An Empirical Comparison of EvoSuite and DSpot for Improving Developer-Written Test Suites with Respect to Mutation Score

Muhammad Firhard Roslan (✉), José Miguel Rojas, and Phil McMinn

University of Sheffield, UK
mfroslan2@sheffield.ac.uk

Abstract. Since software faults are usually unknown, researchers and developers rely on mutation analysis — i.e., seeding artificial defects, called mutants — to measure the quality of their test suites. One aim of *test amplification techniques* is to improve developer-written test cases so that they kill more mutants and potentially find more real faults. However, these tools tend to be limited in the types of changes and improvements they can make to tests, while also receiving little guidance to tests that kill new mutants. Alternatively, a tool like *EvoSuite* can generate tests with the benefit of detailed fitness information and have the benefit of more flexibility in terms of evolving a test's structure. However, the process is typically not based on developer-written tests, and consequently, the resulting test suites are less likely to be accepted by human developers. In this paper, we propose modifications to *EvoSuite*, in a technique we refer to as $EvoSuite_{Amp}$, which starts with developer-written tests as seeds, and then aims to evolve these tests in the direction of killing further mutants. We then empirically compare $EvoSuite_{Amp}$ with a state-of-the-art test amplification tool, *DSpot*, on 42 versions of 29 different classes from the *Defects4J* benchmark, using the original developer-written test suites for each class as the starting point for test generation. In total, $EvoSuite_{Amp}$ achieves a statistically better mutation score for 35 of these 42 versions when compared to *DSpot*.

Keywords: Search-Based Test Case Generation · Test Amplification · Mutation Analysis · Unit Testing.

1 Introduction

One of the challenges in software testing is deriving tests that are good at revealing faults [?]. But also, writing good tests manually is time-consuming, and some consider it to be a tedious task [?]. For this reason, there has been a lot of well-known work in automated test generation techniques, including in the search-based software engineering community [?].

A widely used automatic test generation tool for Java is *EvoSuite* [?], which generates JUnit tests. However, it has some limitations. Fundamentally, it cannot solve the oracle problem [?] — human testers need to check that the assertions it

generates are correct. Furthermore, the tests it generates do not typically involve human input, and require post-processing to make them more readable [?].

In contrast, *test amplification* explicitly aims to strengthen developer-written tests [?]. The aim is to generate a new version of the developer's test suite so that it covers more corner cases and is more effective at finding faults. Since the "amplified" test suite is based on the developers set of tests, it is likely more understandable and acceptable to them [?]. The current state-of-the-art test amplification tool, *DSpot* [?], utilizes developer-written tests to increase the number of mutants (artificially seeded defects [?]) that they kill. *DSpot* "amplifies" developer-written test cases by changing the values of literals in the tests, method calls, or by adding assertions. Test cases that kill more mutants and have fewer modifications are retained. However, test amplification tools themselves are subject to some limitations. Firstly, the types of changes they can make to tests are limited and not as flexible as *EvoSuite*'s evolution process. Furthermore, unlike search-based tools, they do not utilize fine-grained fitness information to guide them to new tests. Test cases generated by *DSpot* that do not kill new mutants, for example, will be discarded even if the test is actually "close" to killing a new mutant and could be usefully improved in future.

The aim of this paper is to evaluate a potential "best of both worlds" approach, in which we evaluate a version of *EvoSuite* that is capable of reading developer-written tests as a starting point for test case generation. Leveraging its ability to make more fundamental changes to the structure of a test case, it then evolves those tests with the benefit of fine-grained fitness information for killing new mutants. We then evaluate whether *EvoSuite*'s evolution and mutation analysis technique could have a better performance in terms of killing mutants when compared to *DSpot*'s amplification technique. The motivation behind this study is to understand how many more mutants *could* be killed by test amplification tools if the principles of test amplification were applied differently, in the flavor of a more flexible and more guided search-based style of approach.

We compare this modified *EvoSuite* version, which we refer to as $EvoSuite_{Amp}$, with *DSpot* using the developer-written tests in open source projects as the starting point for test suite generation — specifically, 42 different versions of 29 different Java classes in 7 different projects of *Defects4J* (v2.0.0) [?]. Our experiments reveal that $EvoSuite_{Amp}$ outperforms *DSpot* for 35 of the 42 Java class versions studied in terms of mutation score achieved. Over 30 repeated runs, $EvoSuite_{Amp}$ was further capable of killing more "unique" mutants that *DSpot* was not able to kill in any run for 36 of these 42 subjects. $EvoSuite_{Amp}$ and all the data collected is available in our replication package [?].

In summary, the contributions of this paper are as follows:

1. A new test improvement strategy that utilizes the flexibility of *EvoSuite*, $EvoSuite_{Amp}$, that evolves test cases and leverages fitness information for killing specific mutants (Section 3).
2. An empirical study with seven open-source projects comparing $EvoSuite_{Amp}$ with an existing state-of-the-art test amplification tool, *DSpot* (Section 4).
3. Results and analysis of the effectiveness of both tools in terms of mutation score and mutants uniquely killed by each tool (Section 5).

2 Background

Mutation Analysis. Mutation Analysis is a way to evaluate the quality of a test suite [?]. The idea is to make small artificial changes, known as mutants, that mimic the mistakes that programmers could make in a program. Mutation Analysis tools generate mutants by applying a set of rules, known as *mutation operators*, to the program. The program that contains the mutants is executed against the test suite, to assess the quality of the tests in it. If the result of running the mutated program is different from the original program, the mutant is considered *killed*, and if it is the same, the mutant is considered *alive* — indicating that the test suite needs some change/improvement to kill it. The proportion of mutants that are killed as a percentage of all the mutants seeded is known as the test suite's *mutation score*. A test suite that achieves a higher mutation score is generally considered better at detecting faults than one with a lower score [?].

