Automated Feedback Framework for Introductory Programming Courses

Jianxiong Gao
University of Illinois at Urbana-Champaign
1308 W Main St
Urbana, USA
gao2@illinois.edu

Bei Pang
University of Illinois at Urbana-Champaign
1308 W Main St
Urbana, USA
beipang2@illinois.edu

Steven S. Lumetta
University of Illinois at Urbana-Champaign
1308 W Main St
Urbana, USA
lumetta@illinois.edu

ABSTRACT

Using automated grading tools to provide feedback to students is common in Computer Science education. The first step of automated grading is to find defects in the student program. However, finding bugs in code has never been easy. Comparing computation results using a fixed set of test cases is still the most common way to determine correctness among current automated grading tools. It takes time and effort to design a good set of test cases that can test the student code thoroughly. In practice, tests used for grading are often insufficient for accurate diagnosis.

In this paper, we present our utilization of industrial automated testing on student assignments in an introductory programming course. We implemented a framework to collect student codes and apply industrial automated testing to their codes. Then we interpreted the results obtained from testing in a way that students can understand easily. We deployed our framework on five different introductory C programming assignments here at the University of Illinois at Urbana-Champaign. The results show that the automated feedback generation framework can discover more errors inside student submissions and can provide timely and useful feedback to both instructors and students. A total of 142 missed bugs were found within 446 submissions. More than 50% of students received their feedback within 3 minutes of submission. We believe that based on the current automated testing tools, an automated feedback framework for the classroom can benefit both students and instructors, thus improving Computer Science education.

Keywords

Computer Science Education; Concolic Testing; Auto Grader

1. INTRODUCTION

The ability to program has become increasingly important in recent years. As a result, more people are trying to learn programming. Becoming a good programmer requires both practice and feedback. The rate at which a novice programmer makes progress depends heavily on how much instructive feedback the programmer receives from qualified instructors. However, skilled programmers who can provide such feedback are a limited resource.

In practice, basic input-output testing is the most common way of assessing student programs. Student programs are executed with a set of test inputs, and the results are compared with correct outputs. However, it takes time and effort to design an adequate set of test cases. Designing test cases that can cover most of the possible bugs is hard, especially for introductory level programming classes. Novice programmers are not as predictable as experienced programmers. The behavior of novice programmers' code is more unpredictable. Thus the task of designing test cases is never as trivial as it sounds. For each of the programming assignments examined in this paper, there were unexposed bugs in student code that were not caught by the normal grading procedure.

This paper describes a framework that uses automated testing tools to detect defects in student code and to provide feedback on those defects in the form of specific examples of incorrect behavior. During the development process, students can submit their code for review by the framework. Feedback is generated within minutes of student code submission.

The technique that our framework uses to identify defects is called concolic testing [8]. Concolic testing executes a program both concretely and symbolically. The aim is to maximize code coverage. In other words, to create tests that execute all parts of a program. The result of concolic testing is a set of inputs that covers different parts of the code. Concolic testing has been successfully commercialized and adopted by the software industry. For example, Microsoft utilizes concolic testing heavily in its Sage tool [3].

We deployed our framework as an automatic feedback generation tool in ECE220, one of the introductory programming courses at the University of Illinois at Urbana-Champaign. All Electrical and Computer Engineering (ECE) students take this course, which thus has roughly 350 students every semester. Our results show that we can identify more defects in student code than can traditional automated grading. Students can learn from the failed test cases generated and can improve their code. Experiments show that the mean processing time for a recursive maze solver is 260 seconds, meaning that one can scale this solution to pro-

ITiCSE '16, July 09 - 13, 2016, Arequipa, Peru

ISBN 978-1-4503-4231-5/16/07...$15.00

DOI: http://dx.doi.org/10.1145/2899415.2899440

vide directed and specific feedback based on a particular student's code within a few minutes.

Compared to current tools, the automated grading tool proposed in this paper has several benefits:

- Improves ability to identify code defects.
- Generates test cases leading to defects as feedback.
- Improves grading fairness.
- Has Comparable processing time to current methods.

We provide an overview of the sections of this paper. Section 2 gives additional detail on our motivation for building a framework based on industrial tools. Section 3 gives background information on concolic testing. Section 4 describes our framework and workflow for applying concolic testing to automated grading. Section 5 describes our experiment with student assignments. Section 6 discusses the results. We conclude in Section 7.

