Open access • Proceedings Article • DOI:10.1109/ASE.2015.86

# Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T) — Source link

Sina Shamshiri, René Just, José Miguel Rojas, Gordon Fraser ...+2 more authors

**Institutions:** University of Sheffield, University of Washington, University of Luxembourg

**Published on:** 09 Nov 2015 - Automated Software Engineering

**Topics:** Fault coverage, Test case, Code coverage, Automatic test pattern generation and Test suite

# Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges

Sina Shamshiri*, René Just†, José Miguel Rojas*, Gordon Fraser*, Phil McMinn* and Andrea Arcuri‡
*Department of Computer Science, University of Sheffield, UK
†Department of Computer Science & Engineering, University of Washington, Seattle, WA, USA
‡Scienta, Norway, and University of Luxembourg
*{sina.shamshiri, j.rojas, gordon.fraser, p.mcminn}@sheffield.ac.uk, †rjust@cs.washington.edu, ‡aa@scienta.no

***Abstract*—Rather than tediously writing unit tests manually, tools can be used to generate them automatically — sometimes even resulting in higher code coverage than manual testing. But how good are these tests at actually finding faults? To answer this question, we applied three state-of-the-art unit test generation tools for Java (Randoop, EvoSuite, and Agitar) to the 357 real faults in the Defects4J dataset and investigated how well the generated test suites perform at detecting these faults. Although the automatically generated test suites detected 55.7% of the faults overall, only 19.9% of all the individual test suites detected a fault. By studying the effectiveness and problems of the individual tools and the tests they generate, we derive insights to support the development of automated unit test generators that achieve a higher fault detection rate. These insights include 1) improving the obtained code coverage so that faulty statements are executed in the first instance, 2) improving the propagation of faulty program states to an observable output, coupled with the generation of more sensitive assertions, and 3) improving the simulation of the execution environment to detect faults that are dependent on external factors such as date and time.**

## I. Introduction

Unit testing is a common practice in object oriented programming, where each class is typically accompanied by a set of small, quickly executable test cases. There are many benefits to unit testing; unit tests serve as documentation and specification, and they help in finding faults. However, writing unit tests can also be tedious, and writing *good* unit tests can sometimes be more of an art than a science.

To support developers in unit testing, researchers have explored different approaches to automatically generate unit tests, thus relieving the developers of part of their hard work. These automated unit test generation tools have become very effective — automatically generated unit tests may even cover more code than those written by developers [14]. However, a high degree of code coverage does not imply that a test is actually effective at detecting faults.

Because code coverage cannot measure fault-finding effectiveness, a standard method is to compare test suites in terms of their mutation scores — that is, by measuring how many *seeded* faults a test suite finds. Even though there is evidence suggesting that test suites that are good at finding seeded faults are also good at finding real faults [3], [26], the quantification in terms of the mutation score can be misleading: Just because a test suite finds a high number of seeded faults does not necessarily mean that it will find the faults that *matter* — the ones that developers actually make.

In this paper, we empirically evaluate automatic unit test generation using the Defects4J dataset, which contains 357 real faults from open source projects [25]. We applied three state-of-the-art unit test generation tools for Java, RANDOOP [30], EVOSUITE [13], and AGITARONE [1], on the Defects4J dataset, and investigated whether the resulting test suites can find the faults. Specifically, we aimed to answer the following research question:

"How do automated unit test generators perform at finding faults?"

Our experiments indicate that, while it is possible for automated test generation tools to find more than half of the faults, running any individual tool on a given software project is far from providing any confidence about finding faults. In fact, only 19.9% of all the test suites generated as part of our experiments find a fault. This raises a second research question:

"How do automated unit test generators need to be improved to find more faults?"

Our experiments indicate that in many cases, code coverage remains a major problem, preventing automatically generated unit tests from detecting faults. More concretely, 16.2% of all faults and 36.7% of the non-found faults were never even executed by the generated tests in the first place. However, code coverage is not the only problem: 63.3% of the non-found faults *were* covered by automatically generated tests at least once, but the tests did not manage to reveal them. This calls for improved techniques to achieve fault propagation and for improved generation of assertions to detect the propagated faults. Unexpectedly, we also found that 15.2% of all tests were flaky — that is, their passing/failing behavior was temporary due to environmental and other dependencies, rendering them useless and providing reinforcement for recent research on test isolation [5].

The contributions of this paper are as follows:

1) A large-scale experiment, applying three state-of-the-art automated unit test generators for Java to the 357 faults in the Defects4J dataset.
2) A detailed analysis of how well the generated suites performed at detecting the faults in the dataset.
3) The presentation of a series of insights gained from the study as to how the test generators could be improved to support better fault discovery in the future.

Section II describes the methodology behind our experiments, including a discussion of the automatic unit test generators studied, how we designed the experiments, and threats to validity inherent in our study. Sections III and IV detail our results and answer our research questions. Section V discusses related work and Section VI concludes.

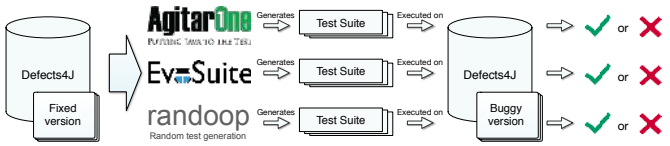



Fig. 1: Overview of the experimental setup. For each fault, the Defects4J dataset provides a *buggy* (i.e., faulty) and a *fixed* version. Test suites are generated with all tools on the fixed version, and executed on the buggy version.

## II. Methodology

In order to answer our research questions, we designed an experiment according to the high-level methodology shown in Figure 1. We considered Defects4J, a database of known, real faults, where each fault is represented by a buggy and a fixed program version. We applied three automated test generation tools (AgitarOne, EvoSuite, and Randoop) on the fixed version, and executed each generated test suite on the buggy version to determine whether it detects the fault — that is, whether is fails on the buggy version. This means that we are considering a regression testing scenario, where a developer would apply test generation to guard against future faults. We then analyzed in detail which faults were detected and how, and which faults were not detected, and why. This section describes the experimental setup in detail.

### A. Subject Programs

The Defects4J [25] dataset consists of 357 real faults from five open source projects: *JFreeChart* (26 faults), *Google Closure compiler* (133 faults), *Apache Commons Lang* (65 faults), *Apache Commons Math* (106 faults), and *Joda Time* (27 faults). Defects4J makes analyzing real faults easy: For each fault, it provides 1) commands to access the buggy and the fixed program version, which differ by a minimized change that represents the isolated bug fix, 2) a developer-written test suite containing at least one test case that can expose the fault, and 3) a list of classes relevant to the fault — that is, all classes modified by the bug fix. As per the regression testing scenario (i.e., guarding against future faults in modified code), we used this list of relevant classes for test generation.

### B. Automated Unit Test Generation Tools

All software projects in Defects4J are written in Java, therefore we also considered test generation tools for Java. NightHawk [4], JCrasher [10], Carfast [31], T3 [33], and Randoop [30] are instances of random unit test generation tools. We chose Randoop as it is the most stable and popular representative of these random testing tools.

TestFul [6], eToc [40], and EvoSuite [13] are instances of search-based unit test generation tools, which use meta-heuristic search techniques to optimize test suites with respect to code coverage criteria. We chose EvoSuite as it subsumes the other tools in terms of functionalities, and is the only actively maintained tool out of the three.

Dynamic Symbolic Execution (DSE) testing tools for Java such as DSC [22], Symbolic PathFinder [32], and jCute [36] require test drivers, whereas our scenario of unit test generation assumes that the tests generated include these test drivers — as typical unit tests do (i.e., in terms of method call sequences). While EvoSuite has an experimental DSE extension [17], it is not yet enabled by default, and therefore was not used in our experiments. Note that for C#, tools such as Seeker [38] or Symstra [45] can generate method call sequences, but the only publicly available tool — Pex [39] — has only a limited ability to do so. However, our experimental setup based on Defects4J requires tools that apply to Java.

Finally, there are commercial unit test generation tools for Java, of which naturally less is known about implementation details. We tried Analytix CodePro [9], Parasoft JTest [24], and AgitarOne [1]. CodePro achieved low coverage and is no longer officially supported. Although we were able to generate test suites with JTest, we were unable to execute the tests with our analysis framework (even with help from Parasoft's support team) because the tests depend on JTest's proprietary stub generation utility. As AgitarOne exhibited fewer problems with test dependencies compared to JTest, we chose AgitarOne for our experiments.