Test Amplification. Unit test suites are usually written by developers manually. This is a common practice as developers who wrote the program have domain knowledge about the program [?]. A study by Grano et al. [?] shows that tests that have been written by a developer tend to be more readable than those automatically generated by a tool. However, the biggest challenge is to create a good test suite that can detect faults [?]. Test amplification, a technique that improves a test suite by utilizing the existing developer-written tests could improve a variety of goals, such as improving code coverage and mutation score [?]. The main distinction between test amplification and general automated test generation tools is that test amplifiers use existing developer-written tests as a starting point, that they aim to improve/"amplify".

Two main parts of the process of test amplification are *input amplification* and *assertion amplification*. Input amplification involves forming new test cases by changing values, literals, objects, or method calls in some original, developer-written test case. Assertion amplification involves adding new assertion statements to the test that verify the expected output of the amplified inputs. After amplifying the inputs and assertions, a test amplification tool will derive several new test cases from the developer-written tests. A test selector then selects the test cases that kill new mutants with the fewest modifications from the original developer-written tests they were based on.

DSpot [?] is a well-known test amplification tool that amplifies developer-written tests. It takes, as input, the developer-written tests and the class that will be tested. The amplifiers used in input amplification of *DSpot* are the changing of literal values and method calls (by duplicating calls, removing them, or adding new invocations). After the tool amplifies the inputs, it further changes a test case by adding new assertions. Finally, to select which test cases are to be kept, *DSpot* measures the test cases based on the criteria that are used by the test selector it is configured with. The default test selector of *DSpot* uses the *PITest* mutation analysis tool [?] (a configuration of *DSpot* we refer to as $DSpot_{Mut}$), which keeps test cases that kill mutants not killed by the original test suite, and the number of changes from the original test case (with smaller changes preferred over bigger ones). Another test selector option available on *DSpot* uses

the *JaCoCo*[1] coverage test selector (that we refer to as $DSpot_{Cov}$), which keeps test cases that increase code coverage and execute unique paths.

Search-Based Test Generation. *EvoSuite* is an automatic test generation for Java that uses genetic algorithms to generate a test suite [?] that has been evaluated on many open-source projects in terms of code coverage and detecting faults [?]. The default configuration of *EvoSuite* can produce a JUnit test suite that maximizes the code coverage for each class. However, it can also be configured to use a fitness function that aims to maximize the generated test suite's mutation score [?]. The fitness function that guides test generation towards strongly killing mutants is formulated using three different distance metrics. Firstly, it calculates the distance of the calling function on the test case if it does not contain the function of the mutated statement. Secondly, it calculates the distance to executing the mutant using the approach level and branch distance Finally, it calculates the mutation impact, where the mutants need to infect the state and could propagate to an observable state.

EvoSuite typically starts by generating a random initial population of tests that calls the class methods. However, this population can bee seeded using a technique called *carving* [?] that harvests sequences of statements from the test cases of an existing test suite. Assuming that developers have written some tests, *EvoSuite* can take those tests and execute them to collect all potential reusable objects. The objects will then be inserted as part of a newly created test case (initialization). However, there are two limitations of this technique. It needs the developer-written tests to be converted into a representation that could be used in the *EvoSuite* search algorithm, and all the assertions from the developer-written tests will be removed. This means that it does not preserve exactly the same format that is being written by a developer.

3 Modifications made to *EvoSuite* — $EvoSuite_{Amp}$

EvoSuite was originally designed to generate a test suite from scratch. In this study, we need *EvoSuite* to read developer-written tests, remove mutants that are killed by developer-written tests, and not to add new random test cases during the search. With this in mind, we made four different modifications to *EvoSuite*, which we refer to as $EvoSuite_{Amp}$, and are as follows:

1. Removing Killed Mutants by the Developer-Written Tests. We set the fitness criteria of $EvoSuite_{Amp}$ to both branch coverage and strong mutation testing. Before starting the evolution, we remove the goals that were met by the developer-written tests. This is to make sure that the search focuses on the goals that are not covered yet. As an example, the class under test will have mutant A, X, Y, and Z. If the developer-written test could kill mutant X, Y, and Z, the only criteria that it needs to meet is to kill mutant A only.
2. Seeding Developer-Written Tests into the Initial Population of the GA. The second modification we made is on the initial population of the test

[1] Available at: https://www.eclemma.org/jacoco/.

cases. The default behavior of *EvoSuite* is to randomly generate new test cases. Instead of randomly generating new test cases, we used the developer-written tests as the initial population of the search. This utilizes the developer's domain knowledge of the program. The initial population size on the *EvoSuite* is set to 50 individuals, but in our study, we changed the population size depending on the developer-written test suite size. This is all done by using the carving technique that has been implemented in *EvoSuite* [**?**], introduced in Section 2.

3. Tuning the Add New Random Test Case Rate to Zero. We tune the settings of the parameter values of the evolutionary algorithm responsible for generating the test suite. The default configuration of *EvoSuite* is to use crossover, mutation, and randomly add new test cases into the population. However, we change the rate of adding new random test cases to the population of test cases to zero. This change means that the developer-written tests are kept during evolution, without the addition of completely new, randomly generated tests. This is crucial for maintaining similarity of the generated tests to the original test suite, and keeping the test suite free of tests or part of tests that are completely new or alien to the original developer. We still allow modifications to inputs featuring in the tests, however, so that there is scope for improving the original tests to kill more mutants, and for tests to be recombined by the crossover operator. After a few generations, the fitness of all individual chromosomes improves, where it will stop if it meets all the criteria or if the search budget is exhausted. A study by Aniche et al. shows that developers tend to copy and paste from previous test methods and modify their name, inputs, and assertions [**?**]. This effect is simulated, in part, by crossover, with mutation focussed on modifying the developer-written test inputs only.

4. Turning Off Test Suite Minimization. We turned off the *EvoSuite* test suite post-process minimization feature in order to maintain the developer-written tests, else they may be discarded following test suite evolution.

