PROGRAM REPAIR FOR INTELLIGENT TUTORING AND PROGRAMMING EDUCATION

ZHIYU FAN

(B.E., Southern University of Science and Technology)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

SCHOOL OF COMPUTING

NATIONAL UNIVERSITY OF SINGAPORE

2024

Advisor:

Professor Abhik Roychoudhury

Declaration

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Zhiyu Fan

November 27, 2024

First and foremost, I would like to express my deepest gratitude to my PhD advisor, Prof. Abhik Roychoudhury, for his generous and consistent support, rigorous training, and invaluable guidance over the past several years. Abhik taught me the importance of impactful research and how to think with a long-term vision. His imagination has always inspired me to pursue influential research. Beyond this, his wisdom has greatly enhanced my communication skills, critical thinking, and professional attitude. I am profoundly grateful for his patience and valuable advice, which have been instrumental in my achievements.

I would like to especially thank my undergraduate advisor, PhD co-advisor, and friend, Prof. Shin Hwei Tan. Shin Hwei introduced me to the world of research and encouraged me to pursue an academic path. She is a talented educator who imparted essential knowledge and concepts in Software Engineering, significantly improving my research abilities. Her guidance helped me navigate the most challenging periods of my early PhD. Without her support, I would not have reached this stage.

I am also grateful to Prof. Damith Chatura Rajapakse and Prof. Olivier Danvy for their valuable time and insights as members of my thesis committee.

I extend my heartfelt thanks to my fantastic collaborators Yannic Noller, Umair Z. Ahmed, Xiang Gao, Jooyong Yi, Omar I. Al-Bataineh, Sergey Mechtaev, Martin Mirchev, Yuntong Zhang, Haifeng Ruan, Zhenxin Huang, Ashish Dandekar, and others for their brilliant minds, fruitful discussions, and significant help. Their contributions have been invaluable to my work.

I sincerely thank Xiang Gao and Ruijie Meng for their kind support and useful suggestions during tough times. I am also grateful to my friends and colleagues Ridwan Shariffdeen, Huan Zhao, Robbie Lin, Dylan Wolff, Jiang Zhang, Jinsheng Ba, Andreea Costea, Gregory J. Duck, Jiawei Wang, Yu Liu, Xiao Liang Yu, Mingyuan Gao, Zhen Dong, Bo Wang, and Lanfeng Liang for making my journey at NUS and Singapore memorable.

I would like to thank my parents and family for their unwavering support and guidance throughout my life. Their encouragement has enabled me to pursue my academic interests. I also thank my childhood friends Kai Guo and Jiyue Tu for their lifelong companionship.

Last but not least, I express my deepest love and sincere thanks to my wife, Qing Xiao, for her unconditional love. Her understanding, support, patience, and

encouragement have been my strongest shields and driving force. Her luminous presence has graced my life ever since we met, and I am profoundly grateful for her.

Contents

Abstract

Automated program repair is a technology for automated rectification of errors and vulnerabilities in programs. This technology can be used for intelligent tutoring of programming – where student assignments are compared with reference assignments to compute feedback to students. In this thesis, we report on the design, implementation, and real-life evaluation, as well as the experience of hundreds of students, for such an intelligent tutoring system. We also show how the core program repair technology can combined with grading rubrics to provide automated grading of programming assignments. Such a grading goes beyond test-suite based grading since the aim is to find concepts which are not understood by students. The intelligent tutoring system has been conducted as a real-life project in the course of teaching and has been used by tutors / students of a first-year programming course at the National University of Singapore. In the last part of the thesis, we report on the experience of using the intelligent tutoring system in conjunction with Large Language Models (LLMs) to teach data structure and algorithms in a second-year course. Since well-known data structures have been ingested by LLMs – this could provide a forward-looking perspective about the teaching of algorithms in the future. The intelligent tutoring system powered by program repair is aided by LLMs to provide algorithm level feedback.

List of Figures

List of Tables

Chapter 1

Introduction

Computer Science education has experienced a tremendously increasing number of enrolments in the past decade, and there is no sign of slowing down the process. This phenomenon might occur because of various reasons. For example, the attractive compensation in the job market as a software developer, the high demand for programming and data analytic skills for professional career development, the inevitable trend of AI applications in every domain, etc. As a consequence, the enrolments imposed significant stress on computer science education, especially programming teaching, despite the advancements in digital technology. Online learning platforms (e.g., Coursera, Khan Academy, EdX, and Udacity) are one of the earliest attempts to scale the teaching capability. These platforms provide programming courses known as Massive Open Online Courses (MOOC) taken by thousands of learners worldwide. Learners can register for a MOOC at any time, anywhere to learn about programming with a lower entry barrier. However, MOOCs still struggle with the inherent challenge: *Lack of individual learning feedback for students, particularly novice students.* Automatically providing high-quality learning support for individual students and alleviating the workload of teaching teams have become more crucial in computer science education than ever.

Automated program repair (APR) [38] is a well-studied technology that automatically rectifies bugs in large software. Given a buggy problem P_b, and a test-suite T. APR aims to find a set of edit changes, such that P_b passes all tests in T after applying those edits. The changes made to P_b are referred as *fixing patches*. In addition to fixing bugs in large software codebases, a prior study [107] has shown that APR can also be applied in the computer science education context to generate

feedback for students' programming assignments automatically. Given an incorrect student's solution for a particular programming assignment and a test suite for assessment, the fixing patches generated by APR techniques for the incorrect solution can be seen as basic barebone feedback/suggestions that guide the student toward the correct solution. However, the existence of a complete set of high-quality test cases to validate the repairs as feedback cannot be assumed. Over-fitting the repair to an incomplete specification is a well-known problem of test-based APR tools [35]. The overfitted repairs may pass the given test suite for validation, but still be buggy on the input outside the test suite. Generating complex incorrect feedback that merely passes explicit tests can potentially confuse the end-users, particularly in the computer science education context, where the end-users are often novice students.

In addition to trustworthy programming feedback for individual students, the huge enrolments also place significant strain on teaching resources such as manpower to grade students' programming assignments. Hence, there is also an urgent need to scale up the grading procedure both efficiently and accurately. One widely adopted common practice in existing programming assessment systems (e.g., online judge system) evaluates student's performance based on the number of passing tests in a given test suite (i.e., test-based automated grading). However, test-based systems are not ideal, because of the fact that a manual inspection of students' attempts that fail the majority of the test suite which is deemed as severely incorrect can reveal some conceptually correct understanding of the students. This strict assessment may discourage students from improvement.

Beyond the scope of supporting first-year programming courses, another typical challenge in computer science education is the design of software engineering development projects. Software engineering is one of the foundation courses for a Computer Science student. It provides students with hands-on software development opportunities as a team project, which helps them get experience and be prepared before entering the job market. Despite the importance, software engineering projects are often a one-time effort to build a toy software for a specific course. Students can hardly get the accomplishment of contributing to a mature software codebase and experience the software evolution, which is more crucial in the industrial context.

To handle the above limitations in computer science education, the goals of this thesis are to design and implement a full-fledged, ready-to-use Intelligent Tutoring

System that scales high-quality programming teaching by generating trustworthy feedback to students as well as producing precise automated programming assignment assessments for instructors and tutors. Furthermore, the Intelligent Tutoring System can serve as an in-house software engineering project framework that iteratively evolves over time. To achieve this goal, we propose a series of tightly connected high-level ideas and techniques:

- **Verified Repair as Programming Feedback.** To provide trustworthy feedback for student programming assignments, we propose a general approach to verified repair. Verified repair engenders greater trust in the output of the automatic repair tool, which has been identified to be a key hindrance in the deployment of automated program repair [83]. In this work, we developed a tool Verifix, and we show that verified repair from Verifix is feasible and achievable in a reasonable time scale for student programming assignments at a large public university. This shows the promise of using verified repair to generate high-confidence live feedback in programming pedagogy settings.

- **Conceptual Programming Assignment Assessment.** Since test-based grading may not adequately capture the student's understanding of the programming concepts needed to solve a programming task, we propose the notion of a concept graph which is essentially an abstracted control flow graph. Given the concept graphs extracted from a student's solution and a reference solution, we define concept graph matching and comparing differing concepts. Specifically, the concept-based grading is (experimentally) shown to be closer to the grade manually assigned by the tutor. Apart from grading, the concept graph used by our approach is also useful for providing feedback to struggling students, as confirmed by our user study among tutors. In this work, we present and implement ConceptGrader, a new automated grading approach that uses the differences between the student concept graph and reference concept graph to generate a score for a given incorrect student submission.

- **Design of Intelligent Tutoring System for Programming.** Based on the research advances in recent years, specifically in automated program repair and synthesis, we have designed and built an intelligent tutoring system that

has the capability to provide automated personalized feedback for students. The main techniques rely on the precise low-level patches from automated program repair and high-level conceptual explanation by the natural language description capability from large language models. We discuss the design principles and architecture of the Intelligent Tutoring System and show its effectiveness in CS-1 educational settings through a live deployment. The previously developed Verifix and ConceptGrader are included as essential components that provide program repair and autograding capability in our Intelligent Tutoring System.

- **Linking Software Engineering and Programming Teaching.** Further, on top of the built Intelligent Tutoring System, we designed a software engineering course that naturally links software engineering and programming teaching. The development projects in the SE course guide third-year undergraduate students in incrementally developing more features for the Intelligent Tutoring System over several years. Each year, students will make contributions that improve the current implementation, while at the same time, we can deploy the current system for junior students to use for learning programming. This work describes our teaching concept and our experience in the advanced computing course CS2040S at the National University of Singapore. This software engineering project for the students has the key advantage that the users of the system are available in-house (i.e., students, tutors, and lecturers from the first-year programming courses). This helps to organize requirements engineering sessions and builds awareness about their contribution to a "to–be–deployed" software project. In this multi-year teaching effort, we have incrementally built a tutoring system for first-year programming courses.

Contributions. The proposed approaches in this thesis are delivered as an Intelligent Tutoring System for Programming (ITS). It impacts the current computer science education by improving the quality of auto-generated feedback and providing precise assessments for programming assignments. Specifically, the ITS can efficiently and confidently repair students' incorrect programming submissions with a formal guarantee and high success rate, which is essential for practical usage. Moreover, the ITS can assess students' performance in programming based on high-level conceptual

understanding to produce positive learning feedback with the proposed concept graphs and automated program repair engines. Finally, the ITS can be seen as a mature software engineering project framework that computer science students can contribute to, which facilitates software engineering education and eventually benefits the teaching of other CS courses. The ITS has been deployed in CS1010S Programming Methodology at the National University of Singapore, and Monash University. In summary, this thesis can scale the capability and reduce the burden of computer science education.

Research Scope. This thesis aims to provide trustworthy and conceptual feedback to students and instructors for programming assignments in computer science education. Although the presented feedback generation approaches are explored specifically for first-year introductory programming courses, they present high-level research ideas that have the potential to be extended in programming assignments of more advanced CS courses. For example, data structure, database management, and networking. To provide constructive feedback to students, while the proposed approaches do not guarantee to generate feedback for *all incorrect submissions*, but they do guarantee that every generated feedback is trustworthy from the pedagogy perspective. In addition to feedback generation, we also applied the presented techniques in automated programming assignment assessment to alleviate the high workload of the teaching team. These two usage scenarios are oriented at different stakeholders but target common goals, which is to scale the programming teaching capability and provide a better programming learning environment. Moreover, we also integrated the proposed approaches into a self-sustained Intelligent Tutoring System that can be evolved and evaluated iteratively over a longer period. This ITS has been deployed at the National University of Singapore to gain practical experience in a real-world context. Beyond first-year (CS-1) programming courses, we believe the proposed ideas can also be generalized to other computer science courses.

Thesis Organization. The remainder of this thesis is organized as follows. We first provide an overview of the existing technical background in automated intelligent programming education, including automated program repair/synthesis, feedback generation, and automated assignment assessment in Chapter 2. In Chapter 3, we present our approach that addresses the overfitting problem of program repair in

computer science education via the program equivalence checking technique. Chapter 4 introduces a conceptual-level automated grading mechanism for programming assignments based on our novel concept graph that assesses students' conceptual understanding. Chapter 5 describes the design principles and architecture of our Intelligent Tutoring System for programming education and our experience of using it in CS1010S Programming Methodology. Chapter 6 describes our novel idea of linking software engineering and introductory programming teaching via the gradual implementation and improvement of our Intelligent Tutor System for Programming which is designed in Chapter 5. Chapter 7 presents the related work of our proposed techniques. Chapter 8 concludes this thesis and discusses the potential research directions for future improvement. The detailed introduction of our Intelligent Tutoring System (ITS) for Programming and Algorithms Education can be found at `https://nus-its.github.io/index.html`.

Chapter 2

Background

In this chapter, we introduce the background knowledge for this thesis, including automated program repair, programming feedback generation.

2.1 Automated Program Repair

Automated program repair (APR) is an emerging area for automated rectification of programming errors [38]. APR techniques take as inputs a buggy program and a correctness specification (often provided in the form of a test suite). APR techniques aim to produce a fixed program by slightly changing the buggy program to satisfy the given specification (i.e., passing all given tests). There are a few well-studied automatic program repair techniques in the literature to generate patches for buggy programs.

2.1.1 Search-Based Repair

Search-based repair tools (e.g., GenProg [58], Prophet [64], TBar [61]) search for correct patches that can pass all the given tests among a pre-defined patch space S. Typically, the patch space S is curated with a set of program repair operators [58] or manually-summarized repair templates [61]. These repair operators define how we can mutate or change the buggy programs to derive different program variants that form the patch space S. Once the patch space S is determined, different search algorithms can be leveraged to find the correct patches from S. For example, evolutionary search [58, 109], random search [80] and enumeration search [61, 55]. The patch search procedure terminates when a patch makes the buggy program pass

all tests is found, or the entire patch space S is evaluated.

2.1.2 Semantic-Based Repair

Semantic-based repair tools (e.g., SemFix [75], Angelix [69]) generate patches by formulating a repair constraint that needs to be satisfied based on a given test suite specification by leveraging symbolic execution and then solving the repair constraint to generate patch via program synthesis. Semantic-based repair tools first replace the suspicious expressions (usually identified by fault localization [1]) of a program with symbolic expressions, then execute each test from the beginning. When the execution reaches the suspicious location, it then starts to execute the program symbolically. This symbolic execution process produces a set of symbolic formulas representing the program specification inferred from the test suite. Then the program synthesis techniques e.g., component-based synthesis [49], are used to find a concrete program expression that satisfies the inferred constraints which were collected from the given test suite.

2.1.3 Learning-Based Repair

The application of deep learning techniques in program repair has been explored in the past few years. This line of techniques [101, 42, 20, 60, 65, 50, 115] formulates the automated program repair as a next token prediction problem. They often train a deep learning model by collecting corresponding bug-inducing and bug-fixing commits from open-source software repositories with the goal of generating the correct patch for any unseen bug. During the training process, the patch generation is often guided by a specific representation of code syntax (e.g., abstract syntax tree) and semantics to predict the next tokens that are most likely to be a correct patch. Different from search-based repair tools, the learned repair strategies from historical human bug-fixing experiences tremendously enrich the patch space so that can go beyond the predefined repair operators.

