An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding

Benjamin Simon Clegg
University of Sheffield

Phil McMinn
University of Sheffield

Gordon Fraser
University of Passau

ABSTRACT

Automated grading often requires automated test suites to identify students' faults. However, tests may not detect some faults, limiting feedback, and providing inaccurate grades. This issue can be mitigated by first ensuring that tests can detect faults. Mutation analysis is a technique that generates artificial faulty variants of a program for this purpose, called mutants. Mutants that are not detected by tests reveal their inadequacies, providing knowledge on how they can be improved. By using mutants to improve test suites, tutors can gain the confidence that: a) generated grades will not be biased by unidentified faults, and b) students will receive appropriate feedback for their mistakes. Existing work has shown that mutants are suitable substitutes for faults in real world software, but no work has shown that this holds for students' faults. In this paper, we investigate whether mutants are capable of replicating mistakes made by students. We conducted a quantitative study on 197 Java classes written by students across three introductory programming assignments, and mutants generated from the assignments' model solutions. We found that generated mutants capture the observed faulty behaviour of students' solutions. We also found that mutants better assess test adequacy than code coverage in some cases. Our results indicate that tutors can use mutants to identify and remedy deficiencies in grading test suites.

CCS CONCEPTS

• **Social and professional topics** → **CS1**; • **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

autograding, mutation analysis, introductory programming

ACM Reference Format:
Benjamin Simon Clegg, Phil McMinn, and Gordon Fraser. 2021. An Empirical Study to Determine if Mutants Can Effectively Simulate Students' Programming Mistakes to Increase Tutors' Confidence in Autograding. In *Proceedings of the 52nd ACM Technical Symposium on Computer Science Education (SIGCSE '21), March 13–20, 2021, Virtual Event, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3408877.3432411

SIGCSE '21, March 13–20, 2021, Virtual Event, USA

ACM ISBN 978-1-4503-8062-1/21/03…$15.00
https://doi.org/10.1145/3408877.3432411

1 INTRODUCTION

Automated grading sees widespread use in programming courses, as it allows for a growing number of students' programs to be assessed without a similar increase in the use of tutors' time [9, 21]. This is especially important for Massive Open Online Courses (MOOCs), where students far outnumber tutors, and can learn at any time of the day [14, 22]. However, automated grading produces its own challenges. Tutors must define how a grader determines a solution's correctness, often using a suite of automated tests. Test suites can vary in quality; lower quality test suites may not detect some faults. This is problematic for two reasons. First, students who make mistakes that are not detected by tests will receive higher grades than students whose mistakes are detected. Second, students cannot receive feedback on undetected mistakes. The prevalence of this issue may be exacerbated by students making mistakes at a frequency that tutors may not anticipate [2]; tutors may be unaware of their test suites' inadequacies in revealing faults. Tutors may be able to use students' programs to find deficiencies in their test suites, but may be limited by assessment deadlines, and awareness of undetected faults is impossible without costly manual analysis. Similarly, the instantaneous generation of grades and feedback by online graders [15] can be a double edged sword; they benefit students' learning, but cannot be tuned using knowledge gained from students' solutions before they are assessed. In addition, new programming tasks will not have existing students' solutions for test suite evaluation, so their adequacy cannot be determined directly.

Test goals can be used in place of real faults to estimate a test suite's adequacy. A test suite capable of achieving an entire set of test goals is more likely to reveal real faults. One popular technique to construct test goals is code coverage, where each line of a program is a test goal, which a test achieves by executing. If some lines are not executed by tests, faults cannot be revealed within them; failure to achieve test goals reflects lower test adequacy. However, it is possible for a test to execute code without checking its effect. As such, faults that do not cause an error simply by being executed may not be detected; covering every line does not guarantee test adequacy. An alternate technique is mutation analysis, which involves the generation of multiple artificial faulty variants of a program, called *mutants*, to use as test goals. If a test suite fails on a mutant, it detects it, achieving a test goal. Mutation analysis has been shown to be an effective means of determining the quality of a test suite [10, 12]. Using mutants to automatically identify and address deficiencies in a grading test suite would resolve the issues caused by undetected faults, without needing to manually analyse students' programs. This would provide a tutor with a greater degree of confidence that they will not have to make corrections to their test suite either under an encroaching deadline in traditional

assessment, or after students' faulty solutions have been accepted in the case of online graders. However, if real faults behave differently to mutants, mutation analysis may not be an appropriate means to measure test adequacy; some tests may pass on every mutant, but fail for some real faults. It is therefore important to evaluate how well mutants capture the behaviour of real faults.

