Evaluating Quality of Student-Written Tests

Zalia Shams and Stephen H. Edwards
Department of Computer Science
Virginia Tech
2202 Kraft Drive
Blacksburg, VA 24060 USA
{zalia18, edwards} @cs.vt.edu

ABSTRACT

Software testing is important, but judging whether a set of software tests are effective is hard. The same problem also appears in the classroom as educators frequently include software testing activities in assignments. While tests can be hand-graded, some educators use objective performance metrics to assess software tests, just as professionals do. The most common measures used at present are code coverage measures— tracking how much of the student's code (in terms of statements, branches, or some combination) is exercised by the corresponding software tests. Code coverage has limitations as it does not assess whether computational results from the executed code are checked against expectations, and sometimes it overestimates the true quality of the tests. We alternatively evaluate students' tests on how many defects the tests can detect from injected errors—mutation testing— and actual errors present is others code—all-pairs testing. We overcome a number of technical challenges to apply these two approaches in classroom assessment systems. Afterwards, we compare all three methods—all-pairs testing, mutation testing and code coverage—in terms of how well they predict defect detection capabilities of student-written tests when run against a large collection of known, authentic, human-written errors. Experimental results encompassing over 700,000 test runs show that all-pairs testing is the most effective predictor of the underlying bug revealing capability of a test suite. Further, no strong correlation was found between bug revealing capability and either code coverage or mutation analysis scores. Investigating effectiveness of student written tests we find that, students are mainly "happy path" testers – purpose of their writing tests is to show that their code "worked" rather than finding real errors.

1. PROBLEM AND MOTIVATION

Testing accounts for 50% of the cost of software development. Because of the necessity of testing, educators are including more and more software testing as a part of programming and software engineering assignments. Automatic 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 a student's code—e.g., branches or statements—that are executed from his test cases. It may overestimate test quality as executed code may not be checked against expected behaviors. Moreover, students frequently submit incomplete and incorrect assignments. If a student completes 60% required features and executes all the code from his tests, he will achieve 100% coverage. As a result, code coverage calculated on incomplete or incorrect solutions will fail to indicate the true quality of their tests.

Alternatively, we evaluate students' tests based on how many bugs they can detect from other students submissions—known as all-pairs testing—and from a correct solution having artificially injected errors in it—known as mutation testing. In all-pairs testing a student's tests are run against the other students' programs. This mechanism gives students a greater realization of the density of bugs in their code and their ability to write tests that find defects in others' solutions. Mutation analysis seeds artificial errors into code and then checks whether a test suite can detect them. Test suites that detect more errors are better than those that detect less. Both all-pairs testing and mutation testing have a number of technical challenges to apply them in classroom assessment systems. We investigate three main research questions in this paper:

1) Is it feasible to use all-pairs and mutation testing for evaluation of student-written tests?

2) Are they better indicators of test quality in terms of defect-revealing capability of tests?

3) How effective are student-written tests in finding real bugs?

We devise novel techniques to address challenges of all-pairs testing and mutation analysis and apply them in automated assessment systems. We also directly compare all three methods of measuring test quality in terms of how well they predict the observed bug revealing capabilities of student-written tests when run against a naturally occurring collection of student-produced defects. Finally, we investigate quality and pattern of students' tests. Experimental results encompassing over 700,000 test runs show that the all-pairs testing approach is the best predictor of defect detection ability of student-written tests, while neither code coverage nor mutation analysis scores were significantly correlated with defect detection capability. Analyzing the results, we found that students write very similar test cases. While they achieved an average branch coverage of 95.4% on their own solutions, their test suites were only able to detect an average of 13.6% of the faults present in the entire program population. Thus, students are naïve "happy path testers" - they write tests to show that their code "worked", rather than finding errors or faults.

2. BACKGROUND & RELATED WORK

To incorporate software testing as a part of coding, Goldwasser [1] proposed an idea of requiring students to turn in tests along with their solutions, and then running every student's tests against every other's programs. Embracing this idea, automated assessment tools (e.g., Web-CAT, ASSYST, and Marmoset) evaluate student written code against instructor's tests and use some form of coverage analysis to assess quality of students' tests. As tests written in compiled languages, such as Java, do not compile against a solution that differs in structure from the author's solution, no capability for all-pairs testing was available.

Edwards and Shams [3] are the first to provide a practical solution for all-pairs testing. Aaltonen *et al.* [2] proposed using mutation analysis to evaluate adequacy of the tests but computational overhead makes it impractical to generate real-time feedback and use in classroom assessment tools. We proposed solutions for these obstacles and evaluated mutation analysis against an alternative, code coverage [4].

