Automated Assessment of Students' Testing Skills for Improving Correctness of Their Code

Zalia Shams
Virginia Tech
2202 Kraft Drive
Blacksburg
VA-24060,USA
zalia18@cs.vt.edu

ABSTRACT

Although software testing is included as a regular part of many programming courses, current assessment techniques used in automated grading tools for evaluating student-written software tests are imperfect. Code coverage measures are typically used in practice, but that approach does not assess how much of the expected behavior is checked by the tests and sometimes, overestimates the true quality of the tests. Two robust and thorough measures for evaluating student-written tests are running each students' tests against others' solutions(known as all-pairs testing) and injecting artificial bugs to determine if tests can detect them (also known as mutation analysis). Even though they are better indicators of test quality, both of them posed a number of practical obstacles to classroom use. This proposal describes technical obstacles behind using these two approaches in automated grading. We propose novel and practical solutions to apply all-pairs testing and mutation analysis of student-written tests, especially in the context of classroom grading tools. Experimental results of applying our techniques in eight CS1 and CS2 assignments submitted by 147 students show the feasibility of our solution. Finally, we discuss our plan to combine the approaches to evaluate tests of assignments having variable amounts of design freedom and explain their evaluation method.

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.

Keywords

Test driven development, automated grading, mutation testing, software testing,test coverage, reflection, bytecode transformation.

1. INTRODUCTION

Testing is an inaugural part of software development. Even though it accounts for 50% cost of software development, practitioners perceive testing as a tedious, uncreative, boring work and less than 15% of them ever receive any formal training in the subject [12]. Students are not accustomed to test their code. They usually focus on output correctness on instructor's sample data [5] and do less testing on their own [4]. Considering the necessity of testing, more educators are including software tests [3] in programming and software engineering courses [5, 8]. To support software testing as a part of regular programming assignments, current classroom assessment systems (e.g.,Web-CAT, ASSYST, Marmoset) allow students to turn in their programs along with tests. These automated grading tools evaluate student-written tests using coverage metrics. Code coverage measures the percentage of code—e.g., statements or branches—that is executed by running tests. The rationale behind code coverage is: the more code executed during testing the higher the chance of finding flaws in them. However, code coverage may falsely indicate test quality as it does not check if the executed code has been tested against expected behavior [1, 9]. Moreover a students' solution may be incomplete or incorrect. Large percentage execution of an incomplete solution will result in high code coverage missing all the omissions.

Two robust and thorough mechanisms for evaluating student written tests are: 1) *all-pairs* student testing, and 2) *mutation testing*. Even though they are strong indicators of test quality and adequacy, because of technical difficulties they are rarely used for assessing student-written tests, especially when programs are written in object-oriented languages such as Java. All-pairs testing [7] involve running every student's tests against the others' 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. However, implementation of this all-pairs model of executing tests is rare because student-written tests, such as JUnit tests written for Java programs, depend on individual aspects of the author's solution and may not compile against another student's program. Automated grading systems face similar issues when running instructor-provided JUnit-style reference tests against student submissions. Student solutions that fail to conform to the all requirements of the assignment cannot be compiled or assessed using reference tests. As a result, partial or incomplete submissions would have no results against reference tests, and most grading systems would assign no credit for the corresponding portion of the assignment grade. Thus, to run each student's tests against every other student's solution, a way must be devised to ensure a uniform interface against which tests can be executed regardless of differences between solutions or divergence from the assignment requirements. The other mechanism, mutation testing, seeds artificial errors into code (generating buggy versions called mutants) and then checks whether a test suite can detect them. Mutant score is computed from the number of mutants detected vs. number of mutant generated, and is used to indicate adequacy of test suites. Mutation testing is difficult to use in an educational setting for three main reasons [1]. First, mutation testing is computationally expensive and time consuming as it involves a manual

SPLASH'13, October 26–31, 2013, Indianapolis, Indiana, USA.
ACM 978-1-4503-1995-9/13/10.
http://dx.doi.org/10.1145/2508075.2508078

determination process of whether an injected error is a true bug or an innocuous solution. Second, mutants must be generated from a solution that is presumably correct and complete. A student solution is not a reliable candidate for mutant generation as it may be incorrect or partial. Third, students' tests may have dependencies on their own solutions and may fail to compile against mutants. Thus, to use mutation testing for assessment of student written tests and provide students immediate feedbacks, mutants should be generated from a solution that covers all the aspects of the assignment, they must be classified into true bugs or equivalent of an original solution using an automated efficient mechanism, and the mutants must run against students' tests irrespective of the tests' internal structure.

