Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests

Zalia Shams and Stephen H. Edwards
Department of Computer Science
Virginia Tech
114 McBryde Hall (0106)
Blacksburg, VA 24061
{zalia18, edwards}@cs.vt.edu

ABSTRACT

Software testing is being added to programming courses at many schools, but current assessment techniques for evaluating student-written tests are imperfect. Code coverage measures are typically used in practice, but they have limitations and sometimes overestimate the true quality of tests. Others have proposed using mutation analysis instead, but mutation analysis poses a number of practical obstacles to classroom use. This paper describes a new approach to mutation analysis of student-written tests that is more practical for educational use, especially in an automated grading context. This approach combines several techniques to produce a novel solution that addresses the shortcomings raised by more traditional mutation analysis. An evaluation of this approach in the context of both CS1 and CS2 courses illustrates how it differs from code coverage analysis. At the same time, however, the evaluation results also raise questions of concern for CS educators regarding the relative value of more comprehensive assessment of test quality, the value of more open-ended assignments that offer significant design freedom for students, the cost of providing higher-quality reference solutions in order to support better quality assessment, and the cost of supporting assignments that require more intensive testing, such as GUI assignments.

Categories and Subject Descriptors

K.3.2 [**Computers and Education**]: Computer and Information Science Education; D.1.5 [**Programming Techniques**]: Object oriented Programming; D.2.5 [**Software Engineering**]: Testing and Debugging—testing tools.

General Terms

Experimentation, Verification.

Keywords

Test-driven development, automated assessment, automated grading, mutation testing, software testing, programming assignments, test coverage, reflection, bytecode transformation.

1. INTRODUCTION

Testing accounts for 50% cost of software development. Because of the necessity of testing, more educators are including software testing [1, 2] in programming and software engineering courses [3, 4]. Current classroom assessment systems (e.g.,Web-CAT, ASSYST, Marmoset) use code coverage to evaluate how well students test their own code. Code coverage measures the percentage of code—e.g., statements or branches—that is executed by running tests.

Code coverage measurements sometimes overestimate test quality for three reasons. First, a software test may cause a bug to be executed, but contain insufficient confirmation of expected behavior (e.g., assertion statements about resulting changes in program state) to detect that the code did not behave as intended. Second, the test may have appropriate behavioral checks, but the effects of the bug may only be visible for particular data values that aren't used in the test. Third, a student's solution may be incomplete because it completely omits some required behaviors, but code coverage measures miss these omissions. In effect, code coverage only indicates how much of the *student's program* is exercised, not how much of the *required problem* has been correctly implemented.

Inspired by Aaltonen *et al.* [5], this discussion paper reports on an experimental evaluation of a new approach to assessing the quality of student-written tests using *mutation analysis*. In short, mutation analysis involves creating a collection of "buggy" versions of a complete solution by injecting errors, and then running software tests against this collection in order to determine how many of the bugs are actually detected by the tests. This produces a measure in terms of "bugs detected" rather than "statements executed," which is arguably more meaningful in evaluating the quality of tests. Mutation analysis poses three practical obstacles to classroom use, however:

1. The student's own solution cannot be used for mutation analysis, since it may be incomplete or buggy.
2. The student's software tests may not compile against an alternative (correct) reference implementation for the problem.
3. When automatically injecting faults in a solution to generate buggy versions, the problem of determining whether an injected fault results in a true bug (as opposed to a behaviorally equivalent but innocuous solution) is not computable.

The new mutation approach presented here cleanly solves all three of these problems, allowing mutation analysis to be applied to student programs. An evaluation in the context of CS1 and CS2 assignments is presented, comparing the results of mutation analysis to code coverage analysis. However, this evaluation also raises new questions about alternative approaches to quality assessment such as the mutation approach. These questions merit a broader discussion among CS educators regarding the community view on the benefits and costs: the relative bene-

ICER'13, August 12–14, 2013, San Diego, California, USA.

http://dx.doi.org/10.1145/2493394.2493402

fit of more comprehensive assessment of test quality, in contrast to the benefit of more open-ended assignments that offer significant design freedom for students, the cost of providing higher-quality reference solutions in order to support better quality assessment, and the cost of supporting assignments that require more intensive testing, such as GUI assignments.