2. MOTIVATION

Teaching programming is hard. The dropout rate of introductory programming courses can be as high as 50% [9]. Because more students tend to take such class, the number of students enrolled in introductory programming courses has been growing in recent years. Outside of universities, people are also in massive online open courses (MOOC) to learn programming. Due to the large number of students, these introductory programming courses are usually taught in a lecture+laboratory structure [4]. Lectures are important for students to learn concepts, language definitions, and common mistakes. However, the most important part of learning engineering is to practice.

For programming courses, students practice by doing programming assignments. When students finish their programming assignments, they need to know whether their programs are correct. However identifying defects in code is not an easy task. Even experienced programmers in industry produce programs that have bugs in them. In industry, code review helps programmers to get feedback about their work. In programming courses, students are supposed to get feedback about their work from instructors. Human inspection is yet still heavily used as a way of code review. Ganssle [2] and Kemerer and Pavik [6] reported that an ideal rate of 150-200 lines per hour is most effective for manual inspection. For student code, the efficiency may be even less. However, manual inspection can only be done by experienced programmers. With large class sizes, the process requires hiring more teaching assistants, who cost less than instructors, but are more variable in the quality of their code review. The increasing number of students learning programming and the limited time of instructors are causing problems in programming education.

3. CONCOLIC TESTING

Concolic testing [7] is a software verification technique that combines symbolic execution with concrete values. The goal of concolic testing is to generate test cases that can achieve high code coverage. Concolic testing starts with a set of variables that are marked as symbolic. Then random concrete values are generated for these variables. The program executes using the concrete values, and the control flow is recorded. Code that has been executed as part of the program execution is marked as covered. In order to expand the covered part of the program, one symbolic constraint in the path is negated and solved using SMT solvers. A new set of concrete values is thus generated that directs the program into a different execution path. The process continues until no more execution paths can be explored.

Concolic testing combines random testing and symbolic execution. Concrete values are used to overcome some limitations of fully symbolic execution, while symbolic execution is used to generate better coverage than random testing.

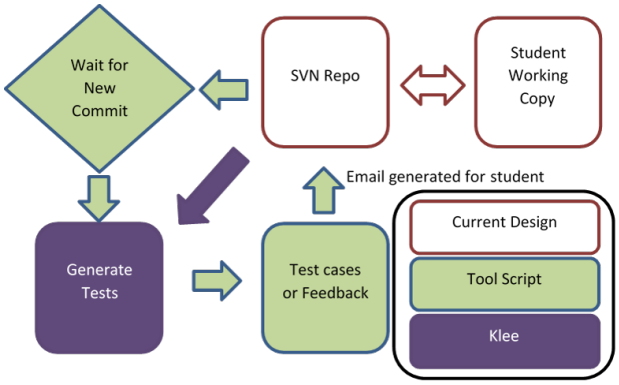

Figure 1: Collect Student Code/Generate Feedback: We implemented tool scripts to react to student commits. When a student commits a new version, we generate tests and send feedback to the student.

4. FRAMEWORK SETUP

The framework developed in this paper consists of two main parts. The frontend collects student-submitted code, initiates the grading process on the backend, distributes graded code back to students and notifies students. The backend first compiles student code into LLVM byte code, then executes the student code using KLEE [1], a concolic testing tool. If there is any defect inside student code, a test case leading to that defect is generated.

4.1 Frontend

The system implemented for students to submit their code for feedback revolves around the flow diagram in Figure 1. The framework implements a system that assesses the student code and distributes feedback to students. When new submissions are detected, the system pulls the latest commit of student work from the Subversion server. The student work is passed to the backend for assessment. The backend assesses student work and generates test input sets. After test inputs are generated, we collect the test input sets that lead to errors in student code. We analyze the test input sets and generate a feedback message to the corresponding Subversion repository. After the grading process is finished, we generate an email to the student to notify the student that the auto-grading process has finished. On receiving the email, the student can collect generated feedback from Subversion. The whole frontend system is written in Ruby.