2.1.4 Overfitting in Program Repair

Overfitting is one of the well-known and challenging issues in automated repair tools that rely on test suite to serve as intended program behavior (i.e., test-based

APR tools). This is because the test suite is often limited and incomplete program specifications, which only represents an input-output relationship of a given program. It is possible that the patched program passes all given tests by generating incomplete program fixing patches, yet the patched program still fails test cases outside the given test suite. We call those incorrect program patches that pass the validation test suite as *plausible patches*. There are a few lines of work targeted to alleviate the overfitting problem in program repair because it is critical for APR systems to be deployed in real-world codebase. One typical approach [64, 106] is to rank the generated patches with heuristics or machine learning algorithms so that the correct patches are more likely to be found and suggested to developers earlier. Another approach [91] introduces the Anti-Pattern concept, which disabled the generation of specific incorrect program patches in search-based APR tools. There are also attempts to combine fuzzing [16] strategies to enhance the given test suite and filter out more potentially plausible patches.

2.2 Automated Feedback Generation

The typical high student-to-teacher ratio in computer science education makes it challenging to manually write personal feedback to guide students' learning processes for programming assignments. This high time latency (usually 1-2 weeks) between finalizing submission and receiving feedback may have a negative effect on student's learning motivation. Given a programming assignment and a student's submission. We wish to automatically generate constructive personalized feedback regarding specific students' submissions in a short period so that the student can identify their mistakes and further improve.

2.2.1 Debugging-Based Feedback Generation

The debugging-based approach is a systematic method to provide basic feedback for students. There are a few different typical debugging-based feedback generation systems that leverage program debugging output as feedback to guide students to rectify their programming assignments. For example, one common strategy is to predefine a comprehensive test suite for each programming assignment, when the student submits an incorrect solution, the failed test cases can be seen as a

hint and shown to students. This test-based approach is widely adopted in online judge platforms such as LeetCode and online programming teaching platforms such as Coursemology [85]. There are also other approaches that suggest syntax-level feedback to students, which is often based on compiler or static analyzer output (e.g., warning messages, etc). They focus on postprocessing the complex compiler output into easy-to-read natural language hints for students to resolve syntactical errors. For example, incorrect function usage, use of undefined variables, unexpected return types, etc. This is often achieved by manually defining natural language templates and parsing rules for specific warning/error types.

2.2.2 Program Equivalence based Feedback Generation

Program equivalence checking [22] is a technique that formally proves whether two programs are semantically equivalent, and it is applied in CodeAssit [53] to find the behavioral difference between the instructor's reference solution P_r and the student's solution P_s. The identified discrepancies in input-output relations are then reported as feedback to students. CodeAssit [53] manually inspects correct reference solutions and repairs incorrect student's solutions at program contract granularity instead of generating a concrete expression that the student needs to change. Moreover, CodeAssist was specifically designed to work for dynamic programming assignments, which further limits its capability.

2.2.3 Program Repair based Feedback Generation

Automated program repair techniques are originally designed to fix bugs in large codebases for experienced software developers. Prior work [107] has shown the potential of applying general-purpose APR tools to fix introductory programming assignments and provide the generated program patches as feedback for students. After that, new automated program repair techniques have been proposed to specifically help novice programmers to fix their programming mistakes. Different from general-purpose APR tools, those APR tools that specialize in fixing programming assignments often assume the existence of reference solutions as additional program specifications in addition to test suites, which are available in the computer science education context. Typical APR-based feedback generation tools e.g., Refactory [47],

Clara [41] and Sarfgen [99] take in the reference solution P_r and incorrect solutions P_i for a particular programming assignment. Then they try to align the program structure between P_r and P_i based on control-flow structure or basic-block similarity to extract fine-grained program specification that P_i needs to satisfied from P_r. Further, they leverage the code snippet in P_r as available patch ingredients to fix the bugs in P_i, which is often done by program synthesis [39]. In addition to program synthesis based patch generation, there are also approaches to repair student's programming assignments. For example, learning-based repair systems [42, 82] used deep learning and programming-by-example [43] respectively to learn program transformation rules to fix incorrect submissions from historical data, whereas error model based system [87] requires course instructor to define how an incorrect solution can be changed for each specific programming task, then search for a correct modification efficiently.

2.2.4 Large Language Model based Feedback Generation

The emergence of powerful LLMs has gained popularity in the computer science education field due to their ability in code generation and code explanation. Researchers have proposed various methods to leverage LLMs for feedback generation in programming education [12, 66, 59, 92, 19, 44, 54, 56, 62, 77]. For example, Balse et al [12] and Hellas et al [44] found that LLMs struggles to identify all issues in student's questions and false positives are common in the feedback generated by LLM and many works [92, 59, 77] focus on generating feedback on syntax problems and error messages. While [66, 62, 54] tried to build LLM-based feedback generation systems that are capable of handling general students' questions. These systems heavily rely on specifically curated prompts, and it remains unclear how effectively they can be adopted by educators worldwide.

Chapter 3

Verified Repair of Programming Assignments

3.1 Introduction

CS-1, the introductory programming course, is an undergraduate course offered by Universities and Massive Open Online Courses (MOOCs) across disciplines. Several programming assignments are typically attempted by the students as a part of this course, which are evaluated and graded against pre-defined test-cases. Given the importance of programming education and the difficulty of providing relevant feedback for the massive number of students, there has been increasing interest in automated program repair (APR) techniques for providing automated feedback to student assignments [107, 41, 99, 47, 53, 87].

Existing approaches and their limitations Table 3.1 provides a summary of state-of-the-art APR works for introductory programming assignment, and compares them with our approach Verifix. The repair rate of the state-of-the-art techniques [41, 99, 47] is around 90%. However, different from general test-based APR technique, these works make certain assumptions such as the presence of multiple reference programs and a high quality tests.

Many student assignment feedback generation approaches [41, 99, 47] assume the existence of a complete set of high quality test-cases to validate their repairs. Over-fitting the repair to an incomplete specification is a well known problem of test-based APR tools [38, 81, 108]. Prior studies have shown that trivial repairs such as functionality deletion alone can achieve ~50% repair success rate on buggy

Tool	Completely Automated	Beyond Identical Reference CFG	Verified Repair	Target Language	Tool Availability	Dataset Availability
Clara [41]	✓	✗	✗	C, Python	✓	✗
SarfGen [99]	✓	✗	✗	C#	✗	✗
ITSP [107]	✓	✓	✗	C	✓	✓
Refactory [47]	✓	✓	✗	Python	✓	✓
CoderAssist [53]	✗	✓	✓	DP for C	✓	✗
Verifix	✓	✓	✓	C	✓	✓

Table 3.1: Programming assignment repair tools comparison. Most existing APR tools are completely automated and rely on test case evaluation (generate unverified repair).

student programs given a weak oracle [21]. Generating complex incorrect feedback that merely passes all tests can potentially confuse novice students more than expert programmers. Indeed, a prior study [107] shows that novice students when provided with incorrect/partial repair feedback that merely passes more tests, have been shown to struggle more, as compared to expert programmers given the same feedback. Hence, we suggest that the feedback given to novice students needs rigorous quality assurance, whenever possible.

In a related vein, some approaches, in particular recent ones [41, 99], assume the existence of multiple reference programs. This assumption is made to overcome the difficulty of generating feedback when the Control-Flow Graph (CFG) structure of the student program is different from the instructor provided reference program.[1] Using multiple reference programs can also diversify the solution space, and thereby a feedback can be made more customized to a student solution [40]. However, the problem is that the existing approaches collect multiple reference programs manually or based on testing (student submissions that pass all tests are considered correct), without formally verifying their correctness. Automatic equivalence checking remains challenging despite recent advances [22].

Insight Many of the aforementioned problems of the existing APR techniques can be addressed with a verified repair. We assume the presence of at least one reference solution, which is always available in educational settings and can be given by an instructor. This setting is simpler than most existing approaches [41, 99, 47] requiring both multiple reference solutions and a test-suite. We then create a verifiably correct

[1]SarfGen [99] and Clara [41] require that the control-flow structure of student and reference programs should be exactly the same. Clara also demands aligned variables to be evaluated into the same sequence of values at runtime.

repair of the student assignment. In other words, the repaired student assignment will be semantically equivalent to the reference assignment given by the instructor. In terms of workflow, the repair engine indicates when it can generate a verified repair as feedback, and when it does, the students can receive a feedback which is guaranteed to be correct. In other words, we can have greater confidence or trust on the feedback generated by the repair tool. Furthermore, student programs that are verified to be correct after repair can be used as additional trustworthy reference programs in future.

Contribution: Verified repair In this chapter, we propose a general approach to verified repair. Verified repair engenders greater trust in the output of the automatic repair tool, which has been identified to be a key hindrance in deployment of automated program repair [83]. We show that verified repair is feasible and achievable in a reasonable time scale (on average 29.5 seconds) for student programming assignments of a large public university. This shows the promise of using verified repair to generate high confidence live feedback in programming pedagogy settings.[2] To the best of our knowledge, ours is the first work to espouse verified repair for general purpose programming education. The only previous attempt on verified repair [53] is tightly tied to a specific structure of programs implementing dynamic programming.

Repair tool: Verifix We build our verified-repair technique by extending the existing program equivalence checking technique. Although automatically proving the equivalence between two programs remains challenging (mainly due to the difficulty of automatically finding loop invariants), we found that student programs are in many cases amenable for equivalence checking. This is because there is usually a reference program whose structure is similar to the student program, as shown in earlier works [41, 99]. Exploiting this, Verifix produces a verified repair. Note that, Verifix performs repair and equivalence checking at once. More concretely, Verifix aligns the incorrect student program with the reference program into an aligned automaton, derives alignment relation to relate the variable names

[2]According to an earlier user study [107], students spend about 100s on average to resolve semantic errors.

```
1   int check_prime(int n)
2   {
3     if (n == 1)
4       return 0;
5     int j;
6     for(j=2; j<n; j++)
7     {
8       if (n%j == 0)
9         return 0;
10    }
11    return 1;
12  }
```

(a) A reference program

```
1   int check_prime(int n)
2   {
3
4
5     int i;
6     for(i=1;i<=n-1;i++)
7     {
8       if (n%i == 0)
9         break;
10    }
11    return 1;
12  }
```

(b) An incorrect student program

```
1   int check_prime(int n)
2   {
3     if (n == 1)
4       return 0;
5     int i;
6     for(i=2;i<=n-1;i++)
7     {
8       if (n%i == 0)
9         return 0;
10    }
11    return 1;
12  }
```

(c) The Verifix-generated repair

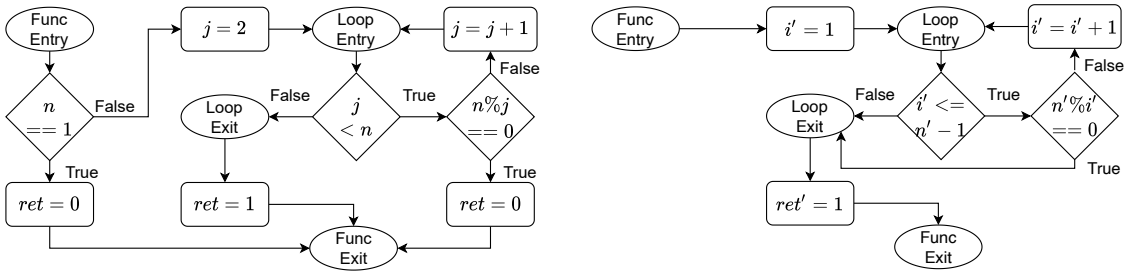
Figure 3.1: Motivating example for the *Prime Number* programming assignment. Existing tools such as Clara [41] and Sarfgen [99] cannot repair the incorrect student program in Fig 3.1(b) since its Control-Flow Graph (CFG) differs from the CFG of instructor designed reference program in Fig 3.1(a). Our tool Verifix generates the repaired program in Fig 3.1(c), which is verifiably equivalent to the reference implementation, due to superior Control-Flow Automata (CFA) based abstraction.

of the two programs, and suggests repairs for the code captured by the edges of the aligned automaton via Maximum Satisfiability-Modulo-Theories (MaxSMT) solving. We use MaxSMT to find a minimal repair. Our approach can generate a program behaviourally equivalent to the reference program while preserving the original control-flow of the student program as much as possible. This leads to smaller patches/feedback which we believe are easier to comprehend, in general. We evaluate our approach on student programming submissions curated from a widely used intelligent tutoring system. Our approach produces small-sized verified patches as feedback, which, whenever available, can be used by struggling students with high confidence. Our tool Verifix is available at `https://github.com/zhiyufan/Verifix`.

3.2 Overview

Consider a simple programming assignment for checking whether a given number n is a prime number. Figure 3.1(a) shows a reference implementation prepared by an instructor, and Figure 3.1(b) shows an incorrect program submitted by a student.

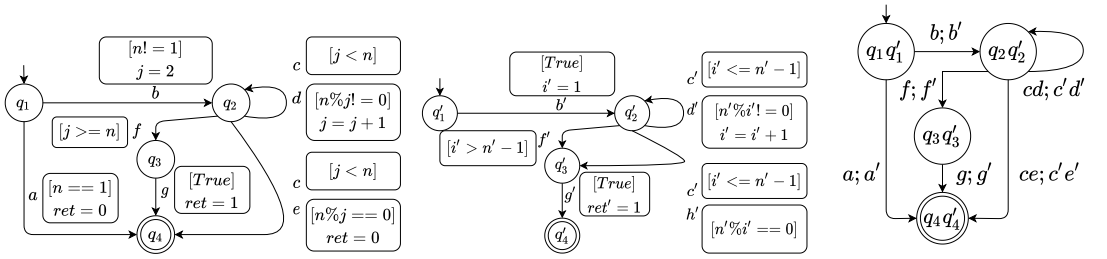
Limitations of the Existing Approaches The state-of-the-art approaches such as Clara [41] and Sarfgen [99] make the same-control-flow assumption described as



(a) CFG of the reference program in Fig 3.1(a)

(b) CFG of the incorrect student program in Fig 3.1(b)

Figure 3.2: Control Flow Graph (CFG) of the reference and incorrect program listed in Fig 3.1. Incorrect program CFG in Fig 3.2(b) differs from reference program CFG in Fig 3.2(a) due to a missing return node. Existing tools like Clara [41], Sarfgen [99] cannot repair the incorrect program.



(a) Reference CFA $\mathcal{A}_R$ **(b) Student's CFA $\mathcal{A}_S$** **(c) Aligned CFA $\mathcal{A}_F$**

Figure 3.3: Control Flow Automata (CFA) of the reference and incorrect program listed in Fig 3.1. CFA $\mathcal{A}_R$ of reference program in Fig 3.3(a) is structurally aligned with CFA $\mathcal{A}_S$ of student program in Fig 3.3(b) to obtain an aligned CFA $\mathcal{A}_F$ in Fig 3.3(c).

follows.

To perform a repair, a given incorrect program and its reference implementation should have the same control-flow structure.