Randoop [30] implements feedback-directed random test generation for object-oriented programs. This means that it iteratively extends sequences of method calls with randomly selected candidates until the generated sequence raises an undeclared exception or violates a general code contract. Randoop also executes its generated sequences and creates assertions that capture the behavior of the class under test. However, Randoop cannot target a specific class under test since it uses a bottom-up approach that requires that all dependencies of a method call (i.e., arguments) have been instantiated in a prior iteration. Randoop therefore requires, as input, a list of all classes it should explore during test generation. For each class under test we provided a list containing the class under test and all its dependencies as determined by Defects4J[1]. We applied Randoop for three minutes on each run, using a unique random seed since Randoop is deterministic by default (i.e., the random seed is always 0). For all other settings, we applied the defaults, with the exception of enabling null values as method arguments with a probability of 0.1; this improves Randoop's effectiveness in our experience [26].

EvoSuite [13] applies a genetic algorithm in order to evolve a set of test cases that maximizes code coverage. It starts with a population of test suites of random test cases, and then iteratively applies search operators such as selection, mutation, and crossover to evolve them. The evolution is guided by a fitness function based on a coverage criterion, which is branch coverage by default. Once the search ends, the test suite with the highest code coverage is minimized with respect to the coverage criterion and regression test assertions are added [16]. EvoSuite then checks for each test whether it is syntactically valid by compiling it, and executes it to check whether it is stable (i.e., passing). Any failing assertions at this point are commented out by EvoSuite. Since EvoSuite can target a specific class under test, we generated, for each fault, a test suite targeting only classes relevant to that fault, as reported by Defects4J. We used EvoSuite with its default options, except for two settings: 1) We set the stopping criterion for the search to three minutes per class and 2) we deactivated the mutation-based filtering of assertions such that all possible regression assertions are included, because the filtering can be computationally costly and caused some timeouts.

AgitarOne [1] is a commercial test generation tool for Java developed by Agitar Technologies, which is advertised as being able to achieve 80% code coverage or better on any class. The tool consists of a client and a server application. According to AgitarOne's support, the test generation is

[1]Defects4J dynamically determines dependencies by monitoring the class loader during the execution of the developer-written tests.

"fairly" deterministic, and a user should *not* need to generate multiple test suites for a given program version[2]. The client is an Eclipse plugin that connects to the server[3] to request the generation of test cases, and the server takes a variable amount of time to do so, depending on the class under test. As we did not succeed in automating AGITARONE, we imported the fixed project version of each Defects4J bug into Eclipse and manually invoked AGITARONE's Eclipse plugin to generate test suites. AGITARONE can target a specific class under test, and we therefore generated a test suite for each class relevant to the fault. If AGITARONE failed to generate tests for a class, we re-attempted the request five times. Nevertheless, the tool was not able to generate a test suite for 34 (9.5%) faults (RANDOOP and EVOSUITE were not able to generate any test suites for 2 and 1 (<1%) faults). AGITARONE makes heavy use of a proprietary mocking system that requires AGITARONE's own test runner. As we could not integrate this test runner in Defects4J's infrastructure, which uses Apache Ant's JUnit test runner, we collected the same experimental data as for the other tools using AGITARONE's dedicated test runner.

### C. Experiment Procedure

We applied the following procedure.

*1) Test Generation:* RANDOOP and EVOSUITE are both randomized tools, capable of producing different results on each invocation. To account for this randomness, we generated 10 test suites for each tool and fault on the fixed version as provided by Defects4J. As AGITARONE required manual effort to generate and analyze a test suite (e.g., manually selecting the classes under test and starting the tool in Eclipse), we only generated 1 test suite for each fault with AGITARONE.

*2) Flaky Tests:* In order to determine whether a test detects a fault, we require it to pass on the fixed version and fail on the buggy version. However, tools may generate *flaky* (unstable) tests, which may also fail on the fixed version. For example, a test case that contains an assertion that refers to the system time will only pass during generation, and will fail when re-executed later. We applied the following automated process to remove flaky tests: First, all non-compiling test classes were removed. Then, each compilable test suite was executed on the fixed version five times. If any of these executions revealed flaky tests, then these tests were removed, and the test suite re-compiled and re-executed. This process was repeated until all remaining tests passed five times in a row.

Technically, a test may reveal its flakiness only at a later stage (e.g., if a test depends on the current date, it might fail only after the current day is over). By that time, the source code of the class under test might have been changed, and one would need to spend time to understand whether a failing test has found a fault or it is just flaky. Although possible, we did not encounter any such cases in our experiments.

*3) False Positives:* Even if a test is not flaky, it might still fail on the buggy version for reasons that are unrelated to the actual fault — that is, it is a *false positive*. Flaky tests can technically also be false positives, but false positives mainly happen when test generation tools break the object-oriented principle of encapsulation, for example by calling private methods directly through reflection, or by capturing outdated behavior of dependency classes with mocks.

We identified false positives as follows: For each test that failed on a buggy version, we compared the failure message and stack trace produced by the failing tests of the developer-written test suite included in Defects4J with that of the generated test. If a test failed with the same exception or a similar assertion, we considered it a true positive. If the exception or assertion differed, we manually validated whether the failure was caused by the fault or whether it is a false positive.

We found false positives for RANDOOP and AGITARONE but not for EVOSUITE. Note that RANDOOP suffered false positives only for Closure, and only with tests that executed a dependency class rather than any of the relevant classes. For AGITARONE we identified two common types of false positives: test failures due to mocking and accessing (missing) private class members. Consequently, we automatically classified an AGITARONE test as false positive if it only failed because of mocking or missing private class members (see Section IV-D). While we could have prevented this problem to a certain extent by changing the faults provided by Defects4J (i.e., inlining all changes or maintaining unused code), we argue that this would not reflect common practice.

*4) Fault Detection:* To determine fault detection we executed the test suites against the buggy version of each bug as provided by Defects4J. For RANDOOP and EVOSUITE, we executed the test suites using Defects4J's JUnit test runner; for AGITARONE we used its proprietary JUnit test runner. For each executed test, we collected information on whether it passed or failed, and if it failed we logged the reason (i.e., the failure message and stack trace).

*5) Coverage Analysis:* In order to study how code coverage relates to fault detection, we measured statement coverage on each class relevant to the fault. Furthermore, given the set of program statements modified by the bug fix (i.e., the difference between the buggy and the fixed version), we measured *bug coverage* — that is, whether a fault was 1) fully covered (all modified statements covered), 2) partially covered (some modified statements covered), or 3) not covered.

For RANDOOP and EVOSUITE, we used Cobertura[4] to measure code coverage. As AGITARONE's proprietary coverage tracking mechanism conflicts with Cobertura, we relied on the proprietary coverage files generated by AGITARONE. To that end, we extended Crap4j[5] [35] and extracted the code coverage ratio from the coverage files produced by AGITARONE's test runner for each relevant class. To determine whether a fault was fully, partially, or not covered, we manually inspected the visual code coverage indicators of the modified statements using AGITARONE's Eclipse plugin.

### D. Threats to Validity

In this study, we used bugs taken from only five Java open source projects, which may not generalize to all programming languages and different program characteristics, and thus constitutes a threat to *external validity*. Our study considered three state-of-the-art test generation tools, of which one is commercially available and actively used by developers. However, our study does not include tools based on Dynamic Symbolic Execution or other specialized techniques, and such tools may be more effective at some challenges we identified than the tools used in our study (e.g., complex conditions).