4.2 Backend System: KLEE

The work flow of the backend system is shown in Figure 2. The backend system takes two inputs: a student solution

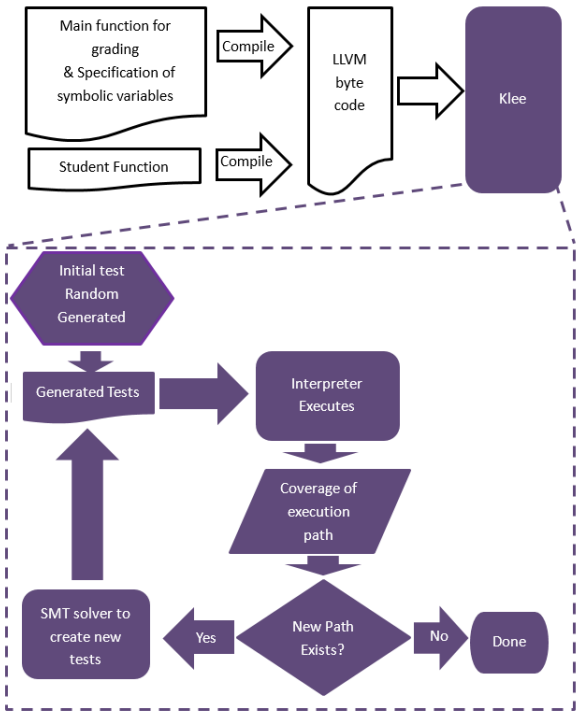

Figure 2: Workflow: Main function (contribution of paper) and student function are compiled into Llvm byte code. The KLEE is applied to generate tests that can cover the most execution paths.

```
#include "stdio.h"
#include "assert.h"
int gold_calculate(a,b,operator){
  if(operator=='+'){
    return a+b;
  }
  else if(operator=='-'){
    return a-b;
  }
  return 0;
}

int main(){
  int a,b;
  char operator;
  klee_make_symbolic(&a, sizeof(a),"a");
  klee_make_symbolic(&operator,
      sizeof(operator),"operator");
  klee_make_symbolic(&b, sizeof(b),"b");
  if(stu_calculate(a,b,operator) !=
    gold_calculate(a,b,operator)){
    assert("Student Calculate Function Error.\n");
  }
  return 0;
}
```

Figure 3: Symbolic Variable Marked: Selected variables are marked as symbolic using the *klee_make_symbolic* function. Different sets of values are generated and assigned to these variables to explore all the possible execution paths. Each set of values forms a test case.

and a wrapper main function. Both input files are compiled into LLVM byte code. This LLVM code is then passed to KLEE for assessment.

In order for KLEE to work, some setup work needs to be done. The setup work is divided into two parts: marking symbolic variables and checking functional correctness. Symbolic variables are used to generate test inputs and are also the key to achieving maximum coverage. To check functional correctness, we need to verify that the program can achieve the desired goal. KLEE by itself only tries to achieve high code coverage. The testing of functional correctness of the program is not the aim of KLEE, but is supported by our wrappers and checking code.

4.2.1 *Selecting Symbolic Variables*

The symbolic variables are usually chosen from inputs to the student program. Figure 3 is an example calculator program. The inputs are two numbers and an operand. To use KLEE to find maximal coverage, we need to mark the two numbers and the operand as symbolic. Marking these variables as symbolic can be done by calling a function call *klee_make_symbolic*. All the initialization of a,b and operator is now done by KLEE.

4.2.2 *Functional Correctness Checking*

KLEE was designed to achieve high code coverage, so by default KLEE only generates test cases that can cover most of the code. To verify that the program under test has the desired behavior, we add post-condition checking with the *assert* function. A predicate that checks the result of the student function can be used. An assert function can be done on the predicate. KLEE will try to a find path that leads to an assert failure (to cover that code). In Figure 3, the return value of student function *stu_calculate* is checked against the return value from the reference implementation *gold_calculate*. Because KLEE uses an SMT solver to find new execution paths, the SMT solver tries to solve for a set of values that can lead the execution to the *assert* function. These assertions are effectively branches under the execution of KLEE. The SMT solvers effectively checks for functional correctness by trying to maximize code coverage. If any execution path that leads to an assertion failure exists, the student program has functionality bugs. Assertions are usually added at the end of the wrapper code, after the execution of student code.

5. EXPERIMENTS

The automated feedback framework has been applied to five different assignments here at the University of Illinois at Urbana-Champaign. All five assignments are for an introductory level programming class, ECE220. The framework has successfully done the job of providing quick and accurate feedback to students. Here in this section, some background information about the assignments and the feedback framework is provided.