2. RELATED WORK

Students often consider software testing to be boring. They may instead focus on output correctness on provided sample data [3] and do less testing on their own [6]. To incorporate software testing as a part of coding, Goldwasser [7] proposed requiring students to turn in tests along with their solutions, and then running every student's tests against every other's program. Widely used automated assessment tools (e.g.,Web-CAT [8], ASSYST[9] and Marmoset [10]) evaluate students' code along with their software tests.

2.1 Automated Grading Systems and Their Evaluation Measures

Web-CAT executes instructor-provided reference tests, and for some programming languages, such as Java, provides static analysis tools that can assess code style, adherence to coding standards, and some structural aspects of commenting conventions. It calculates a student's score based on three factors: percentage of instructor's reference tests passed, percentage of the student's own tests passed, and code coverage achieved by the student's tests on his or her own solution. ASSYST also runs instructor-written reference tests against student submissions, and measures code coverage (i.e., statement coverage) of student solutions when the instructor-written tests are executed. Marmoset[10] evaluates student programs against two test sets: 1) public test sets, and 2) "release" test sets. The public test sets are available to students, and feedback generated from running public test sets are provided to students immediately. Release test sets are run against submissions that pass all the public tests. Results from the release test sets are delayed and limited.

All three of these tools run instructor-provided reference tests against student submissions. However, reference tests written in compiled languages, such as Java, do not compile against solutions that fail to provide all the required features in an assignment or that have incorrect method signatures. As a result, these systems cannot evaluate partial or incomplete submissions and gives no credit in those situations. Edwards et al. first presented a solution for assessing partial or incomplete Java programs by applying late binding to test cases [11].

Web-CAT, ASSYST, and Marmoset all provide some form of code coverage analysis. Although code coverage provides feedback about what percentage of solution code was executed, it does not assess test adequacy or test quality. As an alternative, Aaltonen *et al.* [5] proposed using mutation analysis to assess the quality of student-written tests in Java assignments.

2.2 Mutation Analysis for Assessing Tests

Mutation testing seeds artificial defects, such as changes in arithmetic operators, boolean operators, or numeric constants, into a program and checks whether test cases can distinguish the mutated version from the original program. The defective versions, which are called mutants, are intended to be representative of the faults that programmers are likely to make in practice. The key principle of mutation analysis is that complex faults are coupled to simple faults in such a way that a test suite that detects all the simple faults is thorough enough to detect most complex faults [12, 13]. In mutation testing, a large collection of mutants is generated from the original program, and then test suites are run against all of these mutants. A test suite's effectiveness is measured by its mutation score—the percentage of mutants detected (or "killed") by the test suite.

There are many automated tools for generating mutants. Mutants for java programs can be generated at the source level or at the bytecode level μJava[14] is the most popular source-level mutant generator for Java. μJava is efficient in generating mutants and easy to use but has some limitations. For example, it does not handle Java generics and does not support JUnit-style tests. In addition, the source code of μJava is not publicly available.

Bytecode-level mutation testing is more efficient and scalable than source-level testing. Javalanche [15] is the most widely used bytecode-level mutation analysis tool for Java. It replaces numerical constants, negates jump conditions, omits method calls and replaces arithmetic operators. Javalanche handles JUnit test cases. It combines the process of mutation generation, running user supplied test cases, and analyzing mutant coverage from test pass-fail rates all as a single action.

Aaltonen *et al.* [5] proposed the use of mutation analysis for assessing the quality of student-written tests in Java assignments. They used Javalanche to generate mutants from a student's solution and to run the student's test cases to check how many mutants were detected. This approach appears to provide a deeper perspective on the adequacy of the student-written tests than code coverage. However, some limitations make it impractical to use in the classroom. For example, they could not provide instant feedback to students because the mutation analysis required significant processing time. Moreover, the generated mutants were implementation dependent. Thus, complex solutions had many mutants, which could lead to an artificially lowered mutation score. Similarly, testing unspecified behaviors that were not part of the assignment could reward a student with a higher mutation score. Further, their approach required manual inspection of the generated mutants to weed out those that were not true bugs. In total, their procedure was inefficient and not uniform, so cannot be used for automated classroom assessment tools.