[2]Personal communication, August 2015

[3]The test generation server was deployed on a Linux computer with 64GB of RAM and 32 CPU cores @2.1GHz

[4]http://cobertura.github.io/cobertura/, accessed May 2015

[5]Java project quality assessment tool, originally written by the developers at Agitar Technologies

TABLE I: Overall outcome of the test generation and execution process. For each *project* and *tool*, the table shows the percentage of *compilable* test classes in all test suites, the average number of generated *tests* in them, the percentage of how many of these tests are *flaky*, the percentage of failing non-flaky tests that were *false positives*, and the average code *coverage* ratio for all non-flaky test suites on classes relevant to the bug. It also shows the *max* and *average* number of bugs per project that each tool detected (excluding false positives), and details how the bugs were detected (i.e., a failing *assertion*, an unhandled *exception*, or a *timeout*). Note that for EVOSUITE and RANDOOP, a bug might have been detected by only a subset of the 10 generated test suites.

| Project | Tool | Compilable | Tests | Flaky | False Pos. | Coverage | Max Bugs | Avg. Bugs | Assertion | Exception | Timeout |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Chart | AGITARONE | 100.0% | 131.2 | 0.2% | 30.6% | 84.7% | 17 | 17.0 | 10.0 | 11.0 | 0.0 |
| | EVOSUITE | 100.0% | 45.9 | 3.5% | 0.0% | 68.1% | 18 | 9.7 | 5.4 | 5.2 | 0.3 |
| | RANDOOP | 100.0% | 4874.9 | 36.8% | 0.0% | 54.8% | 18 | 14.1 | 7.5 | 9.1 | 0.0 |
| | Manual | 100.0% | 230.6 | 0.0% | 0.0% | 70.5% | 26 | 26.0 | 17.0 | 12.0 | 0.0 |
| Closure | AGITARONE | 100.0% | 199.4 | 0.4% | 79.3% | 79.1% | 25 | 25.0 | 16.0 | 10.0 | 0.0 |
| | EVOSUITE | 100.0% | 34.9 | 1.7% | 0.0% | 34.5% | 27 | 11.8 | 10.5 | 1.4 | 0.0 |
| | RANDOOP | 98.4% | 5518.4 | 19.8% | 15.8% | 9.8% | 9 | 2.2 | 0.5 | 1.7 | 0.0 |
| | Manual | 100.0% | 3511.1 | 0.0% | 0.0% | 90.9% | 133 | 133.0 | 103.0 | 42.0 | 0.0 |
| Lang | AGITARONE | 100.0% | 127.7 | 1.0% | 23.5% | 50.9% | 22 | 22.0 | 10.0 | 14.0 | 0.0 |
| | EVOSUITE | 79.5% | 48.6 | 5.4% | 0.0% | 55.4% | 18 | 9.2 | 5.5 | 3.3 | 0.9 |
| | RANDOOP | 68.3% | 11450.7 | 5.7% | 0.0% | 50.7% | 10 | 7.0 | 1.7 | 6.3 | 0.0 |
| | Manual | 100.0% | 169.2 | 0.0% | 0.0% | 91.4% | 65 | 65.0 | 31.0 | 36.0 | 0.0 |
| Math | AGITARONE | 100.0% | 105.8 | 0.1% | 8.9% | 83.5% | 53 | 53.0 | 34.0 | 25.0 | 0.0 |
| | EVOSUITE | 99.8% | 29.7 | 0.2% | 0.0% | 77.9% | 66 | 42.9 | 26.1 | 17.7 | 0.3 |
| | RANDOOP | 97.8% | 7371.4 | 15.6% | 0.0% | 43.4% | 41 | 26.0 | 17.8 | 10.8 | 0.0 |
| | Manual | 100.0% | 167.8 | 0.0% | 0.0% | 91.1% | 106 | 106.0 | 76.0 | 31.0 | 0.0 |
| Time | AGITARONE | 100.0% | 187.2 | 3.3% | 30.9% | 86.7% | 13 | 13.0 | 10.0 | 8.0 | 0.0 |
| | EVOSUITE | 100.0% | 58.0 | 2.8% | 0.0% | 86.7% | 16 | 8.5 | 4.9 | 4.0 | 0.0 |
| | RANDOOP | 81.1% | 2807.1 | 25.3% | 0.0% | 43.0% | 15 | 4.5 | 3.8 | 1.1 | 0.0 |
| | Manual | 100.0% | 2532.7 | 0.0% | 0.0% | 91.8% | 27 | 27.0 | 13.0 | 17.0 | 0.0 |

A potential threat to *internal validity* is that not all tests that detected a fault were manually investigated to ensure the validity of the bug detection result. However, we mitigated this threat by using several sanity checks, such as comparing failure reasons of generated and developer-written test suites. Furthermore, we manually inspected a large number of test suites, in particular the ones that exhibited an unexpected failure reason. AGITARONE may produce different test suites when invoked on the same class several times, depending on external factors such as available resources. Thus, there is a potential threat as we only generated a single test suite for each bug with AGITARONE. However, we experimentally validated the claim of Agitars support of the tool being fairly deterministic by sampling 10 faults, and found that the tool is generally consistent in whether it detects a bug. Moreover, AGITARONE spent significantly more than three minutes on some classes, and allowing RANDOOP and EVOSUITE more time for test generation may produce better results. However, based on our experience, a search budget of three minutes is sufficient for the search in EVOSUITE to converge in most cases, such that more time would not further change the tests. For RANDOOP we observed that code coverage saturated already within <1 min, and the test suites exhibit a very high degree of redundancy. Therefore, we do not expect our choice of test generation time to affect effectiveness.

A potential threat to *construct validity* is the use of all bugs in the Defects4J dataset, as Defects4J does not distinguish the type or severity of faults. Furthermore, each bug is represented by a minimized diff between the buggy and a later fixed version, rather than the actual code change that introduced the bug. Thus, although the bugs are real bugs, not all may be representative for regression faults. Therefore, our study might underestimate the effectiveness of automated test generation tools for regression testing if some types of the undetected faults are unlikely to be inadvertently introduced by a developer in the future. Likewise, our study might overestimate the effectiveness if some types of the detected faults are unlikely to be inadvertently introduced in the future. Furthermore, our study relies on the versions of code committed to a public repository and does not include regressions that a developer identified before committing the changes to the repository. This may underestimate the effectiveness of test generation tools if these uncommitted changes are easier to detect. This threat is mitigated somewhat as the bugs are taken from all stages of the project history, including earlier stages of development.

## III. DO AUTOMATED UNIT TEST GENERATION TOOLS FIND REAL BUGS?

### A. How many usable tests are generated?

The left-hand side of Table I reports the outcome of the test generation. Unlike AGITARONE, both EVOSUITE and RANDOOP generated test classes that did not compile. Unsurprisingly, RANDOOP generated the largest number of tests for all projects as it does not target a specific class under test. In contrast, EVOSUITE and AGITARONE generate tests specifically for the selected classes under test, resulting in substantially fewer tests. For reference, we also include the number of tests in the developer-written test suites (manual). Note that these numbers refer to all relevant tests, as reported by Defects4J: A test is considered relevant if it directly or indirectly covers any of the classes relevant to the fault.

AGITARONE generated the lowest ratio of flaky tests overall, with a maximum of 3.3% for Time. As Time makes heavy use of the system time, this is not surprising. EVOSUITE suffered between 0.2%–5.4% flaky tests, but interestingly fewer for Time — presumably due to its built-in test isolation and check for flaky tests. Unlike AGITARONE and EVOSUITE, RANDOOP does not isolate tests from the environment, and as a result suffered between 5.7%–36.8% flaky tests. We mainly observed false positives for AGITARONE, in particular for Closure, due to the use of mocking and reflection to increase code coverage.

Out of the three test generation tools, AGITARONE generally achieved the highest code coverage ratio, except for Lang.

EvoSuite and Randoop struggled to achieve code coverage on Closure — most likely because of the large number of private methods. In comparison to the developer-written test suites, the test generation tools achieved lower code coverage overall, except for Chart. Yet, all tools achieved higher code coverage than developer-written test suites for some classes.

> EvoSuite *and* Randoop *generated 3.4% and 8.3% non-compilable test suites on average. Moreover, on average, 21% of* Randoop*'s tests were flaky, and 46% of* AgitarOne*'s failing tests were false positives.*

### B. How many bugs are found?

Overall, the generated test suites found 199 out of the 357 bugs (55.7%). On the face of it, finding more than half of the bugs sounds like an encouraging result. However, consider Table III, which gives a visual overview of the bug-finding results of the analyzed tools on the complete set of bugs: Found bugs are denoted with filled boxes, there is one row for each bug, and one column for each execution of a test generation tool. Clearly, the filled boxes are sparse in this table as only 19.9% of all executions detected a bug.