5.1 Design of Assignments

These programming assignments are designed by the in-

structors of the course without considering any aspect of our framework. Thus, these assignments are not designed specially for the automated feedback framework. The automated feedback framework works in parallel with the normal grading process, which forms the basis against which we compare our results. Student grades are not affected by the feedback and are solely decided by the normal grading procedure. Even if additional bugs are identified inside student code, this information is reported to the student only, but not to the instructors.

Also, the normal grading procedure is mostly kept unknown to us. The only access that we have is to the test cases released to students. Any additional grading test cases used at the time of final grading are not released.

5.2 Assignments

There are 5 different assignments in total, each of which is now explained in detail:

Code Breaker The Code Breaker assignment requires students to implement the logic for a code-breaking game. A code sequence of four numbers from 1 to 8 is first chosen at random. These four numbers are the solution code, and are also referred to as pegs. The player guesses a sequence of four numbers and is given feedback on each guess, including the number of correct values that appear in the same place in the solution code (these are called perfect matches). The user is also told the number of values that appear in a different place in the solution code (these are called misplaced matches). Values guessed are matched pairwise with the solution, so a given guess value can count either as a perfect match or as a single mismatch, but cannot count as both types, nor as multiple mismatches. Similarly, a given solution value can only count as one match of one type. Functions to implement: set_seed, start_game, and make_guess.

Learning objectives: if statement, logical operators, function calls. Typical student code length: 100 lines.

Image Editor An image is represented as four 1-D arrays, one for each channel (Red, Green, Blue, and Alpha). Each array has the same number of elements as there are pixels in the image. In student functions, these arrays are passed as pointers (for example uint8_t *inRed, uint8_t *inBlue, uint8_t *inGreen, uint8_t *inAlpha). Students are required to implement five functions that modify the image: calculate_cosine_filter, invert_image, convolve_image, convert_to_gray, color_thr- eshold.

Learning objectives: for loops, arrays. Typical student code length: 100 lines.

Sudoku Game The Sudoku assignment is to implement a C program that solves a standard Sudoku puzzle using recursive backtracking. Functions to implement: is_val_in_row, is_val_in_col, is_val_in_3x3_zone, int is_val_valid, solve_sudoku.

Learning objective: recursion. Typical student code length: 150 lines.

2048 Game Students are required to implement the 2048 game with an arbitrarily-sized game board. The game 2048 is played on a grid, with numbered tiles that slide smoothly when a player moves them using the four arrow keys. Every turn, a new tile appears randomly in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they merge into a tile with the summation of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move.

Functions to implement: make_game, destroy_game, get_cell, move_up, move_down, move_left, move_right, legal_move_check, remake_game.

Learning objectives: structures, dynamic allocation (malloc and free). Typical student code length: 200 lines.

Maze Solver Students are required to find a solution for a maze game. An acyclic map of the maze is given to them. The starting location is marked as "S", and the ending location is marked as "E". If the maze is solvable, any point along the solution path has to be marked as ".". Other locations that have been searched but are not on the solution path have to be marked as "∼". The starting and ending location symbols "S" and "E" cannot be overwritten.

Functions to implement: find_start, print_maze, solve_maze_DFS.

Learning objective: recursion. Typical student code length: 250 lines.

Automated feedback generation for Maze Solver was applied in both Spring and Fall semesters of 2015, while the rest were applied to the Fall 2015 semester only. For both semesters, students were asked to volunteer to allow us to provide real-time feedback depending on their code. The process is totally voluntarily and does not affect the normal grading procedure at all. Participating students do not need to do any extra work. For Spring 2015 semester, 82 out of 349 students volunteered; for Fall 2015, 91 out of 393 students volunteered to participate in the experiment.

6. RESULTS

In this section, the quality of the generated feedback is discussed. Because students are allowed to use the automated feedback framework multiple times before the deadline, multiple versions of feedback may exist. Only the last submission by each student, the versions that are actually graded, are considered in this section.

Functional correctness is the most important aspect of a program. Missed bugs inside student code may leave students with misunderstood concepts. The quality of the generated feedback is assessed based on the ability to find defects inside student codes. The results show that the automated feedback framework does a better job than the original grading procedure at discovering defects.

With regard to the time required for creating grading scripts and processing student submissions, space allows only a few brief comments. An undergraduate teaching assistant was able to produce a grading script for an assignment in less than eight hours. And more than half of student submissions received feedback in less than three minutes. See [5] for more detail.

solution peg	1	2	3	4
guesses	1	1	5	6

Table 1: Example for Code Breaker

6.1 Code Breaker

For the Code Breaker assignment, every student submission with defects reported was human-inspected to assess the quality of auto-generated feedback. Among 91 total submissions, three pairs of duplicate code were found, thus only one copy for each pair is considered in the discussion.

