Evaluating Students' Assignments by Assessing How Well Students Test Their Own Code.

Zalia Shams, zalia18@cs.vt.edu, Virginia Tech
Advisor: Stephen H. Edwards, Edwards@vt.edu, Virginia Tech

Problem and Motivation:

Testing accounts for 50% cost of software development. Because of the necessity of testing, educators are including more and more software testing[1] in programming [2] 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., branches or statements—that are executed during a test. It may falsely indicate test quality because code execution may not be initiated from a test oracle (e.g., assert statement) and the solution code may be partial, incomplete, or incorrect. 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 involve running one student's program against the other students' tests, which provides a more thorough mechanism for evaluating the quality of that solution, possibly revealing bugs that might be missed by instructor-written tests. 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, commonly include dependencies on individual aspects of the author's solution and may not compile against another student's program. Therefore, a uniform mechanism (e.g., textual interface) is required to run any student's tests against any other program, regardless of its internal design. However, JUnit tests do not work that way. 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 written in JUnit-style with 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. Test suites that detect more errors are better than those that detect less. However, mutation testing is computationally expensive and time consuming. It is not suitable to use in the classroom where immediate feedback is needed. Moreover, students' written tests might have dependencies on their solutions and fail to compile against mutants. Thus, to use mutation testing for assessment of student written tests, mutants should be generated from a solution that covers all the aspects of the assignment and that will run against all students' tests. Also, computational cost should be paid in advance so that immediate feedback can be generated.

A combination of the two mechanisms, mutation testing and all-pairs testing, is even better as an approach to assess student written tests. Generating mutants from instructor written solutions and running students' tests against them will indicate adequacy of the test suites. All-pairs testing will show how robust the tests are in detecting bugs that are actually present in the students' solutions.

Background and Related Work:

Students take software testing as a boring task. They usually focus on output correctness on instructor's sample data [3] and do less testing on their own [5]. To incorporate software testing as a part of coding, Goldwasser [6] 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 program. Widely used automated assessment tools (e.g.,Web-CAT [7], ASSYST[8] and Marmoset [9]) evaluate students' codes along with their tests.

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 students' score based on three factors: percentage of instructor's reference tests passing, percentage of students' own tests passing, and code coverage of students' tests. 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[9], another automated grading system for Java assignments, evaluates student programs against two test sets: 1) public test sets, where students see all results, and 2) "release" test sets provided by the instructor—that is, private instructor-written reference tests. 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. Both test results and feedbacks obtained from the release test sets are delayed and limited. Marmoset investigates code coverage from the release test sets and from the student's own test sets. This system gives students feedback about their programs using the static program analysis tool FindBugs[10]. All the three 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 or that have incorrect operation signatures. These systems cannot evaluate such partial or incomplete submissions and gives no credits to students. As a result, they are not capable of all-pairs test execution. To our knowledge, we first provided a solution for all-pair test execution by transforming test-cases so that they have no compile time dependency. An earlier version of our work is published at SIGCSE 2012[11].

All three systems, Web-CAT, ASSYST and Marmoset, use 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.* [12] propose using mutation analysis to assess the quality of student-written tests in Java assignments. Mutation analysis seeds artificial errors (mutants) in source code—in this case, in the student's own solution—and then runs the student-written tests to see if the seeded defects are detected. This approach provides a deeper perspective on the adequacy of the student-written tests, going well beyond code coverage metrics, but computational overhead makes it impractical

to use for real-time feedback generation. Moreover, this system generates mutants from a student's solution and runs his test cases against those mutants. As a result, proposed mutation analysis is implementation specific and does not ensure compatibility across all students' test scores.

Approach and Uniqueness:

We provide a Java-specific solution to run student-written test cases against instructor-provided reference solutions and across their programs. This novel solution uses byte-code rewriting to transform test suites into a form that can be applied against any solution. 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 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[13] into purely reflective forms. Java reflection can be complicated and error-prone to use, but we use ReflectionSupport[14], 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 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. Hence, the portion of the student's work that was completed can be assessed, even if other required features are missing. ReflectionSupport library provides appropriate diagnostic messages for test cases that fail because of missing or broken features, reporting to the student the specific class and/or method that is missing or incorrectly declared.

Transformed student-written test suites are run against instructor provided reference solutions first. Three cases may arise from the execution:

1. A test suite may pass.
2. A test suite may fail inside the reflection infrastructure, indicating some dependency on that student's written solution.
3. A test suite may fail, indicating incorrect behavior.

The first and third categories are valid and invalid tests respectively, whereas the second type of test case possesses student-specific dependencies. We collect valid test suites only and run them against all students' solutions. Test cases that reveal more bugs are better than those that reveal less. Using this technique in the classroom gives students an eye opening effect on the density of bugs in their own code and their own ability to find defects in others' solutions. To evaluate adequacy of their tests, mutation analysis is used.

In the mutation analysis, we create mutants from the instructor provided reference solution. Mutants can be generated at the source code level or at the byte-code level. We generate byte-code mutants using our modified version of Javalache[15]. The original version of Javalanche internally generates mutants, runs a list of given test suites, and analyzes coverage of the test

suites. It does not store mutants and regenerate them every time a new test suite is given to the tool. We modified this tool to generate mutants only and to store them in a file so that when a new test suite is submitted by a student, we can reuse the mutants. Mutation analysis is computationally expensive as there is no automated mechanism to weed out equivalent mutants. We generate mutants from the reference solution upfront and run reflective versions of students' valid test cases (certified by reference solution) are run across mutants. Mutants that produce same result as the original reference solution are considered equivalent. This strategy is based on the fact that, equivalent mutants show same behavior as the original code. Though our automated mutant detection procedure was conservative, it reduced computational cost a great amount. It allows us to produce real-time feedback and to use mutation analysis in the classroom. It also ensures compatibility of test score as all the tests run against the same mutants.