4 Empirical Study

This section details the experiment design of the empirical study we conducted to assess $EvoSuite_{Amp}$, $DSpot_{Mut}$, and $DSpot_{Cov}$ with respect to killing mutants. We also include $DSpot_{Cov}$ into the experiment because the $EvoSuite_{Amp}$ fitness criteria includes branch coverage. In the following, we refer to $EvoSuite_{Amp}$ and *DSpot* as distinct "tools", while we breakdown the analysis of *DSpot* in terms of the two configurations $DSpot_{Mut}$ and $DSpot_{Cov}$ (Section 2 for more information). We designed our empirical study to answer the following four research questions:

RQ1: Which tool ($EvoSuite_{Amp}$ or *DSpot*) kills the most mutants?

RQ2: Which tool kills the most "unique" mutants (mutants not killed by the alternative tool)?

RQ3: Which tool kills the most mutants with the smallest test suites?

Table 1. Subject programs used in this study

Subject	Acronym	Lines of Code			Avg. # of Mutants	# of Unique Classes Evaluated	# of Versions Evaluated
		Min	Max	Avg.			
Commons-**Cli**	Cli	56	200	104	261	2	3
Commons-**Codec**	Cdc	162	355	242	253	3	4
Commons-**Compress**	Crs	92	370	205	182	5	5
Commons-**Csv**	Csv	105	1152	675	117	3	9
Jsoup	Jsp	85	280	193	48	5	7
Commons-**Lang**	Lng	52	1366	907	568	3	5
Commons-**Math**	Mth	148	1091	469	662	8	9
Total						29	42

RQ4: Which tool provides the most consistent results when re-run multiple times?

Subjects. We performed our experiment on the widely used benchmark *Defects4J* (v2.0.0) [**?**], which contains 835 reproducible real faults on 17 open-source projects. Although we are not specifically interested in the individual bugs provided by this benchmark, it provides us with an ideal set of subject classes and utilities with which we can evaluate the performance of both the $EvoSuite_{Amp}$ and *DSpot* tools. This includes an interface for test generation, which among other things help with removing flaky tests — tests that pass and fail without any changes to code [**?**]. It also incorporates the *Major* [**?**] mutation analysis tool, which we use as independent arbiter of the mutants killed by the test suites generated by both $EvoSuite_{Amp}$ and *DSpot* tools (since *DSpot* relies on *PITest* [**?**], while *EvoSuite* uses its own in-built mutation analysis).

We selected subject classes from *Defects4J* with which to perform our experiment based on the following rules:

1. The project includes developer-written tests;
2. $DSpot_{Mut}$, $DSpot_{Cov}$, and $EvoSuite_{Amp}$ were capable of using the provided original developer-written tests,
3. *Major* [**?**], *PITest* [**?**], and *EvoSuite* could generate mutants for the project.

After running every faulty version of each project in the *Defects4J* dataset, 42 faulty versions of 29 unique classes in 7 libraries met the requirements above. Table 1 shows the details of these subjects. We found a large number of *Defects4J*'s classes/versions to be unusable for our study due to an issue with *DSpot*'s interface with its mutation analysis tool *PITest* needed for the study, and problems compiling the class under test. We have contacted the owner of the *DSpot* project, and it could not be resolved to date. Despite this, our final subject set comprises a wide and diverse set of classes over a number of projects that are suitable for our study.

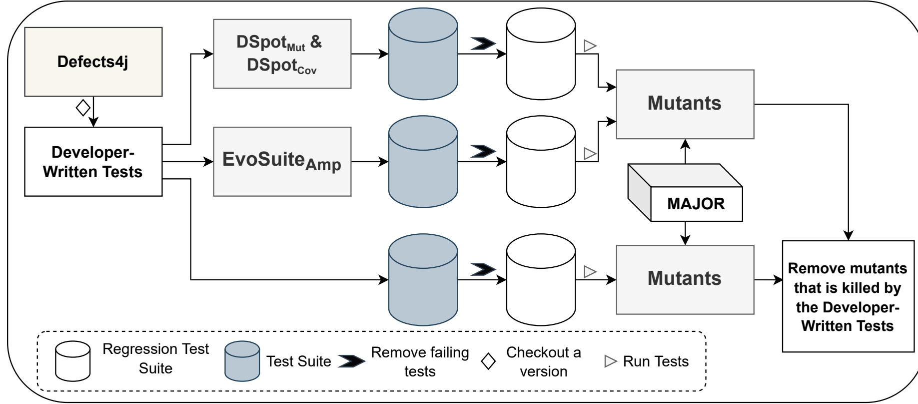

Fig. 1. Overview of the experimental setup. We amplified the test suite using $EvoSuite_{Amp}$, $DSpot_{Mut}$, and $DSpot_{Cov}$.

4.1 Experimental Procedure

Figure 1 shows the overview of our experiment. The tools are fed with developer-written tests that were gathered from the test files in every project version (with a particular fault) from *Defects4J*. For each version, as shown in Table 2, we improved each class of the study's original developer-written test suite (as provided by *Defects4J*) using $EvoSuite_{Amp}$ (using *EvoSuite* v1.2.0), and $DSpot_{Mut}$ and $DSpot_{Cov}$ (using v3.2.0 of *DSpot*). We ran all experiments on the same workstation, with 32GB RAM and Intel i5 CPU @ 3.10GHz, running Ubuntu 20.04.4 LTS. For both tools, we set the search budget time limit to 120 seconds, a commonly used value for test suite generation, and one that is applied in the search-based testing tool competition [**?**].

To take into account the non-deterministic nature of the tools, we repeat test suite generation 30 times for each tool/configuration studied. While we did not perform any internal modifications to *DSpot*— the build was downloaded from their repository[2] and configured to form $DSpot_{Mut}$ and $DSpot_{Cov}$. We made the modifications to *EvoSuite* to form $EvoSuite_{Amp}$ detailed in Section 3.