3. OBSTACLES OF USING MUTATION TESING INAUTOMATED GRADING

Code coverage is often used in automated grading tools for evaluation of student written code quality by calculating what percentage of a student's solution has been executed by their tests. It may overestimate test quality because tests may not have comprehensive checks of expected behavior (assertions), because tests may not use critical data values, or because the student's solution may be incomplete or incorrect. Mutation analysis is a stronger indicator of test adequacy and effectiveness. However, three main obstacles make it impractical to use in classroom set-up where feedback is provided in real-time.

First, to check that a student's tests are effective for the *assigned problem*, one must assess the tests against a complete, correct representation of the entire problem. The student's own solution may not meet these requirements, which gives rise to one of the limitations of code coverage. In effect, lack of completeness or correctness brings the same limitations to bear on mutation analysis. If the student's solution is incomplete, the set of mutants generated from the student's solution will not cover the space of all bugs possible in the assignment. If the student's solution contains additional extra features that are not required, the number of mutants generated may be larger than necessary.

```
int fact = 1;
for (int i=1; i<=n; i++){
    fact = fact * i;
}
```

Figure 1(a): Original code

```
int fact = 1;
for (int i=2; i<=n; i++){
    fact = fact * i;
}
```

Figure 1(b): Equivalent mutant

```
int fact = 1;
for (int i=0; i<=n; i++){
    fact = fact * i;
}
```

Figure 1(c): Non-Equivalent mutant

But most importantly, if the student's solution contains bugs already, then some generated mutants that differ from the student's original may in fact be correct, and there is no way to reliably distinguish when a non-passing test indicates that a buggy mutant has been discovered, versus when the test itself has behavioral expectations that differ from the assignment so that it will pass on the student's (buggy) solution. As a result, generating mutants from a reference solution that is known to be complete and correct, such as one written (and thoroughly tested) by the instructor, is preferable. This reference solution, in effect, serves as an executable model of correct behavior for analyzing the effectiveness of student-written tests.

Second, if a reference solution is used to assess student-written tests, those tests may not even compile against the reference solution. Tests written in object-oriented languages like Java are often written in the form of program code, and tests may depend on any aspect of their author's solution. A student may write tests that refer to any visible features (i.e., public methods, constructors, and fields) in their solution, which may or may not be required by the assignment. For example, a student may have included a helper method in a class as a personal design decision. If a test refers to such a feature, but that feature is not required to be present in all solutions—including the reference solution—the test will fail to compile against the reference solution, and all of its mutants.

Third, mutation testing is manually expensive because not all generated mutants necessarily represent faults. While mutation analysis is founded on the idea of seeding faults, the approach used to seed faults still occasionally produces mutants that are not buggy—they behave indistinguishably from the original. For example, Figure 1 shows a small piece of code, together with two possible mutants generated by simply substituting a different constant value for the start of the loop counter—something intended to represent a kind of "off-by-one" error that a programmer might make. Figure 1(a) shows the original code before mutation. In Figure 1(b), the loop counter's initial value has been altered to be 2, while in Figure 1(c) it has been altered to be 0. Unfortunately, while Figure 1(c) shows a true bug (known as a *non-equivalent mutant*), Figure 1(b) uses the same mutation process but accidentally generates a mutant that is behaviorally equivalent to the original code (an *equivalent mutant*). Unfortunately, it is theoretically impossible to automatically decide whether a mutant is a true bug (observably differs from the original) or is a benign (but useless) equivalent mutant that will not fail any tests. The presence of equivalent mutants introduces errors in the mutation analysis, since they cannot, in principle, ever be distinguished from the original code by any test cases. However, since it is impossible to identify them automatically, weeding out equivalent mutants is a manual process. To use mutation analysis in the classroom for automated feedback, it is necessary to devise a practical approach to detecting and weeding out such equivalent mutants.