3. UNIQUENESS OF THE APPROACH

Our research is conducted in three main steps:

1) Overcoming the challenges of all-pairs and mutation testing in classroom assessment,

2) Comparing how accurately all-pairs testing, mutation testing and code coverage predict defect revealing capability of students' tests, and

3) Analyzing student-written tests to find effectiveness.

We describe the three steps below.

3.1 Overcoming the challenges of all-pairs and mutation testing

The main obstacle of using all-pairs testing and mutation analysis (when run on buggy versions generated from any solution other than the student's) is the compile-time dependency of the tests on its author's solution. In object-oriented languages, such as Java, tests are written as a part of the solution and may refer to any visible or public feature of the solution. For example, a student may decide to add a helper method in his solution to assist some computation. If a student tests such components that arise from his personal design decisions that are not present in others' code, then his tests will not compile against others' or reference solutions. We provided a novel way to resolve this issue in Java by transforming the student-written tests so that they use reflection to defer binding to specific features of a solution until run-time [3]. Test sets can be compiled against the particular solution for which they were written. Similarly, instructors typically provide their own implementation to double-check reference test sets, so instructor-written reference tests will compile against the implementation. The byte-code of the compiled test sets are then transformed into reflective forms so that they use late binding. This allows the tests to be run against any solution, with student-specific test cases failing at run-time because of failed reflective method lookups.

Reflection is a feature of Java that can be used to reduce compile-time dependencies between code components. In the previous work [3], we transformed the byte-codes of the test cases using Javassist into purely reflective forms. Java reflection can be complicated and error-prone to use, but we use ReflectionSupport [5], a library of methods that completely encapsulates the details of using reflection underneath a powerful, streamlined interface ideal for writing test actions. As a result, test cases written using this library will have no compile-time dependencies on the software under test.

The purely reflective test cases will run against any student submission. Individual test cases that depend on features that are missing or incorrectly declared in the student's work fail at run-time, while other test cases run normally. Therefore, any test set will run against all solutions, even incomplete ones. This strategy of removing compile-time dependencies from test sets is a key technique to applying all-pairs testing.

