AUTO GRADING TOOL FOR INTRODUCTORY PROGRAMMING COURSES

BY

JIANXIONG GAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2015

Urbana, Illinois

Adviser:

Associate Professor Steven S. Lumetta

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. Current automated grading tools do not focus on identifying defects inside student code. 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.

Meanwhile, automated testing tools have been developing for some time. Even though it still takes some effort to apply automated testing tools to real software development, we believe that automated testing tools are ready for automated feedback generation in the classroom. The reason is that for classroom assignments, the code is relatively simple. A well understood reference implementation provided by the instructor also makes automated testing tools more effective.

In this thesis, 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 interpret the results obtained from testing in a way that students can understand easily. Furthermore, we use the test results to classify erroneous student codes into different categories. Instructors can use the category information to address the most common conceptual errors efficiently.

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 are found within 446 submissions. More than 50% of students receive their feedback within 3 minutes of submission. By doing grouping on one of the assignments with 91 submissions, two groups of student submissions of 15 and 6 are identified to have the same type of error. The average grading code setup time is estimated to be less than 8 hours for each assignment. 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.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I would like to express my gratitude to my advisor, Prof. Steven Lumetta, for the useful comments, remarks and support through the learning process of this master's thesis.

Furthermore I would like to thank Prof. Sayan Mitra for his guidance and support on both theory and application of this thesis.

Also I would like to thank Andy Gong and Bei Pang for their collaboration and discussion, as well as for their work.

Last but not least, I would like to thank my family and loved ones, who have supported me throughout the entire process, both by keeping me positive and by helping me put pieces together. I will be grateful forever for your love.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

In this thesis we present an automated feedback framework for introductory level programming classes. We believe that by providing high quality and timely feedback to both students and instructors, we can improve the quality of computer science education.

The ability to program is increasingly important in recent years, with application in a broad range of careers. As a result, interest among university students in learning how to program has also risen, and more people are trying to learn programming on their own in order to gain an edge in their careers. 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, the most common way of assessing student programs is basic input-output testing. Student programs are examined with a set of test inputs, and the results are compared with correct outputs. Each output correctly generated by the student program earns a certain number of points. Students may also be given a subset of these test inputs and encouraged to test their own programs, or given access to a compiled version of the correct solution.

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 their produced code is even more unpredictable. Thus the task of designing test cases is never as trivial as it sounds. For all the programming assignments examined in this thesis, there exist unexposed bugs inside student code that were not caught by the normal grading procedure.

Furthermore, with only failed test cases it is hard to tell what actually causes the student code to fail. There are different reasons for a student code to fail a test case; for example, a bug may be caused by a typo or by conceptual errors. Current grading procedure lacks the ability to differentiate different types of bugs inside student code. It takes too much human effort to look at each failed student code and identify the bugs. If students cannot learn from their mistakes, there is no point for them to do their assignments.

It is also unfair to assign points according to failed test cases. A typo in student code may cause it to fail all the test cases while a conceptual error inside student code may only fail a few of the test cases. Assigning points according to the type of bugs inside student code is much more fair than the current scheme.

This thesis 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, a student can submit their code for review by the framework. Tests are automatically generated for the student's code. After being examined by these automatically generated test cases, if the execution of student code does not match the desired behavior, the framework generates a test case that exposes the defective code to students. The whole feedback generation process are done within minutes of student code submission. Our framework tries actively to find bugs inside the student code. In particular, our framework explores all the execution paths and checks if any of them leads to an error.

In addition to providing the test case back to students, we also try to cluster students' codes with similar errors into groups. Students can get additional information about the quality of their code based on the clustering information. The clustering is done by doing cross testing. All generated tests for different students are collected and then run against every student code. The whole set of generated tests is then reduced to a smaller set of core tests. The reduction is done by removing the tests that are "contained" in other tests. Test A contains Test B is defined by the following: if all the student codes that fail Test B also fail Test A, then Test B is contained in Test A. After doing the reduction, all the core tests are NOT contained in each other. For any student code, the fail/pass information for all these core tests forms a signature vector. Any two student codes with the same

signature vector are then clustered into the same group. By doing clustering, our framework reduces the effort needed from instructors to analyze student code and provide human feedback as well. Instead of reading each and every student code and providing feedback, instructors now can do analysis on several student codes in each group, and it is likely that all student codes in the same group have similar errors. With our experiment, out of 91 student submissions, 15 are clustered into one group and 6 into another. After human inspection of both groups of student submissions, every student submission in each group has the same type of error.

The technique that our framework uses to identify defects in student code is called concolic testing [1]. Concolic testing executes a program both concretely and symbolically. The aim of the test 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 cover 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 [2].

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 ECE students take this course, which thus has roughly 300 students every semester. Our results show that we can identify more defects in student code than traditional automated grading. Students can learn from the failed test cases generated and can improve their code. Experiments show that the average processing time for a recursive maze solver is 260 seconds, meaning that one can scale this solution to provide 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 thesis has several benefits:

- Improved ability to identify code defects.
- Generates test cases leading to defects as feedback.
- Improved grading fairness.
- Short processing time: scalable and quick feedback.

We provide an overview of the chapters of this thesis. Chapter 2 gives additional detail on our motivation for deploying industrial tools in computer science education. Chapter 3 gives background information on concolic testing. Chapter 4 describes our framework and workflow for applying concolic testing to automated grading. Chapter 5 describes our experiment with a student assignment and discusses the results. We conclude in Chapter 6.

CHAPTER 2

BACKGROUND

2.1 Why Teaching Programming Is Difficult

Teaching programming is hard. Both students and instructors are struggling to do a good job. The dropout rate of introductory programming courses can be as high as 50% [3]. Because more students tend to take an introductory programming class, the number of students enrolled in introductory programming courses has been growing in recent years. Outside universities, people are also enrolled in massive online open courses (MOOC) to learn programming. Due to the large number of enrolled 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. Atkins et al. [5] indicate that British science and engineering students spend 50 to 70 percent of their time in laboratory work. These laboratories can help students to build up their problem solving skills, which prepare them for their future jobs.

For programming courses, these labs are usually done as programming assignments. There may exist laboratories done in a form of group meeting session led by an instructor; however, because the instructor's time is limited, the chances for students to interact with instructors are limited. For MOOC platforms, there may be no interactions at all between the instructor and the students. The opportunities for students to get direct feedback from instructors are limited.

When students finish their programming assignments, they need to know if their programs are correct. However the correctness of a computer program is hard to determine. In 2002, Newman estimated that software bugs have cost $59.5 billion annually [6]. Stamat and Humphries further concluded that

in 2006 over 8000 vulnerabilities were cataloged by the Computer Emergency Response Team [7]. While producing correct programs for experienced programmers remains a hard task, students who are novice programmers are more likely to produce buggy code. It is the responsibility of the instructors to help students identify bugs in their code, and to learn from these bugs.

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 [8] and Kemerer and Pavik [9] 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 it is expensive for instructors to manually review each and every student's code. The process requires hiring more teaching assistants, who costs less than instructors, but are more variable in the quality of their code review.

After identifying bugs in student programs, explanations are also needed to help students understand what mistakes they have made. Providing these explanations is sometimes easy if the bug is obvious, for example typos or off-by-1 errors. Bugs caused by misunderstanding concepts may be hidden deeply inside the student code. Explaining these bugs to students requires time and effort from instructors, who have limited resources. The increasing number of students learning programming and the limited time of instructors are causing problems in programming education.

2.2 Previous Work

For programming courses, students are usually given assignments that specify the goal they need to reach. Along with the assignment, students usually also get some references that provide feedback to them. In most cases, the references consist of test cases and desired outputs. The only feedback produced is a pass or fail notification. This kind of feedback provides limited information to students. Code reviews require too much human effort and do not provide timely feedback. Other methods like discussion boards lack

consistent and timely feedback information. Significant effort in both research and practice has focused on automatically assessing student work. In order to provide quick and accurate feedback, automatic tools are needed. Tools have been proposed to assess several aspects of student submitted programs. Some automated tools help to regulate the programming style of students [10] and are more objective than manual grading. Srikant and Aggarwal developed a tool to grade based on machine learning on a number of keywords [11]. Other tools focus more on assessing functionality. Pears et al. [12] point out that for tools that assess functionality of student programs, the grading is typically done by comparing a student's output with results provided by the teacher's model solution program. For any given set of inputs, the teacher's model solution program generates a model set of output. Student programs take the same input set to generate corresponding output, which is then compared with the modeled output.

However, how to generate a set of test inputs that cover all the possible defects in student program is challenging. Yet most of the automated tools today rely on the instructor to provide sets of test inputs. Instructors may fail to provide a full set of tests that can cover every part of the student code. Without a test case that covers the defective part of student code, current grading tools may leave bugs in student programs undetected. Defective code passing tests can deprive the student of knowing that they have made mistakes. What is more dangerous, the student may believe that the implementation is correct, which encourages him or her to make the same mistake in the future.

2.3 Recent Automated Assessment Tools

Amelung et al. [13] implemented a system that separates all concerns of managing students, assignments, and submissions from the actual testing. The system consists of two parts. The frontend manages storage of assignments and solutions, proper treatment of submission periods and re-submissions, communication of results to students, grading of the results, and statistics for individual students and whole cohorts. The backend tests the functionality of the student submitted code. The submitted code is executed and the output of a student solution can be compared to that of a model solution for

a set of test data, or the assignment can be tested for properties which must be fulfilled by correct programs.

Marmoset [14] is a system that hides some of the instructor-provided test cases from students. Students are encouraged to write their own test cases before they submit their work. When student code fails to pass multiple test cases, only a portion of the failed test cases are revealed to students.

Web-CAT [15] is a framework for automated testing. It requires students to submit their own tests along with their solutions. The instructor has an implementation of model solution on the server side. Feedback is generated based on execution of student code on student tests, and execution of student tests is based on the instructor-provided model solution. The model solution is instrumented with statement as well as branch coverage information to arrive at a concrete measure of the breadth of the executed test cases. Edwards and Perez-Quinones [16] further added per-assertion diagnostic messages to the Webcat framework to provide better feedback to students.

Easyaccept [17] uses scripting language to define tests and expected results. Easyaccept provides an easier way for instructors to write tests for students.

AWAT [18] is an automated web application testing system. AWAT provides an interface for instructors to write tests that simulate the actions of human testers to extract information from web pages, and verify expected outcome based on the test case specification.

Naudé et al. [19] and Wang et al. [20] both use graph similarity to assess student programs. Student programs are first analyzed and transformed into system dependence graphs. The assessment is done by a graph similarity measure against a pool of model system dependence graphs.

2.4 Concolic Testing

Concolic testing [21] 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 a set of random concrete values are generated for these variables. With this set of concrete values the program executes with the execution path recorded. The code executed is marked as having been covered. To cover other parts, 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 symbolic execution, while symbolic execution is used to generate better coverage than random testing.