Considering tools individually, EvoSuite, AgitarOne, and Randoop found 145, 130, and 93 bugs, respectively. That is, the number of bugs found by each tool is comparable — around one third of all bugs, which is already substantially less than the overall number of bugs found (199). However, as the sparsity of black boxes in Table III shows — even for bugs that were found — tools like Randoop and EvoSuite use randomized algorithms, so the properties of the generated tests differ for each run. Consequently, a bug may be found in one run, but not in the next. If we say a bug is "likely to be found" if it was found in more than half of the executions of the tool, the number of bugs found for Randoop and EvoSuite changes to 54 and 83, respectively. If we consider a bug as found only if *all* executions of a tool detected the bug, then these numbers decrease to a sobering 28 and 38.

Regarding the effectiveness of the tools per project, there are some distinct differences: The part of Table III for the Chart project is quite densely populated, whereas the Closure project seems to be generally more problematic for test generation tools. Table I summarizes the visual presentation of Table III in numbers, and confirms this intuition: For the Chart project, Randoop and AgitarOne found more than half of the bugs on average (only EvoSuite struggled and only detected 9.7 out of 26 bugs on average). For the Closure project, even AgitarOne discovered only 25 out of 133 bugs, and Randoop found just 2.2 bugs on average. The bug detection results for the Lang and Time project look similarly grim. The Math project seems to be slightly better suited for test generation, with AgitarOne finding half of the bugs, and EvoSuite coming close to this result.

The picture painted overall is that if one wants to find all bugs, one should not rely solely on an automated unit test generation tool. This suggests that plenty remains to be done to improve automated test generation tools. Nevertheless, the fact that 199 bugs were found fully automatically does showcase the potential of such tools for widely used practices such as regression testing. The results also show that none of the test generation approaches is strictly superior to the other two.

> *Automated test generation tools found 55.7% of the bugs we considered, but no tool alone found more than 40.6%.*

TABLE II: The percentage of detected bugs, categorized by whether the bug in question was found by the developer-written test suite with either an assertion, an exception, or both.

| Project | Tool | Assertions | Exceptions | Both |
|---|---|---|---|---|
| Chart | AgitarOne | 64.3% | 55.6% | 100.0% |
| | EvoSuite | 57.1% | 77.8% | 100.0% |
| | Randoop | 57.1% | 77.8% | 100.0% |
| Closure | AgitarOne | 18.7% | 16.7% | 25.0% |
| | EvoSuite | 17.6% | 30.0% | 16.7% |
| | Randoop | 3.3% | 16.7% | 8.3% |
| Lang | AgitarOne | 31.0% | 35.3% | 50.0% |
| | EvoSuite | 20.7% | 29.4% | 100.0% |
| | Randoop | 6.9% | 23.5% | 0.0% |
| Math | AgitarOne | 42.7% | 70.0% | 0.0% |
| | EvoSuite | 56.0% | 80.0% | 0.0% |
| | Randoop | 34.7% | 50.0% | 0.0% |
| Time | AgitarOne | 30.0% | 71.4% | 0.0% |
| | EvoSuite | 80.0% | 42.9% | 66.7% |
| | Randoop | 40.0% | 64.3% | 66.7% |

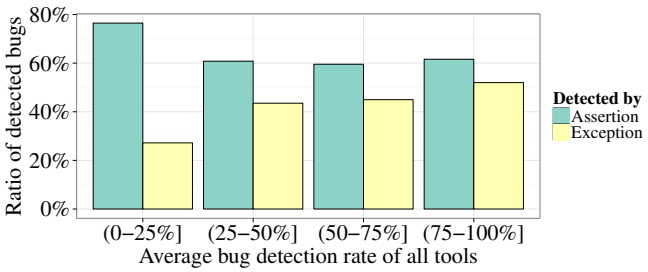



Fig. 2: Ratio of bugs that were detected by an assertion or an exception, grouped by the intervals of the average bug detection rate of all tools. The total number of bugs is 199 and 46, 77, 46, and 30, respectively for each interval. Note that a bug may be detected by both an assertion and an exception.

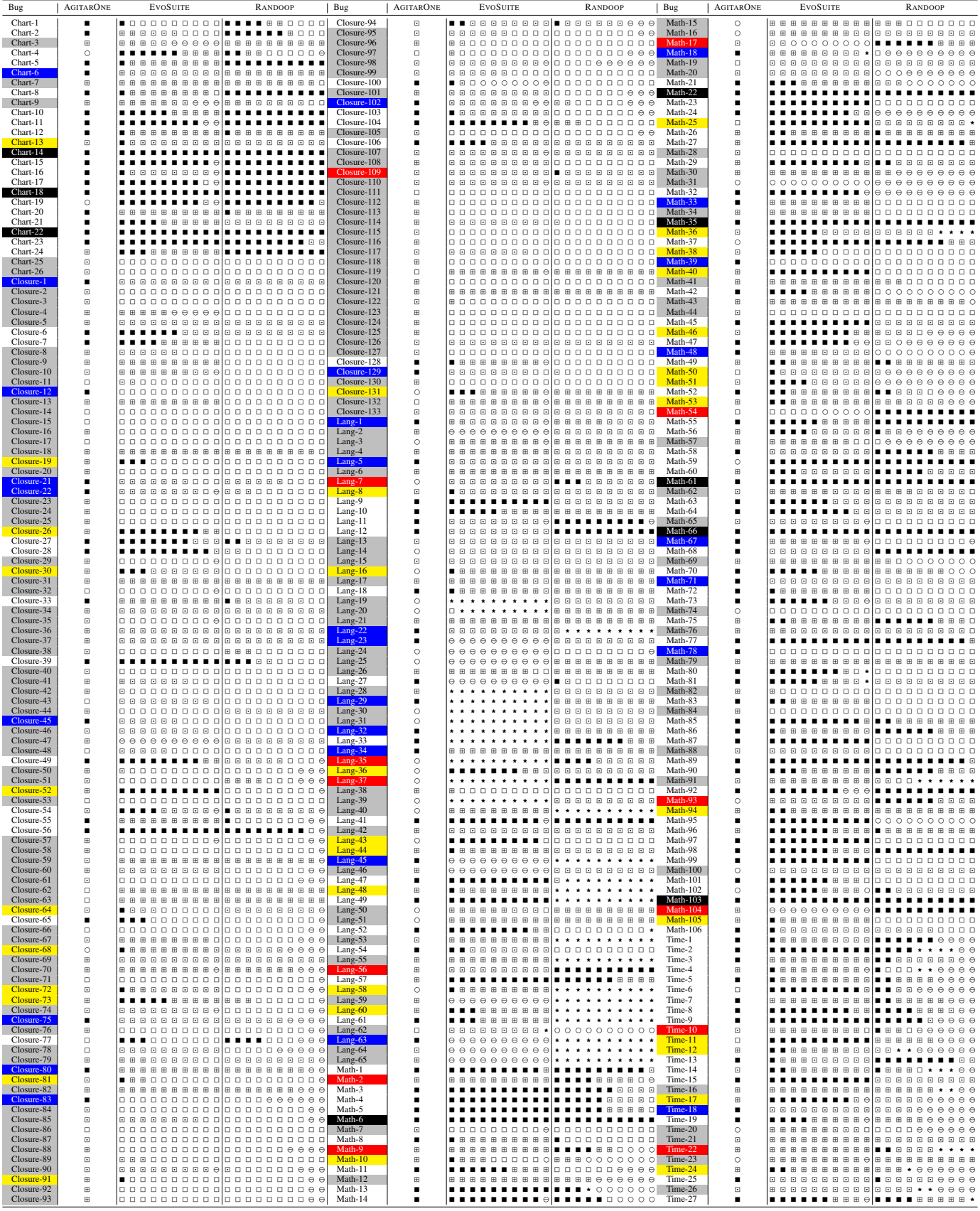
### C. How are the bugs found?

Overall, the generated test suites detected more bugs with an assertion (146) than with an exception (109). Note that 56 bugs were detected with both an assertion and an exception, by different tests. EvoSuite is the only tool that detected 5 bugs via timeout; we treat this as a case of exception for the following discussion. The right-hand side of Table I details the results and shows how effective assertions and exceptions are for each project.