To make sure that there will not be any failing (*flaky*) tests generated by either tool, we used the *fix_test_suite* feature of *Defects4J* that removes failing tests from the test suite until all of them pass. Without removing the failing tests, flaky tests could interfere with the mutation score result. For all the developer-written and generated regression test suites, we used the *Major* mutation testing tool [**?**] to compute the mutation score. *Major* includes a summary of which mutants are killed by each test suite. The summary helps in finding the additional number of mutants that the generated test suites kill. We calculate the relative increase of mutation score for each automatically improved test suite, over the original developer-written version as:

$$\%IncreaseKilled = \frac{AverageMutantsKilled_{Amplified}}{TotalNumberOfMutants} \times 100$$

[2] Available at: https://github.com/STAMP-project/dspot.

Generated Test Suites. *EvoSuite*$_{Amp}$ and *DSpot* generate the test cases in a single test file. There are some cases where it has dependencies from other test files that developers wrote, such as a utility class. Without importing dependencies in *EvoSuite*$_{Amp}$ and *DSpot*, the improved test suite files will have compilation errors. For this reason, we made sure the improved test suite files always imported these test suite dependencies.

Handling of Mutation Analysis in the Experiment. In this experiment, we used strong mutation testing to evaluate the amplified tests. There were, in effect, three different mutation testing tools involved in the study. *EvoSuite* uses its own mutation analysis tool, while *DSpot* uses the *PITest* mutation analysis tool as part of its test amplification process. Since both *EvoSuite* and *DSpot* use different mutation analysis tools, it is not fair to compare the number of mutants it kills with different tools, which could produce different results for the same test suite. To avoid any bias in our study, we used a third mutation analysis tool, in the form of *Major* [**?**] to perform mutation analysis after both *EvoSuite* and *DSpot* generate the improved test suite. Since *Major* is *Defects4J*'s default mutation analysis tool, it was straightforward for us to apply this analysis.

Statistical Analysis. Since we are assessing algorithms that are making random choices, we analyzed the data that we collected using well-established statistical analysis recommendations [**?**]. We repeated each experiment 30 times. We then used the Mann-Whitney U-test to check for the significant differences regarding the number of mutants killed, comparing *EvoSuite*$_{Amp}$ with *DSpot*$_{Mut}$ improved test suites, and then *EvoSuite*$_{Amp}$ with *DSpot*$_{Cov}$ test suites, for each version of each subject class. We used the 99% confidence interval, which means that if the p-value is less than 0.01, our result is statistically significant. We further calculate effect sizes, using Vargha-Delaney's ($\hat{A}$) test. Again, we compared *EvoSuite*$_{Amp}$ with *DSpot*$_{Mut}$, and then *EvoSuite*$_{Amp}$ with *DSpot*$_{Cov}$. An $\hat{A}$ value that is over 0.5 indicates that *EvoSuite*$_{Amp}$ outperforms *DSpot*. Another statistical analysis that we performed was finding the correlation between the size of the test suite, and the mutation score. We used Spearman's rank correlation coefficient to find the relationship between the two variables. We used 99% confidence interval to indicate if the result is statistically significant.

4.2 Threats to Validity

Naturally, there are threats to validity associated with our study. The first is associated with subject selection. We chose to use versions of classes that are part of the *Defects4J* benchmark, yet not all of the classes it provides could be used in our study, due to problems in getting *DSpot* to work. However, we were able to use 42 versions of 29 unique classes in 7 projects, which still provides a suitable number and diversity of subjects with which to carry out our experiments and draw conclusions from the results. Another threat is related to how mutation score is calculated, since *EvoSuite* and *DSpot* use different mechanisms. *EvoSuite* provides its own implementation of a mutation analysis pipeline, while *DSpot* uses *PITest*. To control this threat, we used a third tool, *Major*, to

provide an unbiased assessment across the results of the two tools. To control the threats related to the non-deterministic behavior of both tools, we repeated our experiments 30 times. To mitigate the threats associated with our statistical analysis, and assumptions about the normality of the statistical distributions of our results, we used non-parametric statistical tests. Finally, after generating the test cases using both $EvoSuite_{Amp}$ and *DSpot*, there are some cases where it needs other test files to run, due to dependencies. This could have an impact when calculating the mutation score. To mitigate this problem, we ensured all improved test suites retained access to any dependent libraries and code.

5 Results

Answer to RQ1: Mutation Score. Table 2, part B, shows the mean of the mutants killed by $EvoSuite_{Amp}$, $DSpot_{Mut}$, and $DSpot_{Cov}$. The table further shows that $EvoSuite_{Amp}$ is more effective at killing mutants for 35 out of the 42 versions (83.3%) than $DSpot_{Mut}$. It is also better at killing mutants for 27 out of the 42 (64.3%) versions compared to $DSpot_{Cov}$. $EvoSuite_{Amp}$ is most effective at killing mutants with classes from the *Math* project. All the class versions that $EvoSuite_{Amp}$ achieves a better mutation score have a p-value less than 0.01. Where $DSpot_{Cov}$ and $DSpot_{Mut}$ achieve a better mutation score than $EvoSuite_{Amp}$, the p-value is less than 0.01. When using the $\hat{A}$ statistic to measure effect size, we found that $EvoSuite_{Amp}$ has a score that favours it over $DSpot_{Mut}$ in 35 out of the 42 projects and $DSpot_{Cov}$ in 25 out of 42 versions (59.5%), each time with an $\hat{A}$ value greater than 0.8 (i.e., a large effect size).

Conclusion for RQ1. $EvoSuite_{Amp}$ performs better than $DSpot_{Mut}$ and $DSpot_{Cov}$ in terms of killing mutants.

Answer to RQ2: Unique Mutants. We also evaluated the cumulative number of uniquely killed mutants after executing each of the 30 test suites on every tool. This means that mutants that are still alive after 30 runs are considered as either stubborn mutants or equivalent mutants. We found that $EvoSuite_{Amp}$ killed more unique mutants in 36 out of the 42 versions (85.7%) when compared to $DSpot_{Mut}$, and 27 out of the 42 versions (64.3%) when compared to $DSpot_{Cov}$. This and more detailed information regarding the performance of each tool on each class version can be seen in part D of Table 2.