There are several limitations of concolic testing:

Complicated Programs

Theoretically, given a powerful constraint solver, concolic testing can generate test cases that can cover all reachable statements or branches [22]. Even though such a constraint solver does not exist, high coverage can still be achieved given that the code under test is not too complicated. Concolic testing has been successfully employed with reasonably-sized and heavily-tested software packages. Because student assignments for introductory programming courses are much simpler programs, concolic testing is effective in practice.

Float/Double Data Type

Another limitation of concolic testing is that float/double type variables are not fully supported. The reason is because most constraint solvers do not support these data types. However we can always avoid these data types when applying concolic testing to student programs. Using these data types in student assignments is usually not an essential part of the assignment.

Native Call

Concolic testing cannot handle calls to code for which source code is unavailable. However, the concolic testing tool Klee [23] has integrated support for UClibc [24], a subset of the standard C library. This kind of support is sufficient for an introductory level programming course.

CHAPTER 3

DESIGN

The implementation of this thesis 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 a modified version of Klee [23]. If there is any defect inside student code, a test case leading to that defect is generated. In addition to these two main parts, a separate clustering tool is used to do analysis on generated test cases. Defective student codes are clustered into different groups according to the test cases generated. The clustering tool runs separately with the main tool, and can be invoked at any time.

3.1 Frontend

The system implemented for students to submit their code for feedback revolves around the flow diagram in Figure 3.1.

3.1.1 Original System

The original system was set up using Subversion, a software versioning and revision control system. Programmers use Subversion to maintain current and historical versions of files. Students can work on their assignments following the instructions. Each time they make progress towards completion, they can commit their work to the Subversion server. Students can test their own work with released test cases from the instructor on their local development environment. Students can modify their work and commit to the Subversion server anytime before the deadline. After the deadline, all of the latest committed student submissions are assessed by instructor-

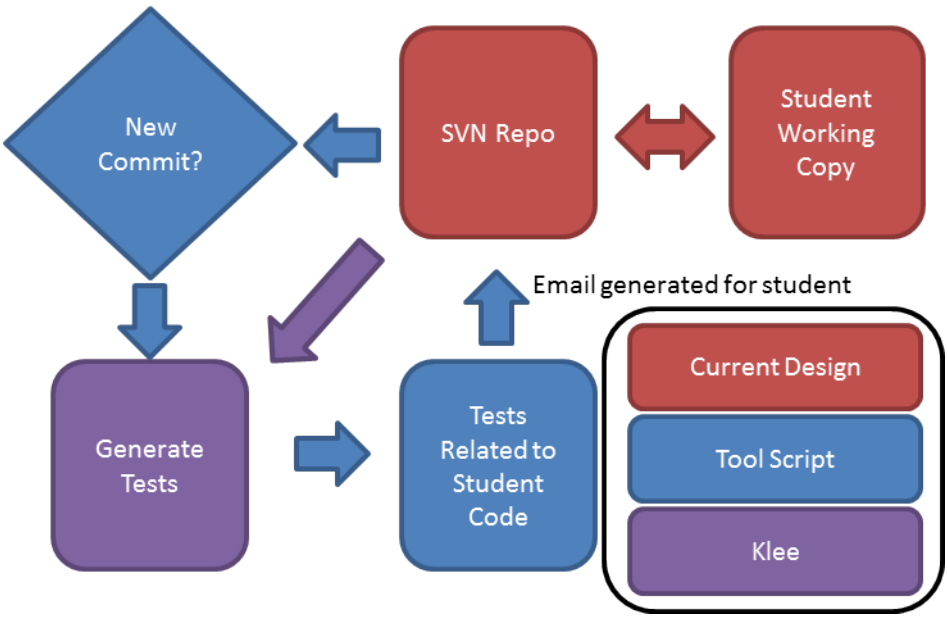

Figure 3.1: Collect Student Code/Generate Feedback: We implemented tool scripts to check student commits. When there are new commits we generate tests and send these tests back to students.

provided scripts. These scripts execute the student program with several sets of instructor-provided inputs, and compare these results with the instructor-provided model solution. A score is given based on the comparison result.

3.1.2 Frontend System

This thesis implements a system that assesses the student code and distributes feedback to students. The system checks for a new submission from students every minute. 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. If there exist any such test inputs, we add them to the corresponding Subversion repository. After the grading process is finished, we generate an email to the student no matter whether the submission is correct or not. Students receiving the email can check their Subversion directory to collect generated input sets. The whole frontend system is written in Ruby.

3.2 Backend System: Klee

The work flow of the backend system is shown in Figure 3.2. The backend system takes two inputs: a student solution and a wrapper main function. Both input files are compiled into LLVM byte code. This LLVM code is then passed to the modified version of Klee for assessment.

Klee [23] is a concolic testing framework. As a concolic testing tool, Klee tries to maximize code coverage for the tested code. An initial set of values for the symbolic variables is generated randomly. An interpreter is used to execute the program. An execution path with the initial set of random values is recorded. After each execution, the information about coverage of execution path is generated for the SMT solver. If there exists any new set of values for the symbolic variables that can increase the coverage, the new set of values is recorded and a new execution generated.

However, Klee was designed for experienced programmers. The information generated by Klee is hard to understand for novice programmers. It also takes some setup work for Klee to work properly. Thus in our framework, Klee is hidden from students in order not to confuse them. The setup work is handled by the main function for grading provided by the instructor. After Klee finishes execution, the result is interpreted into simple feedback as test cases that are useful and understandable for students.

3.2.1 Setup

In order for Klee to work, some setup work needs to be done. The setup work is divided into two parts: mark symbolic variables and check functional correctness. Symbolic variables are used to generate test inputs and are also the key to achieve 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.

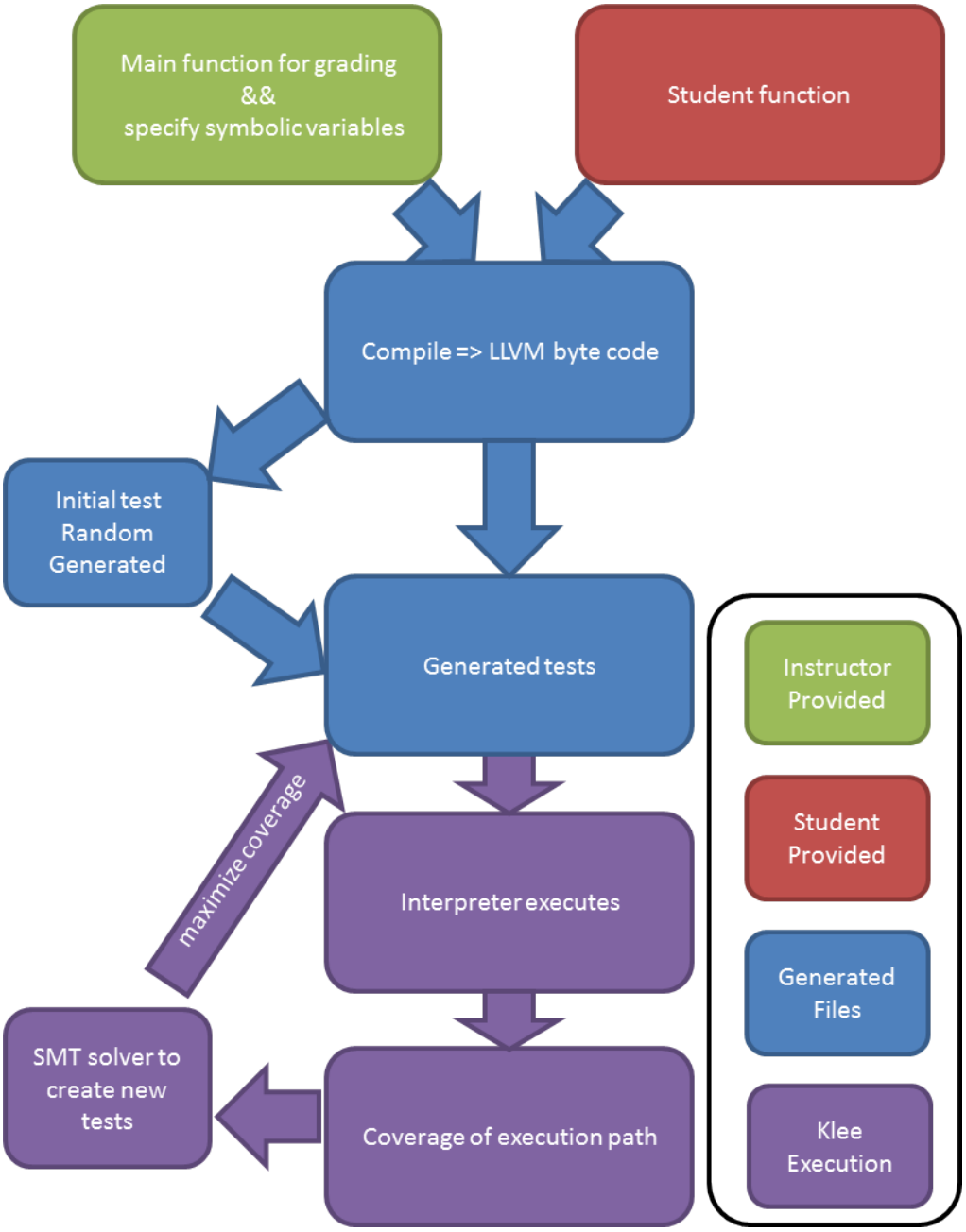

Figure 3.2: Workflow: Main function (contribution of thesis) and student function are compiled into LLVM byte code. The Klee is applied to generate tests that can cover the most execution paths.

```c
#include "stdio.h"
#include "assert.h"
int main(){
  int a,b;
  char operator;
  if(scanf("%d",&a)==1){
    if(scanf(" %c",&operator)){
      if(scanf("%d",&b)){
        if(operator=='+'){
          return a+b;
        }
        else if(operator=='-'){
          return a-b;
        }
      }
    }
  }
  assert(0);
  //execution should not reach this point
  return 0;
}
```

Figure 3.3: Sample Program

Selecting Symbolic Variable

The symbolic variables are usually chosen from inputs to the student program. For example, for a calculator program, the input can be two numbers and an operand. A sample program is in Figure 3.3. To use Klee to find maximal coverage we need to mark a,b and operator as symbolic. Marking these variables as symbolic can be done through a function call klee_make_symbolic. The resulting program is in Figure 3.4. All the initialization of a,b and operator through stdin is now done by Klee. Executing the program using Klee produces the result in Figure 3.5.