Previous studies have shown that mutants are a valid substitute for real faults in software projects when assessing a test suite's quality [10, 12]. This is performed by measuring how many real faults are *coupled* to simple mutants, which occurs when a test that fails on a real fault also fails on a set of mutants. By showing that real faults are coupled to a set of mutants, there is evidence that a test suite that detects all mutants should also detect real faults. Such a study has not yet been conducted to verify that mutants can be used to simulate mistakes present in students' programs.

In this paper, we aim to remedy this by presenting the results of our empirical study. We used faulty students' solution programs for real introductory programming tasks in place of faults made by experienced programmers. We found that most detected students' faults are coupled to mutants that were generated by existing mutation tools. This indicates that available tools generate mutants that can simulate students' faults, since they cause the same test failures. Using sampled test suites, we identified a positive correlation between the proportion of detected mutants and the detection rate of faulty students' solutions. We also compared the performance of mutants to another adequacy measure, code coverage. We found that mutant detection and code coverage are similarly correlated to real fault detection, though mutants and real faults can remain undetected even if 100% code coverage is achieved, which only requires few, weak tests in small programming tasks. Mutants provide additional information on how to improve test suites when coverage cannot; tutors can gain more confidence in their test suites' adequacies by using mutants in addition to code coverage metrics.

2 MUTATION ANALYSIS

Mutation analysis involves the generation of artificial faults from a correct version of a program, with the primary goal of informing the improvement of software tests [10]. These artificial faults, *mutants*, are generated by applying a set of rules, called *mutation operators*, to the original program. For example, the *arithmetic operator replacement* mutation operator could replace the statement `a = b + 1` with `a = b - 1`. A typical application of mutation analysis is to determine how effectively a suite of automated software tests can detect faults. By executing a test suite on every mutant, the proportion of mutants that it detects can be calculated. This is referred to as the *mutation score*, where 0% indicates that no mutants are detected, and 100% shows that every mutant is detected.

Educational Applications: The principal application of mutation analysis is to evaluate the adequacy of a test suite's capability to detect faults [10]. A test suite that detects more mutants (i.e. has a high mutation score) should also detect more real faults, as shown by Just et al. [12] High mutation scores therefore provide testers with more confidence in the fault detection capabilities of test suites. Similarly, high mutation scores can provide tutors with more confidence that their grading test suites will identify students' faults, even when no existing students' faults are available.

There is some potential for mutation analysis to enhance the generation of feedback. There are several existing approaches to automated feedback generation [16, 19, 20]. Such approaches are typically data dependent, and often require a set of previously observed faults. Mutation analysis could be used to increase the number of observable faults, potentially improving the quality of the resulting feedback. Additionally, if a new programming task is defined, no existing students' faults will be available. In these cases, generated mutants could be used in their place. Mutation analysis can also be used in fault localisation [17, 18]. By executing tests on mutants with known locations, it is possible to generate a model to estimate the locations of other faults based on their test results. Providing students with the estimated locations of their faults may help them to improve their code, serving as effective feedback.

Coupling: Mutation analysis relies on the assumption that these generated mutants can replicate the behaviour of real faults, despite typically being less complex. This is supported by the *coupling effect*, first defined by DeMillo et al. [5]; if a test suite is sensitive enough to detect simple faults, it must also be capable of detecting more complex faults. As such, a real fault is said to be *coupled* to a set of mutants if the tests that detect the real fault also detect the mutants. Just et al. have shown that the majority of real faults from five open source programs couple to mutants, and that there is a correlation between real fault detection and mutation score [12]. However, this has not yet been demonstrated for students' faults. As students' faults may differ significantly from those made by experienced programmers, it is important to demonstrate that mutants are capable of simulating them before using mutants to inform the development of grading test suites.

Chen et al. have proposed another coupling measure, called *probabilistic coupling* [3], which derives an estimated probability, p, that a real fault, f, is detected given that a test goal, g_i, is satisfied (e.g. a mutant is detected), or $p = \mathbb{P}(\text{detect } f \mid g_i \text{ is achieved})$. As such, probabilistic coupling offers some insight into how well a test goal captures a test suite's adequacy in revealing a real fault. Using the maximum probability for a set of test goals for the real fault allows this insight to be gained without knowledge of every possible real fault, and without an impact from irrelevant test goals (e.g. mutants that are not covered by a test). While conventional coupling shows that a test suite that detects real faults also detects mutants, probabilistic coupling essentially reveals the inverse; a test suite that can detect mutants is also able to detect a real fault.