Results and Contributions:

This is an ongoing research. We applied our solution for all-pairs testing in two programming assignments and student written test sets, in two different courses, Fall 2007 CS1 and Spring 2012 CS2.

The CS1 assignment had 46 submissions with 46 test sets consisting of 463 test cases. All the test cases were transformed to reflective tests and were run against an instructor-provided reference solution. Among 463test cases, 405 (87.5%) were passed, 27 (5.8%) showed incorrect behaviors and 31 (6.7%) were student specific. Next, all 46 test sets were executed against all 46 solutions in approximately 30 minutes. 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 submissions as shown in Figure 1. Figure 2 shows that 63% of the submissions passed every test case written by every student, and every submission passed at least 50% of the test cases. These high pass rates are typical of an assignment that students perform well on, as was the case here. While just over 50% of test cases failed to find any defects, the remaining test cases varied in effectiveness and did uncover defects in many other submissions. Also, **no** transformed test suite failed to compile or run because of their dependency on author's solution.

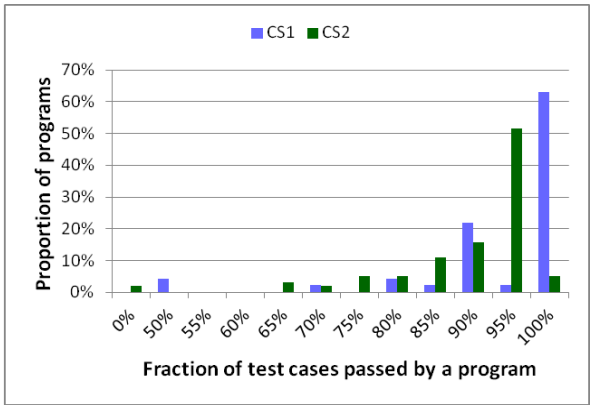

Figure 1: Distribution of program performances

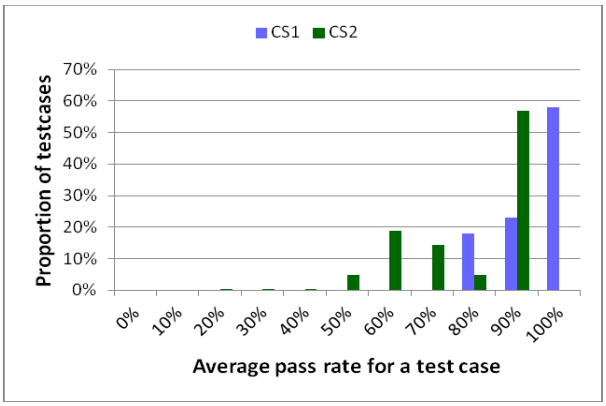

Figure2: Distribution of pass rate for test cases

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 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. The process to run 101 test sets against 101 solutions took approximately two hours and produced a total of 200,100 tests run after excluding invalid and student specific test cases. 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. However, over half the class passed 90% or more of the test cases, with a rapid drop-off in number of students for successively lower test case pass rates. The distribution of test case pass rates shown in Figure 2 illustrates 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.

We used the transformed test cases of CS1 assignment for mutation analysis. First, we generated 27 mutants from instructor provided reference solution. We run the mutants against 405 valid test cases. It took about 15 minutes. All the 27 mutants were detected as non-equivalent. Every test suite was able to detect at least one mutant. Average mutation detection rate was 87.21% and 83.33% test suites were able to detect 90% or more non-equivalent mutants. Thus, students performed well in writing adequate test suites in their first assignment. We also examined composite (branch and statement) code coverage (of students' solutions) of the test suites. Code coverage outcome is similar to mutation testing. In fact, correlation between composite code coverage was 0.7779. We are conducting experiments to get insight into what percentage of the executed code (calculated by code coverage) was actually initiated from an oracle (e.g., assertion statement) in students' written tests.

From the above results, it is clear that it is indeed possible to remove compile-time dependencies automatically, replacing them with dynamic resolution using reflection. This resolves the primary obstacle to the execution of student-written tests against other's code and to run reference test cases against complete or partial student submissions in the context of JUnit tests. Similar techniques can be applied in other languages where reflection or interpretation is available. We proved that it is feasible to apply mutation testing to evaluate student-written tests. We automatically detected equivalent mutants in a conservative but efficient way, and assessed adequacy of test cases. Application of these mechanisms will give students a greater realization of the density of bugs in their codes and their ability to write quality tests that find defects in other solutions, and will equip them well for future carrier.

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-driven 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] 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.

[6] M. H. Goldwasser, "A gimmick to integrate software testing throughout the curriculum," *SIGCSE Bull.*, vol. 34, pp. 271-275, 2002.
[7] 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.
[8] D. Jackson and M. Usher, "Grading student programs using ASSYST," *SIGCSE Bull.*, vol. 29, pp. 335-339, 1997.
[9] 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.
[10] B. Cole, *et al.*, "Improving your software using static analysis to find bugs," 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] 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.
[13] 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.
[14] *ReflectionSupport*. Available: http://sourceforge.net/projects/web-cat/files/ReflectionSupport/,last accessed on 04/15/2013
[15] *Javalanche*. Available: https://github.com/david-schuler/javalanche/, last accessed on 04/15/2013.