Setting a variable as symbolic is easy. However, these variables can take any value that Klee assigns to them. Sometimes the input to the program needs to have some desired properties. For example we may require that one variable be positive. We can use the klee_assume function (see Figure 3.6) to describe such constraints to Klee. The predicate passed to klee_assume call is always evaluated by the SMT solver. The SMT solver then tries to return a set of concrete values for the symbolic variables that evaluates the predicate

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

Figure 3.4: Symbolic Variable Marked

```
KLEE: done: total instructions = 2052
KLEE: done: completed paths = 3
KLEE: done: generated tests = 3
```

Figure 3.5: Result of Klee

```c
klee_assume(a>0);
```

Figure 3.6: klee_assume

```c
#include "stdio.h"
#include "assert.h"
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 3.7: Correctness Checking

to be true. In another sense, klee_assume can help us to define preconditions [25] for the whole program. It is safe to assume that the precondition is true after the klee_assume call.

Functional Correctness Checking

Klee was designed to achieve high code coverage, so by default Klee does not check functional correctness. All Klee does is to generate test cases that can cover most of the code. Thus, in order to verify that the program under test has the designated behavior, we add post-condition checking, which is done 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. In Figure 3.7, the return value of student function *stu_calculate* is checked against the return value from the reference implementation *gold_calculate*. Because Klee uses 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. To maximize code coverage, SMT solvers try to solve for a set of inputs that can lead to the false part of a branch. 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.

3.2.2 Execution

The execution of Klee is an iterative process of trying to achieve higher code coverage. Klee first generates an initial set of random concrete values for marked symbolic variables. An execution path with these concrete values is recorded. Klee picks a part of the code that was not covered. The code must begin at some sort of branch from code that was covered. Klee uses the branch condition for that decision along with other constraints needed to reach that branch and packages the constraint set for solution by the SMT solver. If a solution can be found, that solution will cause execution of the previously uncovered code. All sets of concrete values are recorded as generated tests. When there is no more path to explore, Klee finishes execution.

When generating feedback using Klee, some execution parameters need to be set in order to generate the right number of test cases. By default Klee outputs every set of concrete values as a test input. This behavior generates lots of redundant test cases. In order to limit the number of test cases generated, we have Klee execute with the $-only-output-states-covering-new$ parameter set. In this way, each test case generated covers a different execution path inside the program.

Because student code may include standard library code, Klee provides a C library, using uClibc [24]. Any function call supported by uClibc can be supported by Klee. Common function calls such as printf and assert are all supported.

For execution paths that have errors in them, Klee generates an error report indicating what error has occurred, and at which line. The type of error includes memory accesses out of bound, division by zero, assertion failures and so forth. With this information, we can easily detect if there are potential bugs inside the student code.

3.2.3 Extensions To Klee

While using Klee as the grading engine for the framework, we found that Klee was not optimized for some of the heavily utilized functions, particularly the I/O functions. Klee by itself depends on a third party library, uClibc [24], which does not support the c99 standard scanf functions. It is also complicated to set up symbolic files for the uClibc library. Thus an I/O extension for Klee is implemented.

The I/O extension includes support for the following functions: $fopen$, $fclose$, $fscanf$, $fprintf$, $fputc$, $fread$, $fwrite$, $sscanf$.

In addition to these functions, a new helper function $klee\_make\_IO\_buffer$ is also implemented.

In order to use the extension, the "–symbolicFileIO" flag has to be set for Klee at execution. Inside the main function, a char array needs to be allocated. The char array and a file name are then passed to the $klee\_make\_IO\_buffer$ function. The char array works just as a file. Any $fopen$ function call to the same file name opens the char array, and any operation done to the opened file is reflected in the char array. Like any other array, the char array can be set as symbolic.

The implementation is done by adding new special function handlers to Klee. LLVM byte code with support for symbolic variables is used to support the I/O functions. With the native usage of LLVM byte code instead of compiling uClibc C code to LLVM byte code, the extension also achieves 10× speedup and 100× less memory usage.

The functionality correctness of these functions is tested with sample code collected from 20 students during the Summer 2015 semester. The $sscanf$ function is also tested with 91 student submissions from the Fall 2015 semester. No error has been caused by these I/O functions.

3.3 Clustering Tool

The clustering tool is used to gather student submissions into groups. The goal of the tool is that each group of student submissions contains similar bugs. There are several benefits: first of all, it reduces the work load for instructors. With student programs clustered into groups, instructors can identify conceptual errors for multiple students without looking at all the

Table 3.1: Total Number Of Tests For All Students

Assignment	Total Tests
Code Breaker	48697
Sudoku Game	682
2048 Game	492
Maze Solver	855

students' codes. Secondly, students can get more accurate feedback. By utilizing existing group information, if a student submission can be categorized into any of the existing groups, this particular submission is also very likely to contain similar bugs as other submissions in this group. Instructors can provide more information to students depending on which group the student submission can be clustered into.

The clustering of student programs takes two steps: signature tests identification and student submission categorization.

3.3.1 Signature Tests Identification

Because for each student program there is usually more than one test generated, it takes too much computational power if all the generated tests are used to group student submissions. Table 3.1 shows the total number of tests generated for the final submissions from 91 students. The goal of signature tests identification is to reduce the number of tests that are used to identify the type of bugs inside student submissions. Among all the generated tests, there are tests that expose bugs inside student submissions and tests that do not. The first step of clustering is to collect all the test cases that expose bugs inside student submissions. However, as the number of student submissions increases, the number of test cases in the collection increases as well. Thus another step is taken to further reduce the number of signature tests.

For the second step, all the collected test cases are used to test all the student submissions, and the results are recorded. The results include the student submission that fails to pass the test, and the bug type information if possible. After all the results are collected, the results of all test cases are compared against each other using a superset/subset algorithm. A test case A is considered to be the superset of a test case B if all the test cases failed by test B are also failed by test A. If any test case is a subset of the other,

the subset test case is removed from the signature test set.

After the second step, the signature test set is guaranteed to expose each and every known bug inside student submissions, while maintaining a reasonable size.

3.3.2 Student Submission Categorization

After the generation of signature tests, all student submission test results obtained using the signature test set are used to categorize the student submission. For each student submission, the pass or fail information for each signature test is used as a signature vector. Any two student submissions having the same signature vector are considered to have similar bugs, and thus are placed in the same group of student submissions.

CHAPTER 4

EXPERIMENTS

4.1 Overview

The automated feedback framework has been applied to five different assignments as an experiment 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.

4.1.1 Design of Assignments

These programming assignments are designed by the instructors 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 also works in parallel with the normal grading process, which forms the basis that we compare the results to. 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 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.

4.1.2 Assignments

There are 5 different assignments in total, each will be explained in detail:

Code Breaker

Detailed description in Section 4.2. Learning objectives: basic logic, function call. Typical code length: 100 lines.

Image Editor

Detailed description in Section 4.3. Learning objectives: for loop, array. Typical code length: 100 lines.

Sudoku Game

Detailed description in Section 4.4. Learning objectives: recursion. Typical code length: 150 lines.

2048 Game

Detailed description in Section 4.5. Learning objectives: memory operations. Typical code length: 200 lines.

Maze Solver

Detailed description in Section 4.6. Learning objectives: recursion. Typical 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. There are various forms of feedback generated as well. For Maze Solver, Code Breaker and Sudoku Game, test cases that expose bugs inside student code are generated. For the other two assignments, either the information about which rule from the specification is violated or the name of erroneous function is generated for students. The choice of what kind of feedback information is generated depends on the type of assignment. Maze Solver and Sudoku Game are assignments on recursion, thus it is hard for students to construct a test case that specifically exposes the bug found by the framework. For the other assignments, students should be able to produce their own test cases given the error message. However, the choice of the type of feedback generated is decided by the designer of the feedback script.

4.1.3 Student Participants

For both semesters, students were asked to volunteer to allow us to generate 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. As stated in Chapter 3, we collect student submissions directly from the Subversion server. For Spring 2015 semester, 82 out of 349 students volunteered; for Fall 2015, 91 out of 393 students volunteered to participate the experiment.

4.2 Code Breaker

4.2.1 Problem Description

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 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 somewhere in the solution code but 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. If the player manages to guess the correct sequence in 12 or fewer guesses, they win the game. Otherwise, they lose.

Some Example Guesses:

8 2 4 5: Solution Peg

1 6 3 8: Guess 1, one misplaced match

8 6 5 4: Guess 2, one perfect match, two misplaced matches

8 2 5 4: Guess 3, two perfect matches, two misplaced matches

8 2 4 5: Guess 3, solution found!

The objective for this assignment is for students to gain some experience with basic I/O, implement code using multiple functions, practice using pointers, and solve a problem that requires moderately sophisticated reasoning and control logic.

Students are required to implement three different functions:

int32_t set_seed (const char* seed_str)

The function receives a string (a pointer to a character) as its input and produces a 32-bit signed integer as output. Students need to verify that the string specifies exactly one integer. The integer is then used to seed the pseudo-random number generator. The return value from the function *set_seed* indicates whether the input string did in fact correspond to a number. When the string represents an integer number (and only a number), the function returns 1. Otherwise, the function returns 0 and does not set the random seed, instead printing "*set_seed*: invalid seed". Students are instructed to use the *srand*, *rand* and *sscanf* functions. There is a lab section where instructors demonstrate how to write this function to students.

int32_t start_game (int32_t* one, int32_t* two, int32_t* three, int32_t* four)

The *start_game* routine selects the solution peg at random, a set of four numbers from 1 to 8. The *rand*() function is used to generate a solution peg. In order to maintain the consistency between the normal grading procedure and the student code, the order of assigning results from the *rand* function to solution code is fixed.

int_t make_guess (const char* guess_str, int32_t* one, int32_t* two, int32_t* three, int32_t* four)

The *make_guess* routine compares a player's guess with the solution peg. The inputs to this routine include a string (the player's input) and four pointers to integers. The routine must validate the string in the same way as the *set_seed* function. A valid string contains exactly four numbers (and no extra garbage at the end). All four numbers in the string must be between 1 and 8. Student code must check these cases, using *sscanf* and other logic as necessary. If the string is invalid, the routine must print an error message "make_guess: invalid guess", return 0. For a valid string, the student must store a copy of the guessed code in order in the four addresses provided as input parameters (one, two, three, four). The routine must then compare the guessed code with the solution code (which should be stored in static variables, peg1, peg2, peg3, and peg4 done in the *start_game* function) to count the number of perfect and misplaced matches, then print a message informing the player of the results.

4.2.2 Grading Script Design

C Wrapper

The grading script consists of two parts. The first part is the C wrapper file, which consists of around 250 lines of code, of which 150 lines are reference model code. In order to set the solution code as symbolic, the *rand* function call is substituted with the grading version. In order to catch the result produced by the student function, the *printf* function is also substituted with the grading version.

rand

The grading version of *rand* function returns symbolic solution codes in order, thus the same solution codes are set for both student code and the reference model code.

printf

The parameters passed to the grading *printf* function are extracted and used to compare with the result from the reference model code.

In addition to these two substitution functions written in C, the *sscanf* function is directly supported with the modified version of Klee as stated in Chapter 3.

Because the *set_seed* function is both too simple to test and taught directly in the discussion section, it is assumed to be correct for the student code. No specific grading script is written for testing the *set_seed* function.

The testing for both *start_game* and *make_guess* is combined because these two functions are closely related with each other. Student versions of the *start_game* function and the *make_guess* function are executed in sequence with the set of symbolic variables. The same set of symbolic variables is used for the execution of reference model code. Finally, if any of the results from the student implementation differ from the reference model implementation, an assertion is fired. The following values from the student code are tested for equality with those produced by the model code.

- Return value of the *make_guess* function
- Number of perfect matches
- Number of misplaced matches

- Each of the guess variable written according to the input string
- Turn number

Feedback Generation

The second part of the grading script is used for compiling student code, linking with the c wrapper code, executing with Klee, and generating feedback information for students.