4. SOLUTION STRATEGIES AND IMPLEMENTATION

To move toward a practical mutation analysis approach that can be used in the classroom, the three problems outlined in Section 3 must be resolved.

4.1 Using a Reference Implementation

As described in Section 3, effective mutation analysis requires that we assess the student's tests against a solution to the problem that is known to be complete and bug-free—such as a reference solution written (and suitably tested) by the instructor. Student-written solutions may be incomplete or erroneous. Therefore, an instructor-provided reference solution is a more reliable candidate to be used as the mutation source.

Mutant generation also takes time. If mutants are generated from a reference solution available when an assignment is created, it is possible to pre-generate the full set of mutants from the reference solution ahead of time, so that mutant generation will not slow down analysis of student-written tests. For this paper, we used a modified version of Javalache [15] to perform this task. The original version of Javalanche internally generates mutants, runs a list of given test suites, and analyzes coverage of the test suites, all as one action. It does not store mutants; instead, it regenerates them every time a new test suite is analyzed. We modified Javalanche to separate out the mutant generation step and to store the generated mutants so that when a student submits a new test suite, the generated mutants can be reused. This minimizes the overhead of mutation generation, with the aim of supporting real-time feedback to students.

4.2 Removing Compile-Time Dependencies in Test Suites

While the intent is to run student-written tests against the pre-generated mutants from a reference implementation, those tests may not even compile against the mutants. If student-written tests are provided in source code form, they may have implicit or explicit dependencies on specific, individual design decisions present only in that student's solution. A novel way to resolve this issue in Java is to transform the student-written tests so that they use *reflection*, a mechanism to introspect and modify program components at runtime. Using reflection, a Java program can loop up methods available on a class, or dynamically invoke a method found by such a lookup. We rewrite the bytecode of a student-written test suite into a pure reflective form so that program components such as classes, methods, and fields will be looked up at runtime, rather than being directly referenced by the test's compiled code. As a result, reflective tests can be applied against any solution, regardless of which features it defines or omits. Each student's tests will compile against his or her own code, if they compile at all, and so we need not worry about syntactically invalid test sets.

Our solution uses Javassist [16] to transform every method call, constructor call, or field reference into purely reflective forms via the ReflectionSupport [17] library, a library that completely

encapsulates the details of using reflection under a streamlined interface. As a result, test cases written using this library have no compile-time dependencies on the software under test.

The purely reflective test cases will run against any mutant. Individual test cases that depend on features that are missing from the reference implementation fail at run-time, while other test cases run normally. This makes it possible to separate out "student-specific" tests from those that are generally applicable to any solution, and then mutation analysis can be restricted to just those that tests that are generally applicable.

4.3 Automatic Detection of Equivalent Mutants

While it is not possible to automatically determine if a mutant is equivalent to the original program, we can instead use a conservative approach to classify mutants as *provably different* from the original, or *not provably different*. Initially, all mutants are placed in the not provably different category, since we have no evidence they are true bugs. When the instructor's reference tests are run against the mutants, any mutants detected can be moved into the provable different category immediately.

Then as each student submits tests, we can screen the tests using the original (non-mutated) reference solution. This allows invalid tests that do not correctly capture expectations of the problem to be weeded out. If the valid tests are run against all mutants, any mutant failing a valid test is then *provably different* from the original. Any mutants remaining in the not provably different set can be ignored for the purposes of mutation analysis, since they potentially are equivalent mutants.

As more and more students submit tests, the *provably different* set can increase in size, including every mutant to date that some valid test has definitively shown is observably different from the reference solution. This allows the strength of the mutation analysis to increase over time. This automatic mutant detection process is conservative because there may be cases where no test suites find behavioral differences for a given mutant that is actually a bug. However, initial results from the evaluation presented in Section 5 indicate that this is not a significant issue.