Application of all-pairs testing and mutation analysis will give students a deeper realization on the quality of their tests, which will help them thoroughly test their own code. As a result, they will learn to write software having fewer bugs.

2. PROBLEM

Automated grading tools allow students to turn in software tests along with their solutions. Assessment of the quality of student-written tests helps students learning software testing and programming at the same time. These grading tools evaluate their tests using code coverage which may overestimate test quality as it does not assess how much of the expected behavior has been checked correctly. Therefore, to assess true quality of students' tests we must identify: **Which test quality measures actually assess how much of the expected behavior is checked by the tests**?

Even though some test quality measures may work well in an industry setting, they may not be practical for evaluating students tests for three main reasons: 1) students should have varying amount of design freedom to learn mapping requirements to software modules, 2) they must get immediate feedback on their work, and 3) no formal tracking system for bugs or code changes is available for small class assignments. Therefore, we need to find out: **What are the practical obstacles of using identified test quality measures in an educational setting**? Once we identify the main problems, we will devise: **How can we resolve the obstacles to apply the measures in classroom tools**?

Since students have varying amounts of design freedom in their assignments, we will investigate: **Which approach is more appropriate for open-ended assignments, and what measure works better for close-ended assignments**? Finally, we will determine: **What combination of the approaches works well as a hybrid measure to separately evaluate tests of the assignments having variable amounts of design freedom**?

3. APPROACH

All-pairs testing and mutation analysis are two robust measures to evaluate the quality of student written tests. The main obstacle of using all-pairs testing 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, and are not present in others' code, then his tests will not compile against others' solutions. A novel way to resolve this issue in Java is to transform the student-written tests so that they use reflection to defer binding to specific features of a solution until run-time. Test sets that depend on the internal details on one particular solution can be compiled against the particular solution they were written for. For example, one 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. Similarly, instructors typically provide their own implementation to double-check reference test sets, so the reference tests will compile against the implementation. We transform the byte-code of the compiled test sets into reflective forms so that they use late binding. Reflection is a feature of Java that is used to reduce compile time dependency between code components. Hence, the byte-codes of the test cases are transformed using Javassist [2] into purely reflective forms. Java reflection can be complicated and error-prone to use, but we use ReflectionSupport [10], 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 compile and run against any student submission. Individual test cases that depend on features that are missing or indirectly declared in the student's work fail at run-time, while other test cases run normally. Therefore, the test-sets will run against all solutions, even if some of them are incomplete. Finally, we collect how many bugs a student-written test has revealed to assess its robustness.

The second approach, mutation testing, injects artificial bugs in the code and checks if test-cases can detect the bugs. Mutants (buggy versions) must be generated from a correct and complete solution. Usually instructors provide a reference solution, which is presumably correct and includes all the required features of the assignment, to an automated grader. We choose the reference solution to generate mutants. 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. Later, student-written test-cases are transformed to remove compile-time dependencies so they will run against any mutants. Afterwards, we validate students' tests against the reference solution and run mutants against only the valid tests. Mutants that produce different results from the original solution are considered as bugs. Our conservative process reduces computational overhead of manual differentiation, determining whether injected faults in a mutant are true bugs or a behavioral equivalent of an innocuous solution. Then, we evaluate the quality of a student's test from how many mutants it has detected.

All-pairs testing evaluates robustness and mutation analysis measures adequacy of the tests. The approaches may need some modification to evaluate open-ended assignments where large percentage of test-cases examines student-specific features. For example, to evaluate student-specific tests, we can use mutants generated from the same author's solution. Finally, using massive test-suites collected from all students' tests with the same assignments in different semesters, we determine if students are producing fewer bugs than they used to using code coverage, after getting feedback from all-pairs testing and mutation analysis.