Most of the script is the same across different assignments. The only part specific to each individual assignment is the part that generates feedback information for students. Depending on the assertion result generated by Klee, either a test case leading to an erroneous result is generated, or the actual type of error is generated as feedback to students.

For the error of the number of perfect and misplaced matches, a test case is generated. The reason to generate a test case instead of simply giving students the type of error is that, by only knowing the type of error, it is hard for students to produce a test case that leads to erroneous code. In other words, bugs are hidden inside the student code and can only be exposed by specific test cases. Thus, the feedback provides the test case to students to avoid confusion and to help students better understand the problem. The test case is extracted from the results produced by Klee.

For the other three errors, students should be able to identify the bug inside their code easily, and they are input independent. Thus only the type of error is generated as feedback to students.

4.2.3 Normal Grading Procedure

Three test inputs are provided to students, each consisting of a sequence of sets of inputs to the program and the desired output from the program. For final grading, three similar test inputs are used, and some additional testing for each individual student function is used as well. However, no detailed information about all these tests is released.

4.3 Image Editor

4.3.1 Problem Description

In this assignment, one 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 5 functions:

calculateCosineFilter

This function takes an empty filter and a radius value, then changes the value of the filter according to the radius.

invertImage

This function inverts the colors of the input image and stores the result in a separate output image.

convolveImage

This function performs convolution on the RGB channels of the input image. The result is written to a separate output image. The Alpha channel should remain the same.

convertToGray

This function generates a grayscale version of the input image and stores it in a separate output image.

colorThreshold

This function creates an output image which contains only the pixels of the input image which exceed the color threshold. All other pixels are set to black.

One important aspect of this assignment is that floating point operations are involved. In particular, the Cosine Filter in the calculateCosineFilter function, the grayscale weights in the convertToGray function and the filter in the convolveImage function are all arrays of type *double*.

4.3.2 Grading Script Design

C Wrapper

As stated in the Chapter 2, Klee does not support symbolic representation of floating point values. All such values must thus be chosen as concrete values for testing. Because there is a floating point type of variable in this assignment, any floating point type variables have to be prevented from being set to symbolic, or appear in the control flow. Thus all the floating point type variables are set as concrete values for this assignment.

Furthermore, when checking the functional correctness of the student function, instead of comparing two floating point type variables directly, the values are first converted to integers, and the difference between the resulting integers (which should be 0) is used to decide correctness.

The first step of grading Image Editor is the initialization (120 lines) of the input images. In order to compare the student output and that of the model code, the initialization step allocates two identical images that have the same set of symbolic RGB pixel values. The image processing functions all manipulate the pixel value and store the result to output image channel pointers which are parameters passed into the image processing function. Therefore, output images are also allocated and used to catch the outputs of both student and reference model functions. After comparing the outputs, the correctness of student code can be determined.

To reduce the computation time, small images are used: 8 pixels wide and 6 pixels high. After the initialization, the student and the reference model implementation are executed with their own symbolic input. The student result is checked against the model result. The image processing functions are tested separately. Only one function is tested in each Klee run.

The following two functions are used to compare the student and reference model result:

bool checkImage(Image *stud,Image *sol)

checkImage() compares two Images by comparing the pixels at the same position of both images. If any of the pixel pairs are not the same, the student code contains error.

bool imageError(Image *stud,Image *sol, int error_rate)

imageError() function calculates the difference between student and

model results. Because floating point is not supported well for Klee, one threshold integer err_range is used. The condition is thus: $err\_range * (\sum(student\_val - gold\_val))^2 < (\sum(gold\_val))^2$. If the condition holds, the student function is determined to be correct. Note that there is a floating point variable inside the condition, which results in the concretization of symbolic values. However, the concretization is allowed here because, this is the final step of the grading procedure, thus no more paths need to be explored.

Feedback Generation

The script code handles the compilation of the main wrapper function and student function, as well as multiple executions of Klee. Feedback is generated based on which function fails to pass the test. The failure type information generated by Klee is also used to generate feedback. In particular, the memory out of bound pointer error is used to determine that the student code has a bug affecting array accesses, which is common for this assignment.

4.3.3 Normal Grading Procedure

Only one input image is provided to students as test input. For the final grading multiple images may be used, but no detailed information is released.

4.4 Sudoku Game

4.4.1 Problem Description

The Sudoku assignment is to implement a C program that solves the Sudoku puzzle using recursive backtracking. A standard Sudoku puzzle contains 81 cells, in a 9 by 9 grid, and has 9 zones. Each zone is the intersection of 3 rows and 3 columns (i.e. size 3x3). Each cell may contain a number from 1 to 9 and each number can only occur once in each 3x3 zone, row, and column of the grid. At the beginning of the game, several cells begin with numbers, and the goal is to fill in the remaining cells with numbers satisfying the puzzle rule.

There are five different functions students need to implement:

int is_val_in_row(const int val, const int i, const int sudoku[9][9])

Checks if the given number val can be filled in row i.

int is_val_in_col(const int val, const int j, const int sudoku[9][9])

Checks if the given number val can be filled in col j.

int is_val_in_3x3_zone(const int val, const int i, const int j, const int sudoku[9][9])

Checks if the given number can be filled in the 3x3 zone corresponding to the cell (i, j).

int is_val_valid(const int val, const int i, const int j, const int sudoku[9][9])

Checks if the given number val can be filled in position (i,j).

int solve_sudoku(int sudoku[9][9])

main recursive function to solve the Sudoku.

There is also a challenge question for this assignment. In addition to the constraint that each number can only occur once in each row, column, and 3x3 zone, the challenge implementation imposes another rule that each number can appear exactly once in each of the two diagonal lines of the matrix. Students should directly modify their code in the function solve_sudoku such that the solution returned satisfies the diagonal constraint as well as the regular constraints. Note that a valid solution for the challenge implementation also infers a valid solution for the non-challenge implementation.

4.4.2 Grading Script Design

C Wrapper

To explore as many execution paths as possible, a symbolic Sudoku game board is needed. However, because this assignment is an assignment on recursion, it is impossible to set the whole game board as symbolic, which causes Klee to never finish execution. For the backtracking algorithm in this assignment, it requires a large number of recursive calls.

There is no need to generate the whole Sudoku game board every time. Instead, 1 initial Sudoku game board with exactly 1 solution is used, and 9

additional symbolic numbers are set to be mutually unequal with each other. These symbolic values can only take values between 1 and 9, which generates a set of symbolic values that can represent the input domain of the Sudoku game. By mapping symbolic values to the hard-coded Sudoku game board one by one, 9! different Sudoku game boards that are guaranteed to have an unique solution can be generated. However, 9! different Sudoku game boards are still too many for Klee to handle. With some experiments the number of symbolic numbers is reduced to 5, such that 120 different game boards are generated.

For the *check_valid* function, the only thing that matters is the return value. Therefore, the script compares the result of student and reference model code with same input parameters. The function checks are in a sequential order: $is\_val\_in\_row$, $is\_val\_in\_col$, $is\_val\_in\_3x3\_zone$, and is_val_valid. Assertion is fired at the first function with return value different from that of the model code.

For the solve_sudoku function, because the generated game board has only one solution, it is enough to check if the student-solved game board is exactly the same as the reference solution game board. The student result game board is first re-mapped back to the original game board according to the symbolic variables, and then compared with the hard coded reference solution.

4.4.3 Normal Grading Procedure

Three sample Sudoku game boards are supplied as tests to students. The final grading uses a single game board that has multiple solutions to test the student code. Student code is determined to pass the challenge question if the resulting game board passes the restriction of the challenge question.

4.5 2048 Game

4.5.1 Problem Description

The game 2048 is played on a 4×4 grid, with numbered tiles that slide smoothly when a player moves them using the four arrow keys. Every turn,

a new tile will randomly appear 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 will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move.

For this MP, students are implementing the same game on a variably sized grid. The game is controlled using the keys w, a, s, d, with n to restart the game and q to quit. The key w causes the tiles to slide up, a to the left, s down, and d right. Each turn, the program waits for a keyboard input. When the program receives a directional input (w, a, s, d), it will slide the tiles in the corresponding direction. If this causes no change, the turn is over and the program waits for the next input. If the input causes at least one tile to slide, a new random tile is added and the game is printed to the terminal before the turn ends and the program waits for a new input. The key n will recreate the game board with new dimensions, setting all cells to zero, and randomly adding one tile. The key q will quit the program.

Each turn, the program will ask for one keyboard press as an input. The code for inputs n and q are already done for students. A directional input will call one of the *move_* functions. These functions modify the game board and update the score. When two tiles merge, the score increases by the new value. For example, two 4 tiles will merge into an 8, so the score should be incremented by 8.

Students are required to implement the following 9 functions:

game * make_game(int rows, int cols)

Generates a new game board with dimension rows * cols.

void destroy_game(game * cur_game)

Frees up all the memory allocated.

cell * get_cell(game * cur_game, int row, int col)

Returns the value in cell(row,col).

int move_w(game * cur_game)

Performs a upward move for the game board.

int move_s(game * cur_game)

Performs a downward move for the game board.

int move_a(game * cur_game)

Performs a leftward move for the game board.

int move_d(game * cur_game)

Performs a rightward move for the game board.

int legal_move_check(game * cur_game)

Checks if the game is over.

void remake_game(game _cur_game_ptr, int new_rows, int new_cols)**

Cleans up the current game and regenerate a new game board.

4.5.2 Grading Script Design

The intention of this assignment is to teach students about memory allocation. However, because Klee is not designed to detect memory leaks, and there are already tools such as Valgrind to test memory allocation and deallocation, the target of the automated feedback framework is focused on the logic of four move functions.

C Wrapper

First of all, it is noticed that for each move operation, rows (or columns) are independent of each other. Thus, instead of making a game board of arbitrary size, a symbolic board of size 5 * 1 is used to test row operations, and a symbolic board of size 1 * 5 is used to test column operations. The element in each cell of the game board is a symbolic variable. The symbolic variable can take its value from the set {-1, 2, 4, 6, 8, 16}.

The choice of size 5 for the game board is a balance of execution time and test completeness. After some experiments with the assignment, it is clear that a game board of size 5 can cover most of the situations in a real game. Increasing the size lengthens the execution time by too much with little gain in functionality checking.

For each student function, two game boards consisting of the same symbolic variables are created. One gameboard is a duplicate of the other. Cells in each board with the same coordinates have the same symbolic variable, and thus are mapped to the same value. One game board is passed to the student

function and the other is passed to the reference model implementation. The return value of the function, the updated game board and the updated score are checked for correctness, and the checking process stops when a bug is found. To reduce execution time, students can always fix the error and resubmit for more feedback.

Feedback Generation

The feedback generated for this assignment is failure information about the *move_* functions.

- Return value does not match for Move_
- Incorrect score update for Move_
- Board not same for Move_

More detailed information about exactly what game board can cause the error can be generated. However, because for the student code the map generation is random and the map generation code is provided, it is still hard for students to utilize the game board information. Thus only the failure information is provided.

4.5.3 Normal Grading Procedure

For functionality of the student function, the normal grading procedure tests each student function with several fixed game boards, and compares the results with a reference solution. However, there is no check for invalid moves for the *move_* functions. For memory allocation and de-allocation, the normal grading procedure utilizes Valgrind to test.

4.6 Maze Solver

4.6.1 Problem Description

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

```
//maze is a 2-d array of
//size width * height

//Student is supposed to find "S" symbol,
//return its coordinate back in x and y
void findStart(char ** maze, int width, int height, int * x,
    int * y);

void printMaze(char ** maze, int width, int height);

//Solve the maze with starting location
//xPos and yPos.
//If unsolvable return 0, otherwise 1.
int solveMazeDFS(char ** maze, int width, int height, int
    xPos, int yPos);

//Check if the solution is valid.
//Return 1 if solution is correct,
//otherwise 0.
int checkMaze(char ** maze, int width, int height, int x, int
     y);
```

Figure 4.1: Student Function

the ending location is marked as E. Students are required to implement four different functions, as shown in Figure 4.1. Initially only the test for the $solveMazeDFS$ function was implemented. After receiving positive feedback from students using our system, feedback for the $findStart$ and $checkMaze$ functions was added. The feedback generation for different functions is based on two separate runs of the backend, one for solveMazeDFS and one for both findStart and checkMaze. An example maze is shown in Figure 4.2.

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. An example of a correct solution is shown in Figure 4.2.