Assuming that a developer-written test that reveals a bug is indicative of whether an assertion or an exception is required for that bug, we can determine if the tools are better at detecting bugs requiring an assertion or an exception. Table II shows how many of the bugs (detected by the developer-written test suites with an assertion or an exception) were detected by the generated test suites. In the majority of cases, clearly more of the bugs that trigger an exception were found than of those that require an assertion. The main exception is EvoSuite for the Time project, where 80% of the bugs requiring an assertion were found, compared to only 42.9% of the bugs requiring an exception.

Figure 2 shows the ratio of bugs detected by assertion or exception for different intervals of the average bug detection rate of all tools. Note that the average bug detection rate is computed as the mean across tools rather than the mean across

TABLE III: Summary of bug-finding results for each bug, tool, and execution. For each tool execution, if a compilable test suite was generated, a bug may have been detected (■), fully covered but not detected (⊞), partially covered but not detected (⊡), or not covered at all (□). Otherwise, the test suite was not compilable (⋆), resulted in configuration errors in our evaluation framework (⊖), or it was not even generated (○). Bugs that were either never found, always found, or found by only one tool, are highlighted with different colors (no tool, all tools, only AGITARONE, only EVOSUITE, or only RANDOOP).



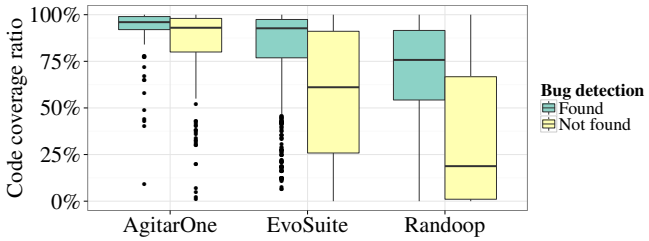



Fig. 3: Code coverage ratios for generated test suites that found a bug and generated test suites that did not. The differences are significant for all tools (Mann-Whitney U test, $p < 0.001$).

TABLE IV: Bug coverage of test suites that did not detect the bug. For each project and tool, *Total* gives the number of bugs that were not always detected by all generated test suites.

| Project | Tool | Total | Not | Partially | Full |
|---|---|---|---|---|---|
| Chart | AGITARONE | 7 | 28.6% | 28.6% | 42.9% |
| | EVOSUITE | 22 | 36.0% | 18.6% | 45.4% |
| | RANDOOP | 15 | 29.8% | 30.0% | 40.2% |
| Closure | AGITARONE | 47 | 44.7% | 31.9% | 23.4% |
| | EVOSUITE | 130 | 50.0% | 30.7% | 19.3% |
| | RANDOOP | 133 | 73.2% | 17.6% | 9.2% |
| Lang | AGITARONE | 30 | 66.7% | 0.0% | 33.3% |
| | EVOSUITE | 61 | 42.0% | 23.7% | 34.4% |
| | RANDOOP | 61 | 37.2% | 34.1% | 28.7% |
| Math | AGITARONE | 46 | 28.3% | 21.7% | 50.0% |
| | EVOSUITE | 83 | 23.4% | 27.2% | 49.3% |
| | RANDOOP | 93 | 54.8% | 25.4% | 19.8% |
| Time | AGITARONE | 11 | 27.3% | 36.4% | 36.4% |
| | EVOSUITE | 23 | 5.8% | 27.4% | 66.8% |
| | RANDOOP | 27 | 52.1% | 25.3% | 22.6% |

all test suites because the numbers of generated test suites differ for the tools. The plot suggests that hard to find bugs (i.e., bugs with a low average detection rate) are more often detected by an assertion than by an exception. Furthermore, bugs that are easier to find are more often detected by an exception than bugs that are harder to find.

*More bugs were detected with a test assertion than with an exception (146 vs. 109), but the detection ratio is lower for bugs requiring assertions (37.34% vs. 49.4% avg. per tool).*

### D. Are bugs that are covered usually found?

Figure 3 compares the code coverage ratios of generated test suites that detected a bug and generated test suites that did not, clearly showing that code coverage matters when it comes to detecting faults. This is also confirmed by a strong correlation between code coverage and bug detection (Pearson correlation of 0.40 on average per tool).

However, a high code coverage ratio does not necessarily indicate that the bug was covered. Table IV therefore reports the bug coverage (as described in Section II-C5) for all test suites that did *not* detect a bug. In general, fewer than 50% of the undetected bugs are fully covered by the test suites. A bug that is fully covered but not detected is indicative of a problem in generating an assertion since a test suite very likely detects a fully covered bug, even without an assertion, if that bug raises an exception. A notable exception in terms of fully covered bugs is given by EVOSUITE for the Time project, where 66.8% of the undetected bugs are fully covered. We surmise that in these cases EVOSUITE removed assertions while checking for unstable tests. Although full bug coverage is not sufficient to trigger the bug, the tools struggled to achieve full bug coverage for the majority of bugs.

*Of all test suites that did not reveal a fault, 46.8% did not fully cover it, and 26.6% did not even cover it partially.*

## IV. HOW CAN THE TOOLS BE IMPROVED?

Table III shows that the majority of the generated test suites did not detect the corresponding fault. The faults that were always detected by all generated test suites[6] are simple faults, such as a `NullPointerException` caused by a missing input validation, or easily executable and observable changes, such as Math-22:

```
public boolean isSupportLowerBoundInclusive() {
-   return true;
+   return false;
}
```

Note that in this and all following code snippets, '+' indicates an added line and '−' a removed line, in the fixed version.

However, most bugs are not this trivial. To gain insights on how to increase the fault detection rate of test generation tools, we investigated the challenges that prevent fault detection. To this end, we first looked at the 12 faults that no tool managed to cover (not even partially) and the 4 undetected faults that were fully covered. We then looked at the 76 faults that only one tool managed to detect — 28 by AGITARONE, 35 by EVOSUITE, and 13 by RANDOOP; these reveal strengths of a particular tool that others are missing. Finally, we studied the root causes for flaky and false-positive tests. The remainder of this section presents our findings.

### A. Improving Coverage

A test has to cover a fault to detect it, and Figure 3 shows that test suites that detect a fault achieve significantly higher code coverage. We identified four challenges that inhibit test generation tools from achieving such high code coverage.

*1) Creation of Complex Objects:* Out of the 12 faults[7] that were not covered by any of the generated test suites, the majority (9) of them are from the project Closure. For these 9 faults, reaching the buggy code requires the creation of a complex data structure (e.g., a control flow graph). This is a highly complex and challenging task for an automated test generation tool since it requires a certain sequence of method calls prior to exercising the target method. This task is also demanding for human testers, as evident in Closure's developer-written tests that detect the fault: The tests create a complex string (i.e., program text) and re-use the compiler infrastructure to create an appropriate data structure. For an automated test generation tool, however, generating such a complex input string is as challenging as initializing a complex data structure. Consider the following fault (Closure-14):

```
for (Node finallyNode : cfa.finallyMap.get(parent)) {
-    cfa.createEdge(fromNode, Branch.UNCOND, finallyNode);
+    cfa.createEdge(fromNode, Branch.ON_EX, finallyNode);
}
```

Covering this fault requires constructing a control flow graph in advance, with an edge connecting two consecutive `finally` blocks. The developer-written test achieves this by constructing a control flow graph object from a complex input string:

[6] Chart-{14, 18, 22}, and Math-{6, 22, 35, 61, 66, 103}

[7] Closure-{14, 15, 17, 32, 53, 66, 71, 86, 111}, Lang-24, and Math-31

```
String src = "X:while(1){try{while(2){try{var a;break X;}" +
      "finally{}}}finally{}}";
ControlFlowGraph<Node> cfg = createCfg(src);
assertCrossEdge(cfg,Token.BLOCK,Token.BLOCK,Branch.ON_EX);
```

In contrast, no coverage guidance exists for generating such complex strings within an automated test generation tool.