Clara fails to repair the incorrect program shown in Figure 3.1(b) when the reference implementation shown in Figure 3.1(a) is used, reporting that the structures of these two programs do not match. The CFGs of this incorrect program and its reference program are shown in Figure 3.2(b) and Figure 3.2(a), respectively. Notice that that in the reference CFG (Figure 3.2(a)), the LoopExit node has one incoming edge, whereas in the student program's CFG (Figure 3.2(b)) the matching LoopExit node has two incoming edges where the additional edge of Figure 3.2(b) comes from "$n'\%i' == 0$". The problem is that "$n'\%i' == 0$" does not match "n %

j ==0" since the downward edge of node "n % j ==0" does not reach LoopExit, unlike in "$n'\%i' == 0$", and hence the structures of the two CFGs do not match. The fact that Clara treats a loop-free segment of the code as a single block does not help. In Clara, two adjacent nodes, "n % j ==0" and "ret = 0", of Figure 3.2(a) are grouped together, but the outgoing edge of this group still does not reach LoopExit.

A common approach that has been used to overcome this problem is to use multiple reference programs of diverse control-flow structures [41, 99, 53]. Since it would be labor-intensive for an instructor to prepare multiple reference implementations, recent works (e.g., [41, 99]) gets around this problem by using student submissions. That is, student submissions that pass all tests are added into a pool of reference implementations. However, this approach exposes students to the risk of getting wrong feedback generated based on an incorrect program that happens to pass all tests.

Our Approach We show how we address the aforementioned limitations. Essentially, we do not make the same control-flow-structure assumption. Instead, we conduct repair with Control Flow Automata (CFA) where its nodes represent program locations and its edges represent guarded actions. Figure 3.3 shows examples of CFAs, as will be described shortly in Section 3.2.1. Also, we extend the existing equivalence checking technique into a verified repair technique. We traverse each edge of the CFA obtained from a student submission and check its semantic equivalence with the corresponding edge of the CFA obtained from a reference program. Note that each edge represents a loop-free segment of a program. Equivalence checking is performed by encoding the problem into an SMT (Satisfiability Modulo Theories) formula. If equivalence checking fails, we reformulate the equivalence checking problem into a repair problem; we allow the expressions of the student submission to be replaced with the expressions of the reference program (after converting variable names). The number of replacements is minimized by encoding the repair problem into a MaxSMT (Maximum Satisfiability Modulo Theories) formula.

In the following, we show how our repair algorithm works through the following three phases: the setup phase, the verification phase, and the repair phase. The last two steps occur simultaneously as explained in the following.

3.2.1 Setup Phase

In the setup phase, we model the given reference and student programs as Control Flow Automata (CFA) with the nodes representing control-flow locations and the edges representing guarded actions. Figure 3.3(a) and 3.3(b) show the CFA for the reference program ($\mathcal{A}_\mathcal{R}$) and the CFA for the student program ($\mathcal{A}_\mathcal{S}$), respectively. Notice that each edge of a CFA is annotated with a sequence of guarded actions. For example, in Figure 3.3(a), the edge between q_1 and q_2 is annotated with "$[n\,!=1]\,j=2$" where an assignment command $j=2$ is guarded with the conditional expression $n\,!=1$. In the figure, we label this guarded action with "b". As another example, the self-edge of node q_2 is annotated with a sequence of two guarded actions, c and d, which indicates that c and d should be executed in sequence. As in the case of c, a guarded action can have only a conditional expression φ, which means that the NOP command is guarded with φ.

To perform verification/repair in the next phase, we build an aligned CFA $\mathcal{A}_\mathcal{F}$ by aligning the nodes and edges of $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{S}$. Figure 3.3(c) shows the aligned CFA for our running example. Notation q_1q_1' used in the entry node of Figure 3.3(c) denotes that node q_1 of $\mathcal{A}_\mathcal{R}$ and node q_1' of $\mathcal{A}_\mathcal{S}$ are aligned with each other. The other nodes of $\mathcal{A}_\mathcal{F}$ are interpreted similarly. Meanwhile, notation $cd;c'd'$ used in the edge between q_2q_2' and q_2q_2' denotes that guarded-command-sequence cd of $\mathcal{A}_\mathcal{R}$ is aligned with guarded-command-sequence $c'd'$ of $\mathcal{A}_\mathcal{S}$. To align nodes and edges, we use lightweight syntax-based approaches, as will be detailed in Section 3.4.1. Recall that the existing approaches [41, 99] fail to handle our running example, due to their same-CFG assumption. We relax this assumption by conducting node alignment and edge alignment separately. In our running example, after aligning node q_1 with q_1' and q_4 with q_4', we conduct edge alignment for the edge between q_1 and q_4 (annotated with guarded action a) by creating a fresh edge between q_1' and q_4' (annotated with a' in Figure 3.3(c)). Similarly, a new edge $c'e'$ is constructed between q_2' and q_4', corresponding to the edge ce between $q2$ and $q4$, during the alignment stage since no such edge exists in the student automata. Conversely, the edge $c'h'$ between q_2' and q_3' of the student's CFA is removed because no matching edge exists in the reference automata. Our experimental results show that this simple extension alone reduces the structural alignment mismatch rate by 13% (see Table 3.4).

While in our example, only one aligned automaton can be constructed, there can be multiple ways to align $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{S}$ when multiple edges exist between two aligned nodes (Figure 3.5 shows an example). In such a case, we construct all possible aligned CFAs, and in the next phase (verification/repair phase), each aligned automaton is investigated to generate a minimal repair.

To conduct verification/repair, we also need to align variables used in $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{S}$. To align variables, we use a syntax-based approach similar to [99]. For each edge of $\mathcal{A}_\mathcal{F}$, we align variables whose usage patterns are similar to each other (see Section 3.4.2). For example, Verifix infers the following variable alignment predicate for the edge $q_1q_1' \rightarrow q_2q_2'$: $\{ret \leftrightarrow ret', n \leftrightarrow n', j \leftrightarrow i'\}$ where ret is a special variable holding the return value of the function under verification/repair.

3.2.2 Verification Phase

We perform verification for all aligned automata $\mathcal{A}_F$. If verification succeeds for $\mathcal{A}_F$ or its repaired variation, semantic equivalence between student and reference programs is guaranteed (see Theorem 1). Verification is performed inductively for individual edge, starting from the outgoing edges of the initial node of $\mathcal{A}_F$ ($a; a'$ and $b; b'$ for our Figure 3.3(c)). More specifically, we perform verification by checking whether $q \sim q'$ (i.e., q is bisimilar to q') holds for each aligned nodes q and q' of $\mathcal{A}_F$.

Consider the edge $q_1q_1' \xrightarrow{b;b'} q_2q_2'$. Given this edge, we should prove the following: when $q_1 \sim q_1'$ is assumed, $q_2 \sim q_2'$ holds after executing $b; b'$. We achieve this by checking

$$\varphi^1_{edge} : \phi_{q_1q_1'} \wedge \psi_r \wedge \psi^1_s \wedge \neg\phi_{q_2q_2'}$$

where $\phi_{q_1q_1'}$ and $\phi_{q_2q_2'}$ denote the variable alignment predicates at node q_1q_1' and q_2q_2', respectively.

$$\begin{aligned} \phi_{q_1q_1'}\colon &\quad (ret_0 = ret_0') \wedge (n_0 = n_0') \wedge (j_0 = i_0') \\ \phi_{q_2q_2'}\colon &\quad (ret_1 = ret_1') \wedge (n_1 = n_1') \wedge (j_1 = i_1') \end{aligned}$$

Meanwhile, ψ_r and ψ^1_s denote the guarded actions of b and b', respectively, in a Single Static Assignment (SSA) form, where

$$\begin{aligned} \psi_r\colon &\quad (n_0 \neq 1 \implies j_1 = 2) \quad \wedge \quad (\neg(n_0 \neq 1) \implies j_1 = j_0) \\ \psi^1_s\colon &\quad (True \implies i_1' = 1) \quad \wedge \quad (\neg True \implies i_1' = i_0') \end{aligned}$$

Block	Student Transition		Repaired Transition	
a'	$\emptyset$		$[n' == 1]$	$ret' = 0$
b'	$[True]$	$i' = 1$	$[n'! = 1]$	$i' = 2$
c'	$[i' <= n' - 1]$		$[i' <= n' - 1]$	
d'	$[n'\%i'! = 0]$	$i' = i' + 1$	$[n'\%i'! = 0]$	$i' = i' + 1$
e'	$\emptyset$		$[n'\%i' == 0]$	$ret' = 0$
f'	$[i' > n' - 1]$		$[i' > n' - 1]$	
g'	$[True]$	$ret' = 1$	$[True]$	$ret' = 1$
h'	$[n'\%i' == 0]$		$\emptyset$	

Table 3.2: Incorrect student blocks and their corresponding repairs generated by Verifix, after multiple rounds of edge verification-repair of Figure 3.3 aligned automaton. The blocks a' and e' are created in the automata, while the block h' is removed.

If φ^1_{edge} is satisfiable, then $q_2 \sim q'_2$ does not hold, indicating verification failure. We check the satisfiability of φ^1_{edge} using an off-the-shelf SMT solver, Z3 [73].

3.2.3 Repair Phase

For our running example, the SMT solver Z3 finds that φ^1_{edge} is satisfiable under a certain assignment ϕ^1_{ce} which is

$$\phi^1_{ce} : n_0 = n'_0 = 1, j_0 = i'_0 = 0$$

where ϕ^1_{ce} can be viewed as a counter-example to the edge verification. When ϕ^1_{ce} holds, variable j_1 of the reference program has a value 0 (since $\neg(n_0 \neq 1) \implies j_1 = j_0$) by ψ_r), whereas variable i'_1 of the student program (aligned with j_1) has a different value 1 (since $True \implies i'_1 = 1$ by ψ^1_s), violating $\phi_{q_2 q'_2}$. Using this counter-example, we perform a repair based on counter-example-guided inductive synthesis or CEGIS strategy [89] (see Section 3.5.2). Following CEGIS strategy, we look for a repair of ψ^1_s which rules out the counter-example ϕ^1_{ce}. Verifix returns two potential repair candidates.

$$\psi^2_s\text{:}\quad (False \implies i'_1 = 1) \quad \wedge \quad (\neg False \implies i'_1 = i'_0)$$
$$\psi^3_s\text{:}\quad (n'_0 \neq 1 \implies i'_1 = 1) \quad \wedge \quad (\neg(n'_0 \neq 1) \implies i'_1 = i'_0)$$

When ψ^2_s (or ψ^3_s) is substituted for ψ^1_s in $\varphi^1_{edge} \wedge \phi^1_{ce}$ (notice that the original formula φ^1_{edge} is conjoined with ϕ^1_{ce}), the modified formula is not satisfiable, indicating that under the context of the counterexample (i.e., ϕ^1_{ce}), ψ^2_s (or ψ^3_s) is a repair. Notice how the original formula ψ^1_s is repaired. In ψ^2_s and ψ^3_s, the original expression $True$

is replaced with $False$ and $n'_0 \neq 1$, respectively. To obtain $n'_0 \neq 1$, we use the expression $n_0 \neq 1$ appearing in ψ_r, the guarded action for the reference program. This copy mechanism that exploits the existence of a reference program is a de-facto standard technique in recent works [41, 99].

So far, we only showed that ψ_s^2 (or ψ_s^3) is a repair only in the context of ϕ_{ce}^1. It is not known yet whether ψ_s^2 (or ψ_s^3) is a repair in a general context. To check this, we retry edge verification for φ_{edge}^1 after replacing ψ_s^1 with ψ_s^2 (or ψ_s^3) in φ_{edge}^1. In our example, verification attempt fails again for both ψ_s^2 and ψ_s^3 (that is, the repaired φ_{edge}^1 is still satisfiable), and the following new counter-example ϕ_{ce}^2 is obtained.

$$\phi_{ce}^2 : n_0 = n'_0 = 2, i_0 = i'_0 = 0$$

By considering both ϕ_{ce}^1 and ϕ_{ce}^2, Verifix returns a new repair candidate ψ_s^4,

$$\psi_s^4\text{:} \quad (n'_0 \neq 1 \implies i'_1 = 2) \quad \wedge \quad (\neg(n'_0 \neq 1) \implies i'_1 = i'_0)$$

As compared with ψ_s^1, two sub-expressions of ψ_s^1 are repaired. As in ψ_s^3, $True$ is replaced with $n'_0 \neq 1$. Also, $i'_1 = 1$ is replaced with $i'_1 = 2$ based on $j_1 = 2$ appearing in ψ_r. This updated repair candidate ψ_s^4 rules out all counter-examples seen so far, and no further satisfying assignments of φ_{edge}^1 are found. This completes the verification and repair, thereby repairing the edge b' in Figure 3.3(b). The remaining edges are similarly verified/repaired, and Table 3.2 summarizes the buggy student automata $\mathcal{A}_S$ edges and their corresponding repairs generated by our repair tool Verifix.

We note that Verifix generates a minimal repair for each aligned edge under consideration. That is, a generated edge repair modifies the minimum number of expressions required to repair the edge (see Theorem 4). To obtain a minimal edge repair, we formulate a repair problem as a partial MaxSMT problem, as described in Section 3.5.2. Essentially, Verifix tries to preserve as many original expressions as possible, by assigning a higher weight penalty to the original expressions (hence, replacing an original expression increases the cost of repair). While combining minimal edge repairs does not necessarily lead to a globally minimal repair, our experimental results suggest that our greedy approach works well in practice. Verifix tends to generate smaller repairs than a state-of-the-art tool Clara (see Section 3.7.4).

3.3 Program Model

Prior to explaining our alignment and verification-repair procedures, we introduce the key structures used to model programs.

3.3.1 Abstract Syntax Tree (AST)

An Abstract Syntax Tree (AST) consists of a set of nodes representing the abstract programming constructs. With the tree hierarchy, or edges, representing the relative ordering between the appearance of these constructs. We extend the standard AST with special labels for two node types: *Func-Entry* and *Loop-Entry*. Each AST consists of a root node corresponding to a function definition, which is labelled as a function-entry node. Similarly, every loop construct in the AST is labelled as a loop-entry node.

The AST for motivating example shown in Figure 3.1 consists of two labelled nodes: a *Func-Entry* node q_1 which maps to the *check_prime* function definition and a *Loop-Entry* node q_2 which maps to the for-loop construct. We note that some existing APR techniques for programming assignments, like ITSP [107] which uses GenProg [58], operate on program ASTs directly.

3.3.2 Control Flow Graph (CFG)

Existing state-of-art APR techniques like Clara [41] and SarfGen [99] operate at the level of CFG, whose nodes are basic blocks and edges denote control transfer. We extend the standard CFG by introducing four types of special labelled nodes: {*Func-Entry*, *Loop-Entry*, *Func-Exit*, and *Loop-Exit*}; denoting the program states when control enters a function or a loop, and when control exits a function or a loop, respectively. The *Func-Entry* and *Loop-Entry* CFG nodes correspond with control entering AST nodes of the same type. The *Func-Exit* and *Loop-Exit* CFG nodes correspond with the program state after control visits the last child of *Func-Entry* and *Loop-Entry* AST node, respectively. These *Func-Exit* and *Loop-Exit* program states can also be reached by altering the control-flow using *return* and *break* statements, respectively.

Figures 3.2(a) and 3.2(b) depict the CFG of the reference and student program in Figures 3.1(a) and 3.1(b), respectively. These CFGs contain four special nodes

denoting *Func-Entry* (q_1/q_1'), *Loop-Entry* (q_2/q_2'), *Loop-Exit* (q_3/q_3'), and *Func-Exit* (q_4/q_4') program states.