4. EVALUATION METHODOLOGY

We identified that all-pairs testing and mutation analysis are better indicators of test quality than code coverage. Then, we investigated obstacles for using them in automated grading systems, and devised novel solutions to resolve the problems. We applied our solution for all-pairs testing in one CS1 and one CS2 assignments to evaluate our primary hypothesis.

The CS1 assignment had 46 submissions with 46 test sets consisting of 463 test cases. We removed compile-time dependencies from the test-cases and screened them against the reference solution. This resulted in a total of 18,225 individual tests after removing invalid or student-specific test cases. Every test case was passed by at least 65% of the programs and 63% of the submissions passed every test case written by every student. Keeping in mind that it was the first assignment for the beginners, we consider that students were able to write 35% effective tests that did uncover defects in many other submissions. The CS2 assignment had 101 student submissions with 101 test sets consisting of 2155 test cases. We used byte code translator to convert the test sets to their reflective versions like before. After validating them against an instructor-provided reference solution we got 2001 (92.9%) valid test cases and ran them against 101 solutions. The performance of the programs was representative of a more challenging assignment. The average portion of the test cases passed by a solution was 83.5%. Only five student programs passed all valid test cases, and it shows that students on the whole are quite capable of writing test cases that will reveal bugs. Unlike the CS1 assignment, there were ***no*** test cases that were passed by every program. Also, ***no*** transformed test suite failed to compile or run because of their dependency on the author's solution. To our knowledge, we are the first to successfully apply all-pairs testing [6] in automated grading.

To evaluate the practicality of our solution for mutation analysis, we applied it to six CS1 and CS2 assignments, where students were required to write their own software tests for each of their solutions. 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 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. Total number of valid test cases from the CS1 assignments was 672, where 42 among 47students completed all the assignments. In the CS2, 107 students completed assignments where valid student test cases were 2224. In all the assignments, the mutant detection ratio (M = 42.2% ~ 87.2%, sd = 13.5%~ 25.9%) was significantly lower (Wilcoxon signed rank test, 855.5+, $p < 0.0001$) than the test coverage achieved (M = 93.8%~96.9%, sd = 17.2%~7.7%). From this result, 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. 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 [11] in ICER, 2013. We are researching effective ways of generating feedback from the two approaches without revealing reference solutions or other students' solutions. Afterwards, we will develop a hybrid approach of all-pair testing and mutation analysis to evaluate student-written tests having variable amounts of design freedom.

Finally, to evaluate our secondary hypothesis we will create massive test-suites collecting all the students' tests over different semesters with the same assignments where students will receive feedback from three different measures: code coverage, all-pairs testing, and mutation analysis. We will analyze defect density, the number of bugs per thousands non-commented line of code, of students' solutions when they receive feedback from the three different approaches.

Our research outcome will provide educators and students insight into the quality of students' testing skills. Implementation of our solution to remove compile-time dependencies from the test cases will enable automated graders to evaluate partial solutions. Application of all-pairs testing and mutation analysis will encourage students to practice testing skills in many classes and will give them concrete feedback on their testing performance. As a result, students will learn to test their code well which will improve accuracy of their solution.

5. REFERENCES

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

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

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

[4] Edwards, S. H. 2003. Using test-driven development in the classroom: Providing students with concrete feedback . *Proc. of the International Conference on Education and Information Systems: Technologies and Applications.*

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

[6] Edwards, S. H. *et al.* 2012. Running students' software tests against each others' code: new life for an old "gimmick". *Proc. of SIGCSE'12* , pp. 221-226, Raleigh, NC, USA.

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

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

[9] Schuler, D. and Zeller, A. 2011. Assessing Oracle Quality with Checked Coverage. *Proc. of ICST'11*, pp. 90-99.

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

[11] Shams, Z., and Edwards, S. H. 2013. Toward Practical Mutation Analysis for Evaluating the Quality of Student-Written Software Tests. *Proc. of ICER*, pp. 53-58.

[12] Wilson, R. C. 1995. *UNIX test tools and benchmarks: methods and tools to design, develop, and execute functional, structural, reliability, and regression tests*: Prentice-Hall, Inc.