5. EVALUATING THE SOLUTION

To evaluate the practicality of this solution, we applied it to seven assignments that were originally given in CS1 and CS2, where students were required to write their own software tests for each of their solutions.

5.1 CS1 Assignments

Our evaluation included four CS1-level assignments. Project 1 involved writing a "screen scraping" program that used a utility class to extract meaningful content from a web page. Students had great flexibility in choosing the web pages of interest, and the targeted information to extract. Project 2 involved writing classes to read from a set of RSS (or atom) feeds on the web and pretty-print the aggregate series of stories. In Project 3, students wrote a web-based video and news browser that allows one to search multiple RSS feeds for given keywords and view the resulting stories, including embedded videos where present. Project 4 involved writing two different "virtual pet" implementations—one entirely of their own design—that obeyed simple finite state machine models.

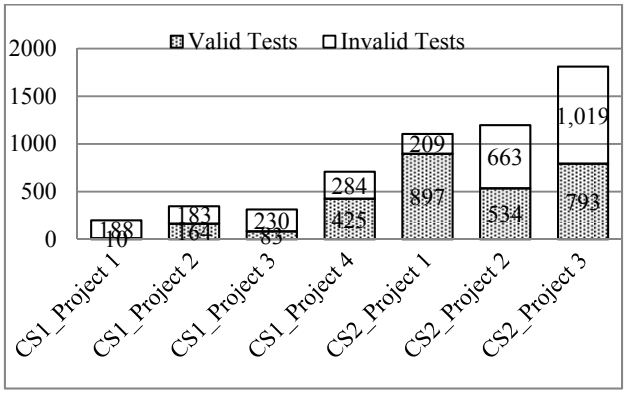


Figure 2: Test-Suites Summary

For all assignments, a complete set of mutants was generated from an instructor-provided reference solution. Student tests were then compiled against their author's original solution, and then transformed using bytecode rewriting to a fully reflective implementation. Tests were then run against the non-mutated reference solution to weed out invalid tests, as well as tests that depended only on student-specific details of that student's personal solution. The remaining valid test cases were then run against all mutants to determine which were detected.

The number of students completing the CS1 assignments varied from 42-47. Surprisingly, however, many of the tests written by students were invalid, in that they failed to pass on the reference solution, usually because they depended on some individual choices about that student's own solution. In the first CS1 project, for example, 188 of the 198 test cases (95%!) written by students could not be run on the reference implementation because they included student-specific details. Figure 2 summarizes the number of test cases, both valid and invalid, written by students for each of the assignments. Figure 3 summarizes the total number of mutants generated from the reference solutions for each assignment, with Projects 1 through 4 having 47, 45, 43, and 27 mutants, respectively.

From the surprising number of non-applicable tests, two things are clear. First, one cannot effectively evaluate student tests if large proportions of those tests are not considered because they are too specific to a student's individual choices—mutation analysis as discussed here may require explicitly giving students a well-defined interface to test against. Second, despite the belief of many instructors that assignments written for automated grading (or for student testing) are over-constrained and lack opportunities for students to exercise design flexibility, the experiences here indicate this is not true. Even in the CS1 assignments shown here, there is clear room for individual choices and alternative design strategies, and student-written tests appear often to be tightly coupled to such decisions. Even though the instructor-written reference tests compiled against and exercised every student solution, student-written tests lacked this level of generality and transportability across solutions.