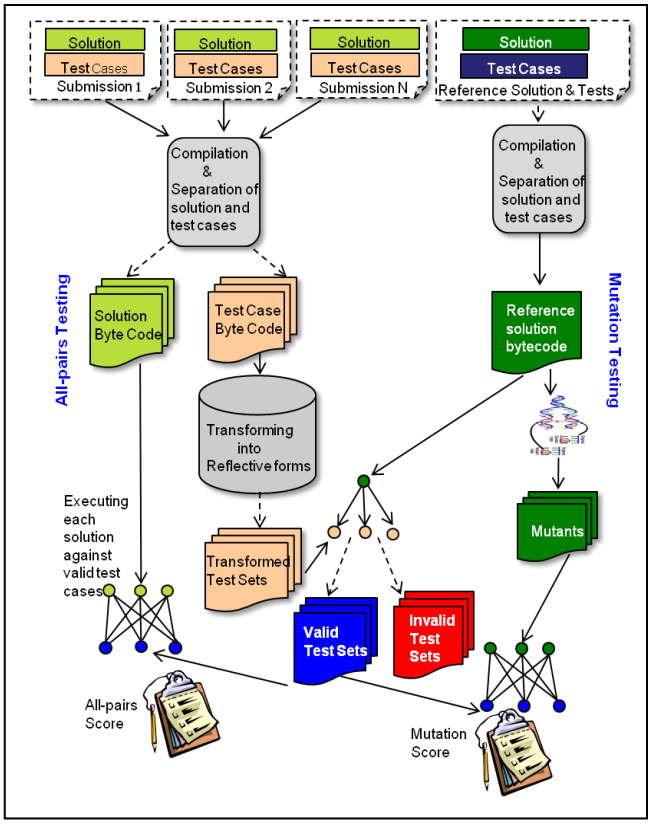

Figure 1: The procedure for executing all-pairs testing

solution, 2) removing compile time dependencies from students' tests so that they run against the mutants, and 3) automatically detecting if a mutant is a true defect or equivalent to the original. Usually, instructors write a reference solution, which is presumably correct and includes all the required features of the assignment. In earlier work [4], we chose the reference solution to generate mutants, so that mutants would cover all required behavior, and so that non-equivalent mutants could be considered to truly deviate from the required behavior. Mutant generation also takes time that slows down analysis of student-written tests. By pre-generating the full set of mutants from the reference solution ahead of time, mutant generation does not have to be performed for each student submission, so it will not slow down analysis of student-written tests. Later, student-written test cases are transformed to remove compile-time dependencies. Afterwards, we validate students' tests by running them against the reference solution, and then run only the valid test cases against the collection of mutants. Further, we can opportunistically (and conservatively) consider mutants as being non-equivalent as soon as any valid test case that passes on the reference solution (whether the test is written by the instructor or a student) fails against that mutant. Mutants for which no test case "witnesses" are discovered during the assignment can be (conservatively) judged as being "possibly equivalent" to the original, and can be eliminated from the analysis. This approach eliminates any need for manual inspection for equivalent mutants. Thus, we resolve the challenges of application of mutation analysis in an educational setting and evaluate the quality of a student's test from how many mutants it has detected.

3.2 Comparing All-pairs testing, mutation testing and code coverage

We compare the three measures of test quality: all-pairs testing, mutation analysis and code coverage for Java programs submitted in a CS2 assignment. The goal of this experiment is to evaluate the likelihood that a student-written test suite will discover any given bug, which we can call the suite's *bug-revealing capability*. Unfortunately, measuring this capability directly is both challenging and expensive in general. Because of the size of the collection of programs, manual debugging and manual defect counting were cost-prohibitive. Instead, inspired by Edwards [6], we constructed a proxy for the set of bugs contained in the student programs. All student-written tests were combined into a single large test suite, along with the instructor-written reference tests for the assignment. This "master" test suite was then run against all student programs, collecting all results in a large matrix—one column per student program, and one row per test case. From this matrix, we could identify *equivalent test cases*, where every program that passed one test case also passed the other, and every program that failed one also failed the other. The master test suite could then be reduced by eliminating redundant test cases, keeping only one representative from each equivalent group. After this reduction, all test cases in the master test suite were behaviorally distinguishable, in the sense that for any pair of test cases, there was at least one program that passed one test case in the pair but failed the other. This does not guarantee that the test suite is *orthogonal*, in the sense that test cases do not overlap or test the same behaviors. However, it does indicate that there is at least one bug that is uniquely detected by each test case. In other words, each test case differs from all others in the specific defect(s) that it detects—the sets of defects detected by any two test cases may overlap, but cannot be identical. This master test suite is then a proxy for the set of all bugs contained in all of the student-written solutions produced for this assignment. While each individual test case in the suite may not necessarily represent a single defect, it does represent an "equivalence class" of defects, where all bugs in the equivalence class cause the corresponding test case to fail. Further, there may still be some bugs present in one remaining equivalence class—those that cannot be detected by any of the test cases in the master suite. However, for sufficiently large numbers of distinct test cases, and distinct bugs written in solutions, the equivalence classes proxied by each test case grow small, as does the number of bugs that are detected by no test cases. Prior work indicates that test cases in a test suite created in this fashion are strongly correlated with the bugs revealed by manual means, justifying the use of this proxy approach [6].

Once the master suite is established, the number of test cases from the master suite that are failed by any given program is a strong estimate of the number of bugs present in the program. The frequency of the (equivalence class of) bug represented by a master suite test case can be determined by how many of the student programs fail that test case. Further, by comparing the tests in any given student-written test suite against the master suite, it is possible to tell which of the (equivalence classes of) bugs it can detect.

3.3 Analyzing effectiveness of students' tests

We gather all the students' tests and screen them by executing against the reference solution. Student-written tests fall into three categories: valid, invalid and student-specific. Tests that pass reference solution are valid, whereas tests that fail are invalid. The third category, student-specific tests, has at least one referral to any component of the author's solution that is not present in other students' solutions. We use only valid tests in effectiveness analysis. We use the master suite created in comparing all-pairs testing, mutation analysis and code coverage to investigate effectiveness. To determine effectiveness of tests, we calculated how many test cases from the equivalence class were covered by each student-written tests. We also analyzed if student-written tests have variation among them or not.

4. RESULTS AND CONTRIBUTIONS

We have presented the results in three sections based on the three main research questions.