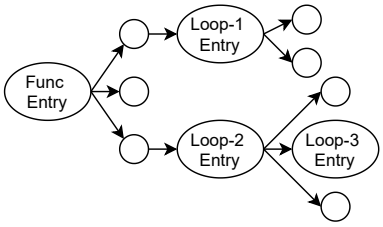
3.3.3 Control Flow Automaton (CFA)

Our tool Verifix operates at the level of the control flow automaton (CFA), often used by model-checking and verification communities [45]. The CFA is essentially the CFG, with code statements labeling the edges of CFA, instead of code statements labeling nodes as in CFG. The nodes of our CFA are annotated with the node types mentioned earlier: *Func-Entry*, *Loop-Entry*, *Func-Exit*, and *Loop-Exit*. The edges of our CFA are constructed by choosing all possible code transitions between the program states in CFG. Depending on the reason for control-flow transition, these edges can be of three types: *normal*, *return* or *break*. Figures 3.3(a) and 3.3(b) depict the CFA modeled using the reference and student CFG in Figures 3.2(a) and 3.2(b), respectively. We provide our precise definition of CFA in the following.

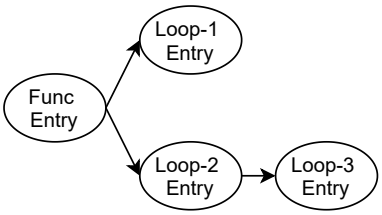
A Control Flow Automaton (CFA) is a tuple of the form $\langle V, E, v^0, v^t, \Omega, \Psi, Var \rangle$, where:

- V : is a finite set of vertices (or nodes) of the automata, representing function and loop entry/exit program states,
- $E \subseteq V \times V$, is a finite set of edges of the automata representing normal, break, and return transitions between program states,
- v^0 : is the initial node representing function entry state,
- v^t : is the terminal node representing function exit state,
- Ω : $\{u \leftrightarrow v \mid \forall u \in V, \exists v \in V\}$, for each function/loop entry node, maintains a mapping to the corresponding exit node,
- Ψ is a mapping from edge e to ψ_e for all edges e, where ψ_e is the set of guarded actions labeling e, and
- Var is the set of variables used in $\bigcup_e \psi_e$.

For edge e in the CFA, ψ_e is thus the code statements labeling e. How we build a CFA is described in Section 3.4.

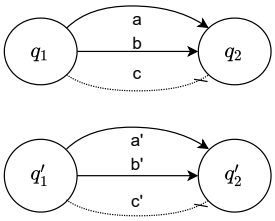


(a) Example AST with labelled and unlabelled nodes.

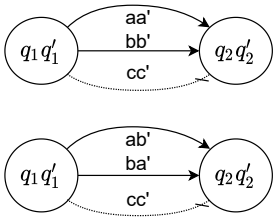


(b) Example AST^L after deletion of unlabelled nodes.

Figure 3.4: Example demonstrating Abstract Syntax Tree (AST) transformation to retain nodes labelled as function and loop entry.



(a) Example $\mathcal{A}_\mathcal{R}$ and $\mathcal{A}_\mathcal{S}$.



(b) Example $\mathcal{A}_\mathcal{F}$ after alignment.

Figure 3.5: Example demonstrating edge alignment. Given node alignment $V : \{q_1 q'_1, q_2 q'_2\}$, the edges are aligned based on type. The single *break* transitions c and c' are aligned with each other, while the multiple *normal* edges are aligned combinatorially to produce two unique aligned automata.

3.4 Aligned Automata

Our methodology for repairing incorrect student programs relies on constructing an aligned automaton $\mathcal{A}_F$ from the given student automaton $\mathcal{A}_S$ and the reference automaton $\mathcal{A}_R$. The construction of the automaton $\mathcal{A}_F$ consists of following steps: (i) modeling the student and reference programs as Control Flow Automaton (CFA) $\mathcal{A}_S$ and $\mathcal{A}_R$, (ii) the structural alignment of $\mathcal{A}_S$ and $\mathcal{A}_R$, and (iii) the inference of the variable alignment predicates.

3.4.1 Structurally Aligning $\mathcal{A}_S$ and $\mathcal{A}_R$

To construct an aligned automaton $\mathcal{A}_\mathcal{F}$, we first conduct node alignment between the nodes of $\mathcal{A}_S$ and $\mathcal{A}_R$. This step is followed by aligning the transition edges between $\mathcal{A}_S$ and $\mathcal{A}_R$. A more detailed description is provided below.

Node Alignment Given two CFAs $\mathcal{A}_\mathcal{S}$ and $\mathcal{A}_\mathcal{R}$, and their corresponding Abstract Syntax Trees AST_S and AST_R for student and reference program, respectively, we

construct node alignment $V : V_S \leftrightarrow V_R$ as follows.

1. Delete all unlabelled nodes from AST_S and AST_R to obtain AST_S^L and AST_R^L, respectively. An AST^L consists of only *Func-Entry* and *Loop-Entry* labelled nodes.

2. If the syntactic tree structures of AST_S^L and AST_R^L are identical with each other, align each node of AST_S^L with AST_R^L and add to V. This step aligns the *Func-Entry* and *Loop-Entry* nodes of $\mathcal{A}_\mathcal{S}$ and $\mathcal{A}_\mathcal{R}$.

3. For each pair of entry nodes (either *Func-Entry* or *Loop-Entry*) that are aligned with each other, their corresponding exit nodes (either *Func-Exit* or *Loop-Exit*) are aligned with each other.

For constructing node alignment V, we first align the labelled nodes of student and reference Abstract Syntax Tree (AST). The labelled AST nodes can be of two types: *Func-Entry* and *Loop-Entry*. These labels are same as those in $\mathcal{A}_S$ and $\mathcal{A}_R$, but we take advantage of the tree structure in the AST. Figure 3.4 demonstrates unlabelled AST node deletion in step-1 through an example, after which only the *Func-Entry* and *Loop-Entry* labelled nodes are retained. For the reference program (respectively student program) listed in Figure 3.1, the labelled AST_R^L (resp. AST_S^L) consists of two nodes $q_1 \rightarrow q_2$ (resp. $q_1' \rightarrow q_2'$). Since both the AST^L trees are structurally the same, the node alignment V consists of $\{q_1q_1', q_2q_2'\}$ after step-2 of node alignment, denoting the *Func-Entry* and the *Loop-Entry* aligned nodes.

The step-3 of node-alignment finally aligns the function and loop exit nodes. Given the student and reference automata in Figure 3.3, q_4, which is the *Func-Exit* node corresponding to q_1, is aligned with q_4', which is the *Func-Exit* node corresponding to q_1'. Similarly, the *Loop-Exit* nodes q_3 and q_3' are aligned, since their corresponding *Loop-Entry* nodes q_2 and q_2' were aligned in step-2.

The node alignment constructed thus, if successful, will lead to a bijective mapping from nodes of $\mathcal{A}_\mathcal{S}$ to nodes of $\mathcal{A}_\mathcal{R}$. Node alignment fails if the two programs have different different function/looping structure from each other. While limited, our approach can handle more diverse programs than the state-of-the-art approaches [41, 99] which require not only bijective mapping between nodes but also bijective mapping between edges. In these approaches, q_4 and q_4' of Figure 3.3

cannot be aligned with each other, since the edge $q_2 \to q_4$ of $\mathcal{A}_\mathcal{R}$ does not have a corresponding edge in $\mathcal{A}_\mathcal{S}$.

Edge Alignment Given two CFAs $\mathcal{A}_\mathcal{S}$ and $\mathcal{A}_\mathcal{R}$, and their corresponding node alignment $V : V_S \leftrightarrow V_R$, we construct an aligned CFA $\mathcal{A}_\mathcal{F}$ by aligning the edges of $\mathcal{A}_\mathcal{S}$ and $\mathcal{A}_\mathcal{R}$. Suppose that $u_S \leftrightarrow u_R$ (i.e., node u_S in $\mathcal{A}_\mathcal{S}$ is aligned with u_R in $\mathcal{A}_\mathcal{R}$) and $v_S \leftrightarrow v_R$. For each edge of type $t \in \{break, return, normal\}$, we treat the following four cases differently.

1. $\mathcal{A}_\mathcal{S}$ has only one edge from u_S to v_S of type t, and $\mathcal{A}_\mathcal{R}$ has only one edge from u_R to v_R of the same type t.

2. Only $\mathcal{A}_\mathcal{R}$ has an edge from u_R to v_R of type t, while $\mathcal{A}_\mathcal{S}$ has no edge from u_S to v_S of type t.

3. Only $\mathcal{A}_\mathcal{S}$ has an edge from u_S to v_S of type t, while $\mathcal{A}_\mathcal{R}$ has no edge from u_R to v_R of type t.

4. None of the above matches, and $\mathcal{A}_\mathcal{S}$ (or $\mathcal{A}_\mathcal{R}$) has multiple edges from u_S to v_S (or from u_R to v_R) of type t.

In the first case, we simply align the matching edges. For example, in Figure 3.3, $\mathcal{A}_\mathcal{R}$ contains only one *normal* edge b between q_1 and q_2 and $\mathcal{A}_\mathcal{S}$ contains only one *normal* edge b' between q'_1 and q'_2. Hence, the aligned CFA $\mathcal{A}_\mathcal{F}$ has an edge $b; b'$ as shown in Figure 3.3(c). An example of the second case is shown with the two nodes, $q_1q'_1$ and $q_4q'_4$, of $\mathcal{A}_\mathcal{F}$. While $\mathcal{A}_\mathcal{R}$ has one edge a between q_1 and q_4, $\mathcal{A}_\mathcal{S}$ has no edge between q'_1 and q'_4. In this case, we insert an edge $a; a'$ to $\mathcal{A}_\mathcal{F}$ where a' has an empty guarded action. The third case is the opposite of the second one. In this case, we remove the edge between u_S and v_S since there is no matching edge in the reference automata.

Lastly, in the fourth case, there exist several possible edge alignments of the order of $\binom{M}{N} \times N!$, where M is the number of edges from $u_R \to v_R$ and N is the number of edges from $u_S \to v_S$. Figure 3.5 demonstrates this case through an example, resulting in two possible edge alignments. The single *break* transitions c and c' are aligned with each other, while the remaining *normal* edges (i.e., a, b, a' and b') are

aligned combinatorially to produce two unique aligned automata. When multiple aligned automata can be constructed, we choose the edge alignment which maximizes the number of verified-equivalent edges in the resultant aligned automaton $\mathcal{A}_{\mathcal{F}}$. The formal structure of aligned automaton is described in the following.

The automaton $\mathcal{A}_{\mathcal{F}}$ that results from aligning the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ is a tuple of the form $\langle V, E, v^0, v^t, \Omega, \Psi, Pred \rangle$, where:

- $V : V_S \leftrightarrow V_R$, is a finite set of one-to-one bijective mappings between the nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$,
- $E \subseteq V \times V$, is a finite set of edges representing normal, break, and return transitions between the aligned nodes,
- $v^0 : v_S^0 \leftrightarrow v_R^0$, where v_S^0 and v_R^0 are the initial function entry nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively,
- $v^t : v_S^t \leftrightarrow v_R^t$, where v_S^t and v_R^t are the final function exit nodes of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively
- $\Omega : \{u \leftrightarrow v \mid \forall u \in V, \exists v \in V\}$, for each function/loop entry node, maintains a mapping to the corresponding exit node,
- Ψ is a mapping from edge e to ψ_e for all edges e, where $\psi_e = \psi_s \cup \psi_r$, and ψ_s, ψ_r are the set of guarded actions at the aligned edges e_s and e_r of the automata $\mathcal{A}_S$ and $\mathcal{A}_R$ respectively, and
- $Pred : Var_S \leftrightarrow Var_R$, denoting variable alignment, is a bijective mapping between variables of $\mathcal{A}_S$ and $\mathcal{A}_R$.

3.4.2 Inferring Variable Alignment Predicates

To infer alignment predicates of $\mathcal{A}_F$, we use a syntactic approach based on variable-usage patterns similar to that of SarfGen [99]. Our approach for computing a mapping between two sets of variables proceeds as follows.

For each edge e_i in $\mathcal{A}_F$ we collect the usage set for each variable x/x' in the reference/student program, namely the sets $usage(x, e_i)$ and $usage(x', e_i)$. If the student automaton has fewer variables than reference automaton ($|Var_S| < |Var_R|$),

then fresh variables are defined in Var_S. The goal is to find a variable alignment, a bijective mapping between Var_R and Var_S, which minimizes the average distance between $usage(x, e_i)$ and $usage(x', e_i)$ for each $i \in [1, n]$, where n is the number of edges in $\mathcal{A}_F$. This is done by constructing a distance matrix $\mathcal{M}_{e_i}$ for each edge e_i of size $|Var_R| \times |Var_S|$, where

$$\mathcal{M}_{e_i}(x, x') = \Delta\ (usage(x, e_i), usage(x', e_i))$$

Using the matrices $\mathcal{M}_{e_1}, \ldots, \mathcal{M}_{e_n}$, we construct a global distance matrix $\mathcal{M}_g$ for the entire set of edges in $\mathcal{A}_F$, where

$$\mathcal{M}_g(x, x') = \sum_{i=1}^{n} \frac{\mathcal{M}_{e_i}(x, x')}{n}$$

We then choose to align the variable x in R to the variable x' in S, denoted as $x \leftrightarrow x'$, if the pair (x, x') has the minimum average distance among all possible variable y aligned with x', that is among all variable alignment pairs (y, x').

$$\mathcal{M}_g(x, x^{'}) \leq \min(\forall_{y \in Var^R \setminus x'}(\mathcal{M}_g(x, y)))$$

3.5 Verification and Repair Algorithm

Once the aligned automaton $\mathcal{A}_F$ is constructed, we can initiate the repair process of the incorrect student program. Note that a repaired version of the incorrect student program produced by our algorithm is guaranteed to be semantically equivalent to the given structurally matched reference program. Our algorithm traverses the edges of the automaton $\mathcal{A}_F$ to perform edge verification which basically checks the semantic equivalence between an edge of the student automaton and its corresponding edge of the reference automaton.[3] In case the edge verification fails, we perform edge repair after which edge verification succeeds. While the existing approaches [41, 99] also similarly perform repair for aligned statements/expressions, the correctness of repair is not guaranteed unlike in our algorithm.

We combine the edge verification and repair into a single step by extending the well-known SyGuS (syntax-guided synthesis) approach [8] which can be defined as follows:

[3]Our implementation performs a breadth-first search, while our algorithm is not restricted to a particular search strategy.