6.1.1 Failed to Generate Feedback

The framework failed to generate feedback for 5 out of the 88 valid submissions. Following is a list of failure reasons:

Over time The processing time limit was 3 minutes. Two functionally correct but unnecessarily complex submissions timed out.

Printf Two students used *printf* with the wrong number of arguments.

Set Seed The *set_seed* function should be called only once to set the game up. One submission called *set_seed* multiple times, causing the *rand* function to return erroneous results.

6.1.2 Bugs Missed by Normal Grading

Sixteen student submissions containing bugs passed all tests in the normal grading procedure. Ten of these exhibited the Multiple Mismatch Peg bug, three failed to write back to the output variables, one accessed out-of-bounds pointers, one failed to check negative input values, and two used uninitialized values.

The most common bug was the Multiple Mismatch Peg bug, in which multiple input pegs are counted as mismatches with a single solution peg. As a result the code reports an inflated mismatch value. For the example in Table 1, the correct output should be 0 perfect matches and 1 mismatch. With the Multiple Mismatch Peg bug, the number of mismatches is calculated as 2.

The cause of the bug is usually that when student submissions match an input guess to a solution peg, the solution peg is not marked as being paired with one of the input guesses. In the example, the solution peg with value 1 did not get marked when pairing with any of the 1s in the guesses, so the other guess could be matched with the same solution peg, which results in 2 mismatches instead of 1.

The multiple mismatch peg is also the most common bug missed by the class' normal grading procedure. Ten student submissions, or 63% of the submissions that had this bug, passed the normal grading procedure. Six more student submissions with this bug were caught because of other bugs, while this bug was never exposed by the normal grading procedure.

6.2 Image Editor

Among the student submissions that were determined to be correct by the normal grading procedure, the checking of index bounds for array accesses alone identified 30 buggy submissions. The normal grading procedure only compared the end result, which led to the failure to detect this kind of bug.

```
if(...){
  if(...){
      return 1;
    }
}
else{
  return 0;
}
```

Figure 4: Example Code for Sudoku Game

6.3 Sudoku Game

6.3.1 Compile Failure

Five student submissions failed to compile, of which one failed to compile with the normal grading procedure as well. The main reason for compilation failure is compiler differences. The normal grading procedure used g++ as the default compiler, while the auto-grading framework used llvm-gcc. All five students used C++ syntax supported only by g++.

6.3.2 Time Limit

Because the Sudoku game assignment is a recursion assignment, the number of execution paths may become extremely large with erroneous student submissions. Though the execution time limit was set for 5 minutes, 17 student submissions failed to finish. However, for 4 of the timed-out submissions, bugs were identified within the student code before timing out.

6.3.3 Bugs Missed by Normal Grading

The most frequently missed-bug for the Sudoku game assignment was implementation of a challenge problem (not required for full credit). The specification clearly stated that the check for number uniqueness on diagonals should only be done outside of the *is_val_valid* function. However, 37 student submissions had the diagonal check included inside the *is_val_valid* function. The normal grading procedure did not test if the diagonal check was implemented outside the *is_val_valid* function. There was no individual test for the *is_val_valid* function at all.

Another two missed bugs were related to the problem of misplaced return values. One example is given in Figure 4. Depending on calling convention, if the return value of a function happens to be 0, the functionality of the buggy code is still correct. It is assumed that the normal grading procedure used a compiler that happens to have the return value as 0, as the student submissions passed the test.

6.4 2048 game

6.4.1 Compile Failure

There were two compilation failures; both student submissions failed the class' grading procedure as well.

6.4.2 Out of Time

There were 28 student submissions that ran out of time for automated feedback. Of these, 18 were determined to have errors by the normal grading procedure. With human inspection, all the 10 correct student submissions had unnecessarily complex algorithms for the *move_* functions.

6.4.3 Bugs Missed by Normal Grading

The most obvious missed bug was the a lack of test cases for invalid moves. When a move is invalid, the *move_* functions should return 0. By default, the *move_* functions return 1 in the given code. Missing test cases on invalid moves caused two students, with their *move_up* function empty except for the "return 1" statement, to pass the return value test for the normal grading procedure. One other student submission, which was checked with invalid moves but failed to return 0, also passed the normal grading procedure.