3 EXPERIMENT PROCEDURE

We conducted an empirical study to investigate the following research questions:

- ***RQ1:*** *Do students' faults couple to mutants?*
- ***RQ2:*** *Is there a relationship between the detection of mutants and students' faults?*
- ***RQ3:*** *How do mutants and code coverage compare in evaluating test adequacy?*

Subject Programs: In this study, we used solutions to end of year programming assignments that were written by students enrolled in a first year undergraduate introductory Java programming course. These solutions were collected across three different cohorts of

Table 1: Subject Classes

Assignment	Class	Student Sltns.	Mutants		Tests		LoC
			Major	*Pit*	*Manual*	*EvoSuite*	
Chess	`Board`	45	57	204	18	14	25
	`Queen`	40	94	386	9	2	41
Wine	`Cellar`	36	512	1715	16	15	315
Fitness	`DataLoader`	38	44	195	7	1	71
	`Questions`	38	199	808	20	30	263

students, with each cohort completing a different assignment. We selected five subject classes from these three assignments to use in our study, and collected students' solution classes for each. Table 1 shows the number of compilable solutions we used for each subject class. We also used a correct model solution for each class in our study. Their lines of code counts are shown in Table 1.

The *Chess* assignment required students to implement a game of chess. The `Board` class stores the state of the chess board, including the positions of pieces such as `Queen`, each of which uses this state to evaluate which moves they can make.

Wine tasked students to write a program that processed a set of CSV files containing various properties of different wines. `Cellar` reads and stores the property values for each wine, and allows for these values to be queried via various methods.

Fitness called for students to implement a system that can read structured data files containing readings of several different fitness trackers. `DataLoader` parses the data files and constructs a data structure using several provided classes. `Questions` is the implementation of 20 public methods that each return an answer for a pre-defined question about the data, such as "what is the average heart rate across the whole dataset?"

We were not able to compile some of the students' solutions, usually due to improper packaging or simple syntax errors. We attempted to repair such simple issues for each of these solutions, and removed any solutions which required a more complex fix, or where the intended functionality was unclear. This provided us with additional testable, faulty solutions. In total, we collected 236 individual solution classes, of which 197 were compilable.

Tests: For three classes, `Cellar`, `DataLoader`, and `Questions`, we used the original grading test suites in our analysis. These test suites were not provided to students during the assessment. For `Board` and `Queen`, we did not have access to the grading test suites, so instead we used our own manually written test suites, which both achieve 100% coverage on their respective model solutions.

In order to increase variation in test cases, we expanded these test suites using automated test generation. For this we used EvoSuite [6], a search-based test generation tool, to generate tests from the model solution of each subject class. EvoSuite has been shown to generate test suites which are capable of revealing real world faults [1]. However, since the model solutions implement vague aspects of their specifications, tests generated from the model solutions may test for unspecified functionality. We manually analysed individual tests and solutions and found that some students' solutions fail due to deviations from the model solution rather than actual faults or specification violations, producing false positives. For example, a model solution would throw a `NullPointerException` when a method is called with a parameter of `null`. A generated test that checks for this behaviour would incorrectly fail for students' solutions which defensively handled `null` inputs, punishing such good practices. Of the 126 total generated tests, 64 produced such false positives, which we removed. The remaining tests are shown in Table 1. We also found that no students' solutions passed every test case. We confirmed this by manually analysing every solution; each contained at least one fault which caused a test to fail.

Mutants: We generated mutants using two Java mutation tools; Pit 1.5.2 [4] and Major 1.3.4 [11, 13]. We executed both mutation tools on the model solutions of each of our subject classes, with all mutation operators enabled. In total, Pit implements 29 operators, and Major implements 9. Major 1.3.4 only supports the mutation of classes that are compatible with Java 1.7. However, some of the model solutions use features of newer Java versions, such as streams. Prior to running Major, we modified these solutions to use only Java 1.7 compatible features, while remaining functionally identical. We also created a third set of mutants by combining the mutants generated by Pit and Major. We refer to this combined set as *Both* in our results for RQ2. We did not use this combined set for RQ1, since if a student's solution is coupled to mutants generated by one tool it must be coupled to mutants from both.

Execution: We executed each test on all students' solutions and mutants, and recorded whether the test passes or fails. We treated test errors as failures; for example if the test exceeds a time budget, it indicates that the class under test will not halt due to an endless loop. For each subject class, we also executed all of its tests on the model solution, to ensure that the tests are valid. We confirmed that all of our unit tests were valid. For the model solutions, we also used JaCoCo [8] to record the lines covered by each test, allowing us to compare the use of code coverage metrics and mutation scores.