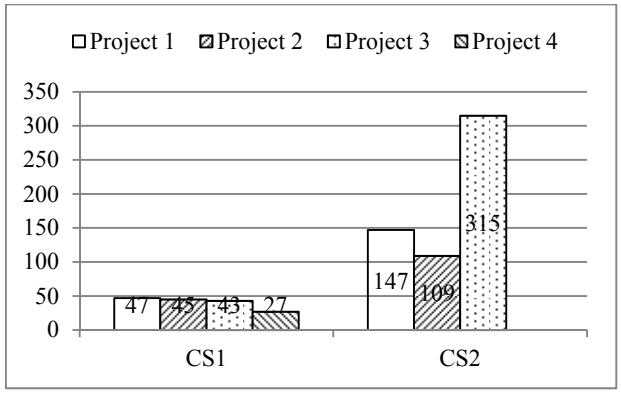
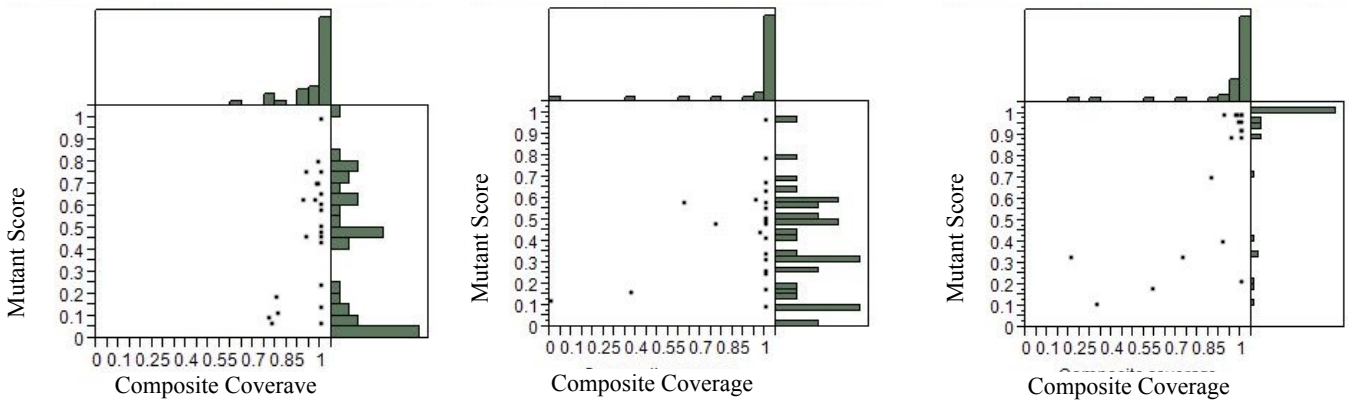


Figure 3: Mutants Summary



(a): CS1-Project 2 **(b): CS1-Project 3** **(c): CS1-Project 4**

Figure 4: Scatter plots of composite code coverage vs. mutation score of CS1 assignments

More than half of the student-written tests were applicable to the reference implementation in only one of the four CS1 assignments, CS1-Project 4. Here, the number of mutants were small and there was a common core of the assignment that was well-specified for students, so tests on this common core were generally applicable to all solutions. Here, the mutant kill ratio (M = 87.2%, sd = 25.9%) was significantly lower (Wilcoxon signed rank test, S = 72, p = 0.0054) than the test coverage achieved (M = 93.8%, sd = 17.2%). Of the 42 students who completed the assignment 26 produced test sets that detected every single mutant, while 28 produced test sets that achieved 100% code coverage (22 students were in both groups). All of the mutants were provably different than the original reference solution.

Figure 4 shows scatter plots of mutant detection percentages against test coverage percentages for CS1 Projects 2-4. Project 1 is omitted, since 95% of tests were invalid and only 5 (of 47) students had test suites that included *any* valid tests that could be run against mutants.

5.2 CS2 Assignments

We also applied our solution to three assignments from a CS2 course. CS2-Project 1 involved implementing a game board for the game "Minesweeper," which students could then play using a provided GUI interface. 107 students completed the assignment, writing 1,106 test cases, of which 209 were non-applicable (invalid or student-specific). The reference solution produced 147 mutants. Once again, the mutant kill ratio (M = 69.1%, sd = 17.2%) was significantly lower (Wilcoxon signed rank test, S = 2724.5, p < 0.0001) than the test coverage achieved by students (M = 95.8%, sd = 13.8%). Figure 5(a) shows a scatter plot of the mutant detection percentages against test coverage percentages. From this, it is clear that achieving a higher mutation score (better bug-revealing capability) was harder than achieving higher test coverage, supporting the belief that mutation analysis provides a better evaluation of test quality.

