFinder: symbolic execution of Java bytecode. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, Vol. 10. 179–180.

F. Pastore, L. Mariani, and G. Fraser. 2013. CrowdOracles: Can the Crowd Solve the Oracle Problem?. In *IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE. (To appear).

R. Ramler, D. Winkler, and M. Schmidt. 2012. 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)*. IEEE, 286–293.

G. Sautter, K. Böhm, F. Padberg, and W. Tichy. 2007. Empirical Evaluation of Semi-Automated XML Annotation of Text Documents with the GoldenGATE Editor. *Research and Advanced Techn. for Digital Libraries* (2007), 357–367.

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

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

M. Staats, G. Gay, and M.P.E. Heimdahl. 2012a. 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)*. 870–880.

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

N. Tillmann and N. J. de Halleux. 2008. Pex — White Box Test Generation for .NET. In *International Conference on Tests And Proofs (TAP)*. 134–253.

N. Tillmann and J. De Halleux. 2008. Pex—white box test generation for .NET. In *Tests and Proofs*. Springer, 134–153.

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

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

Y. Wei, C. Furia, N. Kazmin, and B. Meyer. 2011. Inferring better contracts. In *ACM/IEEE International Conference on Software Engineering (ICSE)*. ACM, 191–200.

S. Yoo and M. Harman. 2012. Test Data Regeneration: Generating New Test Data from Existing Test Data. *Software Testing, Verification and Reliability* 22, 3 (2012), 171–201. http://dx.doi.org/10.1002/stvr.435.

A. APPENDIX

A.1. Study Faults

Listing 1: *Option* study mutants.

```
@@ -610,7 +610,7 @@
        }
        if (longOpt != null ? !longOpt.equals(option.longOpt) : option.longOpt != null)
        {
-           return false;
+           return true;
        }

        return true;



@@ -336,7 +336,7 @@
     */
    public boolean hasArgs()
    {
-       return numberOfArgs > 1 || numberOfArgs == UNLIMITED_VALUES;
+       return numberOfArgs > -1 || numberOfArgs == UNLIMITED_VALUES;
    }

@@ -143,7 +143,7 @@
     */
    public int getId()
    {
-       return getKey().charAt(0);
+       return getKey().charAt(1);
    }

     */
    public boolean hasArgName()
    {
-       return argName != null && argName.length() > 0;
+       return argName != null && argName.length() >= 0;
    }

     */
    private void add(String value)
    {
-       if ((numberOfArgs > 0) && (values.size() > (numberOfArgs - 1)))
+       if ((numberOfArgs > 0) && (values.size() > (numberOfArgs + 1)))
        {
            throw new RuntimeException("Cannot add value, list full.");
        }
```

Listing 2: *Rational* study mutants.

```
@@ -92,7 +92,7 @@
    * @return <code>this</code>+<code>integer</code>.
    */
    public Rational add(long integer) {
-       return add(new Rational(integer, 1L));
```

```
+        return add(new Rational(integer, 0L));
     }

@@ -122,7 +122,7 @@
      * @return <code>this</code>*<code>r</code>.
      */
     public Rational multiply(Rational r) {
-        return new Rational(numerator * r.numerator, denominator * r.denominator).reduce();
+        return new Rational(numerator / r.numerator, denominator * r.denominator).reduce();
     }

@@ -193,7 +193,7 @@

         for(int i = 0; i < numFactors.length; i++) {
             for(int j = 0; j < denomFactors.length; j++) {
-                if(numFactors[i] == denomFactors[j] && numFactors[i] != 1L && denomFactors[j] != 1L) {
+                if(numFactors[i] == denomFactors[j] && numFactors[i] < 1L && denomFactors[j] != 1L) {
                     numFactors[i] = 1L;
                     denomFactors[j] = 1L;
                 }

@@ -112,7 +112,7 @@
      * @return <code>this</code>-<code>integer</code>.
      */
     public Rational subtract(long integer) {
-        return subtract(new Rational(integer, 1L));
+        return subtract(new Rational(integer, -1L));
     }

@@ -141,7 +141,7 @@
      * @return sqrt(<code>this</code>^2).
      */
     public Rational abs() {
-        return new Rational((numerator < 0L) ? -numerator : numerator, (denominator < 0L) ? -denominator :
     denominator).reduce();
+        return new Rational((numerator < 0L) ? -numerator : numerator, (denominator < 0L) ? +denominator :
     denominator).reduce();
     }
```

Listing 3: *DocType* study mutants.

```
@@ -359,7 +359,7 @@

    private void _setSystemID(String id) {

-        if (id == null && publicID != null) {
+        if (id == null) {
            throw new WellformednessException(
              "Cannot remove system ID without removing public ID first"
            );

@@ -591,7 +591,7 @@
            case 'l': return true;
            case 'm': return true;
            case 'n': return true;
-            case 'o': return true;
+            case 'o': return false;
            case 'p': return true;
            case 'q': return true;
            case 'r': return true;

@@ -293,7 +293,7 @@

        if (id != null) {
            int length = id.length();
-            if (length != 0) {
+            if (length != -1) {
                if (Verifier.isXMLSpaceCharacter(id.charAt(0))) {
                    throw new IllegalDataException("Initial white space "
                      + "in public IDs is not round trippable.");
```

```
@@ -233,7 +233,7 @@
     */
     public final void setInternalDTDSubset(String subset) {

-       if (subset != null && subset.length() > 0) {
+       if (subset != null == subset.length() > 0) {
            Verifier.checkInternalDTDSubset(subset);
            fastSetInternalDTDSubset(subset);
        }

@@ -425,7 +425,7 @@
    * @return zero
    */
    public final int getChildCount() {
-       return 0;
+       return 1;
    }
```

Listing 4: *ArrayIntList* study mutants.

```
@@ -230,7 +230,7 @@
    private final void checkRange(int index) {
-       if(index < 0 || index >= _size) {
+       if(index <= 0 || index >= _size) {
            throw new IndexOutOfBoundsException("Should be at least 0 and less than " + _size + ", found " +
                index);
        }
    }

@@ -188,7 +188,7 @@
        checkRangeIncludingEndpoint(index);
        incrModCount();
        ensureCapacity(_size+1);
-       int numtomove = _size-index;
+       int numtomove = _size/index;
        System.arraycopy(_data,index,_data,index+1,numtomove);
        _data[index] = element;
        _size++;

@@ -92,7 +92,7 @@
    public ArrayIntList(int initialCapacity) {
-       if(initialCapacity < 0) {
+       if(initialCapacity < -1) {
            throw new IllegalArgumentException("capacity " + initialCapacity);
        }
        _data = new int[initialCapacity];

@@ -140,7 +140,7 @@
        checkRange(index);
        incrModCount();
        int oldval = _data[index];
-       int numtomove = _size - index - 1;
+       int numtomove = _size - index;
        if(numtomove > 0) {
            System.arraycopy(_data,index+1,_data,index,numtomove);
        }

@@ -237,7 +237,7 @@
    private final void checkRangeIncludingEndpoint(int index) {
-       if(index < 0 || index > _size) {
+       if(index < 0 || index != _size) {
            throw new IndexOutOfBoundsException("Should be at least 0 and at most " + _size + ", found " +
                index);
        }
    }
```