Coupling: If a set of simple faults are detected by a test that also detects a complex fault, the complex fault is coupled to the simple faults. Existing work on coupling uses individual known faults to evaluate the coupling effect [12]. However, it is possible (and likely) that faulty students' solutions contain multiple complex faults; students' solutions cannot be considered as individual faults. It is therefore not sufficient for only one test that fails on a student's solution to also fail on a mutant, as this would only show that the solution contains a coupled fault; it may also contain other uncoupled faults. As such, we performed our coupling analysis on each individual failing test of each solution. For each solution, we identified the failing tests. For each failing test, we determined if the solution is coupled with any mutants; i.e. the test also detects at least one mutant. If the solution is coupled with any mutants for this test, we defined the test as a *coupling* test. Otherwise, we defined it as an *uncoupling* test.

We then calculated the *coupling ratio* for each solution, the proportion of coupling tests to failing tests for the solution. This allows us to determine to what extent a solution's faults are coupled to the available mutants. If a solution has a coupling ratio of > 0, it has at least one coupling test; it is partially coupled to the mutants, as it has at least one fault that is coupled to at least one mutant. If all of a solution's tests either pass or are coupling (coupling ratio $= 1$), then it is absolutely coupled to a set of mutants; every fault in the solution is coupled to at least one mutant. In our results

(Table 2), we show the mean coupling ratio for each subject class. We found that for some subject classes, some generated mutants failed on every test. For example, some mutants would cause an exception to be thrown when calling a class's constructor, causing any executed tests to immediately fail. We removed these trivially detected mutants prior to running our coupling analysis, since they would introduce bias by being coupled to every student's fault.

We also used probabilistic coupling to evaluate the test suite adequacy estimation performance of mutation analysis and code coverage. We grouped test goals into sets: Major's mutants, Pit's mutants, and covered lines of the model solution. We then computed the detection probability for each test goal and each student's solution, by selecting the tests that achieve a test goal (i.e detect a mutant or cover a line) and calculating the proportion of these that fail on the student's solution. Where no tests achieved a test goal, we used a probability of zero. We then found the maximum probability of a test goal in each set for every student's solution.

Correlation: By finding evidence of a relationship between the detection rates of mutants and students' faults, we can assert that mutation analysis is an effective measure of test adequacy. As such, we investigated their correlation to one another. Since we only had one test suite for each subject class, we generated test suites which are subsets of the main test suite for each class. We controlled for test suite size, targeting a consistent number of test suites for each possible size from two tests to 70% of the total number of tests. For each suite size, we generated a set of test suites by randomly sampling from the complete set of tests for the class. We discarded duplicate test suites. Our generation strategy aimed to generate 80 randomly sampled suites for each class, split evenly between each possible test suite size. We repeated this process 30 times, generating 30 sets of ~80 suites for every class. For `Queen` and `DataLoader`, some suite sizes cannot reach the target number of tests. As such, these subjects have fewer test suites overall.

By evaluating the results of tests in each generated suite on students' solutions and mutants, we were able to determine how many mutants and faulty students' solutions are detected by each suite. This allows us to calculate the mutation scores and real fault detection rate for each suite. Since the solutions may contain multiple faults, this real fault detection rate is truly the proportion of faulty students' solutions that have been detected. However, Gopinath et al. note that if tests which detect many mutants are highly likely to detect real faults, the fact that the mutants do not entirely resemble real faults is irrelevant [7]. As such, if we identify a positive correlation between mutation score and the detection of faulty students' solutions, we can demonstrate the ability of simple mutants to simulate students' faults, despite their lower complexity.

We also used the model solution's line coverage of each test to determine the coverage ratio for each generated suite. We only consider the model solution's coverage to simulate a tutor developing a test suite before any students' solutions have been collected.

Since we found that the observations for each repetition were not normally distributed, we required a non-parametric measure of correlation. As such, we computed Spearman's correlation between the mutation score of each group of mutants (Pit, Major, and Both) and the real fault detection rate. We also calculated the correlation between the coverage ratio and the real fault detection rate.