Some bugs requiring the creation of complex objects were found; for example, for Closure-80, an AGITARONE test created a mocked instance of a particular `Node`, and for Closure-109, a RANDOOP test managed to create a valid control flow graph from a complex input string, reusing a string constant from a dependent class. RANDOOP also succeeded in detecting several faults[8] by indirectly testing the class under test through dependencies. In these cases, RANDOOP overcame the object creation problem by exploiting existing logic in classes that are clients of the class under test.

Some viable solutions (which are not implemented in the used tools) exist to address the generally acknowledged problem of complex objects [43]. For example, seeding objects observed at runtime [23], mining of common usage patterns of objects [15] to guide object creation, or carving of complex object states from system tests [11]. However, in the absence of example information the problem is unsolved.

*2) String Optimization:* Complex strings not only occur in the developer-written test suites for Closure, but are also a recurring pattern in the generated test suites for faults that are only detected by one tool. AGITARONE detected the string-related fault in Closure-155, and EVOSUITE detected several faults[9] that require a specific input string. For example, consider Lang-16, a fault whose detection requires the satisfaction of the following condition:

```
if (str.startsWith("0x") || str.startsWith("-0x")
+     || str.startsWith("0X") || str.startsWith("-0X")) {
    return createInteger(str);
}
```

EVOSUITE generates strings using its genetic algorithm and seeded values [12], taken from string constants and runtime observations. The following test detects the fault above:

```
public void test085() throws Throwable {
    String string0 = "-0XeD";
    int int0 = NumberUtils.createNumber(string0);
    assertEquals((-237), int0);
}
```

Search-based tools are capable in principle of generating string inputs [2], but doing so with a search algorithm can take very long. Symbolic approaches using string solvers [18] or dedicated solvers for regular expressions [41] are generally restricted to fixed length strings. If an input grammar is known, then this can be used to generate test data more efficiently [8]. The results of web queries can also serve as useful test data [28]. Nevertheless, our experiments showed that state-of-the-art tools still struggle with string optimization.

*3) Complex Conditions:* Lang-24, a fault that no tool detected, is an example for a complex condition that needs to be satisfied to detect the fault:

```
if (chars[i] == 'l' || chars[i] == 'L') {
-   return foundDigit && !hasExp;
+   return foundDigit && !hasExp && !hasDecPoint;
}
```

Detecting this fault is challenging for two reasons: First, a randomly initialized character array (`chars`) is unlikely to satisfy the outer condition. Second, search-based tools like EVOSUITE suffer from boolean flags such as `foundDigit`, `hasExp`, and `hasDecPoint`, which provide no guidance to the search. This problem of boolean flags is well known, and testability transformation [20] is generally accepted as solution. Note that DSE would not suffer from this problem [17], [19].

Lang-48 a fault that only EVOSUITE detected, exemplifies a problem related to complex conditions involving subtyping:

```
public EqualsBuilder append(Object lhs, Object rhs) {
    ...
    Class lhsClass = lhs.getClass();
    if (!lhsClass.isArray()) {
-       isEquals = lhs.equals(rhs)
+       if (lhs instanceof java.math.BigDecimal) { ... }
+       else { isEquals = lhs.equals(rhs) }
    } ...
}
```

In principle, when creating an input value for a parameter of type `Object`, any class can be used. EVOSUITE addresses this challenge by explicitly using classes that are used in casts or type comparisons, and therefore generates a test case that passes an object of type `BigDecimal` as argument to the `append` method.

*4) Private Methods/Fields:* Many of the faults in the Closure project, in particular those not detected by any tool, exist in private methods. This presents an additional challenge for an automated test generation tool, which usually tests using only the public interface of a class under test. For instance, in Closure-1, a simple change is introduced in a private method:

```
private void removeUnreferencedFunctionArgs(Scope fnScope) {
+   if (!removeGlobals) {
+     return;
+   }
Node function = fnScope.getRootNode();
```

It is, however, difficult to (indirectly) test this method due to the complex class hierarchy and data structures of this project.

AGITARONE tries to sidestep this problem by accessing private fields and methods. While it may not generally be desirable to explicitly call private methods, it enables AGITARONE to cover methods that are hard to reach through the public API of the class. Covering the `removeUnreferencedFunctionArgs` method, AGITARONE triggers a `NullPointerException` by setting the value of the private field `removeGlobals` to `false`, and passing in `null` as `fnScope`:

```
RemoveUnusedVars removeUnusedVars = (RemoveUnusedVars)
  Mockingbird.getProxyObject(RemoveUnusedVars.class, true);
setPrivateField(removeUnusedVars,
        "removeGlobals", Boolean.FALSE);
Mockingbird.enterTestMode(RemoveUnusedVars.class);
callPrivateMethod("com.google.javascript.jscomp.
     RemoveUnusedVars", "removeUnreferencedFunctionArgs",
     new Class[] {Scope.class}, removeUnusedVars, new Object
     [] {null});
```

Although the test *does* find the bug, it is unclear whether such a change could be triggered without modifying the state through private fields and methods. Similarly, AGITARONE detected Closure-{45, 83, 102} and Math-{18, 33, 78} by asserting the value of private fields using reflection. Whether or not accessing private methods and fields is a good approach is debatable but, as shown, it has the potential to reveal faults. However, it can cause false positives, as will be discussed in Section IV-D. This problem can only be overcome by improving test generation tools to achieve coverage of private methods fully through the public API.

[8]Math-{2, 9, 54}, Time-{10, 22}, and Lang-56

[9]Lang-{16, 36, 44, 44, 58, 60}, Time-24, and Closure-73

### B. Improving Propagation and Detection

Even if covered, a fault might not propagate or, if it does, the test oracle might not be able to detect the change in the outcome. Recall that a third of the undetected faults were fully covered and many more were partially covered (Table IV). This section details challenges in revealing those covered faults.

*1) Propagation:* Across all projects and test suites, we found five faults (Closure-{31, 70, 121} and Math-{12, 30}) that were always fully covered but never detected.

Unlike Closure-{31, 70, 121} and Math-30, Math-12 represents a unique case not directly related to propagation, where the fault is simply a forgotten implementation of the `Serializable` interface. This fault is trivially covered, but in order to detect it, a test would need to serialize an object of that class using an `ObjectOutputStream`.

Math-30 is an integer overflow error, which means that not only does the code need to be covered, but it also has to be executed with values that lead to an overflow. Closure-31, 70, and 121 have a potential influence on private members of the faulty class. However, these private members are complex objects, and a change to them can only be observed by involving the faulty object in further complex interactions, rather than simply writing an assertion on a return value of a public method. To some extent, this is the result of focusing on simple structural criteria such as branch coverage, rather than aiming to exercise more complex intra-class data flow dependencies [21].

*2) Assertion Generation:* Considering that more faults are detected with an exception, but the majority of faults require an assertion (Table II), a key challenge for test generation tools is generating adequate assertions (i.e., test oracles).

Assertions are typically generated based on observations of the public API [16], [30], [44] during execution. However, our experiments revealed some particular cases where only AGITARONE was able to generate the appropriate assertions: For Chart-6, Closure-{12, 21, 22, 129}, and Math-48, AGITARONE detected the fault by asserting on the object state in the catch clause. In contrast, EVOSUITE and RANDOOP only verify that an expected exception is thrown. For example, Closure-129 is a fault for which both the buggy and the fixed version throw a `NullPointerException`, but differ in *where* it is thrown. AGITARONE detected this change as follows:

```
try {
    prepareAnnotations.visit(t, n, parent);
    fail("Expected NullPointerException to be thrown");
} catch (NullPointerException ex) {
    ...
    assertThrownBy(PrepareAst.PrepareAnnotations.class, ex);
}
```

### C. Flaky Tests

It is important for regression tests to be deterministic and to produce the same outcome in consecutive runs. Recall that we automatically removed 15.2% of flaky tests to achieve this goal. However, removing such a large portion of tests also means losing any additional coverage gained by such tests.

*1) Environment Dependencies:* Flaky tests are frequently caused by environmental dependencies of the software under test, such as the current time/date of the system. This problem is particularly frequent for Time, but also occurs in the other projects. For example, detecting the fault in Lang-8 requires a call to the method `format` of the `FastDatePrinter` class, which takes a `Calendar` as input — by default, a `Calendar` will be initialized to the current time.