```
Input File:    Solution:
5 5
%%%%%          %%%%%
S   %          S..~%
%% %%          %%.%%
%   E          % ..E
%%%%%          %%%%%
```

Figure 4.2: Example Input Maze File & Solution

4.6.2 Grading Script Design

C Wrapper

A fixed acyclic map size of 6×6 is generated without Start and End symbols. The locations of starting and ending points are marked as symbolic variables for Klee. The student function is tested by calling $solveMazeDFS$ on the generated map.

The return value and returned map are checked for correctness. An assertion will be raised if either of them is not correct. Then the $findStart$ function is tested in the same way as $solveMazeDFS$. The maze is then solved with a referenced model implementation and the $checkMaze$ function is tested with the solved map.

Feedback

For feedback generation, only test cases leading to failure are provided as feedback to students. Even though more detailed information about at which line of code the error occurs is available, this information is hidden from students.

CHAPTER 5

RESULTS

Students submit their programs to the automated feedback framework for feedback. In this chapter, 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, which is also the version that is actually graded, is considered in this chapter. The reason to only consider the last submission is that earlier submissions may not reflect the real quality of student code, as students may submit to test certain functionalities before finishing the whole program, or simply try to save their progress. There is also no normal grade to compare with except for the last submission.

5.1 Bug Identification

Functional correctness is the most important aspect of a program. Missed bugs inside student code may leave students with misunderstood concepts. Hence it is important that every bug inside student code is identified. The automated feedback framework aims at identifying every bug inside student code. Even though successfully finding all the bugs is not guaranteed, the results show that the automated feedback framework does a better job than the original grading procedure in the following aspects.

- Discovers more defects of student submissions.
- Reports fewer false positives.
- Categorizes more accurately the type of defect.

This section presents the comparison between the normal grading procedure and the automated feedback framework.

5.1.1 Code Breaker

For the Code Breaker assignment, every student submission with defects reported is human inspected to assess the quality of auto-generated feedback. Among 91 total submissions, three pairs of duplicate codes were found, thus only 1 copy of these 3 pairs is considered in the discussion.

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

To limit the processing time for each student submission, there is a time limit of 3 minutes for the framework to generate feedback. Two student submissions went over time, thus no feedback was generated. Human inspection of these submissions showed that both were functionally correct but used unnecessarily complex algorithms. When the time limit was removed, the framework generated feedback for both submissions in less than 3.5 minutes.

Printf

To access the functionality of the student submission, the grading script catches the parameters passed to the *printf* function. However, 1 student submission only printed under a certain condition (when the guesses are correct). Failing to catch parameters passed to the *printf* function resulted in the failure of evaluation of the student submission. The same student submission earned 0 points for functionality with the normal grading procedure as well.

Another student split one *printf* call into multiple *printf* calls, which caused the grading script to catch partial parameters, which led to failure of accessing the student submission. The same student submission passed the normal grading procedure, but the use of multiple printf function calls instead of one should be discouraged. The synchronization wrappers on the C library guarantee that each printf is routed automatically to an output stream, whereas multiple printfs can be interleaved with other threads' output to the same stream. So the habit may hurt students in the future.

Set Seed

In order to set the solution peg values as symbolic variables, the *rand* function has been modified to return symbolic values in a fixed order. One student submission called the *set_seed* function and the *rand* function multiple times, causing the *rand* function to return erroneous results that were not symbolic, which caused the failure of evaluation of the student submission. The same student submission passed the normal grading procedure, but the actual implementation should be prohibited.

In addition to these 5 students, one more student, **STUDENT_7**, had an error in the *set_seed* function, which the automated grading framework assumed to be correct and did not test. The error in the *set_seed* function caused the program to fail the normal grading procedure completely. However, the rest of the implementation is correct according to the framework.

Bugs Missed by Normal Grading

The normal grading procedure failed to identify several different types of bugs. A single student submission often contains multiple bugs. For several student submissions, the normal grading procedure caught some but not all of the bugs inside the student submissions. Because of the difficulty of distinguishing which bug the normal grading procedure caught, here in this section only "completely correct student submissions", as determined by the normal grading procedure, are considered as candidates for missed bugs. Sixteen student submissions containing bugs passed all the tests in the normal grading procedure. The bugs in these 16 student submissions are listed below. Please note that there may be other bugs in these student submissions, but only the most common bug in each student submission is listed here.

Multiple Mismatch Peg

The bug appears with the resulting missed match value larger than it should be. As shown in Table 5.1, the correct output should be 0 perfect match and 1 missed match. With multiple mismatch peg bug the missed match is calculated as 2.

The cause of the bug is that when student submissions matched the input guess to the solution peg, the solution peg was not marked when

Table 5.1: Example for Code Breaker

solution peg	1	2	3	4
guesses	1	1	5	6

pairing with one of the input guesses. In the example, the 1 in solution peg 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 resulted in 2 mismatches instead of 1.

The multiple mismatch peg is the most commonly missed bug. Ten student submissions had this bug in their code, yet all these submissions had passed the normal grading procedure. Six more student submissions with this bug inside were caught because of other bugs, while this bug was never exposed by the normal grading procedure.

Similarly, because the solution pegs and guesses are symmetric, solution pegs matched with guesses should get the same result. Some students failed to mark guess pegs, a bug similar to the multiple mismatch peg bug. The framework can catch both type of bugs. However, the normal grading procedure is also capable of detecting this kind of error. Thus there are no missed bugs in this category.

Guess Not Set

The *make_guess* function requires students to use the *sscanf* function to read guesses from the input string to four different guess pointers. However, 2 student submissions failed to write the read values back to the guess pointers, and 1 additional student submission only wrote the read values when they all matched the solution peg. All three student submissions passed the normal grading procedure.

Out of Bound Pointer

One student submission used arrays but accessed locations outside of the array size. The normal grading procedure failed to detect this error, while the framework caught it.

Negative Value

One student implementation failed to check that inputs were in the range 1 to 8 and accepted negative values. The normal grading procedure failed to identify this bug.

Unitialized Value

Two student submissions failed to initialize some of the flag variables, thus causing the logic to be wrong. The normal grading procedure failed to identify this bug, possibly because stack-allocated values often happen to be 0 early in a program's execution.

5.1.2 Image Editor

Because the image editor assignment is designed to help students understand the concept of arrays, the input data does not affect the control flow. Thus for most submissions, there was only one execution path, so only one test was generated by Klee. However, the limitation on execution path exploration did not affect the effectiveness of the automated feedback framework. In particular, Klee as an interpreter checks all the array accesses. Among all the student submissions that were determined to be correct by the normal grading procedure, the checking of array accesses alone identified 30 errors. Seventeen student submissions had erroneous memory accesses in one function, mostly in the $calculateCosFilter$ function. Two student submissions had errors in 4 different functions, and 1 student submission had errors in all 5 functions tested. The functionality of these student submissions were correct; however, students should know that there were hidden bugs in their code and that their concepts of array were wrong. The normal grading procedure only compared the end result, which led to the failure to detect this kind of bug.