Table 2: Coupling Results

Class	Coupling Ratio		% Students' Solutions Fail All Tests	% Solutions Pass All Tests		
	Major	*Pit*		*Students*	*Major*	*Pit*
`Board`	1	1	6.98	0	7.27	14
`Queen`	1	1	0	0	4.26	16.6
`Cellar`	0.99	0.99	8.57	0	52.1	45
`DataLoader`	1	1	0	0	13.2	23.6
`Questions`	1	1	0	0	13.6	16.2

Unlike our analysis of the coupling effect, we did not remove mutants that failed on every test prior to evaluating the correlation of mutation score to real fault detection. We chose to include such mutants as this would better represent how a tutor would use a mutation tool to evaluate their grading test suites; determining how many mutants are detected by a test suites, and identifying those which are not. All of the students' solutions are detected by at least one test; no undetectable solutions will skew the measurement of the real faulty solution detection rate.

Threats to Validity: We only used the solutions of students who volunteered to participate in the study. As such, the collected data may suffer from a degree of self-selection bias. It is possible that other students' solutions may contain different faults. Similarly, the faults present in these solutions may differ to those in other introductory Java programming courses at other institutions, or from those in other learning environments (e.g. MOOCs). This limits generalisability.

It is also possible that our students' solutions may contain additional faults that are not detected by any of the unit tests that we used in this study. This could entail that there are some unidentified faults that are not coupled with any mutants. We attempted to mitigate this issue by using EvoSuite to generate additional tests, but there is still no guarantee that every fault was detected. It may be possible to manually analyse every individual solution for every fault, but such an approach is out of scope for this paper.

4 RESULTS

4.1 RQ1: Do students' faults couple to mutants?

Coupling: Table 2 shows the results of our coupling analysis. We find that most subjects have a coupling ratio of one; each solution's failing tests also fail on at least one mutant. This reveals that these solutions are completely coupled to a set of generated mutants. We however identify that some students' faults for `Cellar` are not fully coupled by the mutants of either tool. We find that two tests fail for five of the students' solutions, but do not fail for any mutants. Upon inspecting the output of these tests for these five solutions, we identify two causes for the test failures. One of these causes is that some solutions use the default scope for the class's constructor, instead of public, which causes the constructor of the super class to be used instead. The other cause is that some solutions modify a reference to a list rather than a copy of the list, causing some data to be incorrectly changed. These faults are therefore not simulated by mutants, but mutation operators could be implemented to simulate them. With the exception of these faults, every detected fault is coupled to at least one mutant. Solutions are coupled to mutants

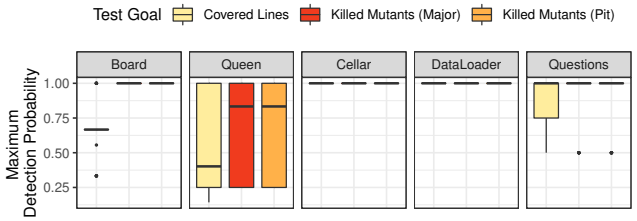
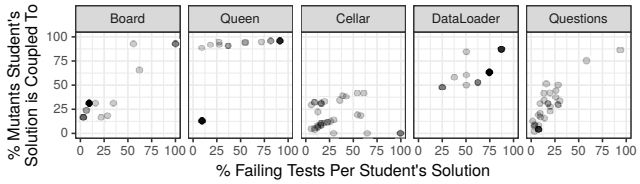
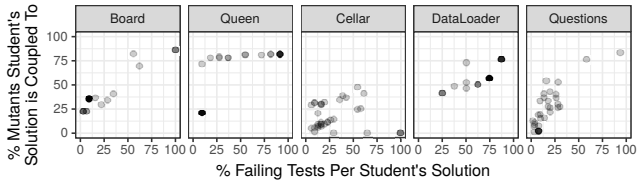


Figure 1: Maximum solution detection probabilities of every student's solution for each set of test goals.



(a) Major's mutants



(b) Pit's mutants

Figure 2: Observations of coupled mutants and failing tests for each student's solution. (Points share the same opacity.)

generated by both Pit and Major. This suggests that existing mutation operators of both tools are typically capable of simulating the behaviour of students' faults in this dataset.

Probabilistic Coupling: Figure 1 shows the results of our probabilistic coupling analysis. We observe that four of the five subjects typically have maximum fault detection probabilities of one. This indicates that mutants can behave similarly to real faults, some mutants fully capture the behaviour of faulty students' solutions. However, we also identify that `Queen` has maximum probabilities of less than one. In this case, mutants can cause more tests to fail than real faults do. As such, in some cases a test suite may detect every mutant, but may not be able to distinguish between solutions with slightly different faults.

Coupled Mutants vs. Failing Tests: Figure 2 shows the proportion of mutants that each student's solution is coupled to with respect to all tests that fail on the student's solution, and the proportion of tests that fail on each student's solution. We find that there is typically a positive relationship between the test failure rate of students' solutions and the quantity of mutants that they are coupled to. This indicates that students' solutions will be coupled to more mutants overall if they have more severe faults or a greater quantity of faults. `Cellar` has less of a clear positive relationship between coupling and failing tests due to uncoupled faults, particularly evident for solutions where every test fails.