Conclusion for RQ2. $EvoSuite_{Amp}$ kills more unique mutants after 30 runs when compared to $DSpot_{Mut}$ and $DSpot_{Cov}$.

Answer to RQ3: Size of Test Suite. Even though $EvoSuite_{Amp}$ can kill more mutants, it generates bigger test suites in general. When comparing $DSpot_{Mut}$ to $EvoSuite_{Amp}$, 39 out of the 42 class versions studied involved an improved test suite that is smaller number of lines of code (KLOC) when $DSpot_{Mut}$ was used, and when comparing $DSpot_{Cov}$ to $EvoSuite_{Amp}$, 26 out of the 42 class versions

Table 2. The result of test amplification on 42 versions after 30 runs for $EvoSuite_{Amp}$ (**Evo**), $DSpot_{Mut}$ (**DS**), and $DSpot_{Cov}$ (**DJ**).

(A) Fault Version	(B) # of Killed Mutants						(C) Standard Deviation			(D) # of Unique Mutants Killed			(E) Original TS Mutation Score %	(F) Increase Killed %			(G) # of KLOC		
	Mean			Median										Mean					
	Evo	DS	DJ	Evo	DS	DJ	Evo	DS	DJ	Evo	DS	DJ		Evo	DS	DJ	Evo	DS	DJ
Cdc-11	†18.2	16.5	20.0	†18.0	16.0	20.0	$^{\circ}$1.2	1.2	0.0	20	20	20	63.9	†21.9	19.9	24.1	*0.2	0.1	0.3
Cdc-16	$^{\dagger\circ}$90.5	2.0	7.0	$^{\dagger\circ}$90.5	2.0	7.0	$^{\dagger\circ}$23.3	1.0	0.0	$^{\dagger\circ}$139.0	3	7	50.6	$^{\dagger\circ}$9.9	0.2	0.8	$^{*\bullet}$0.7	<0.1	0.1
Cdc-17	$^{\dagger\circ}$7.3	1.0	2.0	$^{\dagger\circ}$7.5	1.0	2.0	$^{\dagger\circ}$1.4	0.0	0.0	$^{\dagger\circ}$9.0	1	2	43.5	$^{\dagger\circ}$31.6	4.3	8.7	$^{*\bullet}$0.3	0.2	0.2
Cdc-18	$^{\dagger\circ}$6.9	1.5	2.0	$^{\dagger\circ}$7.0	2.0	2.0	$^{\dagger\circ}$1.5	0.5	0.0	$^{\dagger\circ}$9.0	2	2	43.5	$^{\dagger\circ}$30.0	6.7	8.7	$^{*\bullet}$0.3	0.2	0.3
Cli-37	$^{\dagger\circ}$13.7	1.0	1.0	$^{\dagger\circ}$13.0	1.0	1.0	$^{\dagger\circ}$4.3	0.0	0.0	$^{\dagger\circ}$23.0	1	1	76.5	$^{\dagger\circ}$3.7	0.3	0.3	$^{*\bullet}$1.4	<0.1	<0.1
Cli-38	$^{\dagger\circ}$10.8	2.0	2.0	$^{\dagger\circ}$11.0	2.0	2.0	$^{\dagger\circ}$2.4	0.0	0.0	$^{\dagger\circ}$15.0	2	2	76.1	$^{\dagger\circ}$2.9	0.5	0.5	$^{*\bullet}$1.5	<0.1	<0.1
Cli-39	1.0	2.0	2.0	1.0	2.0	2.0	0.0	0.0	0.0	1	2	2	14.3	7.1	14.3	14.3	$^{*\bullet}$0.2	<0.1	0.2
Crs-34	$^{\dagger\circ}$39.0	35.0	35.0	$^{\dagger\circ}$40.0	35.0	35.0	$^{\dagger\circ}$3.0	0.0	0.0	$^{\dagger\circ}$44.0	35	35	48.2	$^{\dagger\circ}$34.8	31.2	31.2	$^{*\bullet}$0.4	0.2	0.2
Crs-39	$^{\dagger\circ}$57.5	36.0	36.0	$^{\dagger\circ}$58.5	36.0	36.0	$^{\dagger\circ}$4.0	0.0	0.0	$^{\dagger\circ}$62.0	36	36	32.1	$^{\dagger\circ}$51.3	32.1	32.1	$^{*\bullet}$0.8	<0.1	<0.1
Crs-40	†9.0	4.4	17.0	†9.0	4.0	17.0	$^{\circ}$2.1	3.1	0.0	†15.0	12	17	37.2	†0.9	0.5	1.8	$^{*\bullet}$0.2	<0.1	<0.1
Crs-44	12.1	15.3	17.0	12.5	14.0	17.0	$^{\dagger\circ}$3.1	1.5	0.0	$^{\dagger\circ}$18.0	17	17	0.0	52.6	66.5	73.9	$^{*\bullet}$0.2	<0.1	<0.1
Crs-45	$^{\dagger\circ}$108.4	56.3	63.0	$^{\dagger\circ}$110.0	57.0	63.0	$^{\dagger\circ}$14.6	1.3	0.0	$^{\dagger\circ}$132.0	57	63	54.2	$^{\dagger\circ}$18.7	9.7	10.9	$^{*\bullet}$0.8	0.2	0.2
Csv-01	$^{\dagger\circ}$6.7	2.1	3.0	$^{\dagger\circ}$6.5	3.0	3.0	$^{\dagger\circ}$2.9	1.1	0.0	$^{\dagger\circ}$16.0	3	3	41.7	$^{\dagger\circ}$8.0	2.5	3.6	*0.3	<0.1	0.5
Csv-02	$^{\dagger\circ}$9.6	2.0	7.0	$^{\dagger\circ}$12.0	2.0	7.0	$^{\dagger\circ}$3.3	0.0	0.0	$^{\dagger\circ}$12.0	2	7	36.8	$^{\dagger\circ}$50.5	10.5	36.8	$^{*\bullet}$0.3	0.1	0.1
Csv-04	†16.8	14.8	27.0	†17.0	15.0	27.0	$^{\dagger\circ}$1.4	1.0	0.0	†20.0	18	27	40.0	†24.0	21.2	38.6	*0.3	0.3	0.9
Csv-06	$^{\dagger\circ}$12.2	8.4	9.0	$^{\dagger\circ}$12.0	8.0	9.0	$^{\dagger\circ}$0.6	0.5	0.0	$^{\dagger\circ}$13.0	9	9	35.0	$^{\dagger\circ}$61.2	41.8	45.0	$^{*\bullet}$0.4	0.2	0.2
Csv-07	†14.7	12.9	23.0	†14.0	12.0	23.0	$^{\dagger\circ}$2.3	1.5	0.0	†19.0	16	23	47.1	†21.0	18.4	32.9	*0.3	0.2	1.0
Csv-10	16.2	56.0	108.0	14.5	54.5	108.0	$^{\circ}$6.3	10.3	0.2	33	75	109	28.2	5.7	19.7	38.0	$^{*\bullet}$0.6	0.2	0.5