EVOSUITE addresses this problem by using a mocked version of the concrete implementations [5]. This means that if the program accesses the current time on the system, then EVOSUITE provides a mocked time, so that any assertion that depends on the time value will deterministically pass or fail when executed at a later time. The following gives an example for a test, generated with EVOSUITE, which uses a mocked version of the concrete class `GregorianCalendar`, `MockGregorianCalendar`:

```
1 String string0 = "Z,~jsZ/7'{p!wd";
2 int int0 = 0;
3 SimpleTimeZone simpleTimeZone0 = new SimpleTimeZone(int0,
      string0);
4 Locale locale0 = Locale.GERMAN;
5 String string1 = "*z";
6 FastDatePrinter fastDatePrinter0 = new FastDatePrinter(
      string1, simpleTimeZone0, locale0);
7 MockGregorianCalendar mockGregorianCalendar0 = new
      MockGregorianCalendar(locale0);
8 String string2 = fastDatePrinter0.format((Calendar)
      mockGregorianCalendar0);
9 assertEquals("*GMT", string2);
```

In contrast, RANDOOP does not use mocking, and for the 10 runs on the same fault, it generated 84% flaky tests. AGITARONE also applies mocking, but was not able to detect this fault, which additionally requires a specific constraint to hold: the time zone represented by the `SimpleTimeZone` instance (line 3) must differ from the time zone used in the `Calendar` instance (line 7).

The faults Time-12, Time-14, and Lang-65 pose similar challenges to the test generation tools. Note that the problem of environment dependencies is not restricted to objects generated explicitly by the test generator, as other classes may refer to the system time or other external resources directly. EVOSUITE tries to overcome this problem using bytecode instrumentation, such that the environment dependencies can be controlled when directly accessed by the code under test.

*2) Static State:* A local dependency on the static state of the system under test can also result in flaky tests, such that any changes to the state with one test can affect the outcome of the remaining tests. EVOSUITE explicitly tracks changes to static variables, and resets the static state before test execution [5], and as a result was the only tool to detect Time-11, where class `ZoneInfoCompiler` uses a static variable `cVerbose`. The issue of static state has been raised in the context of test generation previously [10], and has recently also been verified in the context of manually written test suites [7], [27], [46].

### D. False Positives

During our evaluation, in particular during the validation of the test results, we encountered a number of false positives. The RANDOOP tests contained a few false positives due to non-deterministic failures unrelated to the fault. In particular, these tests caused an `IllegalStateException` by manipulating the threading behavior in Closure. While these tests are technically flaky tests, their likelihood of failing is very low, explaining why they never failed on the original version when checking for flaky tests. The majority of false positives observed are caused by AGITARONE's access of *private fields/methods/classes* through Java reflection, which breaks object-oriented principles such as *encapsulation*, and AGITARONE's use of *aggressive mocking*,

*1) Accessing Private Fields/Methods:* To maximize coverage, AGITARONE uses Java reflection to access the private API of the class under test. However, developers may add,

remove, or change private fields or methods to improve code quality or optimize the existing implementation. These changes do not affect any client of the class under test as its public API remains unchanged. Detecting regressions that are purely related to the private API therefore increases the likelihood of false positive test results.

Closure-3 is an example of a change of a private method, including its signature:

```
- private boolean canInline() {
+ private boolean canInline(final Scope scope) {
     ...
+    case Token.NAME:
+    Var var = scope.getOwnSlot(input.getString());
+    if (var != null && var.getParentNode().isCatch()) {
+       return true;
+    } ...
```

While AGITARONE generated a test that detects this change with a `NoSuchMethodException`, the test does not fail because of the root cause — that is, the test does not cover the `Token.NAME` case and would therefore pass if the method signature would remain unchanged. Overall, AGITARONE suffered 26 (12%) false positives caused by the use of reflection.

*2) Aggressive Mocking:* Another example for breaking encapsulation is AGITARONE's *aggressive mocking*, which monitors and asserts on the internal state (e.g., the order of method calls) of the class under test, rather than testing the class on what its public method returns, and on the side effects it has on its input parameters once its methods have completed to execute. How such input objects are manipulated is an internal detail that is not part of the public interface specification. As such, it can change without modifying the semantics of the methods. Consider the following example:

```
public int sum(Foo foo){
-   return foo.getX() + foo.getY();
+   return foo.getY() + foo.getX();
}
```

If `getY` and `getX` are pure (i.e., no side effects), then that function can be refactored as shown, and the order in which the function `sum` calls the methods in `Foo` is irrelevant. However, an aggressive mocking strategy would check the order in which the mocks are used, and fail if a different one is encountered.

For example, consider Closure-5, where the developers added a check to handle the special case of a deleted property:

```
+   if (gramps.isDelProp()) {
+       return false;
+   }
```

A valid way to detect this bug would be by using an assertion on the return value. However, with AGITARONE's aggressive mocking, adding such a method call would lead to a failure without even evaluating `isDelProp`, as the method is unexpected and thus triggers a `TestException`, indicating an error originating in AGITARONE's mocking framework. Similarly, deleting the method call would trigger a `TestException` for any subsequent method call on the same object — in neither case would the actual return value be considered.

For 67 faults, AGITARONE generated a test that failed with a `TestException` and a failure message referring to an "unexpected method". Such a mocking error may be a true positive if there is a specification on the order of method calls when communicating with external classes/resources. However, manual investigation of the bug descriptions of these 67 faults suggests that none of them are related to such a specification. We therefore consider each test that fails due to a `TestException` as a false positive caused by mocking.

Applying this interpretation, 31% of all test suites generated by AGITARONE suffered from false positives due to the use of aggressive mocking (10 of these test suites additionally include false positives due to accessing private methods/fields with reflection). However, note that this may be an over-approximation because a mocking exception could be caused by a state change induced by the bug.

## V. RELATED WORK

The most closely related work to ours is that of Xiao et al. [43], who identify two main problems when aiming to achieve high code coverage with generated tests, *external method calls* and *complex object creation*. The external method call problem is related to the environment dependencies issue (Section IV-C1) that we saw in terms of relation to date and time. Just like EVOSUITE uses mocking to overcome dependencies on time or files [5], this is also possible, for example, for database applications [37]. We also saw several instances of the complex object creation problem in the faults we analyzed. Xiao et al. [43] propose a collaborative approach between the tools and the developers, such that the underlying coverage problems is reported back to the developer to provide further guidance to the tool. The aim of our analysis is to identify problems that prevent automated test generation tools from finding faults rather than just covering code, with the hope to improve these tools in the future to find more faults.

There have been studies of the effectiveness of various software defect detection techniques, e.g., [29], [34], [42]. Generally, these studies showed that different techniques are complementary and dependent on the underlying faults; we saw similar results in our study: The individual performance of each of the tools in our study lies beneath the potential of combining all the tools.

## VI. CONCLUSIONS

Automated unit test generation tools are typically evaluated in terms of the code coverage that they can achieve on open source software projects. This paper contributes a systematic study of the fault detection potential of the generated test suites, using three state-of-the-art test generation tools and the Defects4J dataset. The results show that 1) the test generation tools find 55.7% of faults, but no tool alone finds more than 40.6% of faults. 2) Achieving code coverage remains a problem: 16.2% of the faults were never even executed by the generated tests, and 26.6% of the test suites that did not reveal a fault did not even cover the fault partially; however, 3) 63.3% of the non-found faults *were* covered by automatically generated tests at least once, suggesting problems that go beyond code coverage. One specific issue is that 15.2% of all tests were flaky. In order to guide future research on automated unit test generation, we investigated the challenges that need to be addressed in order to improve fault detection, and our qualitative analysis of difficult to find faults reveals specific challenges that prevent tools from achieving the required code coverage and generation of test oracles. We hope that our insights will lead to future research to address these challenges, and our data will serve to evaluate the progress of research on automated unit test generation.

## ACKNOWLEDGMENTS

This work is supported the EPSRC project "EXOGEN" (EP/K030353/1) and the National Research Fund, Luxembourg (FNR/P10/03).