Definition 1 (SyGuS). *SyGuS consists of $\langle \varphi, T, S \rangle$ where φ represents a correctness specification expressed assuming background theory T and S represents the space of possible implementations (S is typically defined through a grammar). The goal of SyGuS is to find out an implementation that satisfies φ.*

While in principle SyGuS can be directly used to perform repair, we have an additional non-functional requirement not considered in SyGuS—that is, we want to preserve the student program as much as possible for pedagogical purposes. To accommodate this additional requirement, we introduce our approach, SyGuR (syntax-guided repair), formulated as follows:

Definition 2 (SyGuR). *Syntax-guided Repair or SyGuR consists of $\langle \varphi, T, S, impl_o \rangle$ where the first three components are identical with those of SyGuS, and $impl_o \in S$ represents the original implementation that should be repaired. The goal of SyGuR is to find out a repaired implementation $impl_r \in S$ that satisfies φ. In addition, differences between $impl_o$ and $impl_r$ should be minimal under a certain minimality criterion.*

We realize SyGuR in the context of automated feedback generation for student programs. In this section, we present the two algorithmic steps we perform to conduct SyGuR: edge verification and edge repair.

3.5.1 Edge Verification

In this section, we describe how we detect faulty expressions in the given incorrect student program. Recall that the edges of the automaton $\mathcal{A}_F$ are constructed by aligning the edges of the student automaton $\mathcal{A}_S$ with the edges of the reference automaton $\mathcal{A}_R$. Recall also that the edges of $\mathcal{A}_S$ can be faulty while the edges of $\mathcal{A}_R$ are considered always as non-faulty.

Each edge $e : u \xrightarrow{\psi_s;\psi_r} v$ of $\mathcal{A}_F$ between nodes u and v asserts the following property:

$$\{\phi_u\}\psi_s;\psi_r\{\phi_v\} \tag{3.1}$$

where ϕ_u and ϕ_v are the variable alignment predicates at the source node u and target node v of the edge e respectively, and ψ_r and ψ_s represent a list of guarded actions of the reference implementation and student implementation, respectively,

expressed in a Single Static Assignment (SSA) form. For example, an original guarded action, $if(x > 1)x+$, is converted into its SSA form, $((x_1 > 1) \implies x_2 = x_1 + 1) \wedge (\neg(x_1 > 1) \implies x_2 = x_1)$. Note that ψ_s and ψ_r do not interfere with each other, since the variables used in ψ_s and ψ_r are disjoint from each other. Also note that ψ_r and ψ_s do not contain a loop (that is, a single edge does not form a loop), and thus an infinite loop does not occur in the edge.

Edge verification succeeds if and only if property (3.1) holds. In SyGuR, property (3.1) expresses a correctness specification φ_e for edge e. To check property (3.1), we use an SMT solver by checking the satisfiability of the following formula:

$$\varphi_e = \phi_u \wedge \psi_s \wedge \psi_r \wedge \neg\phi_v \tag{3.2}$$

The satisfiability of φ_e indicates verification failure, or showing non-equivalence of two implementations along edge e. Conversely, the unsatisfiability of φ_e indicates verification success. Note that there always exists a model m that satisfies $\phi_u \wedge \psi_r \wedge \psi_s$ (this is because ϕ_u is not false, and the SSA forms of ψ_r and ψ_s are defined over disjoint variables), and verification succeeds only when for all such m, $\neg\phi_v$ does not hold. Intuitively, verification succeeds if and only if it is impossible for the post-condition ϕ_v to be false after executing ψ_r and ψ_s under the pre-condition ϕ_u.

As for background theories in the SMT solver, we use: LIA (linear integer arithmetic) for integer expressions, the theory of strings for modeling input/output stream, theory of uninterpreted functions to deal with user-defined function calls such as *check_prime*, and LRA (linear real arithmetic) to approximate floating-point expressions.

3.5.2 Edge repair

Once φ_e is found to be satisfiable for an edge e (which indicates that the edge verification fails), our goal is to repair edge e by modifying the student implementation encoded in ψ_s. Algorithm 1 shows our edge repair algorithm based on the CEGIS (counter-example-guided inductive synthesis) strategy [89]. In step 1, edge verification is attempted, and verification failure results in a counter-example ϕ_{ce} that witnesses verification failure. In the remaining part of the algorithm, we modify ψ_s to ψ'_s in a way that $\{\phi_{ce}\}\psi'_s;\psi_r\{\phi_v\}$ holds. If $\{\phi_u\}\psi'_s;\psi_r\{\phi_v\}$ also happens to hold, edge repair is deemed as completed. Otherwise, an SMT solver

generates a new counter-example ϕ'_{ce}, and our algorithm searches for ψ''_s satisfying both $\{\phi_{ce}\}\psi''_s;\psi_r\{\phi_v\}$ and $\{\phi'_{ce}\}\psi''_s;\psi_r\{\phi_v\}$. This process is repeated until either edge repair is successfully done or it fails. Edge repair can fail either because the search space is exhausted or timeout occurs.

Let us first consider the case where ψ_s and ψ_r have the same number of guarded actions and all guarded actions have the same number of assignments. To ensure this requirement is met, we call function *Extend* (see line 20 of Algorithm 1) which will be described later. Under the current assumption that ψ_s and ψ_r have the same number of guarded actions, $Extend(\psi_s, \psi_r)$ returns ψ_s, and thus, its return value ψ_s^+ equals ψ_s.

To repair guarded actions ψ_s^+, we replace each of the conditional expressions and the update expressions (RHS expressions) with a unique placeholder variable h. This makes an effect of making holes in ψ_s^+, and filling in a hole for repair amounts to equating h with a repair expression. Function *RepairSketch* of the algorithm performs this task of making holes in ψ_s^+ and returns ψ_f defined as $\psi_s^+[e^{(i)} \mapsto h^{(i)}]$. In this definition, notation $\mapsto$ denotes a substitution operator defined over all expressions $e^{(i)}$ appearing in ψ_s^+ and their corresponding placeholder variables $h^{(i)}$. In the following, we use "hole" to refer to a placeholder variable.

In SyGuR (see Definition 2), the expression space of the holes is defined by implementation space S. Previous state-of-the-art works [41, 99] use the expressions of the reference program for repair (generated repairs are not verified in these works unlike in our approach), and we similarly define the implementation space of each hole as follows:

Definition 3 (Implementation space of a hole). *Let C_s (C_r) and U_s (U_r) be respectively the set of conditional and update expressions of ψ_s^+ (ψ_r). Recall that ψ_s^+ (ψ_r) represents guarded actions of the student (reference) program. When a conditional expression e_c is replaced by a hole h_c, the implementation space of h_c is defined as $C_s \mid C'_r \mid true \mid false$, where C'_r represents the set of conditional expressions appearing in ψ_r with all variables of C_r replaced with their aligned variables of the student program (see Section 3.4.2 for variable alignment). Similarly, given an assignment $x = h_u$ where h_u represents a hole for an update expression e_u, the implementation space of h_u is defined as $U_s \mid U'_r \mid x$, where U'_r represents*

the set of update expressions of ψ_r with all variables of U_r replaced with their aligned variables of the student program. The inclusion of an lhs variable x in the implementation space is to allow assignment deletion—replacing $x = e_u$ with $x = x$ simulates assignment deletion.

The repair synthesis process for some faulty expression on the edge e_s relies on four factors: the discovered counter-examples, the set of suspicious expressions in ψ_s^+, the set of reference expressions in ψ_r, and the inferred alignment predicates. These factors collectively determine the set of expressions on the edge e_r that can be exploited to repair the buggy expressions on e_s.

Recall that given a list of counter-examples CE, we search for a repair ψ_s' that satisfies $\forall \phi_{ce} \in CE : \{\phi_{ce}\}\psi_s'; \psi_r\{\phi_v\}$. When searching for a repair, we preserve the expressions of the student program as much as possible for pedagogical reasons. We achieve this by conducting a search for a repair using a pMaxSMT (Partial MaxSMT) solver. Note that an input to a pMaxSMT solver consists of (1) hard constraints which must be satisfied and (2) soft constraints all of which may not be satisfied. Whenever a soft constraint C is not satisfied, cost is increased by the weight associated with C, and a pMaxSMT solver searches for a model that minimizes the overall cost. We pass the following formula to a pMaxSMT solver where hard constraints are underlined.

$$\forall \phi_{ce} \in CE : (\underline{\phi_{ce}} \wedge \underline{\psi_r} \wedge \underline{\psi_f} \wedge \underline{\phi_v} \wedge \bigwedge_{(h^{(i)}, e^{(i)}, S[\![h^{(i)}]\!]) \in holes(\psi_f)} (h^{(i)} = e^{(i)} \wedge h^{(i)} \in S[\![h^{(i)}]\!] \backslash \{e^{(i)}\})) \tag{3.3}$$

where function $holes(\psi_f)$ returns a set of $(h^{(i)}, e^{(i)}, S[\![h^{(i)}]\!])$ in which $h^{(i)}$ represents the placeholder variable appearing in ψ_f (recall that ψ_f is prepared by making holes in ψ_s), $e^{(i)}$ denotes the original expression of $h^{(i)}$ extracted from student program, and $S[\![h^{(i)}]\!]$ represents the implementation space of $h^{(i)}$. Our soft constraints encode the property that each of the original expressions can be either preserved or replaced with an alternative expression in the implementation space. To preserve as many original expressions as possible, we assign a higher weight to $h^{(i)} = e^{(i)}$ than $h^{(i)} \in S[\![h^{(i)}]\!] \backslash \{e^{(i)}\}$.

The Extend function. Previously, we consider only the cases where ψ_s and ψ_r have the same number of guarded actions and all guarded actions have the

same number of assignments. To ensure this requirement, we invoke the *Extend* function which performs the following. First, if ψ_s has a smaller number of guarded actions than ψ_r, then ψ_s^+ (the return value of *Extend*) should contain additional guarded actions, each of which uses the following template: $[False] \implies x = x$, where x is constrained to be the variables of the student program. Notice that these additional guards are initially deactivated to preserve the original semantics of the student program, but they can be activated whenever necessary during repair, since $False$ is replaced with a hole by *RepairSktech*. After this step, *Extend* finds the guarded action of ψ_r that has the maximum number of assignments. Given this maximum number M, we check whether all guarded actions of ψ_s (including additional guarded actions with the $False$ guard) also have M assignments. Any guarded action that has a smaller number of assignments than M is appended with additional assignments, $x = x$ where x is constrained to be the variables of the student program. This process makes sure that for each guarded action, the student program can have as many assignments as the reference program.

3.5.3 Properties preserved by Verifix

Once all the edges of the aligned automaton $\mathcal{A}_F$ are repaired and verified, it is straightforward to produce a repaired student automaton $\mathcal{A}'_S$ by copying repaired expressions from the automaton $\mathcal{A}_F$ to the automaton $\mathcal{A}_S$. In this section, we discuss several interesting properties of our repair algorithm, namely soundness, completeness, and minimality of generated repairs.

Theorem 1 (Soundness). *For all program inputs, $\mathcal{A}'_S$ and $\mathcal{A}_R$ return the same program output.*

Proof. Recall that for each repaired/verified edge $u \xrightarrow{\psi_s;\psi_r} v$ of a repaired automaton $\mathcal{A}'_F$, $\{\phi_u\}\psi_s;\psi_r\{\phi_v\}$ holds. By structural induction on the edges of $\mathcal{A}_F$, the post-condition of $\mathcal{A}'_F$'s final node holds true, and hence $out = out'$ holds for the outputs aligned between $\mathcal{A}_S$ and $\mathcal{A}_R$. Note that for introductory programming assignments, output is clearly known (such as the return value of the program), and we enforce the post-condition of $\mathcal{A}'_F$'s final node to contain $out = out'$. □

Our edge repair algorithm (Algorithm 1) always returns a repaired edge as long

as the underlying MaxSMT/pMaxSMT solver used in the algorithm is complete (that is, UNKNOWN is not returned). This can be stated as follows, using the concept of relative completeness [25]:

Theorem 2 (Relative completeness of edge repair). *The completeness of Algorithm 1 is relative to the completeness of the MaxSMT/pMaxSMT solver.*

Proof. Whenever edge verification fails, Algorithm 1 performs repair in step 4 of the algorithm. In case a repair exists in the repair space, Algorithm 1 reaches line 36, and a pMaxSMT solver is fed with formula (3.3) to find out a repair. Thus, if the MaxSMT/pMaxSMT solver is complete, a repair is always generated. □

Meanwhile, the overall repair algorithm of Verifix is not complete. If $\mathcal{A}_F$ is failed to be constructed, the repair process cannot be started. Theorem 3 identifies the conditions under which Verifix succeeds to generate a repair. In Theorem 3, we use the following definition of alignment consistency:

Definition 4 (Alignment Consistency). *For each edge e of $\mathcal{A}_F$ $\{\phi_u\}\psi_s;\psi_r\{\phi_v\}$, modify ψ_s into the ψ'_s as follows: $\psi'_s \equiv \psi_r[x_r^{(i)} \mapsto x_s^{(i)}]$ where $x_r^{(i)}$ denotes all reference-program variables appearing in ψ_r and $x_s^{(i)}$ denotes student-program variables aligned with $x_r^{(i)}$. Repeat this for all edges of $\mathcal{A}_F$. Then, we say that $\mathcal{A}_F$ is alignment consistent when $\{\phi_u\}\psi'_s;\psi_r\{\phi_v\}$ for all modified edges.*

$\mathcal{A}_F$ is alignment consistent only when the variable alignment predicates are such that a given student program can be verifiably repaired by edge-to-edge copy of the reference program (patch minimality is not considered).

Theorem 3 (Relative completeness). *Our repair algorithm succeeds to generate a repair, under the following assumptions:*

1. *$\mathcal{A}_F$ is constructed,*
2. *$\mathcal{A}_F$ is alignment-consistent, and*
3. *The MaxSMT/pMaxSMT solver used for repair/verification is complete.*

Proof. Assume the three assumptions are met. Since Verifix traverses all edges of $\mathcal{A}_F$ one by one without backtracking, it suffices to show that each edge is repaired

by Algorithm 1 which at a high level consists of the following two parts: verification (step 1 of the algorithm) and repair (step 2, 3, and 4).

First, consider the verification part. Verification is performed via a MaxSMT solver which returns either (a) UNSAT (line 11) or (b) SAT (line 15) for φ^i_{edge} (see line 10). Note that the UNKNOWN case is excluded by the third assumption. In case (a), edge verification is done. In case (b), the algorithm moves to the repair part which we now consider.

In the repair part, a pMaxSMT solver is invoked at line 31 and 36 of Algorithm 1 and returns either (i) UNSAT or (ii) SAT. The UNKNOWN case is excluded by the third assumption. Case (i) happens only when the second assumption is violated (that is, a repair is not in the implementation space), and we exclude this case from consideration. In case (ii), repair candidates are obtained (line 35), and verification is re-attempted to see if one of the obtained candidates ψ'_s satisfies $\{\phi_u\}\psi'_s;\psi_r\{\phi_v\}$. The repetition between repair and verification is guaranteed to terminate, since the implementation space is finite. This concludes the proof. □

Lastly, we consider the minimality of repair. In Verifix, use of MaxSMT guarantees the minimality of edge repair.