Csv-11	†18.1	13.4	31.0	†18.0	13.0	31.0	$^{\circ}$2.0	2.1	0.0	†23.0	18	31	42.0	†22.3	16.5	38.3	*0.3	0.2	1.0
Csv-12	†31.7	0.0	108.0	†33.0	0.0	108.0	$^{\dagger\circ}$3.4	0.0	0.0	†36.0	0	108	50.3	†10.1	0.0	34.6	$^{*\bullet}$2.1	0.1	1.0
Csv-16	†22.4	12.2	46.0	†22.5	8.0	46.0	$^{\circ}$4.3	7.4	0.0	†31.0	26	46	36.8	†19.6	10.7	40.4	*0.8	0.3	1.3
Jsp-58	9.7	20.1	29.0	8.0	19.0	29.0	$^{\dagger\circ}$4.4	2.2	0.0	23	25	29	22.5	13.7	28.4	40.8	*0.1	<0.1	0.2
Jsp-69	†14.3	2.0	24.0	†15.0	2.0	24.0	$^{\dagger\circ}$2.8	0.0	0.0	†18.0	2	24	2.6	†37.6	5.3	63.2	*0.2	0.1	0.4
Jsp-79	$^{\dagger\circ}$2.3	0.0	0.0	$^{\dagger\circ}$2.0	0.0	0.0	$^{\dagger\circ}$0.5	0.0	0.0	$^{\dagger\circ}$4.0	0	0	53.8	$^{\dagger\circ}$9.0	0.0	0.0	*0.2	0.2	0.3
Jsp-80	36.6	46.0	49.0	35.0	46.0	49.0	$^{\dagger\circ}$4.4	2.2	0.0	45	49	49	11.1	45.1	56.8	60.5	$^{*\bullet}$0.3	<0.1	0.2
Jsp-84	16.0	26.0	27.0	16.0	26.0	27.0	$^{\dagger\circ}$2.2	0.0	0.0	20	26	27	0.0	38.2	61.9	64.3	$^{*\bullet}$0.2	0.1	0.1
Jsp-86	$^{\dagger\circ}$6.9	1.5	0.0	$^{\dagger\circ}$7.0	1.5	0.0	$^{\dagger\circ}$2.1	1.5	0.0	$^{\dagger\circ}$12.0	3	0	51.4	$^{\dagger\circ}$19.6	4.3	0.0	*0.2	<0.1	0.2
Jsp-93	†15.7	4.0	18.0	†16.0	4.0	18.0	$^{\dagger\circ}$2.6	0.0	0.0	$^{\dagger\circ}$19.0	4	18	2.5	†39.2	10.0	45.0	*0.3	0.2	0.4
Lng-03	†517.8	484.2	545.0	†517.5	485.0	545.0	$^{\dagger\circ}$12.2	8.9	0.0	†543.0	499	545	0.4	†58.2	54.5	61.3	$^{*\bullet}$2.2	1.2	1.6
Lng-04	$^{\circ}$0.8	1.0	0.0	$^{\circ}$1.0	1.0	0.0	$^{\dagger\circ}$0.5	0.0	0.0	$^{\dagger\circ}$2.0	1	0	82.9	$^{\circ}$2.0	2.4	0.0	$^{*\bullet}$0.3	<0.1	<0.1
Lng-05	$^{\dagger\circ}$104.0	5.0	8.0	$^{\dagger\circ}$104.0	5.0	8.0	$^{\dagger\circ}$3.1	0.0	0.0	$^{\dagger\circ}$112.0	5	8	0.0	$^{\dagger\circ}$73.8	3.5	5.7	$^{*\bullet}$0.4	0.2	0.2
Lng-07	$^{\dagger\circ}$530.8	406.0	492.0	$^{\dagger\circ}$530.5	407.5	492.0	$^{\circ}$11.0	13.3	0.0	$^{\dagger\circ}$554.0	449	492	0.4	$^{\dagger\circ}$59.3	45.4	55.0	$^{*\bullet}$2.4	1.0	1.5
Lng-16	$^{\dagger\circ}$520.7	416.2	490.0	$^{\dagger\circ}$520.0	415.0	490.0	$^{\circ}$10.9	12.8	0.0	$^{\dagger\circ}$545.0	445	490	0.5	$^{\dagger\circ}$59.6	47.7	56.1	$^{*\bullet}$2.3	1.1	1.5
Mth-09	$^{\dagger\circ}$26.2	20.0	20.0	$^{\dagger\circ}$26.0	20.0	20.0	$^{\dagger\circ}$3.4	0.0	0.0	$^{\dagger\circ}$32.0	20	20	51.6	$^{\dagger\circ}$28.8	22.0	22.0	0.4	0.9	1.7
Mth-25	$^{\dagger\circ}$112.9	41.9	82.0	†74.5	32.0	82.0	$^{\dagger\circ}$72.3	13.2	0.0	$^{\dagger\circ}$233.0	59	82	0.0	$^{\dagger\circ}$32.2	12.0	23.4	$^{*\bullet}$0.3	<0.1	0.1
Mth-26	$^{\dagger\circ}$157.0	51.4	65.0	$^{\dagger\circ}$157.0	51.0	65.0	$^{\dagger\circ}$4.2	0.7	0.0	$^{\dagger\circ}$168.0	53	65	45.6	$^{\dagger\circ}$33.3	10.9	13.8	*1.0	0.8	1.2
Mth-27	$^{\dagger\circ}$152.1	51.2	65.0	$^{\dagger\circ}$152.0	51.0	65.0	$^{\dagger\circ}$3.4	0.6	0.2	$^{\dagger\circ}$157.0	53	65	46.0	$^{\dagger\circ}$32.6	11.0	13.9	*1.1	0.8	1.1
Mth-36	$^{\dagger\circ}$85.6	54.0	71.0	$^{\dagger\circ}$86.0	54.0	71.0	$^{\dagger\circ}$10.7	0.0	0.0	$^{\dagger\circ}$101.0	54	71	48.4	$^{\dagger\circ}$23.3	14.7	19.3	1.5	2.2	2.8
Mth-52	$^{\dagger\circ}$1509.6	181.0	187.0	$^{\dagger\circ}$1497.0	181.0	187.0	$^{\dagger\circ}$244.0	0.0	0.0	$^{\dagger\circ}$1869.0	181	187	6.0	$^{\dagger\circ}$55.1	6.6	6.8	$^{*\bullet}$1.1	0.2	0.5
Mth-53	†244.1	88.3	307.0	†255.5	82.5	307.0	$^{\dagger\circ}$60.0	24.2	0.0	$^{\dagger\circ}$311.0	142	307	26.5	†46.5	16.8	58.5	1.4	3.1	13.1
Mth-55	$^{\dagger\circ}$536.2	266.0	266.0	$^{\dagger\circ}$542.0	266.0	266.0	$^{\dagger\circ}$24.3	0.0	0.0	$^{\dagger\circ}$567.0	266	266	19.0	$^{\dagger\circ}$68.5	34.0	34.0	$^{*\bullet}$1.5	0.8	1.0
Mth-56	$^{\dagger\circ}$89.8	45.0	47.0	$^{\dagger\circ}$90.0	45.0	47.0	$^{\dagger\circ}$9.5	0.0	0.0	$^{\dagger\circ}$111.0	45	47	0.0	$^{\dagger\circ}$57.2	28.7	29.9	$^{*\bullet}$0.5	<0.1	<0.1