Beside memory access error, 1 student submission was determined to be correct in functionality with the automated feedback framework, while it was determined to be wrong by the normal grading procedure. Upon human inspection, it is determined that the student's algorithm computed the result in a different order than the reference gold implementation, which caused a difference in the computation result because of lack of associativity with floating-point arithmetic. The difference accumulated with the size of the input image. Because the automated grading framework used a small input to test, the difference was not significant, while the normal grading procedure used a large input image, which caused the error. Because students were not being tested for knowledge of numeric stability, the student's answer should receive full credit for this introductory class.

5.1.3 Sudoku Game

Compile Failure

Five student submissions failed to compile, of which 1 failed to compile with the normal grading procedure as well. The main reason for compilation failure is compiler difference. The normal grading procedure used g++ as the default compiler, while the auto-grading framework used llvm-gcc.

Upon human inspection of the failed student submissions, it was found that 1 student used a type of cast syntax that llvm-gcc does not support; 1 student used pass-by-reference in the function declaration, which is not supported by llvm-gcc, as a C compiler; and 2 students used the goto command in their code which caused the compilation error.

Time Limit

Because the Sudoku game assignment is a recursion assignment, the number of execution paths may become extremely large with erroneous student submissions. Thus the execution time limit set for the assignment was 5 minutes to avoid spending too much time on erroneous or complex student submissions. Seventeen student submissions failed to finish within the 5 minute deadline. However, for 4 of the timed-out submissions, bugs were identified within the student code. These bugs were identified from the execution paths that Klee explored before time-out.

Bugs Missed by Normal Grading

The most missed-bug for the Sudoku game assignment was the implementation of the challenge problem. In the specification, it is clearly stated that the check for number uniqueness on diagonals should only be done outside the *is_val_valid* function. However 37 student submissions had the diagonal check included inside the *is_val_valid* function. Thus when a diagonally invalid, but otherwise valid, Sudoku game board was passed to the student's *is_val_valid* function, the game board was determined to be invalid. The result was detected by the automated feedback framework and the *is_val_valid* function was determined to be incorrect. However, the normal

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

Figure 5.1: Example Code for Sudoku Game 1

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. Thus all these student submissions were determined to be correct.

Another two missed bugs were related to the problem of default return value of the compiler. The example skeleton codes are in Figure 5.1 and 5.2.

It is easy to tell that in the first student code example, the return statement is misplaced, possibly by a typo. However, if the default return value is 0, the functionality of the function is still correct. It is assumed that the normal grading procedure used a compiler that defaults the return value to 0, as the student submission passed the test.

For the second example, if the execution path does not enter any of the two inner if statements, then the return value of the function is also not defined. Clearly student had a conceptual error. However, the normal grading procedure did not detect this bug, and thus the student submission passed the test.

5.1.4 2048 game

Compile Failure

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

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

Figure 5.2: Example Code for Sudoku Game 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 complex algorithms for the *move_* functions. Most of them traverse the game board multiple times to determine if it is a valid move, which resulted in multiple nested for loops that caused the time-out. After the submission deadline, part of the 28 submissions are tested again without time limit. However the tests for them all reached a memory cap, thus halting the execution of Klee.

Bugs Missed by Normal Grading

The most obvious missed bug was the missing test case for invalid moves. When the move is invalid, the *move_* functions should return 0. By default, the *move_* functions return 1 as given code. Missing test cases on invalid moves caused two students, with their *move_w* function empty except the "return 1" statement, to pass the return value test for normal grading pro-

cedure. One more student submission, which was checked for invalid moves but failed to return 0, passed the normal grading procedure.

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

In addition to standard feedback information about failure of the *move_* functions, 14 student submissions were determined to have out of bound pointer errors. These out of bound pointer errors are bugs themselves, and may lead to the failure of the functionality test as well, but were not detected by the normal grading procedure.

5.1.5 Maze Solver

The automated feedback framework tested $solveMazeDFS$ and $findStart$ successfully. However for $checkMaze$, 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 valid solution only. In other words, Klee only generates correct maps to test student functions. A correct map is not enough to identify all the defects in the $checkMaze$ function. To identify all the defects, a map with both correct and incorrect solutions needs to be generated. Simply providing a correct solution with different starting and ending points is not adequate for finding all the bugs. However, the precondition setting prevents Klee from generating wrong solutions. A better solution is to mark all the spaces in the maze as symbolic variables.

Defects

With 82 students volunteering, a total of 241 submissions were graded, 141 of which contained test cases generated, which indicates that 141 of the submissions have defects. 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.

The grading result is then compared with the normal grading procedure. For this assignment all 349 students are graded in the traditional way. Fixed

sets of 3 test cases were generated by the instructor. Then the grading script tried to compare the results of the student- and instructor-provided functions.

For the final submission, 32 volunteering students failed the automated feedback framework. After manual inspection, we found that 15 of them had their defects detected by the normal grading procedure, while 17 passed the normal grading procedure without their defects caught. More than half of the student submissions with defects received full points from the normal grading procedure.

After the Spring 2015 semester, a full test on all 320 student submissions for the Maze Solver was done. Table 5.2 shows the number of student functions containing defects that passed the normal grading procedure.

Table 5.2: Student Code with Defects Passing Traditional Grading

	Missed
checkMaze	55
solveMazeDFS	118
findStart	11

5.2 Feedback Effect

After the grade from the normal grading procedure is available, a comparison between volunteer students and the rest of the students is done. The correction rate here is calculated based on the traditional grade. Each functionality of a submission is only considered correct when it passes all three test cases provided by the traditional grading. The results are listed in Table 5.3.

For each function tested, students with feedback get better understanding of the problem as they get a higher correct rate than students who do not have feedback.

Table 5.3: Correct Rate Volunteer Student vs. Rest of Students
Correction Rate Based on Traditional Grade

	Volunteer Students	Rest of Students
checkMaze	45.12	29.83
solveMazeDFS	74.39	58.82
findStart	91.46	86.55

Table 5.4: Fractions of Fully Correct Solutions for Student Participants Compared with those of Non-Participants

	Volunteer Students	Rest of Students
checkMaze	67.07	35.71
solveMazeDFS	56.10	20.59
findStart	90.24	84.45

The correction rate based on the automated assessment tool is also presented in Table 5.4. For each function, students receiving feedback still do better than those who do not. However, except for the *checkMaze* function, the correction rate is lower compared to the normal grading process. The reason is because with concolic testing, the automated assessment tool can identify more defects in student code.

5.3 Grading Script Production

In order for the automated feedback framework to work correctly, a grading script is needed. The grading script is generated by a graduate student and an senior undergraduate student as assistant. Because instructors gave new assignments or made changes to existing assignments, we received the released assignments at the same time with students. Thus it is important that the grading script can be generated in time. The time it takes to produce the grading script depends on the assignment. However, no grading script took more than two days to be produced. A rough estimation of time taken to produce the grading script is made based on the hourly payment recorded for the undergraduate assistant. Table 5.5 shows the reported work hours from the undergraduate assistant. The Code Breaker assignment was handed out in week 5, the Image Editor assignment was handed out in week 7, the Sudoku Game assignment was handed out in week 8 and the 2048 game assignment was handed out in week 9. The biweekly hours shown indicate that the undergraduate assistant worked around 10 hours each week, in which around 8 hours are estimated to be grading script production time.