Theorem 4 (Minimality of edge repair). *Suppose that our algorithm repairs edge $e : u \to v$ of $\mathcal{A}_F$ by changing $F \subseteq C_s \cup U_s$ (C_s and U_s are defined in Definition 3). There does not exist F' s.t. $|F'| < |F|$ and the pre-/post-conditions of e are satisfied by replacing the expressions of F' with the expressions in $C_r \cup U_r$.*

Proof. Recall that we pass formula (3.3) to a pMaxSMT solver. In the formula, the number placeholder variables $h^{(i)}$ defines the maximum size of edge repair, and a minimal edge repair is obtained when the minimum number of placeholder variables $h^{(i)}$ are equated with expressions different from their original expression $e^{(i)}$, which happens when expression $h^{(i)} = e^{(i)}$ in formula (3.3) is ignored by the pMaxSMT solver. Since a pMaxSMT solver ignores the minimum number of $h^{(i)} = e^{(i)}$, the stated theorem holds. □

Theorem 4 does not necessarily guarantee the global minimality of a generated repair. In the following theorem, we identify the conditions that should be additionally satisfied to guarantee global minimality.

Theorem 5 (Global minimality). *A repaired program generated by our algorithm is minimal if the following conditions hold:*

1. *Node alignment made in $\mathcal{A}_F$ is optimal in the sense that there is no alternative node alignment (other than the one generated by Verifix) that can lead to a smaller repair.*

2. *The variable alignment predicates of $\mathcal{A}_F$ are optimal in the sense that there is no alternative variable alignment that can lead to a smaller repair.*

Proof. Once node alignment and invariants of $\mathcal{A}_F$ are fixed, repairing a student program amounts to repairing each edge of $\mathcal{A}_F$ for which edge verification fails. Since each edge is repaired minimally (Theorem 4), the stated theorem holds. □

Verifix currently does not guarantee the global minimality of repair. Node alignment and variable alignment made by Verifix are not necessarily optimal. Instead of considering all possible alignments, we use a heuristics-based approach for the sake of efficiency. Nonetheless, our experimental results show that Verifix tends to find smaller repairs than Clara. Note that the existing approaches designed to generate minimal repairs [99, 41] also do not consider node/edge alignment in the calculation of the minimality of a repair. Instead, a minimal repair is searched for only after node/edge alignment is made. In fact, unlike those existing approaches that do not consider alignment at all, we consider edge alignment by enumerating all possible edge alignments between aligned nodes.

3.6 Experimental Setup

3.6.1 Research Questions

We address the following research questions in this chapter.

1. RQ1: How does Verifix perform in terms of the repair success rate, as compared to state-of-the-art approaches? While Verifix generates verifiably correct repair, is the repairability comparable to the existing approaches?

2. RQ2: How does Verifix perform in terms of running time? Given that Verifix uses heavy-weight SMT techniques to conduct verification, slowdown in running time as compared to non-verification approaches is expected. How severely is the time performance affected?

3. RQ3: What are the reasons for repair failure in Verifix? The answers to this question can be used to identify where to improve in the future work.

4. RQ4: Does Verifix generate small sized repair? In a pedagogical setting, small repairs are usually desired. While Verifix generates a minimal edge repair, it does not guarantee to generate a globally minimal repair. What is the practical consequence of this greedy approach?

5. RQ5: What is the effect of test-suite quality on repair when a test-based approach is used? We ask this question to compare the existing test-based approaches with Verifix which does not require a test.

6. RQ6: How is the repair success rate of Verifix affected by the number of reference solutions? We ask this question to assess the performance of Verifix when multiple reference implementations are available.

3.6.2 Dataset

Evaluation of a programming assignment feedback tool requires a dataset of incorrect student assignments. For our dataset, we chose a publicly released dataset curated by ITSP [4] [107] for evaluating feasibility of APR techniques on introductory programming assignments. This benchmark consists of incorrect programming assignment submissions by 400+ first year undergraduate students crediting a *CS-1: Introduction to C Programming* course at a large public university. Other datasets used in previous work are either not publicly available [41, 99, 53] or use different programming languages than C [47].

We take students' incorrect attempts from four basic weekly programming labs in ITSP benchmark, where each lab consists of several programming assignments that cover different programming topics. For example, the lab in week 3 (Lab-3 in

[4] `https://github.com/jyi/ITSP#dataset-student-programs`

Table 3.3) consists of four programming assignments which teach students about floating-point expressions, printf, and scanf. Table 3.3 lists the four programming labs partitioned by different programming topics. Students had, on average, a time limit of one hour duration for completing each individual assignment. Our implementation currently does not support all programming language constructs such as pointers, multi-dimensional arrays, and struct, which are necessary to support the remaining labs in the ITSP benchmark. Note that support for more programming language constructs is orthogonal to our verified-repair generation algorithm. As more programming language constructs are supported, our repair algorithm can be used without modification to repair more diverse programs, these are left as future work.

We use 341 compilable incorrect students' submissions from 28 various unique programming assignments as our subject. In addition to the incorrect student submissions, each programming assignment in the ITSP benchmark contains a single reference implementation and a set of test cases designed by the course instructor. Both Verifix and baseline Clara [41] have access to the reference implementation and test cases to repair the incorrect student programs.

Baseline comparison We compare our tool Verifix's performance against the publicly released state-of-art repair tool Clara [5] [41] on the common dataset of 341 incorrect student assignments. A timeout of 5 minutes per incorrect student program was set for both Verifix and Clara to generate repair. We do not directly compare our results against CoderAssist [53] tool since it does not work with our dataset (CoderAssist targets dynamic programming assignments), while Refactory [47] implementation targets Python programming assignments. About SarfGen, we could not obtain access to the tool from its authors due to a copyright issue (SarfGen is commercialized). We instead address these comparisons in our related work Section 7. Our tool Verifix [6] is publicly released to aid further research.

Our experiments were carried out on a machine with Intel® Xeon® E5-2660 v4 @ 2.00 GHz processor and 64 GB of RAM.

[5] `https://github.com/iradicek/clara`
[6] `https://github.com/zhiyufan/Verifix`

Lab-ID	Topics	# Assign-ments	# Prog-rams	Repair (%)		Avg. Time (sec)	
				Clara	Verifix	Clara	Verifix
Lab-3	Floating point, printf, scanf	4	63	54.0%	92.1%	2.0	39.7
Lab-4	Conditionals, Simple Loops	8	117	71.8%	74.4%	32.9	34.2
Lab-5	Nested Loops, Procedures	8	82	22.0%	45.1%	10.2	12.5
Lab-6	Integer Arrays	8	79	12.7%	21.5%	14.2	8.1
Overall	-	28	341	42.8%	58.4%	21.3	29.5

Table 3.3: Lab-wise repair success rate (shown in the Repair column) of our tool Verifix and Clara [41]. Time column represents the average runtime in seconds for all successfully repaired programs. The number of assignments in each lab is shown in the *#Assignments* column, and the number of incorrect student submissions in each lab is shown in the *#Programs* column.

3.6.3 Implementation

Verifix supports repairing compilable incorrect C programs, given a reference C program and optional test cases. Verifix implementation is composed of three components: (1) Setup, (2) Verification, and (3) Repair generation.

For the setup phase, we build on top of Clara [5] [41] parser to convert incorrect and reference C programs into a Control-Flow Graph (CFG) representation. We then convert the obtained CFGs into its dual Control-Flow Automata (CFA), and align the reference CFA with incorrect CFA.

In the verification phase, the reference and student program labels on each aligned edge are converted into a Single Static Assignment (SSA) format using our custom Verification Condition Generator (VCGen) implementation. We make use of Z3 [73] SMT solver to verify if the aligned edges are equivalent.

In the repair phase, we encode each repair candidate using Boolean selectors. Z3 pMaxSMT solver is used to select the repair with minimal cost. The final repaired CFA/CFG internal representation is converted back into a program using a custom concretization module (reverse VCGen). After which, we make use of Zhang-Shasha [7] tree-edit distance algorithm [112] to compute the patch size between incorrect student program and the repaired student program.

[7] `https://github.com/timtadh/zhang-shasha`

3.7 Evaluation

3.7.1 RQ1: Repair success rate

Table 3.3 compares the repair success rate of our tool Verifix against the state-of-art tool Clara [41] on the common dataset of student submissions. Given a single reference implementation per assignment, Verifix achieves an overall repair success rate of 58.4% on the 348 incorrect programs across 28 unique assignments. In comparison, the baseline tool Clara achieves a lower overall repair success rate of 42.8% on the same assignments, a difference of more than 15%. Note that *the repairs generated by Verifix are verifiably equivalent to the reference implementation*, in addition to passing all the instructor provided test cases. That is, Verifix generates a verifiably correct feedback for 58.4% of student submissions in diverse assignments, which is not possible using existing test-based approaches.

The improvement in repair success rate of Verifix over Clara is partly due to the more flexible structural alignment of Verifix than that of Clara. Recall that Verifix uses a more relaxed structural alignment, as compared to the stricter structural alignment used by existing state-of-the-art approaches including Clara, as described in Section 3.4.1. Verifix requires the reference and incorrect Control-Flow Automata (CFA) to have the same number of program states or nodes, denoting functions and loops. While Clara additionally requires the reference and incorrect CFA to have the same number of edges, denoting return/break/continue transitions. In Section 3.7.3, we investigate the common reasons for repair failure.

3.7.2 RQ2: Running time

The *time* column of Table 3.3 shows the average running time of Verifix and Clara, in seconds. Verifix on average takes 29.5s to successfully repair an incorrect program, as compared to 21.3s on average by Clara. The running time of Verifix is particularly high in Lab-3 (39.7s) and Lab-4 (34.2s), whereas in Lab-6, Verifix runs significantly faster than Clara (8.1s vs 14.2s). The high running time of Verifix in Lab-3 and Lab-4 seems due to the fact that Lab-3 and Lab-4 programming assignments involve non-linear arithmetic expressions. For example, one of the Lab-4 assignments is on *computing the distance between two co-ordinate points*, which

Lab-ID	# Prog-	Repair (%)		Struct. Mismatch (%)		Timeout (%)		Unsupported (%)		SMT issues (%)	
	rams	Clara	Verifix	Clara	Verifix	Clara	Verifix	Clara	Verifix	Clara	Verifix
Lab-3	63	54.0%	92.1%	0.0%	0.0%	42.9%	0%	3.2%	3.1%	0%	4.8%
Lab-4	117	71.8%	74.4%	7.7%	7.7%	19.6%	10.3%	0.9%	0.9%	0%	6.8%
Lab-5	82	22.0%	45.1%	75.6%	35.4%	1.2%	11.0%	1.2%	1.2%	0%	7.3%
Lab-6	79	12.7%	21.5%	83.5%	69.6%	2.5%	0%	1.3%	1.3%	0%	7.6%
Overall	341	42.8%	58.4%	40.2%	27.2%	15.5%	6.2%	1.5%	1.5%	0%	6.7%

Table 3.4: The distribution of the four reasons for repair failure, i.e., structural mismatch (4th column), timeout (5th column), unsupported language constructs (6th column), and SMT issues (7th column). The first three columns are copied from Table 3.3.

involves square-root computation. Note that SMT solvers generally run slow when non-linear arithmetic expressions are used in the input formula. There has been an effort to handle non-linear arithmetic more efficiently [32], and Verifix can be benefited from the improvement of the SMT techniques.

We also note that while Clara runs faster than Verifix across the labs except for in Lab-6, its repair success rate is always lower than that of Verifix across all labs. For example, in Lab-3, Clara's average running time is only 2.0s, but its repair success rate is only 54.0%, which is 38.1% lower than that of Verifix (92.1%). Overall, while Verifix, which uses heavy-weight SMT techniques, tends to require more running time than Clara, the overall results are nuanced by the other facts such as repair success rate and correctness guarantee.

3.7.3 RQ3: Reasons for repair failure

Table 3.4 shows the distribution of the repair failure reasons for Verifix and Clara. *Structural Mismatch* (shown in the 4th column) is the primary reason for repair failure of Verifix and Clara, accounting for 27.2% and 40.2% of all the 341 incorrect student programs, respectively. Recall that a single reference solution is used for each assignment in the labs. For simpler programs such as those in Lab-3 and Lab-4, both tools achieve low structural mismatch rate. That is, almost all the incorrect student programs can be structurally aligned with the reference program. As the complexity of the programs increases (in our dataset, as the lab ID increases, the students submissions tend to be more complex), the structural mismatch rate tends to increase in both tools. However, the rate increases more gently in Verifix than in Clara. For example, in Lab-5, while the structural mismatch rate of Clara drastically increases to 75.6%, Verifix maintains a much lower mismatch rate of

35.4%. This difference in structural match rate results in a overall higher repair success rate in Verifix as compared to Clara. For example, 45.1% of Verifix vs 22.0% of Clara for Lab-5. The high structural mismatch rates in Lab-6 are related to the following: many incorrect students' programs use function calls, but the reference programs often do not have functions with matching function signatures.

The second biggest failure reason is *Timeout* (5 minutes), accounting for 6.2% and 15.5% of the dataset for Verifix and Clara, respectively. In Verifix, most of the running time is spent on SMT and pMaxSMT solving by Z3 solver during verification and repair stage, respectively. In 1.5% of student programs, repair failure occurs since our current implementation does not support all programming language constructs used in our datasets. For example, both Verifix and Clara currently do not support the *GOTO* statement. Lastly, in 6.7% of the incorrect programs, Verifix fails to generate a repair due to the incompleteness of SMT solving. Common cases of this kind are when the SMT solver returns UNKNOWN result, instead of SAT or UNSAT, during the verification or repair phase.

3.7.4 RQ4: Minimal repair

To investigate this research question, we compare the sizes of repairs generated from Verifix and our baseline state-of-art tool Clara [41]. Since the size of the student programs vary significantly, we normalize patch size with the size of original incorrect program to obtain Relative Patch Size (RPS), given by: $RPS = Dist(AST_s, AST_f)/Size(AST_s)$. Where, AST_s and AST_f represents the Abstract Syntax Tree (AST) of incorrect student program and fixed/repaired program generated by tool, the *Dist* function computes a *tree-edit-distance* between these ASTs, and the *Size* function computes the #nodes in the AST.

In our benchmark of 341 incorrect programs, Verifix can successfully repair 199 student programs, Clara can successfully repair 146 programs, while Verifix and Clara both can successfully repair 132 common programs. Out of these 132 commonly repaired programs, Verifix generates a patch with smaller RPS in 67 of the cases, Clara generates a patch with smaller RPS in 47 of the cases, and both tools generate a patch of the exact same relative patch size in 18 cases. Note that in the case of Clara, a smaller repair does not necessarily imply better quality repair

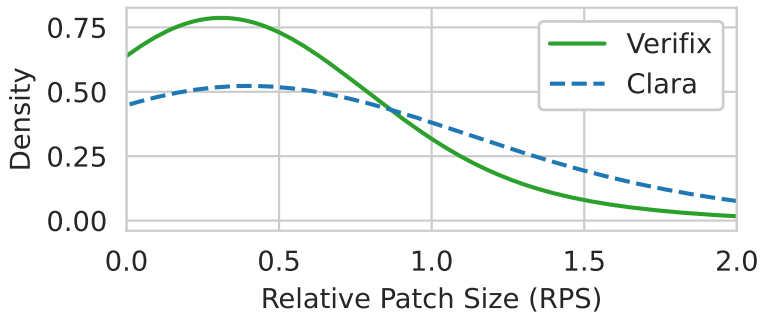


Figure 3.6: Kernel Density Estimate (KDE) plot of Relative Patch Size (RPS) by Verifix and Clara on 132 common successful repairs.

```
1  void main(){
2    int n1, n2, i;
3    scanf("%d %d", &n1, &n2);
4    if(n2 <= 2)          // Repair #1: Delete spurious print
5      printf("%d ", n2); //              Verifix ✓, Clara ✗
6    for(i=n1; i<=n2; i++){
7      if(check_prime(i)==0) // Repair #2: Delete ==0
8        printf("%d ", i);   //   Verifix ✓, Clara ✓
9    }
10 }
```

Figure 3.7: Example from a Lab-5 *Prime Number* assignment. The main function contains two errors, both of which are fixed by Verifix, while Clara's repair overfits given test-suite by ignoring first error.

since these repairs can overfit the test cases (see Section 3.7.5).