† $EvoSuite_{Amp}$ performs significantly better than $DSpot_{Mut}$ (p-value < 0.01)
$^{\circ}$ $EvoSuite_{Amp}$ performs significantly better than $DSpot_{Cov}$ (p-value < 0.01)
* $EvoSuite_{Amp}$ generates more KLOC than $DSpot_{Mut}$
$^{\bullet}$ $EvoSuite_{Amp}$ generates more KLOC than $DSpot_{Cov}$

Table 3. Spearman correlation value (ρ) between test suite size (LOC) and mutation score.

Tool	p-value	Correlation (ρ)
$EvoSuite_{Amp}$	<0.01	0.698
$DSpot_{Mut}$	<0.01	0.588
$DSpot_{Cov}$	<0.01	0.608

had smaller test suites with $DSpot_{Cov}$. Furthermore, when comparing $DSpot_{Mut}$ to $EvoSuite_{Amp}$, the improved test suites for 27 out of 42 class versions (64.3%) had a better ratio of killing mutants per line of code with $DSpot_{Mut}$, and similarly 26 out of 42 versions (61.9%) were better with $DSpot_{Cov}$ than $EvoSuite_{Amp}$. In all seven versions in which $DSpot_{Mut}$ has a better mutation score, it improves test suites with a smaller KLOC compared to $EvoSuite_{Amp}$. As an example, Cli-39 as shown in Table 2 part G, $EvoSuite_{Amp}$ generates 0.2 KLOC to kill one mutant, while $DSpot_{Mut}$ generates 0.1 KLOC to kill two mutants. When comparing $EvoSuite_{Amp}$ to $DSpot_{Cov}$, where $DSpot_{Cov}$ has a better mutation score, 8 out of 15 class versions (53.3%) have a lower number of LOC. As an example, for Cdc-11, $EvoSuite_{Amp}$ generates 0.2 KLOC while killing around 18 mutants and $DSpot_{Cov}$ generates 0.3 KLOC, while killing 20 mutants. On the contrary, Jsp-84, $EvoSuite_{Amp}$ generates 0.2 KLOC while killing around six mutants, and $DSpot_{Cov}$ generates 0.1 KLOC, while killing 27 mutants.

In order to verify whether there is a correlation between the generated test suite KLOC size and the increase of mutation score, we used the Spearman rank correlation measure. Table 3 presents the correlation coefficients of each tool. There is a strong correlation (ρ) between the $EvoSuite_{Amp}$ size of the test, and the increase in mutation score. In both $DSpot_{Mut}$ and $DSpot_{Cov}$, there is a moderate ($\rho > 0.4$) correlation between the size and increase of the mutation score, and high correlation ($\rho > 0.7$) for $EvoSuite_{Amp}$. All the tools' p-values are less than 0.01, which shows that there is statistical significance.

> **Conclusion for RQ3.** $EvoSuite_{Amp}$ generates a larger final test suite when compared to $DSpot_{Mut}$ and $DSpot_{Cov}$.