Table 5.5: Reported Work Hours

	Hours
Week 4 & 5	19.3
Week 6 & 7	15.0
Week 8 & 9	21.2
Week 10 & 11	23.7

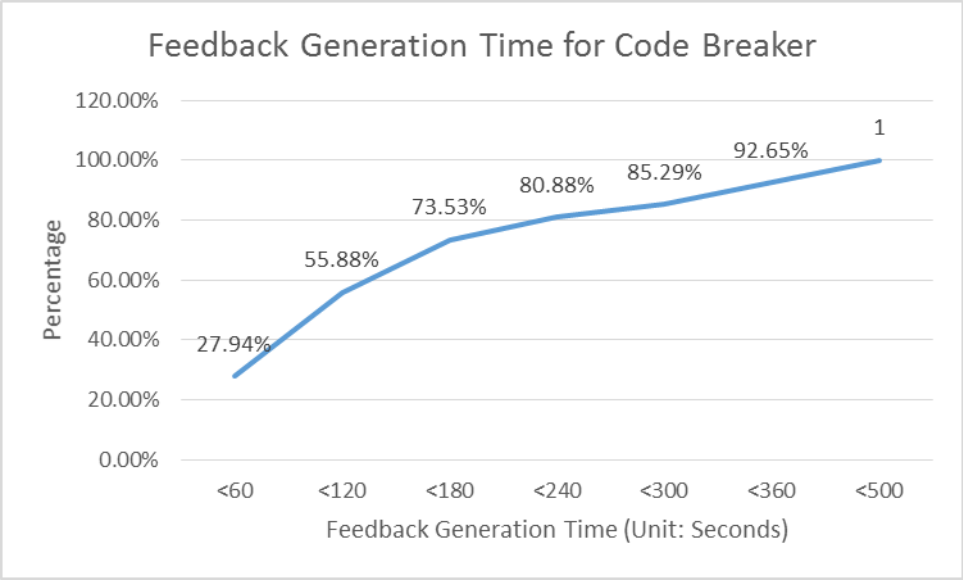

Figure 5.3: Processing Time Distribution for Code Breaker [Unit:Seconds]

5.4 Processing Time

The processing time calculated here is the time difference between student commit time and the generated test case commit time from the subversion server log. For the Fall 2015 semester assignments, the automated feedback framework was executed on a server with Xeon ES-2420 CPU and 96 GB memory. For the Maze Solver assignment which is done in Spring 2015 semester, the automated feedback framework is executed on a desktop, with Intel i5-3470 CPU and 8 GB memory. It is worth noticing that because we are running the grading process on a single machine, only one grading is being done at each time. When two commits happen within two minutes of each other, the latter one is queued up.

Figures 5.3, 5.4, 5.5, 5.6 and 5.7 show the CDF of processing time for each assignment. Figures 5.8 and 5.9 show the average processing time and median processing time for each assignment. Clearly the feedback generation times

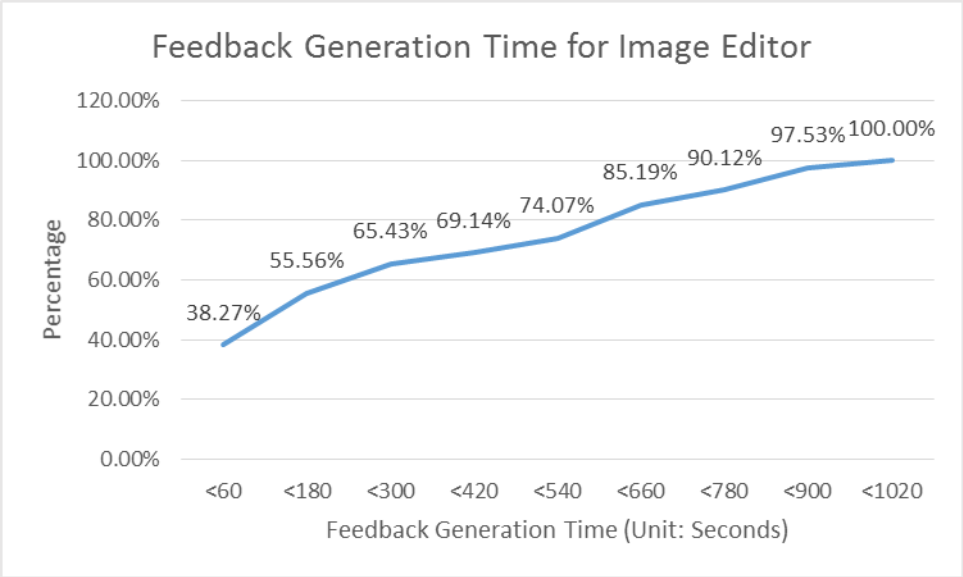

Figure 5.4: Processing Time Distribution for Image Editor [Unit:Seconds]

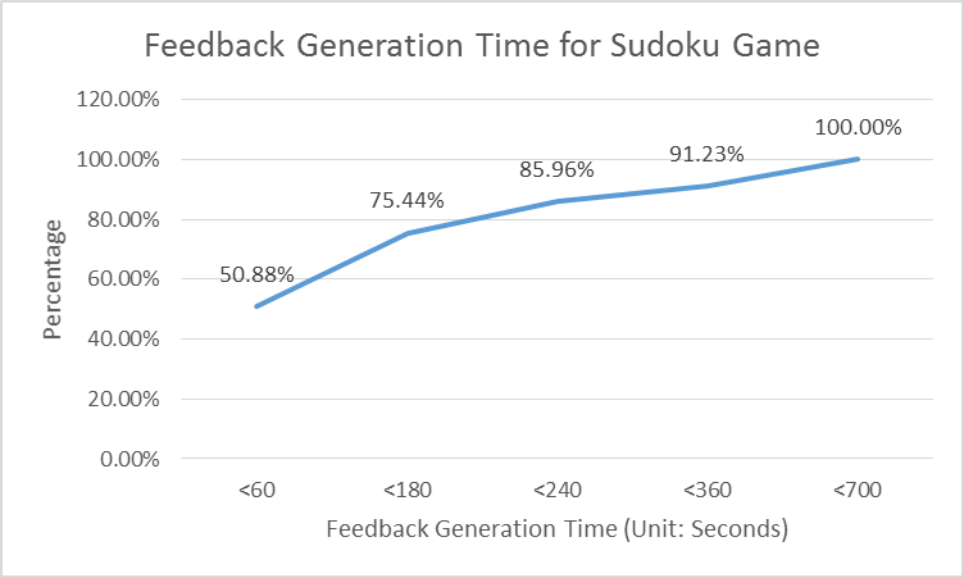

Figure 5.5: Processing Time Distribution for Sudoku Game [Unit:Seconds]

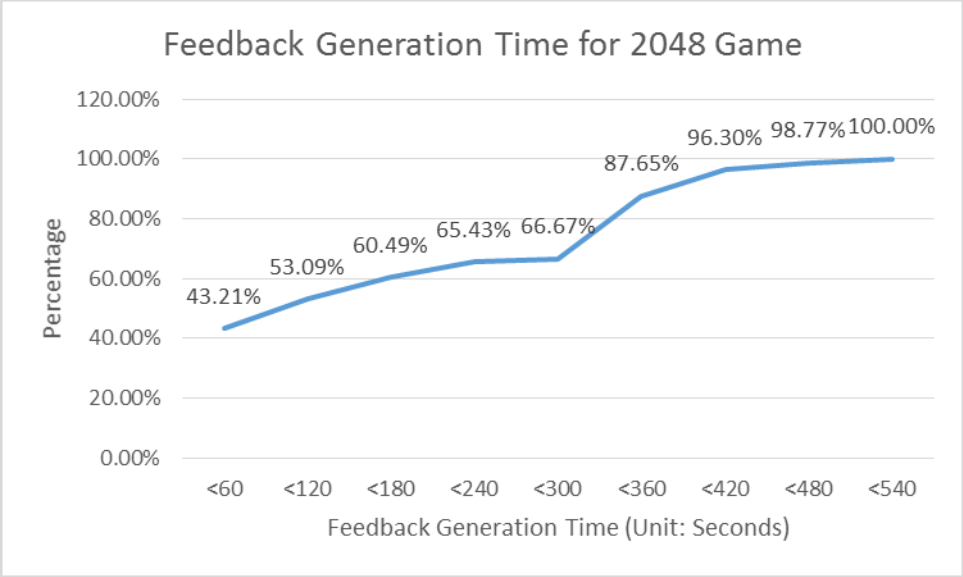

Figure 5.6: Processing Time Distribution for 2048 Game [Unit:Seconds]

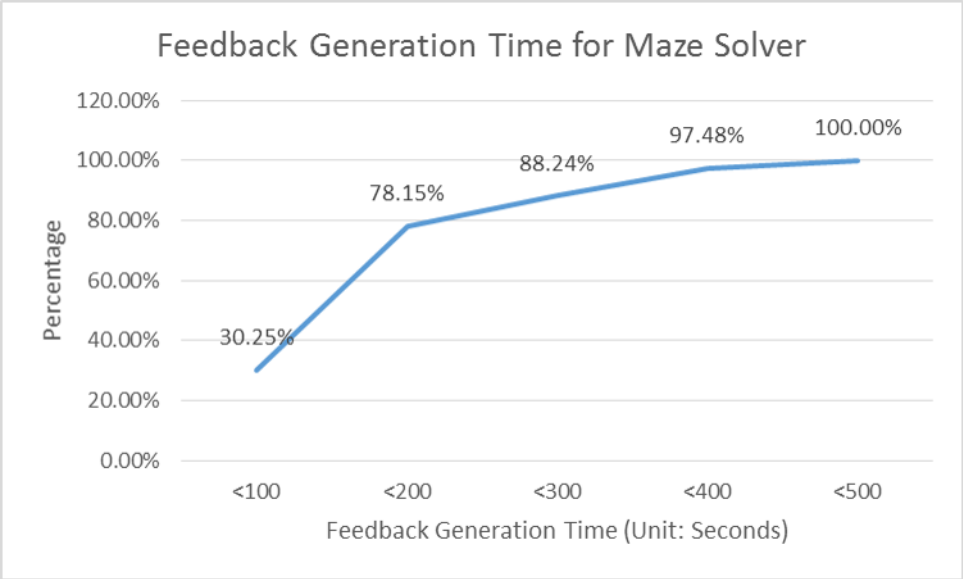

Figure 5.7: Processing Time Distribution for Maze Solver [Unit: Seconds]

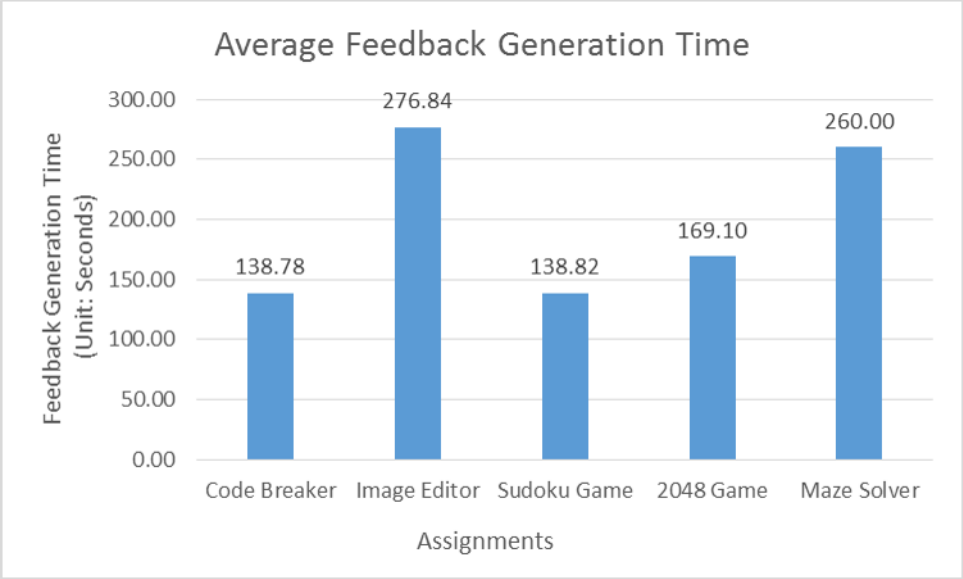

Figure 5.8: Average Feedback Generation Time

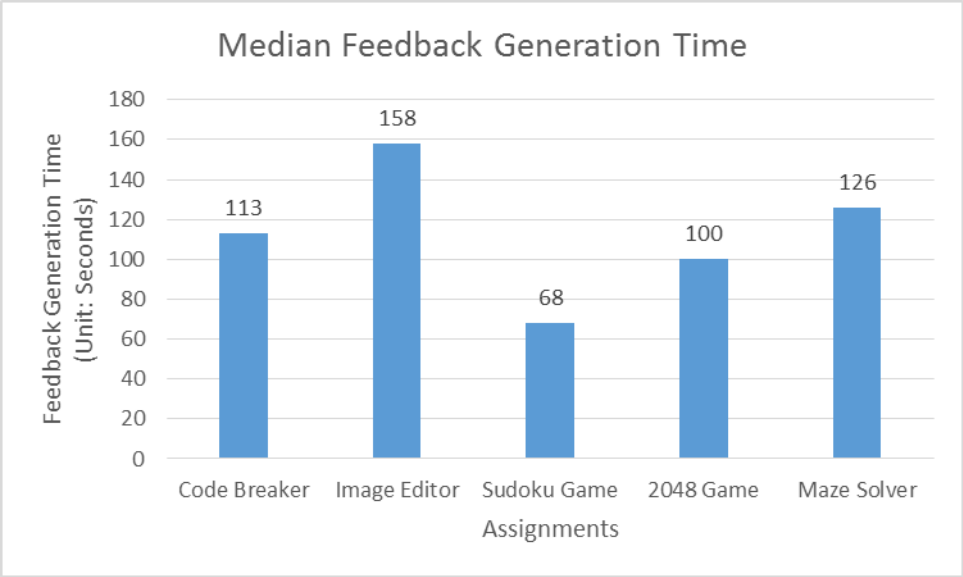

Figure 5.9: Median Feedback Generation Time

Table 5.6: Signature Test Cases Generated for Code Breaker

	Sol 1	Sol 2	Sol 3	Sol 4	Input 1	Input 2	Input 3	Input 4
Test 1	1	1	1	3	1	1	8	1
Test 2	3	1	1	8	1	1	1	3
Test 3	8	8	8	8	8	8	8	8
Test 4	1	1	1	2	2	1	3	1
Test 5	3	1	2	3	1	3	3	1
Test 6	3	1	1	1	1	1	3	3

for different assignments are different. The feedback generation time depends not only on the difficulty of the assignment, but also on the submission pattern of students. When multiple students submit their assignments at the same time, the queuing delay adds up. In addition to the queuing delay, the framework only checks for new submissions every minute, in order to reduce the workload on the Subversion server. However, for all the assignments, more than half of the students can receive their feedback within 3 minutes.

5.5 Student Grouping

Student grouping is done for the Code Breaker assignment. After applying the method described in Chapter 3, two large groups of students failed the same group of signature test cases. One group of 15 students failed because of the Multiple Misplaced Match error described earlier. Another group of 6 students failed to set the Guess variables.

With 91 student submissions, the initial collection of test cases generated 38 different test cases. Applying reduction on these 38 test cases resulted in 6 signature test cases, shown in Table 5.6.

Human examination was done for each student submission. Students who failed signature test cases 5 and 6 were determined to have the same Multiple Misplaced Match error. Students who failed signature test cases 1, 2, 3, 4, 5, 6 were determined to have not set the Guess variables.

5.6 Discussion

5.6.1 Grading Code Design

When doing assessment of a student program, the input to the program under test needs to be marked as symbolic values. However, which variables are marked as symbolic values needs special consideration. Then we need to have Klee generate the right amount of test cases. We would like to identify as many bugs as possible while avoiding duplication of tests for a single bug.

Symbolic Variable Choice

Klee works based on symbolic values, without which it is merely an interpreter. Choosing which variables to be marked as symbolic is not easy. These variables are passed to constraint solvers; thus, the number must be limited to avoid the solver running out of memory or timing out, causing the whole grading process to fail. Usually preventing constraint solvers from failure can be done by limiting the number of variables chosen. Too many variables can lead easily to state explosion. On the other hand, the variation in the value of the chosen variables must also lead to covering different parts of the student-written functions. The balance of these two requirements is the key point of making Klee a successful automatic feedback generation tool. For different programming assignments, different strategies need to be applied to the choice of symbolic variables.

Precondition

The precondition of inputs to a program can be complicated. For example, the Maze Solver assignment requires an input map that is acyclic. Defining an acyclic map using predicate functions can be hard. One way of doing so is to generate an acyclic map using depth first search, and substitute the random generator with *klee_make_symbolic* call. In other words, let Klee control the generation of the acyclic map. However, using symbolic variables in a recursive function causes the SMT solver to require huge memory space and execution time.

When the precondition is complicated such that there is no easy way to generate valid inputs for the program, it may not be necessary to generate the whole set of inputs. The goal of using symbolic variables is to achieve high code coverage. If variation in part of the input suffices for full coverage of student code, we can mark that part of the input as symbolic. For the example of an acyclic map, varying the starting and ending location suffices if the map is constructed carefully.

Though choosing correct symbolic variables requires some insight into the program, it is still easier than creating a complete test set, which requires the instructor to prepare a test case for every possible student code failure. While students are novice programmers, preparing test cases to cover all possible failures is not an easy task. On the other hand, reasoning about the input to the program and finding the key variables that can lead to enough variations to the input require no knowledge about how the student code may fail. Under the limitation of hardware resources, the more variation in input we can create, the better. Only when the hardware resources cannot cover all the possible execution paths do we need to decrease the symbolic space by either reducing the number of symbolic variables or limiting the value range of symbolic variables. The process of decreasing the symbolic space requires knowledge about the assignment. For example in the 2048 Game assignment, even though student submissions are supposed to support an arbitrarily sized game board, the limit of row and column size is set to 5 because the game with the larger game board is based on the game with smaller game board; student submissions passing the test of size 5 are most likely to be correct submissions.

The results presented show that most symbolic variable setups were correct. The only exception is the solution map generation for the Maze Solver problem. The setup failed to generate an invalid solution map to test the *checkMaze* function, and thus failed to prepare an adequate input space. However, the problem of preparing symbolic variables that can cover adequate input space is similar to the problem of selecting an adequate set of test cases with the normal grading procedure. In general the automated feedback framework is more complete than the normal grading procedure.

5.6.2 Bugs Missed by Normal Grading

The traditional way of grading relies on the set of test inputs. When the test inputs do not cover defective code, the grade given can be wrong. 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 Misplaced Matches not detected.

Image Editor

Out of Bound Array Accesses not detected.

Sudoku Game

Return Value not set error not detected.

2048 Game

General logic error not detected.

Maze Game

Starting location right next to Ending location error not detected.

The execution path for Image Editor assignment does not depend on the input data, thus the normal grading procedure covered every bug. However, for all the assignments, there were errors in student code that were covered by the normal test cases, but not detected. In particular, the access of the out of bound array pointer error was not detected. With the normal grading procedure the array accesses are not checked and thus it is hard to discover the access of the out of bound array pointer error.

Last but not least, the normal grading procedure failed to test some aspects of the student submissions according to the specification.

Code Breaker

Write the scanned value back to the Guess variables.

Sudoku Game

Diagonal test should be implemented outside the *is_val_valid* function.

Maze Solver

Maze map with no outside wall.