Feasibility of All-pairs and mutation testing:

We applied our solution for all-pairs testing to two programming assignments and student written test sets in two different courses, Fall 2007 CS1 and Spring 2012 CS2 where we transformed all students' tests into reflective forms. The CS1 assignment had 46 submissions consisting of 463 test cases where 405 (87.5%) were valid, 27 (5.8%) invalid and 31 (6.7%) were student specific. Next, all the valid test sets were executed against all 46 solutions that resulted in a total of 18,225 test runs. Half of the test cases failed to find any defects, 63% of the submissions passed all test cases, and the average passing rate was 94.4%.The CS2 assignment had 101 student submissions with 101 test sets consisting of 2155 test cases. After running the reflective versions of all test sets against an instructor-provided reference solution, we got 2001 (92.9%) valid, 126 invalid (5.8%), and 28 (1.3%) student-specific test cases. Every test case found defects in at least one program and the average passing rate was 83.5%. Most importantly, for both the CS1 and CS2 assignments after bytecode transformation to reflective forms, **no compile time failure occurred** while testing that ensures feasibility of our solution.

To evaluate the practicality of our solution for mutation analysis, we applied it to six CS1 and CS2 assignments. We pre-generated mutants from the reference solution, removed compile-time dependencies from students' tests, validated the tests against the reference solution, automatically detected mutants from the valid tests, and computed the mutant detection ratio of the tests. Among the six assignments, three were from CS1, where the number of mutants varied from 42-47. The other three CS2 assignments had 147, 109 and 305 mutants. In all the assignments, the mutant detection ratio was significantly lower than the test coverage achieved. For example, the CS2 assignments mutation detection scores are 0.685, 0.759, 0.422 and average code coverage scores were 0.949, 0.969, and 0.95. From this result, it is clear that achieving a higher mutation score was harder than achieving higher test coverage. However, we found that when students have larger design freedom in assignments, significant number of their tests examine components related to their personal design decisions. Such student-specific tests could not be evaluated against mutants generated from the reference solution. Outcome of our mutation analysis is published [4] in ICER, 2013.

Comparison of all-pairs testing, mutation testing, and code coverage:

To compare defect-revealing capability we used a CS2 assignment that was used both in all-pairs testing and mutation testing. Evaluating all three test measures in just one assignment required a significant effort—in this case, compiling and analyzing over 700 thousand test case executions. The master suite consists of

Table 1: Correlations between test quality measures and bug-revealing capability.

	Bug-revealing Capability Score
Mutant kill ratio	0.0069
Composite coverage	-0.0634
All-pairs score	**0.6403***

* significant with $p < 0.0001$

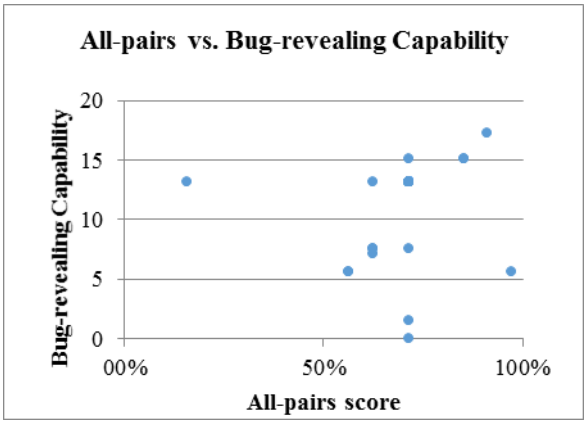

Figure 2: Relationship between all-pairs scores and bug-revealing capability estimates.

2001 valid tests from students and 82 reference tests. After removing redundant tests as described in section 3.2, we found 112 distinct test cases, where each test case represents one (small) equivalence class of bugs that is uniquely identifiable. Every bug that is detectable by any test case written by any student (or the instructor) is represented in this set. On average, each student program passed 76.8% of the test cases in the master suite, with every student program containing at least one bug. Each student program failed an average of 26 test cases in the master suite. Across the 101 programs evaluated, this produced 2,486 individual faults that could potentially be detected by tests. By using this master suite as a proxy for the observable bugs in the student solutions, it is possible to calculate which bugs are detectable by each individual student's test suite. By construction, each student test will overlap with at least one test from the equivalence class of the master suite. We calculate how many tests from the equivalence class were covered by each student's test, add up the total number of observable bugs detected by the equivalence classes, and divide by total number of observable bugs (i.e., 2486). We compare the score of bug-revealing capability with the scores of code coverage, all-pairs testing and mutation analysis for all the students. The co-relation between bug-revealing capability with code coverage, mutation-testing and all-pairs testing are -0.0634, 0.0069, and 0.6403 respectively as shown in Table 1. Thus experimental results show (Figure 2,3,4) that all-pairs testing is the most accurate predictor of defect-revealing capability, while neither code coverage nor mutation analysis scores were significantly correlated with defect detection capability.