## REFERENCES


[1] Agitar One (2014), www.agitar.com/developers/junit_factory.html, Last visited on 01.08.2014

[2] Alshraideh, M., Bottaci, L.: Search-based software test data generation for string data using program-specific search operators. Software Testing, Verification and Reliability (STVR) 16(3), 175–203 (2006)

[3] Andrews, J.H., Briand, L.C., Labiche, Y.: Is mutation an appropriate tool for testing experiments? In: Proc. of the Int. Conference on Software Engineering (ICSE). pp. 402–411. IEEE (2005)

[4] Andrews, J.H., Li, F.C., Menzies, T.: Nighthawk: A two-level genetic-random unit test data generator. In: Proc. of the Int. Conference on Automated Software Engineering (ASE). pp. 144–153. ACM (2007)

[5] Arcuri, A., Fraser, G., Galeotti, J.P.: Automated unit test generation for classes with environment dependencies. In: Proc. of the Int. Conference on Automated Software Engineering (ASE). pp. 79–90. ACM (2014)

[6] Baresi, L., Lanzi, P.L., Miraz, M.: Testful: an evolutionary test approach for java. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 185–194. IEEE (2010)

[7] Bell, J., Kaiser, G.: Unit test virtualization with VMVM. In: Proc. of the Int. Conference on Software Engineering (ICSE). pp. 550–561. ACM (2014)

[8] Beyene, M., Andrews, J.H.: Generating string test data for code coverage. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 270–279. IEEE (2012)

[9] Analytix CodePro (2014), developers.google.com/java-dev-tools/codepro/doc/, Last visited on 01.08.2014

[10] Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for java. Software: Practice and Experience (SP&E) 34(11), 1025–1050 (2004)

[11] Elbaum, S., Chin, H.N., Dwyer, M.B., Dokulil, J.: Carving differential unit test cases from system test cases. In: Proc. of the Symposium on the Foundations of Software Engineering (FSE). pp. 253–264. ACM (2006)

[12] Fraser, G., Arcuri, A.: The seed is strong: Seeding strategies in search-based software testing. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 121–130. IEEE (2012)

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

[14] Fraser, G., Staats, M., McMinn, P., Arcuri, A., Padberg, F.: Does automated white-box test generation really help software testers? In: Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA). pp. 291–301. ACM (2013)

[15] Fraser, G., Zeller, A.: Exploiting common object usage in test case generation. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 80–89. IEEE (2011)

[16] Fraser, G., Zeller, A.: Mutation-driven generation of unit tests and oracles. IEEE Transactions on Software Engineering (TSE) 38(2), 278–292 (2012)

[17] Galeotti, J.P., Fraser, G., Arcuri, A.: Improving search-based test suite generation with dynamic symbolic execution. In: Int. Conference on Software Reliability Engineering (ISSRE). pp. 360–369. IEEE (2013)

[18] Ganesh, V., Kieżun, A., Artzi, S., Guo, P.J., Hooimeijer, P., Ernst, M.: Hampi: A string solver for testing, analysis and vulnerability detection. In: Proc. of the Int. Conference on Computer Aided Verification (CAV). pp. 1–19. Springer (2011)

[19] Godefroid, P., Klarlund, N., Sen, K.: Dart: directed automated random testing. ACM Sigplan Notices 40(6), 213–223 (2005)

[20] Harman, M., Hu, L., Hierons, R., Wegener, J., Sthamer, H., Baresel, A., Roper, M.: Testability transformation. IEEE Transactions on Software Engineering (TSE) 30(1), 3–16 (2004)

[21] Harrold, M.J., Rothermel, G.: Performing data flow testing on classes. ACM SIGSOFT Software Engineering Notes 19(5), 154–163 (1994)

[22] Islam, M., Csallner, C.: Dsc+Mock: A test case + mock class generator in support of coding against interfaces. In: Int. Workshop on Dynamic Analysis (WODA). pp. 26–31. ACM (2010)

[23] Jaygarl, H., Kim, S., Xie, T., Chang, C.K.: OCAT: object capture-based automated testing. In: Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA). pp. 159–170. ACM (2010)

[24] Parasoft JTest (2014), www.parasoft.com/jtest, Last visited on 01.08.2014

[25] Just, R., Jalali, D., Ernst, M.D.: Defects4J: A database of existing faults to enable controlled testing studies for java programs. In: Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA). pp. 437–440. ACM (2014)

[26] Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G.: Are mutants a valid substitute for real faults in software testing? In: Proc. of the Symposium on the Foundations of Software Engineering (FSE). pp. 654–665. ACM (2014)

[27] Luo, Q., Hariri, F., Eloussi, L., Marinov, D.: An empirical analysis of flaky tests. In: Proc. of the Symposium on the Foundations of Software Engineering (FSE). pp. 643–653. ACM (2014)

[28] McMinn, P., Shahbaz, M., Stevenson, M.: Search-based test input generation for string data types using the results of web queries. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 141–150. IEEE (2012)

[29] Mouchawrab, S., Briand, L.C., Labiche, Y., Di Penta, M.: Assessing, comparing, and combining state machine-based testing and structural testing: a series of experiments. IEEE Transactions on Software Engineering (TSE) 37(2), 161–187 (2011)

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

[31] Park, S., Hossain, B.M.M., Hussain, I., Csallner, C., Grechanik, M., Taneja, K., Fu, C., Xie, Q.: CarFast: achieving higher statement coverage faster. In: Proc. of the Symposium on the Foundations of Software Engineering (FSE). pp. 35:1–35:11. ACM (2012)

[32] Păsăreanu, C.S., Rungta, N.: Symbolic PathFinder: symbolic execution of Java bytecode. In: Proc. of the Int. Conference on Automated Software Engineering (ASE). pp. 179–180. ACM (2010)

[33] Prasetya, I.W.B.: T3, a combinator-based random testing tool for java: benchmarking. In: Future Internet Testing, pp. 101–110. Springer (2014)

[34] Runeson, P., Andersson, C., Thelin, T., Andrews, A., Berling, T.: What do we know about defect detection methods? Software, IEEE 23(3), 82–90 (2006)

[35] Savonia, A., Evans, B.: *Crap4J* URL: http://www.crap4j.org/ (2014), Last visited on 19.01.2015

[36] Sen, K., Agha, G.: CUTE and jCUTE: Concolic unit testing and explicit path model-checking tools. In: Proc. of the Int. Conference on Computer Aided Verification (CAV). pp. 419–423. Springer (2006)

[37] Taneja, K., Zhang, Y., Xie, T.: Moda: Automated test generation for database applications via mock objects. In: Proc. of the Int. Conference on Automated Software Engineering (ASE). pp. 289–292. ACM (2010)

[38] Thummalapenta, S., Xie, T., Tillmann, N., De Halleux, J., Su, Z.: Synthesizing method sequences for high-coverage testing. ACM SIGPLAN Notices 46(10), 189–206 (2011)

[39] Tillmann, N., De Halleux, J.: Pex–white box test generation for .NET. In: Tests and Proofs, pp. 134–153. Springer (2008)

[40] Tonella, P.: Evolutionary testing of classes. In: Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA). pp. 119–128. ACM (2004)

[41] Veanes, M., De Halleux, P., Tillmann, N.: Rex: Symbolic regular expression explorer. In: Proc. of the Int. Conference on Software Testing, Verification and Validation (ICST). pp. 498–507. IEEE (2010)

[42] Wood, M., Roper, M., Brooks, A., Miller, J.: Comparing and combining software defect detection techniques: a replicated empirical study. ACM SIGSOFT Software Engineering Notes 22(6), 262–277 (1997)

[43] Xiao, X., Xie, T., Tillmann, N., De Halleux, J.: Precise identification of problems for structural test generation. In: Proc. of the Int. Conference on Software Engineering (ICSE). pp. 611–620. ACM (2011)

[44] Xie, T.: Augmenting automatically generated unit-test suites with regression oracle checking. In: European Conference on Object-Oriented Programming (ECOOP), pp. 380–403. Springer (2006)

[45] Xie, T., Marinov, D., Schulte, W., Notkin, D.: Symstra: A framework for generating object-oriented unit tests using symbolic execution. In: Int. Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS), pp. 365–381. Springer (2005)

[46] Zhang, S., Jalali, D., Wuttke, J., Muşlu, K., Lam, W., Ernst, M.D., Notkin, D.: Empirically revisiting the test independence assumption. In: Proc. of the Int. Symposium on Software Testing and Analysis (ISSTA). pp. 385–396. ACM (2014)