Table 3: Mean Spearman's correlations (r_s) to the detection rate of faulty students' solutions. p = p-value.

Class	Mutation Score						Line Coverage Ratio	
	Both		*Major*		*Pit*			
	r_s	p	r_s	p	r_s	p	r_s	p
`Board`	0.61	<0.01	0.54	<0.01	0.63	<0.01	0.65	<0.01
`Queen`	0.47	<0.01	0.46	<0.01	0.46	<0.01	0.26	0.08
`Cellar`	0.75	<0.01	0.77	<0.01	0.74	<0.01	0.77	<0.01
`DataLoader`	0.29	<0.05	0.21	0.12	0.31	<0.05	0.49	<0.01
`Questions`	0.81	<0.01	0.81	<0.01	0.81	<0.01	0.84	<0.01
Mean	0.58	<0.01	0.56	<0.05	0.59	<0.01	0.61	<0.05

We observe that there are typically a relatively high number of solutions with few test failures. This suggests that students often make mistakes which cause few tests to fail. The impact of this lower number of faults on mutant coupling varies between subject classes. `Questions` has a very low proportion of mutants coupled to students' solutions with few failing tests. This class must have a relatively high degree of isolation between faults; making a mistake in one section of the program does not have greatly detrimental effect on the rest of the code. Comparatively, `DataLoader` has a higher minimum proportion of mutants coupled to faulty solutions. This is likely due to this class depending on reading data from input files; a fault in this functionality will impact the rest of the class, causing multiple tests to fail. As such, programs' architectures influence how students' solutions couple to mutants.

For `Queen`, we observe a separation in the number of coupled mutants, depending on whether one or multiple tests fail on a student's solution. Where students' solutions fail only one test, the fact that this is often a test that fails on few mutants suggests that such tests only fail for more subtle faults. Such tests may only fail under specific conditions. Comparatively, where students' solutions fail multiple tests, these tests also fail on disproportionately many mutants; mutants are detected by them easily. As such, despite the students' faults being coupled to the mutants, they may still behave differently regarding how regularly they fail in some programs. This is a limitation of the coupling effect. It is therefore necessary to also consider the relationship between the detection frequency of mutants and students' solutions.

> ***RQ1:*** *Students' faulty solutions typically couple to mutants generated by existing mutation tools.*

4.2 RQ2: Is there a relationship between the detection of mutants and students' faults?

Table 3 shows the Spearman's correlations between faulty student solution detection rates and mutation scores for our generated test suites. We observe mean correlations (r_s) of 0.58, 0.56, and 0.59 for mutants produced by both tools, Major, and Pit, respectively. Since these correlations are not perfect, mutation score does not necessarily predict the real fault detection rate. However, we find that there is an overall positive correlation between the mutation score and real fault detection rates of our sampled test suites. This indicates that there is a relationship between the detection of generated mutants and real students' solutions. As such, a test suite that detects a majority of mutants is likely to also detect real faulty

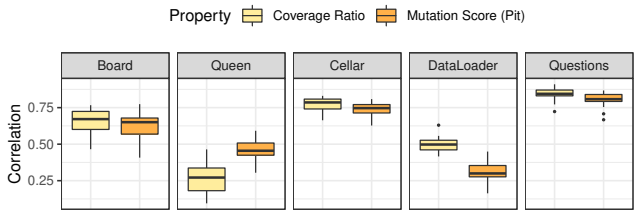


Figure 3: Correlations to detection rate of faulty students' solutions for each test suite, across 30 repetitions.

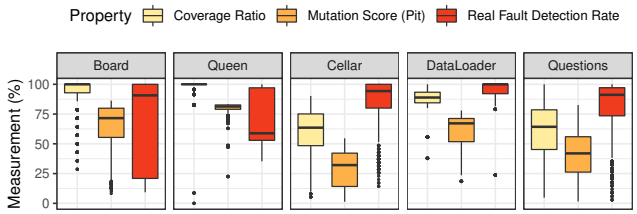


Figure 4: Observed property measurements for each test suite, across 30 repetitions.

students' solutions. This suggests that failing mutants can be used to inform the improvement of a test suite.