Two more student submissions containing logic errors were found to pass the normal grading procedure. The test cases for the normal grading procedure simply failed to discover these logic errors.

Fourteen student submissions were determined to have out-of-bounds pointer errors.

6.5 Maze Solver

The automated feedback framework tested $find\_start$ and $solve\_maze\_DFS$ successfully. However for *check_maze*, even though most of the defects were identified, some bugs were still missed. By inspection, the reason that these bugs were missed is that the precondition was set to test valid solutions only. In other words, our choice of symbolic variables only generated solvable mazes to test student functions. A solvable maze is not enough to identify all of the defects in the *check_maze* function. To identify all the defects, both solvable and unsolvable mazes need to be generated.

6.5.1 Defects

For the Spring 2015 semester, with 82 students volunteering, a total of 241 submissions were graded, 141 of which contained test cases leading to errors generated. Only 16 students have no generated test cases for all of their submitted code. More than 80% of the students had defects in their submissions. After we provided feedback to them, 50 out of 82 students passed our grading tool, a 42% increase in correction rate. Among the 32 buggy submissions, 17 passed the normal grading procedure.

6.6 Summary

The traditional way of grading relies on developing a comprehensive set of test inputs. When test inputs do not cover defective code, the grade does not reflect the defects. For all of the five assignments tested, student submissions with bugs that the normal grading tests did not cover have been identified. A list of the most important missed bugs follows:

Code Breaker Multiple Mismatch Peg.
Image Editor Out of Bound Array Accesses.
Sudoku Game Return Value not set.
2048 Game General logic error.
Maze Game Adjacent Starting and Ending location.

The normal grading procedure also failed to test some aspects of the student submissions according to the specification.

Code Breaker Write back to the output variables.
Sudoku Game Diagonal test should be implemented outside the *is_val_valid* function.
Maze Solver Maze map with no outside wall.

The feedback is for instructors as well. By monitoring the feedback generated, instructors can learn about common errors and address these errors in lecture. Instructors can provide more detailed explanation about the common errors than the default generated message from the framework.

7. CONCLUSION

The use of industrial automatic testing tools on automated grading has been limited. With the implemented framework that utilizes concolic testing tools to identify defects in student code, and that generates feedback to students, students get a better understanding of the course material. The result shows that industrial automated testing tools can identify defects in student code.

With the framework, students can get timely feedback on their submissions. The mean processing times for five assignments in an introductory course are all under 5 minutes. There is also more coverage of student submissions than with the traditional automated grading tool. Results show that more than half of defective student code submissions passed the traditional automated grading tool. Thus, further exploration into the application of an industrial automated testing tool in computer science education should be conducted.

In the future, it would be beneficial to tune these tools toward a better fit for automated assessment. Also, because concolic testing is based on an interpreter, some of the common library calls can be replaced with an instructor-provided versions. More implementations for system calls such as malloc and free are desired in the future.

8. ACKNOWLEDGEMENTS

This project was supported by the Strategic Instructional Innovations Program of the College of Engineering at the University of Illinois at Urbana-Champaign.

9. REFERENCES

[1] C. Cadar, D. Dunbar, and D. R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[2] J. G. Ganssle. A guide to code inspections, 2001.

[3] P. Godefroid, M. Y. Levin, and D. Molnar. Sage: whitebox fuzzing for security testing. *Queue*, 10(1):20, 2012.

[4] I. Huet, O. R. Pacheco, J. Tavares, and G. Weir. New challenges in teaching introductory programming courses: a case study. In *Frontiers in Education, 2004. FIE 2004. 34th Annual*, pages T2H–5. IEEE, 2004.

[5] G. Jianxiong. Auto grading tool for introductory programming courses. Master's thesis, University of Illinois, Champaign, 2015.

[6] C. F. Kemerer and M. C. Paulk. The impact of design and code reviews on software quality: An empirical study based on PSP data. *Software Engineering, IEEE Transactions on*, 35(4):534–550, 2009.

[7] K. Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM International Conference on Automated Software Engineering*, pages 571–572. ACM, 2007.

[8] K. Sen and G. Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Computer Aided Verification*, pages 419–423. Springer, 2006.

[9] A. Yadin. Reducing the dropout rate in an introductory programming course. *ACM Inroads*, 2(4):71–76, Dec. 2011.