Answer to RQ4: Consistency. In order to investigate the non-determinism rate on each tool, we calculated the mean, median, and standard deviation (σ) of the mutation score for all 42 subject class versions over each of the respective 30 re-runs. Table 2 (parts B and C) shows the result of the calculations. The mutation score $EvoSuite_{Amp}$ produces has a greater standard deviation when compared to $DSpot_{Mut}$ and $DSpot_{Cov}$. There were only 7 out of the 42 versions (16.6%) for which the $DSpot_{Mut}$ produced a higher standard deviation, while there was zero for $DSpot_{Cov}$.

> **Conclusion for RQ4.** $EvoSuite_{Amp}$ tends to show more varied behavior when compared to $DSpot_{Mut}$ and $DSpot_{Cov}$.

5.1 Discussion

We now discuss some of the ramifications of our results, along with further observations made during the course of the experiments.

Mutation Score. The $EvoSuite_{Amp}$ tool, in general, kills more mutants than *DSpot*, which shows that using the distance to mutation fitness function that is provided in *EvoSuite* can kill mutants that *DSpot* finds hard to kill. In the case of amplifying developer-written tests using $DSpot_{Cov}$, it is not surprising that an increase in code coverage also helped to increase the mutation score, as mutants that are not reached by developer-written tests could not be detected. Overall, the results show that *EvoSuite*'s evolution and mutation analysis technique is much more suited to improving test suites to kill mutants than *DSpot*.

Unique Mutants. Furthermore, $EvoSuite_{Amp}$ finds and kills more unique mutants after 30 runs when compared to $DSpot_{Mut}$ and $DSpot_{Cov}$. This shows that *EvoSuite* explores more parts of the program than *DSpot* within the 30 runs and that it could find more unique mutants, further adding to our finding that it is better at improving test suites to kill mutants than *DSpot*.

Test Suite Size. In answering RQ3, we found that $EvoSuite_{Amp}$ usually creates a bigger test suite when compared to the two configurations of *DSpot*, and that there is a high correlation between killing mutants and a big test suite. However, by looking at the mutants killed per number of lines of code, the value is not significantly bigger. We set $EvoSuite_{Amp}$ to not run the minimization technique that the default *EvoSuite* does (see Section 3), to avoid original developer-written tests being discarded — however, enabling this technique could reduce the lines of code while maintaining the mutation score. We leave this experiment as an item for future work.

Consistency of Results. Finally, RQ4 shows that both $DSpot_{Mut}$ and $DSpot_{Cov}$ give more consistent results over the 30 runs with each subject class version. This potentially means, however, that $EvoSuite_{Amp}$ has a higher chance of exploring more edge cases due the higher degree of stochasticity that it evolves the developer-written tests, and thereby could find more unique mutants, as shown by the answer to RQ2.

Readability of Final Tests. Anecdotally, we noted that the tests produced by $EvoSuite_{Amp}$ were less readable than *DSpot*'s. Some of this was due to the inevitable disruption caused by the evolutionary operators (although we deliberately turned some of these off for this reason — see Section 4.1). In particular, the carving procedure adapts developer-written tests to EvoSuite's internal test case representation, which causes them to lose some of their original qualities. This is something that needs to be investigated in future work.

6 Related Work

There have been many works that have sought to generate tests based on existing tests, for example to speed up the process of test generation [**?**], or as seeds as the initial population of a search-based technique [**?**,**?**].

Test amplification is a research area that comprises techniques designed to *improve* a developer-written set of test cases in some aspect. One of these aspects is the test suite's coverage and mutation score [?]. There have also been techniques that attempt to improve new tests generated by the amplification process, for example with respect to their readability [?] and potential redundancy [?]. Popular test amplification tools include *DSpot* for Java [?], studied in this paper, and *AmPyfier* for Python [?]. *Test Cube* is a developer-centric test amplification tool for Java [?] that operates as a plugin for the IntelliJ integrated development environment, and builds on the techniques of *DSpot*.

However, none of these works directly compare test amplification tools with techniques capable of fine-grained fitness information to guide the test case search towards strongly killing mutants — functionality that is available in *EvoSuite*.

Elswhere, Olsthoorn et al. [?] applied model seeding that could contribute improving mutation score while maintaining readability of the test cases. There have been some studies on how to amplify tests made by Google [?] and Facebook [?], which asked the professional developer to generate new tests manually that help in increasing mutation score. The two studies are different to ours, however, as they focus on trying to amplify the tests manually, whereas in the study of this paper, we focus on trying to automate this process.

7 Conclusions and Future Work

Test amplification tools aim to improve developer-written tests, but are limited in the changes they can make and are not guided by fine-grained fitness information. Search-based test case generation tools like *EvoSuite*, on the other hand, can benefit from the guidance provided by fitness functions, and have a lot more control over the structure of tests, but are limited in terms of their re-use of developer tests and the final readability of the tests they generate.

In this paper, we formulated a version of *EvoSuite*, $\textit{EvoSuite}_{Amp}$, that uses its carving functionality to start the search on the basis of developer-written test code, and evolves the tests towards killing mutants. When evaluating it against the state-of-the-art Java test amplification tool *DSpot*, $\textit{EvoSuite}_{Amp}$ was better at killing more mutants and killing more unique mutants that *DSpot* was found to never kill in any of the 30 re-runs of our experiments.

In essence, our paper shows that it is possible for automated tools to kill more mutants when starting from developer-written tests, so long as they are given more flexibility in terms of modifying those tests, as well as adequate guidance. However, the downside is less readability of the final tests, since they are further away from the original ones provided by developers. This suggests two possible alternative avenues for future work. Firstly, test amplification tools like *DSpot* could be improved with finer-grained fitness information, and modified to not throw away tests that are improved with respect to fitness goals — with the intention of further improving them so that they eventually kill more mutants; and/or secondly, tools like *EvoSuite* should be adapted, so they are better at

utilizing developer-written tests as a starting point for the search, with the added capability of retaining the characteristics of the original tests, where possible. In particular, work needs to be done in improving *EvoSuite* data structure for encoding tests, so that it can better accommodate the wide variety of styles in which JUnit test cases are written.