Figure 3.6 plots the Kernel Density Estimate (KDE) of Relative Patch Size (RPS) for these 132 common programs that both Verifix and Clara can successfully repair, in order to visualize the RPS distribution for these large number of data points. KDE is an estimated Probability Density Function (PDF) of a random variable, often used as a continuous smooth curve replacement for a discrete histogram. From the Figure 3.6 plot we observe that the density of patch-sizes (y-axis) produced by Verifix is greater than that of Clara when $RPS < 0.8$ (x-axis). On the other hand, the density of patch-sizes generated by Clara is greater than that of Verifix when $RPS \geq 0.8$. That is, a large proportion of repairs generated by Verifix have a small relative patch-size, since the density concentration of repairs is towards lower RPS (x-axis). In comparison, a significantly larger proportion of Clara's repairs have $RPS \geq 0.8$, as compared to Verifix.

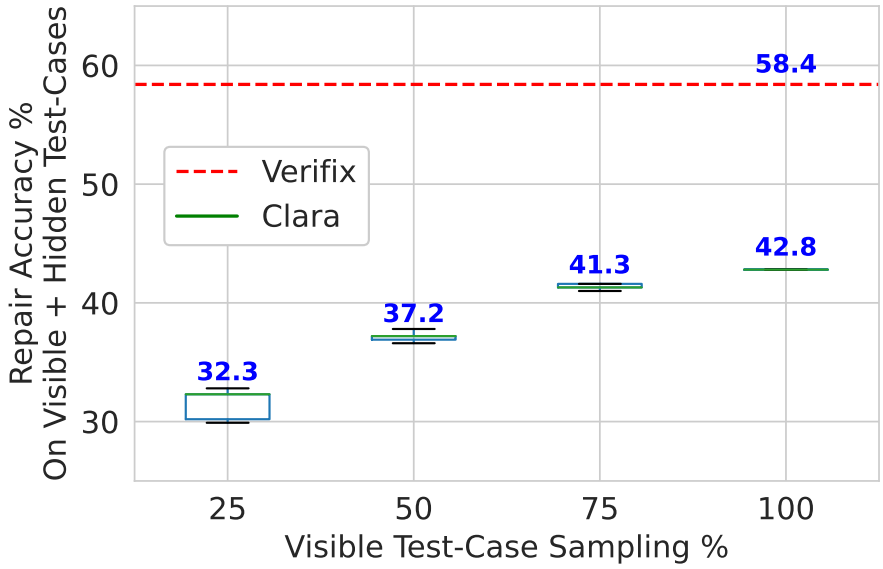


Figure 3.8: Repair accuracy of Clara and Verifix on various test case samplings.

3.7.5 RQ5: Overfitting

Majority of the programming assignment repair tools [107, 41, 99, 47] generate repairs that satisfy a given test suite (incomplete specification). Verifix is distinguished from these existing test based approaches in that it generates a verifiably correct repair. Figure 3.7 demonstrates an example from a Lab-5 *Prime Number* assignment, where Clara's [41] repair overfits the test cases. With the help of a reference implementation, Verifix is able to detect a new counter-example where the student program deviates from correct behavior, when input stream is "1 2" ($n1 = 1$, $n2 = 2$). Given this new unseen test case, the repair suggested by Clara results in an incorrect output "2 2 ", while the repair suggested by Verifix results in the correct behaviour producing output "2 ".

In order to measure the degree of overfitting repairs generated by each tool, we compare the impact of test case quality on repair accuracy. This is done by running Clara and Verifix on our common benchmark of 341 incorrect programs under four different settings, where a percentage of test cases were hidden from tool during repair generation. For each of the 28 unique assignments, with 6 instructor designed test cases on average, we randomly sampled $X\%$ as "visible" test cases. Once the repair was successfully generated by a tool on the limited visible test case sample, we re-evaluated the repaired program on all test cases, including hidden ones. We

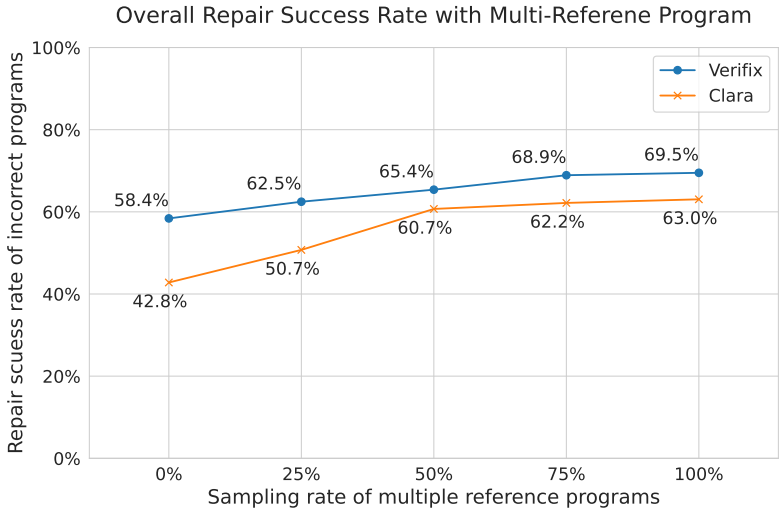
carried out this experiment under four different settings, with a random sampling rate of 25%, 50%, 75%, and 100% of the available test cases. This entire experiment was repeated 5 times, where we randomly sampled test cases each time, and we report on the distribution of repair accuracy achieved by each tool.

Figure 3.8 displays the result of our overfitting experiment, with the X-axis representing the visible test case sampling %, and Y-axis representing the repair accuracy % obtained by APR tool on the entire test-suite (visible and hidden test cases). Each box plot displays the distribution of repair accuracy per test case sampling, by showing the minimum, maximum, upper-quartile, lower-quartile and median values. The median value of each box-plot is shown as text above the box-plot.

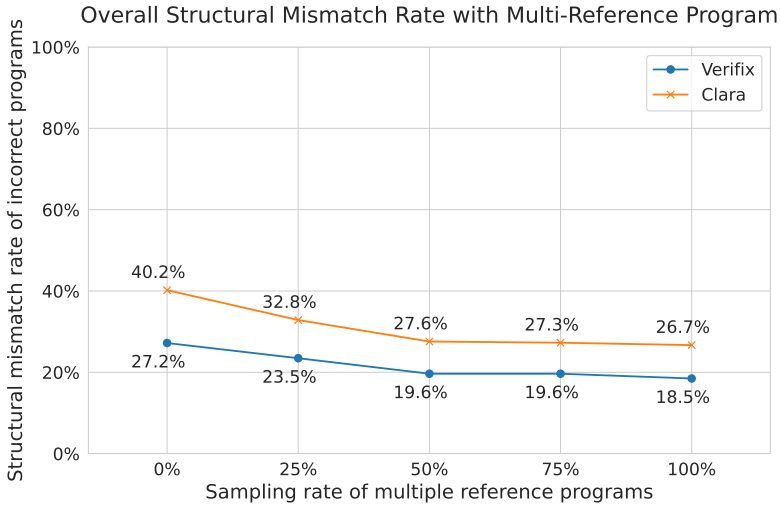
From Figure 3.8 we observe that Verifix's repair accuracy is constant. That is, Verifix's repair does not change based on the percentage of visible test cases provided, since it does not use the available test cases for repair generation or evaluation/verification. On the other hand, Clara's repair accuracy varies from a median value of 42.8% (when all test cases are made visible) to a median value of 32.3% (when only 25% of test cases are made available to Clara). In other words, Clara overfits on $42.8 - 32.3 = 10.5\%$ of our benchmark of 341 incorrect programs, when 25% of test cases are randomly chosen. Similarly, overfitting of $42.8 - 41.3 = 1.5\%$ is observed when visible test case sampling rate is 75%, or 5 visible test cases ($\lceil 75\% \times 6 \rceil = 5$) on average. In other words, when even a single test case on average is hidden from Clara, its generated repair can overfit the test cases.

Moreover, the choice of test case sampling has a large effect on Clara's repair accuracy, as evident from the variation in box-plot distribution. In the case of 25% visible test case sampling, Clara's repair accuracy ranges from a minimum value of 29.9 to maximum of 32.8; depending on which two test cases ($\lceil 25\% \times 6 \rceil = 2$) were made available.

Hence, APR tools such as Clara [41] which rely on availability of good quality test cases for their repair generation and evaluation can suffer from overfitting. Even when the instructor misses out on a single important test case coverage during assignment design. Thereby generating incomplete feedback to students struggling with their incorrect programs. Verifix on the other hand does not suffer from



(a) Overall repair success rates for all labs



(b) Overall structural mismatch rates for all labs

Figure 3.9: Repair success rates and structural mismatch rates across different sampling rates of multiple reference solutions. The X and Y axes represent the sampling rate of the reference solutions and the observed repair success rate, respectively.

overfitting limitation, due to its sole reliance on reference implementation for repair generation and evaluation/verification.

3.7.6 RQ6: Repair success rate with multiple reference implementations

In the previous sections, we conducted experiments with a single reference implementation for each assignment. Several previous works, including Clara [41]

and SarfGen [99], assume the prevalence of multiple reference solutions to help alleviate structural matching issues. In this section, we compare the repair success rate of both Verifix and our baseline tool Clara [41], on being provided access to multiple reference solutions. As additional reference implementations, we use 341 student submissions in the ITSP dataset [107] that pass all test-cases. While passing all tests does not guarantee the correctness of a program, previous works [41, 99] used similar approaches.

To evaluate the change in repair success rate on providing access to multiple reference implementations, we run Verifix and Clara with diverse sampling rates of 0%, 25%, 50%, 75%, and 100%; for each sampling rate of N%, we randomly sample N% of all available reference implementations, in addition to the instructor-provided reference program. For example, 0% sampling rate indicates only the instructor provided reference solution was used (single-reference program). While 100% indicates that all reference programs were made available for the repair tool, in addition to the instructor provided reference program. To prevent a student's incorrect program P being repaired by his/her own final submission P' that passes all test-cases, we exclude P' from the sampled set of multi-reference programs (if it exists) when P is being repaired. We run our baseline tool Clara [41] in its default mode for multi-reference programs; its clustering algorithm is first executed on the set of sampled reference implementations, followed by running its repair algorithm on each incorrect program using the obtained clusters.

The results of multi-reference experiments are shown in Figure 3.9. Figure 3.9(a) shows how repair success rate changes as more reference programs are used, while Figure 3.9(b) shows how structural mismatch rate changes. A student submission S is considered structurally mismatched with a sampled group of reference programs G when no program in G structurally matches S. From Figure 3.9(a) we observe that the repair success rate increases for both Verifix and Clara, as more reference implementations are made available for repair. From Figure 3.9(b) we note that this is primarily due to a reduction in structural alignment mismatch between the set of multiple reference implementations (with more diverse program structures) and the given incorrect program.[8] We note that similar observations have been

[8]Repair failures may also occur due to reasons other than structural mismatch, as discussed in Section 3.7.3.

made regarding the effect of multi reference programs on repair success rate in prior work [47].

From Figure 3.9(a) we observe that Verifix achieves a higher success rate over Clara across all sampling rates. The gap between the repair rate of both tools reduces as more reference programs are provided, indicating that Clara's repair success rate could eventually match that of Verifix's on being provided a large number of reference solutions. From Figure 3.9(b) we observe that Verifix maintains a lower structural mismatch rate over Clara across all sampling rates. When all reference solutions are used, structural mismatch rate of Verifix and Clara drops down to 18.5% and 26.7%, respectively. This result demonstrates the benefit of using Verifix's CFA (Control-Flow-Automata) based structural alignment algorithm over Clara's CFG (Control-Flow-Graph) based alignment algorithm, even in the case of multi-reference solutions.

3.8 User Study

In order to evaluate the usefulness of the repair generated by Verifix, we conducted a user study of tutors of introductory programming courses. Note that students have expressed positive feedback about using feedback generation systems such as Clara [41] and SarfGen [99]. Verifix uses the same copy mechanism for repair as these tools (i.e., parts of a reference implementation are copied) and can generate the same style of feedback. The main difference between Verifix and the existing tools lies in that Verifix generates verifiably correct repairs. We believe that tutors can better appreciate the quality of repairs than novice students, and our user study sheds helpful light on understanding its pedagogical value. A user study with students is left as future work.

3.8.1 User Study Questionnaire

In this user-study, we explored the practical value of Verifix in aiding tutors in the task of grading and providing feedback on incorrect student submissions. This was explored using the following questions:

1. Rate the quality of the generated repair (in terms of semantic correctness, size,

etc).

2. Rate the possibility that you would like to use the repair (either complete or partial) as feedback to the student.

3. Rate the possibility that you would like to use the repair indirectly: to help formulate your own custom feedback to student.

4. Rate the possibility that these repairs can help you in grading?

5. Will examples of student incorrect submissions and repairs like these help you in improving the grading policy?

6. If the repair is known to be verifiably (provably) correct, does it give you more confidence in using it?

3.8.2 User Study Setup

To answer the above questions, we circulated a Google-Form survey among the tutors of introductory programming courses at NUS (National University of Singapore) and UNIST (Ulsan National Institute of Science and Technology). After which, 14 tutors in total volunteered for this survey and completed their responses. For this survey, we randomly selected 10 incorrect student submissions from our benchmark of 341 programs, on which Verifix could successfully generate a repair. For each incorrect student program, the tutors were shown the assignment title, assignment description, a sample testcase, and the differences between a student-written buggy program and its repair generated by Verifix. All the volunteered tutors were shown the same 10 incorrect student submissions in the same order.

The tutors were asked three questions (questions 1–3 listed in Section 3.8.1) for each buggy student submission, followed by three questions (questions 4–6 listed in Section 3.8.1) as an overall summary at the end of the user study. The tutors were asked to provide their ratings on a numeric scale from 1 (very low) to 5 (very high) for each question.