CS2-Project 2 required students to implement a circular, doubly-linked list and use it to provide a basic cyclic image gallery viewer. Students wrote the data structure to represent the "model", and also implemented a Swing graphical interface for the "view". 99 students completed this project, writing 1,179 test cases. However, because of the GUI elements in this assignment (which students did test themselves), a larger percentage (55.4%) of student tests were not applicable to the reference implementation and were excluded from mutation analysis. Again, mutation detection scores (M = 75.9%, sd = 15.0%) were significantly lower (S = 2134.5, p < 0.0001) than test coverage scores (M = 96.9%, sd = 7.0%). The scatter plot is shown in Figure 5(b).

In CS2-Project 3, students wrote both an array-based and a link-based implementation of a queue interface. This was a pure data structure assignment and involved no GUI programming. Students were required to implement iterators (with removal support), along with support for copy construction, equality testing, hash code generation, and the `toString()` method. 99 students completed the assignment, writing 1,197 test cases, 56.2% of which were not applicable to the reference implementation. This high degree of invalid tests was surprising, considering the well-specified nature of the interface students were required to implement. However, students varied greatly in how they designed their solutions internally, with many developing base classes to hold common elements shared between both queue implementations. A surprising number of student-written tests depended on these internal design decisions, limiting the number of tests that could be analyzed.

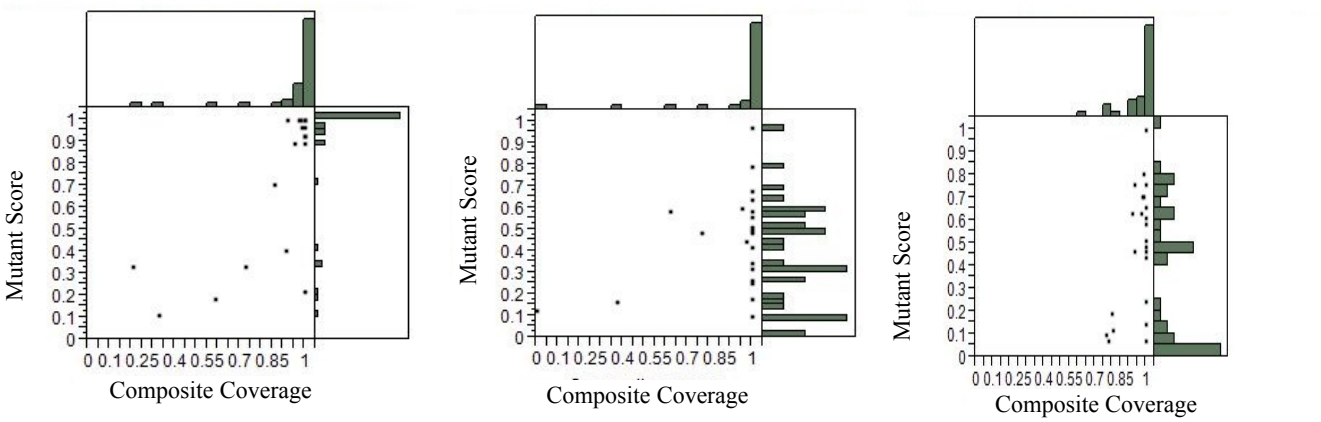
Once more, mutation analysis produced scores (M = 42.2%, sd = 13.5%) that were significantly lower (S = 855.5, p < 0.0001) than test coverage scores (M = 95.1%, sd = 7.7%). The scatter plot is shown in Figure 5(c). However, the lower mutation scores may result from the large number of tests (over half) that could not be analyzed because they depended on individual design choices not present in the reference solution.

6. CONCLUSIONS

This paper describes three key problems with applying mutation analysis to student assignments: student solutions cannot be used as the source of mutants, since they may be buggy or incomplete; student tests may not compile against other (correct, complete) solutions; and weeding out mutants that are indistinguishable from the original must be done manually. Further, we present a solution that addresses all three problems.