The correlations vary across the subject classes for all three mutant groups. In particular, the correlation is much lower for `DataLoader`, and it is the only subject where some correlations are not statistically significant; Major's mutants only achieve a correlation of 0.21 on this subject, with a p-value of 0.12. Upon closer inspection, we identify three tests that fail on every student's solution for this subject. Two of these tests pass on approximately half of all mutants, and one passes on most of the mutants. Any suite that includes these tests will therefore have a faulty solution detection rate of 100%, but could have a low mutation score, depending on how effectively the other tests can detect mutants.

> ***RQ2:*** *There is a positive correlation between mutation score and the detection rate of faulty students' solutions.*

4.3 RQ3: How do mutants and code coverage compare in evaluating test adequacy?

Since Pit's mutants have the highest correlation overall, and each of their correlations are statistically significant, we only use Pit's mutation scores for this comparison. Figure 3 shows the correlations to the faulty solution detection rate for the line coverage ratios and Pit's mutation scores of our randomly sampled test suites. The means of these correlations are also shown in Table 3. Figure 4 shows the observed values for each of the properties.

We find that, overall, the correlation for mutation score is similar to that of coverage, which is also true for three of the five subjects. However, for `Queen` and `DataLoader`, their correlations differ considerably. Coverage has a low correlation for `Queen` (0.26), and is not statistically significant. This class's model solution is very small, so most of the the sampled test suites achieve 100% coverage. As such, most observed test suites have the same coverage level, causing this low correlation. Comparatively, Pit's mutation score has a low correlation for `DataLoader` (0.31), due to the issues discussed in Section 4.2, while coverage has a moderate positive correlation.

We observe that coverage and mutation score have similar overall positive correlations to the real fault detection rate. However, mutation score's correlation is still relatively high when coverage's is low due to 100% coverage being reached. This can be observed in Figure 4, where the generated suites for `Queen` often achieve 100% coverage, but do not always detect all of the real faulty solutions, and never detect all of Pit's mutants. The utility of coverage is exhausted once 100% coverage is reached, but mutants can still guide improvements to a test suite. This is also supported by mutation score's higher probabilities in Figure 1; tests that detect mutants are more likely to fail for students' faults than tests that simply cover lines of code. As such, mutants offer a clear benefit in the development of grading test suites.

> ***RQ3:*** *Mutation score and code coverage have similar positive correlations to the detection rate of faulty students' solutions, but mutants still reveal deficiencies when coverage cannot.*

5 RECOMMENDATION TO TUTORS

We observe that students' faulty solutions are often coupled to mutants generated by existing tools. We also find a positive correlation between mutation score and the detection rate of faulty solutions, even when code coverage cannot yield additional information on test adequacy, such as when all lines are covered by a few tests in small programming tasks. As such, we recommend that tutors use artificial mutants in addition to code coverage when developing grading test suites. First, a suite should achieve 100% coverage on a model solution; uncovered code may contain faults which no test would be able to detect. Coverage in students' solutions may not directly reflect the model solution's 100% coverage, but any uncovered code in their solutions would either be unreachable or implement undefined behaviour. Once 100% coverage has been obtained, tutors should generate mutants using an off the shelf mutation tool, such as Pit, and run the test suite on them. Undetected mutants reveal deficiencies in the test suite, and can be used to inform improvements, via the nature of their faults and their locations, which new tests should exercise more thoroughly.

6 CONCLUSION & FUTURE WORK

In this paper, we have demonstrated that artificial mutants generated by existing tools can simulate students' faults, and recommend that tutors use them in addition to code coverage metrics to ensure that their grading test suites identify students' mistakes. However, more research must be conducted before the key limitations of automated grading are addressed. Detecting mutants does not guarantee that a test suite is fair; redundant tests that exercise the same functionality may introduce bias in a test suite. Existing mutation operators only consider the functionality of programs, but do not consider deficiencies in other aspects of automated grading, such as style checking. Tests must also provide feedback to guide students' growth; merely informing them of failures is not sufficient.

ACKNOWLEDGEMENTS

We would like to thank Maria-Cruz Villa-Uriol, Islam Elgendy, Thomas White, and Richard Clayton for their assistance in preparing and retrieving the subject programs.

Phil McMinn is supported in part by the Institute of Coding, funded by the Office for Students (OfS), England.

REFERENCES

[1] Almasi, M. M., Hemmati, H., Fraser, G., Arcuri, A., and Benefelds, J. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, ICSE-SEIP 2017* (jun 2017), Institute of Electrical and Electronics Engineers Inc., pp. 263–272.

[2] Brown, N. C. C., and Altadmri, A. Novice Java Programming Mistakes: Large-Scale Data vs. Educator Beliefs. *ACM Transactions on Computing Education 17*, 2 (may 2017), 1–21.