Effectiveness of Student-written tests:

All student-written test suites had less than an 18% chance of detecting any given bug occurring in the population of programs

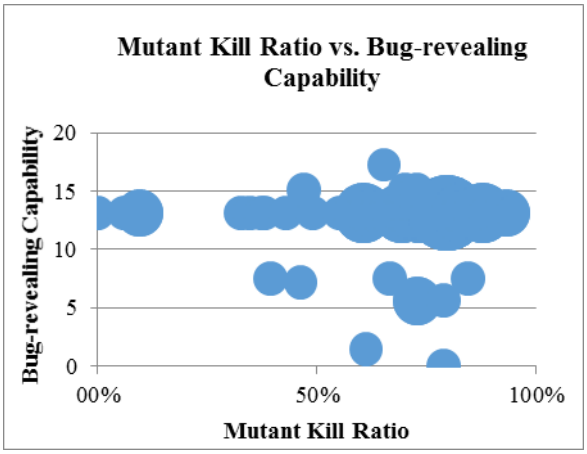

Figure 3: Relationship between mutant kill ratios and bug-revealing capability estimates.

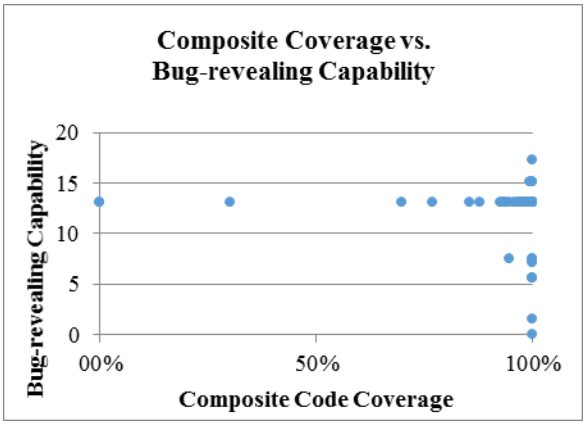

Figure 4: Relationship between code coverage and bug-revealing capability estimates.

being investigated. Interestingly, the average branch coverage was 95.4%, with nearly two-thirds of students (64.6%) achieving perfect 100% coverage of all branches in their solution. Here we need to mention that students were aware that their tests were graded based on coverage and they got feedback from Web-CAT on coverage during submission. Clearly, students were able to write tests that exercised all of the code they wrote, even if these tests were less effective at finding real bugs.

Analyzing the pattern we found that students write very similar tests. The 2001 tests fall into 44 equivalence classes. Examining their tests, we discover a similar pattern. They generally write one test case per method to show that their code "worked". In conclusion, students were following naïve, "happy path" testers, writing basic test cases covering mainstream expected behavior rather than writing tests designed to detect hidden bugs.

We provide practical solutions to apply all-pairs testing and mutation analysis to evaluate student-written tests. To our knowledge, we are the first to compare these two approaches with code coverage in terms of authentic human-written errors. We also analyzed effectiveness and pattern of student-written tests. These results suggest that educators should strive to reinforce test design techniques intended to find bugs, rather than simply confirming that features work as expected.

5. REFERENCES

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

[2] K. Aaltonen, *et al.*, "Mutation analysis vs. code coverage in automated assessment of students' testing skills," *OOPSLA*, pp. 153-160, Nevada, USA, 2010.

[3] S.H. Edwards, Z. Shams, M. Cogswell, and R.C. Senkbeil. Running students' software tests against each others' code: New life for an old "gimmick". In *Proc. 43rd ACM Tech. Symp. Comp. Sci. Education*, ACM, 2012, pp. 221-226.

[4] Z. Shams and S.H. Edwards. Toward practical mutation analysis for evaluating the quality of student-written software tests. In *Proc. 9th Ann. Int'l ACM Conf. Comp. Education Research*, ACM, 2013, pp. 53-58.

[5] Z. Shams and S. H. Edwards, ReflectionSupport: Java Refection Made Easy, to appear to appear at The Open Software Engineering Journal, TOSEJ, 2013.

[6] S.H Edwards. Improving student performance by evaluating how well students test their own programs. *J. Educ. Resour. Comput.***,** 3**(**3**):** Article 1**, 2003.**