Questions for discussion: Although all three of these problems have been addressed, the evaluation of the approach raises concerns that deserve a broader discussion among educators:

- Is the cost of providing complete, yet minimal and error-free reference solutions acceptable to instructors?



(b) CS2-Project 1 **(b) CS2-Project 2** **(c) CS3-Project 3**

Figure 5: Scatter plots of composite code coverage vs. mutation score of CS2 assignments

- For complex assignments (and those with GUIs), the mutation analysis may be time-consuming, simply because of the volume of tests and mutants involved. Is "delayed feedback" on test quality acceptable, and if so, how much delay is reasonable?
- Because this approach relies on incremental improvements as students submit more and more tests, the results of the quality analysis may change (improve, becoming more selective/sensitive) over time as students work on their submissions. How will it affect students to see test quality declining over time as they work on a single assignment, even though their testing has not changed?
- This quality assessment strategy is at odds with open-ended assignments, or assignments with large amounts of design freedom. In those cases, student tests are so diverse that significant numbers cannot be applied to a common reference solution, resulting in an artificially depressed quality measure. Is this acceptable?
- Achieving accurate quality measurement relies on students writing tests to "a common specification" as much as possible, instead of to their own personal design. Will this lead to over-constrained assignments that preclude students from creating their own designs?

REFERENCES

[1] C. Desai*, et al.*, "Implications of integrating test-driven development into CS1/CS2 curricula," *SIGCSE Bull.,* vol. 41, pp. 148-152, 2009.

[2] T. Dvornik*, et al.*, "Supporting introductory test-d
riven labs with WebIDE," presented at the Proceedings of the 2011 24th IEEE-CS Conference on Software Engineering Education and Training, 2011.

[3] S. H. Edwards, "Using software testing to move students from trial-and-error to reflection-in-action," *SIGCSE Bull.,* vol. 36, pp. 26-30, 2004.

[4] D. S. Janzen and H. Saiedian, "Test-driven learning: intrinsic integration of testing into the CS/SE curriculum," *SIGCSE Bull.,* vol. 38, pp. 254–258,, 2006.

[5] K. Aaltonen*, et al.*, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," presented at the Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, Reno/Tahoe, Nevada, USA, 2010.

[6] S. H. Edwards, "Using test-driven development in the classroom: Providing students with concrete feedback " presented at the International Conference on Education and Information Systems: Technologies and Applications, 2003.

[7] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," *SIGCSE Bull.,* vol. 34, pp. 271-275, 2002.

[8] S. H. Edwards, "Rethinking computer science education from a test-first perspective," presented at the Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, Anaheim, CA, USA, 2003.

[9] D. Jackson and M. Usher, "Grading student programs using ASSYST," *SIGCSE Bull.,* vol. 29, pp. 335-339, 1997.

[10] J. Spacco and W. Pugh, "Helping students appreciate test-driven development (TDD)," presented at the Companion to the 21st ACM SIGPLAN symposium on Object-oriented programming systems, languages, and applications, Portland, Oregon, USA, 2006.

[11] S. H. Edwards*, et al.*, "Running students' software tests against each others' code: new life for an old "gimmick"," presented at the Proceedings of the 43rd ACM technical symposium on Computer Science Education, Raleigh, North Carolina, USA, 2012.

[12] R. A. DeMillo*, et al.*, "Hints on Test Data Selection: Help for the Practicing Programmer," *Computer,* vol. 11, pp. 34-41, 1978.

[13] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Trans. Softw. Eng. Methodol.,* vol. 1, pp. 5-20, 1992.

[14] Y.-S. Ma*, et al.*, "MuJava: an automated class mutation system: Research Articles," *Softw. Test. Verif. Reliab.,* vol. 15, pp. 97-133, 2005.

[15] D. Schuler. (04/15/2013). *Javalanche*. Available: https://github.com/david-schuler/javalanche/

[16] S. Chiba and M. Nishizawa, "An easy-to-use toolkit for efficient Java bytecode translators," presented at the Proceedings of the 2nd international conference on Generative programming and component engineering, Erfurt, Germany, 2003.