[3] Chen, Y. T., Tadakamalla, A., Ernst, M. D., Holmes, R., Fraser, G., Ammann, P., and Just, R. Revisiting the Relationship Between Fault Detection, Test Adequacy Criteria, and Test Set Size. *35th IEEE/ACM International Conference on Automated Software Engineering (ASE '20)* (2020), 284–296.

[4] Coles, H. PIT Mutation Testing. [Online; accessed 2020-08-26] https://pitest.org/.

[5] DeMillo, R. A., Lipton, R. J., and Sayward, F. G. Hints on test data selection: Help for the practicing programmer. *Computer 11*, 4 (1978), 34–41.

[6] Fraser, G., and Arcuri, A. EvoSuite: Automatic test suite generation for object-oriented software. *SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering* (2011), 416–419.

[7] Gopinath, R., Jensen, C., and Groce, A. Mutations: How close are they to real faults? In *Proceedings - International Symposium on Software Reliability Engineering, ISSRE* (dec 2014), IEEE Computer Society, pp. 189–200.

[8] Hoffmann, M. R., Mandrikov, E., and Friedenhagen, M. JaCoCo Java Code Coverage Library. [Online; accessed 2020-08-27] http://eclemma.org/jacoco/, 2016.

[9] Insa, D., and Silva, J. Automatic assessment of Java code. *Computer Languages, Systems and Structures 53* (2018), 59–72.

[10] Jia, Y., and Harman, M. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering 37*, 5 (sep 2011), 649–678.

[11] Just, R. The Major Mutation Framework. [Online; accessed 2020-08-26] http://mutation-testing.org/doc/major.pdf, 2018.

[12] Just, R., Jalali, D., Inozemtseva, L., Ernst, M. D., Holmes, R., and Fraser, G. Are mutants a valid substitute for real faults in software testing? *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering 16-21-Nove* (nov 2014), 654–665.

[13] Just, R., Schweiggert, F., and Kapfhammer, G. M. MAJOR: An efficient and extensible tool for mutation analysis in a Java compiler. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering, ASE 2011, Proceedings* (2011), pp. 612–615.

[14] Krusche, S., and Seitz, A. ArTEMiS - An Automatic Assessment Management System for Interactive Learning. In *Proceedings of the 49th ACM Technical Symposium on Computer Science Education - SIGCSE '18* (New York, New York, USA, 2018), ACM Press, pp. 284–289.

[15] Manzoor, H., Naik, A., Shaffer, C. A., North, C., and Edwards, S. H. Autograding Jupyter Notebooks. In *Proceedings of the 51st ACM Technical Symposium on Computer Science Education (SIGCSE '20)* (New York, NY, USA, feb 2020), Association for Computing Machinery, pp. 1139–1144.

[16] Marin, V. J., Pereira, T., Sridharan, S., and Rivero, C. R. Automated personalized feedback in introductory Java programming MOOCs. *Proceedings - International Conference on Data Engineering* (2017), 1259–1270.

[17] Papadakis, M., and Le Traon, Y. Effective Fault Localization via Mutation Analysis: A Selective Mutation Approach. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing - SAC '14* (New York, New York, USA, 2014), ACM Press.

[18] Papadakis, M., and Le Traon, Y. Metallaxis-FL: Mutation-based fault localization. *Software Testing Verification and Reliability 25*, 5-7 (aug 2015), 605–628.

[19] Pu, Y., Narasimhan, K., Solar-Lezama, A., and Barzilay, R. sk_p: a neural program corrector for MOOCs. In *SPLASH Companion 2016 - Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity* (New York, New York, USA, oct 2016), ACM Press, pp. 39–40.

[20] Singh, R., Gulwani, S., and Solar-Lezama, A. Automated feedback generation for introductory programming assignments. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) 48*, 6 (jun 2013), 15–26.

[21] Souza, D. M., Felizardo, K. R., and Barbosa, E. F. A systematic literature review of assessment tools for programming assignments. *Proceedings - 2016 IEEE 29th Conference on Software Engineering Education and Training, CSEEandT 2016* (apr 2016), 147–156.

[22] Staubitz, T., Klement, H., Renz, J., Teusner, R., and Meinel, C. Towards practical programming exercises and automated assessment in Massive Open Online Courses. In *Proceedings of 2015 IEEE International Conference on Teaching, Assessment and Learning for Engineering, TALE 2015* (jan 2016), Institute of Electrical and Electronics Engineers Inc., pp. 23–30.