```
if(maze[yPos][xPos]!=' ' && maze[yPos][xPos]!='S'){
  return 0;
}

if(maze[yPos][xPos]!='S' && maze[yPos][xPos]!='E'){
  maze[yPos][xPos]='.';
}
```

Figure 5.10: Code of Student A

```
if(currentChar == ' ' || currentChar == 'S')  // Check if the
    current space is a valid location
{
  if(currentChar == ' ')       // Check to make sure to not
   overwrite the start, S
    maze[yPos][xPos] = '.';
  /* Recursively call this function at the adjacent spaces
     If a recursive call returns one, then a solution was
     found, and this function can return 1
  */
  ...
}
```

Figure 5.11: Code of Student B

5.6.3 Fairness

Several of the students had the same type of bug in their code; however, some of them got penalized and some not. For the Image Editor, an example has been shown in Section 5.1.1. For the Maze Solver assignment, two student submissions A and B differ only at their orders of recursive calls. The student submissions A and B are listed in Figure 5.10 and 5.11 respectively. They both made the same mistake that multiple recursive calls are made to the starting point. Because the starting point is not overwritten as checked (by marking as "."), all the neighboring cells can recursively call on the starting location. Figure 5.12 is the test case generated by Klee indicating that they both fail. However, because their orders of recursive calls to different directions are different, student A passed the traditional grading tool while student B was penalized.

```
Start of maze at (4, 2)
%  ...%
 %.%.
 %.%S%
 %.%.
 %E%~%
    %~~
```

The "." under the "S" symbol shoule be "~" because it is not on the solution path.

Figure 5.12: Erroneous Result

5.6.4 Limitations

There are several limitations of using concolic testing for automated assessment as well.

Wrapper Code

As discussed in Section 3.2.1, to successfully assess a student function, some setup work needs to be done. Because of the complexity of Klee, the wrapper is hidden from students. In order for students to test their program when they are developing, a normal main function has to be provided to the students. However, if the target student function is inside the same file as the main function, to include the target student function in the wrapper requires extracting student code from the main file. The easiest solution is to provide separate skeleton files to students. One file contains the main function that students do not need to touch, while all the student functions are in a separate file. However, for introductory level programming courses, some students may be confused by the multiple-file configuration.

Floating Point

Because concolic testing tools do not support floating point operations, any assignment with floating point operations is not suitable for automated assessment with concolic testing. However, the floating point itself is not a requirement for most of the assignments. Furthermore, because floating point operations inevitably lose information, they introduce difficulty in grading

assignments even with the normal grading procedure. When floating point operations are involved in an assignment, it is normal to give the result from student function an error range. The reason is because the different order of the floating point operations may result in slightly different answers; for example, the result of $a \times b \times c$ is not necessarily the same as the result of $a \times c \times b$. In some cases, even an error range is not enough. In one of the assignments given to students from ECE220, because of a floor function used on two different values between "4.99999999" and "5.00000000", the student answer is judged as wrong while it is actually the correct implementation. So while we should avoid floating point operations in student assignments when possible, not being able to support floating point operations is acceptable.

Library Calls

Concolic testing supports only limited library calls. For Klee, it uses uClibc which is only a partial implementation of the standard C library. If library calls outside the uClibc library range are made, the automated assessment may fail. However, because concolic testing is based on an interpreter, instructors can implement or even replace library calls. For example when teaching about I/O functions, standard library calls such as fscanf and fprintf can be replaced with an instructor version, which instead of reading/writing from/to a file, reads/writes from/to a buffer. Functional correctness checking can also be done to this internal buffer instead. Hints can also be generated when these function calls are not in the correct format.

CHAPTER 6

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 submission. The average 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. For example, including functional correctness checking against a reference solution inside Klee is a feasible option. Also, because concolic testing is based on an interpreter, some of the common library calls can be replaced with an instructor-provided version. The implementation of IO extension is the first step in this direction. More implementations for system calls such as malloc and free are desired in the future.

REFERENCES

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

[2] P. Godefroid, M. Y. Levin, and D. Molnar, "Sage: whitebox fuzzing for security testing," *Queue*, vol. 10, no. 1, p. 20, 2012.

[3] A. Yadin, "Reducing the dropout rate in an introductory programming course," *ACM Inroads*, vol. 2, no. 4, pp. 71–76, Dec. 2011. [Online]. Available: http://doi.acm.org/10.1145/2038876.2038894

[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*. IEEE, 2004, pp. T2H–5.

[5] M. Atkins, G. A. Brown, and G. Brown, *Effective Teaching in Higher Education*. Routledge, 2002.

[6] M. Newman, "Software errors cost us economy $59.5 billion annually," *NIST Assesses Technical Needs of Industry to Improve Software-Testing*, 2002.

[7] M. L. Stamat and J. W. Humphries, "Training & Education: Putting secure software engineering back in the classroom," in *Proceedings of the 14th Western Canadian Conference on Computing Education*, ser. WCCCE '09. New York, NY, USA: ACM, 2009. [Online]. Available: http://doi.acm.org/10.1145/1536274.1536308 pp. 116–123.

[8] J. G. Ganssle, "A guide to code inspections," 2001. [Online]. Available: http://www2.cs.uni-paderborn.de/cs/ag-schaefer/Lehre/Lehrveranstaltungen/Vorlesungen/ SoftwareQualityAssurance/WS0405/material/Inspections.pdf

[9] 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*, vol. 35, no. 4, pp. 534–550, 2009.

[10] M. Blumenstein, S. Green, S. Fogelman, A. Nguyen, and V. Muthukkumarasamy, "Performance analysis of GAME: A generic automated marking environment," *Computers & Education*, vol. 50, no. 4, pp. 1203–1216, May 2008. [Online]. Available: http://dx.doi.org/10.1016/j.compedu.2006.11.006

[11] S. Srikant and V. Aggarwal, "Automatic grading of computer programs: A machine learning approach," in *Machine Learning and Applications (ICMLA), 2013 12th International Conference on*, vol. 1. IEEE, 2013, pp. 85–92.

[12] A. Pears, S. Seidman, L. Malmi, L. Mannila, E. Adams, J. Bennedsen, M. Devlin, and J. Paterson, "A survey of literature on the teaching of introductory programming," *ACM SIGCSE Bulletin*, vol. 39, no. 4, pp. 204–223, 2007.

[13] M. Amelung, P. Forbrig, and D. Rösner, "Towards generic and flexible web services for e-assessment," *SIGCSE Bull.*, vol. 40, no. 3, pp. 219–224, June 2008. [Online]. Available: http://doi.acm.org/10.1145/1597849.1384330

[14] J. Spacco, D. Hovemeyer, W. Pugh, F. Emad, J. K. Hollingsworth, and N. Padua-Perez, "Experiences with marmoset: Designing and using an advanced submission and testing system for programming courses," *SIGCSE Bull.*, vol. 38, no. 3, pp. 13–17, June 2006. [Online]. Available: http://doi.acm.org/10.1145/1140123.1140131

[15] A. Shah, "Web-cat: A web-based center for automated testing," Ph.D. dissertation, Virginia Tech, Citeseer, 2003.

[16] S. H. Edwards and M. A. Pérez-Quiñones, "Experiences using test-driven development with an automated grader," *J. Comput. Sci. Coll.*, vol. 22, no. 3, pp. 44–50, Jan. 2007. [Online]. Available: http://dl.acm.org/citation.cfm?id=1181849.1181855

[17] J. P. Sauvé and O. L. Abath Neto, "Teaching software development with atdd and easyaccept," *SIGCSE Bull.*, vol. 40, no. 1, pp. 542–546, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1352322.1352317

[18] M. Sztipanovits, K. Qian, and X. Fu, "The automated web application testing (awat) system," in *Proceedings of the 46th Annual Southeast Regional Conference*, ser. ACM-SE 46. New York, NY, USA: ACM, 2008. [Online]. Available: http://doi.acm.org/10.1145/1593105.1593128 pp. 88–93.

[19] K. A. Naudé, J. H. Greyling, and D. Vogts, "Marking student programs using graph similarity," *Computers & Education*, vol. 54, no. 2, pp. 545–561, 2010.

[20] T. Wang, X. Su, Y. Wang, and P. Ma, "Semantic similarity-based grading of student programs," *Information and Software Technology*, vol. 49, no. 2, pp. 99–107, 2007.

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

[22] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Empirical Software Engineering and Measurement (ESEM), 2011 International Symposium on*. IEEE, 2011, pp. 117–126.

[23] C. Cadar, D. Dunbar, and D. R. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[24] E. Andersen, "Uclibc," 2010. [Online]. Available: http://www. uclibc. org

[25] B. Meyer, *Object-oriented Software Construction (2nd Ed.)*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1997.