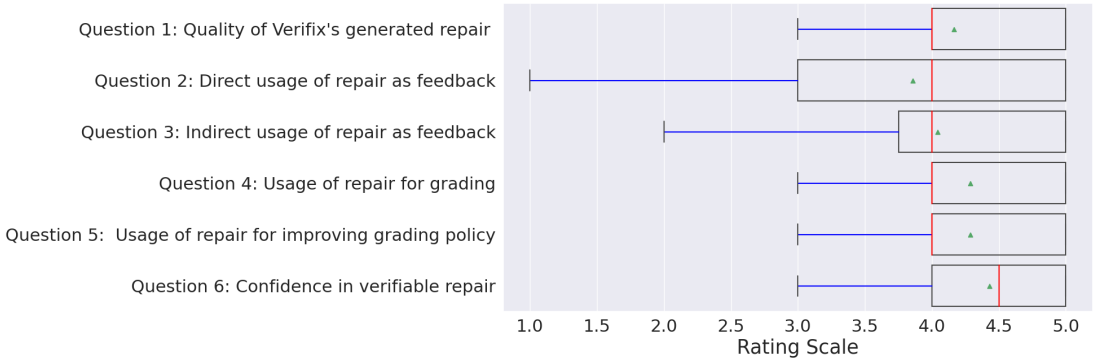


Figure 3.10: Boxplot of the responses–with the scales from 1 (very low) to 5 (very high)–collected from 14 tutors. Red line represents the median value and green triangle represents the mean value. The whiskers denote the minimum/maximum value, and the rectangle denotes the first/third quartile.

3.8.3 User Study Results

The overall result of the 14 tutor responses is summarized using boxplots in Figure 3.10. From Figure 3.10 we note that the tutors responded with an overall positive rating for all six of our questionnaire (Q1–Q6), with a mean/median value of >= 3.8 in all the cases. We observe that the tutors rate the quality of Verifix generated repairs (Q1) highly, with a mean/median rating of ~4.0. Our tool's verification capability improved the tutors' confidence in accepting our generated repairs, with a mean/median rating of ~4.4. The tutors, on average, found Verifix's repair useful for providing feedback to students, both directly (Q2) and indirectly (Q3), giving a mean/median rating between 3.5–4.0. While a larger variation is observed in the case of direct usage of repair as feedback (Q2), this discrepancy reduces for indirect usage of repair as feedback (Q3), where tutors can quickly design customized feedback using the generated repair. The tutors agreed on the utility of Verifix's repair in grading (Q4) and in improving grading policy (Q5), giving a mean/median rating between 4–4.5.

3.9 Threats to Validity

Our aligned automata setup phase consists of a syntactic procedure to obtain a unique edge and variable alignment between the reference and student automata. Producing an incorrect alignment does not affect our soundness or relative complete-

ness guarantees, but can increase the size of a generated patch. This however occurs rarely in practice, as demonstrated by our RQ4 (Section 3.7.4).

The arithmetic theory of SMT solvers is incomplete for non-linear expressions, which can affect our relative completeness. However, this issue affects 6.7% of our dataset of incorrect student programs in practice, as demonstrated by our results in Table 3.4.

Evaluating repair tools using multiple correct student submissions, instead of restricting to a single instructor reference solution, could help improve the repair success rate. We mitigate this risk by noting that such an evaluation has been undertaken earlier [99, 47], and would benefit both Verifix and our baseline tool Clara in terms of reduced structural mismatch rate. Furthermore, we cannot always assume the availability of a large number of reference solutions, in general.

3.10 Discussion

In this chapter, we have presented an approach and tool Verifix, for providing verified repair as feedback to students undertaking introductory programming assignments. The verified repair is generated via relational analysis of the student program and a reference program. Verifix is able to achieve better repair success rate than existing approaches on our common benchmark. The repairs produced by Verifix are of better quality than state-of-art techniques like Clara [41], since they are often smaller in size, while being verifiably equivalent to the instructor provided reference implementation.

We feel that technologies like Verifix have a place in intelligent tutoring systems of the future. Specifically, they may be used to give feedback to struggling students learning programming. Since Verifix generates verifiably correct repairs, it can used first for generating feedback. If Verifix is able to generate a feedback, it can used with confidence. For the cases where Verifix is unable to generate a feedback, other heuristic based student feedback generation approaches may then be used. We envision such a workflow for future intelligent tutoring systems for teaching programming.

Algorithm 1 Edge verification-repair

Input: Aligned *edge*
Output: Verified/Repaired *edge*

Let $\phi_u \equiv$ edge.sourceNode.invariants
Let $\phi_v \equiv$ edge.targetNode.invariants
Let $\psi_r \equiv$ edge.label.reference
Let $\psi_s \equiv$ edge.label.student
$CEs \leftarrow []$ // List of counter-examples
$candidates \leftarrow [\psi_s]$
repeat
    // Step 1: attempt for edge verification
    for each ψ_s^i in *candidates* **do**
        Let $\varphi_{edge}^i \equiv \neg(\phi_u \wedge \psi_r \wedge \psi_s^i \implies \phi_v)$
        if $\not\models \varphi_{edge}^i$ **then** // UNSAT
            $edge.label.student \leftarrow \psi_s^i$ // Update edge
            return ✓ // Verifiably correct
        else
            $\phi_{ce}^i \models \varphi_{edge}^i$ // SAT
            $CEs \leftarrow CEs \cdot \phi_{ce}^i$
        end if
    end for
    // Step 2: make holes in ψ_s
    Let $\psi_s^+ \equiv Extend(\psi_s, \psi_r)$
    Let $\psi_f \equiv RepairSketch(\psi_s^+)$
    // Step 3: define implementation space
    $\varphi_{hard} \leftarrow []$; $\varphi_{soft} \leftarrow []$
    for each ϕ_{ce}^i in CEs **do**
        $\varphi_{hard} \leftarrow \varphi_{hard} \cdot (\phi_{ce}^i \wedge \psi_r \wedge \psi_f \wedge \phi_v)$
    end for
    for each $hole, expr, weight$ in $RepairSpace(\psi_f, \psi_r, \psi_s)$ **do**
        $\varphi_{soft} \leftarrow \varphi_{soft} \cdot (hole = expr, weight)$
    end for
    // Step 4: search for a repair
    if $\not\models (\varphi_{hard}, \varphi_{soft})$ **then** // UNSAT or UNKNOWN
        return ✗ // Repair Failure
    else
        // Update *candidates* using a pMaxSMT solver
        // There can be multiple candidates
        $candidates \models (\varphi_{hard}, \varphi_{soft})$
    end if
until timeout

Chapter 4

Concept-based Automated Grading

4.1 Introduction

There has been a growing interest in computer science education in recent years. Several education initiatives (e.g., Coursera, EdX, and Udacity) provide online courses that are taken by thousands of students all around the world. These online courses are known as Massive Open Online Courses (MOOC), which include many computer science courses that use programming assignments for assessing students' learning outcomes. With the increasing number of student enrollments, the number of submitted programming assignments also grows extensively throughout the year. This motivates the need for an automated grading system that can save the time and effort spent in grading these assignments. In this chapter, we study the problem of automated grading of introductory programming assignments, which is common in first-year programming courses. There exist certain inherent difficulties in grading introductory programming assignments written by a novice programmer. Part of the difficulty comes from the fact that these programming attempts are significantly incorrect, often barely passing any tests [107]. Yet manual inspection of the code can reveal some degree of understanding of the problem by the student which should ideally be taken into account. Overall, the test-based automated grading may be too harsh for introductory programming assignments. In the K-12 computing education domain, promising results have been shown by using rubrics for grading assignments written in a visual programming environment to evaluate whether assignments produced by students demonstrate that they have learned certain algorithms and programming concepts [9]. Although grading based on rubrics provides a reliable way of assessing students' learning, the current rubric-based grading approach in

most universities still relies either on manual grading or semi-automated grading [5], which may be too labor intensive for the instructors and tutors.

Existing approaches in automated programming assignment grading [63, 99, 41, 97] have several limitations. These approaches either (1) generate a patch for the incorrect student's submission as feedback or (2) produce binary (Correct/Incorrect) results via test-based grading, (3) only compare syntactic differences between instructors' reference solution and student solution (CFG-based grading). Although feedback in the form of patches can be useful for experienced developers or graders, prior studies show that novice students may not know how to effectively utilize the generated patches as hints, causing the increase of problem-solving time when patches are given [107]. Meanwhile, despite the widespread adoption of test-based grading approaches for online judges, the binary results provided by the test-based grading approaches may be too coarse-grained and may underestimate students' effort. CFG-based grading approach cannot distinguish the syntactically different but semantic equivalent implementation, which gives inaccurate results if the student's solution is syntactically different from instructors' reference solution. In education literature, *convergent formative assessment* (this kind of assessment "determines if the learners knew, understood or could do a predetermined thing") has been shown to enhance student learning by evaluating if a student knows a concept [78, 14]. In contrast to formative assessment, current test-based grading approaches may be more suitable for *summative assessment* (aims to evaluate student learning) instead of improving learning.

In this chapter, we present ConceptGrader, a novel automated grading approach that evaluates the correctness of students' conceptual understanding in their programming assignments to support convergent formative assessment.

Our key insight is that *introductory programming courses usually teach only a few concepts, and these concepts map well to the topics taught in the course syllabi*. To support convergent formative assessment, we introduce *concept graph*, a form of abstracted control-flow graph (CFG) where we (1) select some *important* (those that correspond to topics covered in the introductory programming course syllabi) nodes and edges of a CFG, and (2) translate the selected nodes/edges into natural-language like expressions (e.g., "insert i to *newlist*" in Figure 4.1 denotes the statement "newlist.append(i)"). ConceptGrader also introduces the idea of

```
1 def remove_extras(lst):
2   newlist = []
3   for i in lst:
4     if i not in newlist:
5       newlist.append(i)
6
7
8   return newlist
```

(a) A reference program

```
1 def remove_extras(lst):
2   new_lst = [lst[0]]
3   for i in lst:
4     if i in new_lst:
5       continue
6     else:
7       new_lst += [i]
8   return new_lst
```

(b) An incorrect student program

Test Inputs	Expected Outputs	Actual Outputs
[1, 1, 1, 2, 3]	[1, 2, 3]	[1, 2] ✗
[1, 5, 1, 1, 3, 2]	[1, 5, 3, 2]	[1, 5] ✗
[]	[]	IndexError ✗
[3, 4, 5, 1, 3])	[3, 4, 5, 1]	[3, 4] ✗

(c) Test cases and actual output of incorrect student program

Figure 4.1: Examples from the *Duplicate Elimination* assignment

automated folding/unfolding of concept nodes for a more abstract level matching of concept graphs. The proposed concept graph can be used for automated grading by calculating a score based on the differences between the concept graph for the reference solution and that for the incorrect solution. Such abstraction allows us to evaluate students' efforts from their comprehension to programming concepts.

Overall, our contributions can be summarized as follows:

- We propose concept graph, an abstracted CFG that highlights programming concepts in submissions of introductory programming assignments. The concept graph contains expressions translated into natural language to enhance readability, and make it more suitable as hints to provide feedback to students. To allow more abstract matching of programs, we introduce concept node folding where we temporarily hide complex expressions in concept nodes for a fuzzy concept matching, and unfold (unhide) the expressions for precise concept matching whenever we detect a likely programming mistake within the folded concept node. Moreover, it can be used for automated grading to provide more accurate scores (with scores close to those given by manual grading) for introductory programming assignments.

- We present and implement ConceptGrader, a new automated grading approach that uses the differences between the student concept graph and reference concept graph to generate a score for a given incorrect student submission. The implemen-

tation is publicly available at *https://github.com/zhiyufan/conceptgrader*.

- We evaluate the effectiveness of ConceptGrader on 1540 student submissions from a publicly available dataset [47]. Our experiments show that compared to baselines (i.e. test-based approach and CFG-based approach), ConceptGrader performs better in terms of cosine similarity, root means squared error (RMSE), and mean absolute error (MAE) score.

- We also conduct a user study to assess the usefulness of the feedback produced by ConceptGrader. Our user study shows that ConceptGrader outperforms existing approaches by providing more useful feedback.

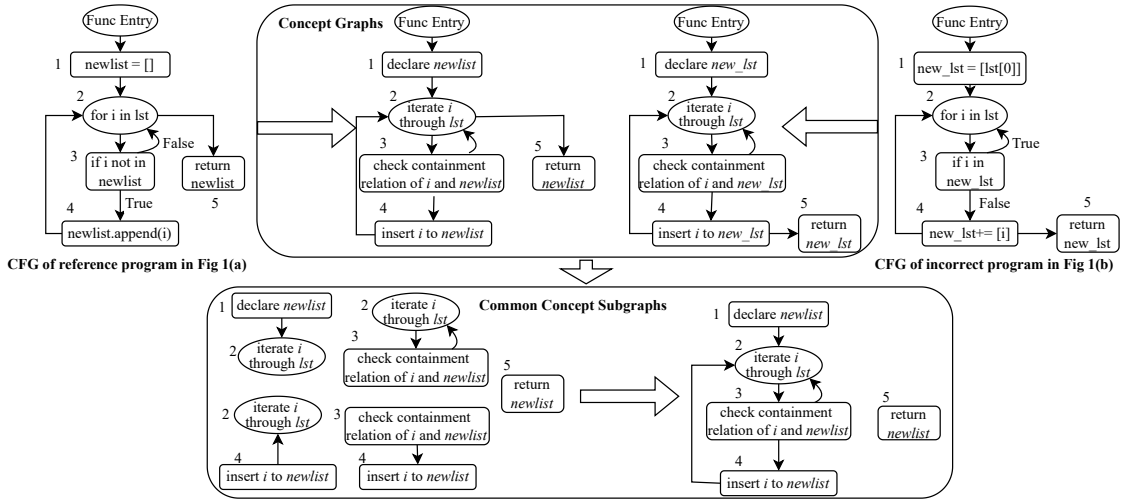
4.2 Overview

We give an overview of our concept-based automated grading approach by presenting a Python programming assignment for removing repeated elements in a list (*Duplicate Elimination*). Figure 4.1 shows the reference solution provided by the instructor, an incorrect solution submitted by a student, and a set of (input, output) pairs used to verify the correctness of each submission.

In the example in Figure 4.1, the student made two mistakes. First, instead of initializing an empty list, the student assumed that the input *lst* is not empty and initialized the new list with the given value at line 2. This incorrect assumption causes the third test case in Figure 4.1(c) to fail. Second, the student has an incorrect indentation of the return statement at line 8, which causes early termination of the program at the end of the second iteration and fails the other three test cases. As all test cases fail, a test-based grading approach will give the student a zero score for the submission. Compared to the tutor's manual inspection which gives 80% scores, the test-based grading approach underestimates the student's effort.

We now describe how we address the problem of inaccurate grading with a concept-based approach.

Concept Graph Abstraction Given the reference and incorrect student program in Figure 4.1, we construct the control flow graph (CFG) for each program. For each CFG, we follow concept abstraction rules described in Section 4.3 to extract



(a) CFGs versus concept graphs (CGs) of the reference program and incorrect program listed in Fig 4.1.

(b) Score and feedback given by the three approaches for the incorrect solution in Fig 4.1.

Approach	Score	Feedback
Test-based	0/100	The solution passes 0/4 test cases
CFG-based	20/100	The solution makes mistakes in "new_lst =[lst[0]]", "i in new_lst", new_lst += [i]"
Concept-based	87/100	The solution makes mistakes in "declare new_lst" and "return new_lst"

Figure 4.2: Examples from the *Duplicate Elimination* assignment

the programming concept represented by each basic block. Figure 4.2 shows the CFG and concept graph of the student program. The student program first declares a new list in block 1, we abstract it as *declare new_lst*. Then in block 2, the student uses a for-loop to iterate through elements in the input list *lst* in block 2, we use a concept *iterate i through lst* to show his/her understanding. In block 3, the reference program and student program use reverse conditions to check whether i exists in the previously declared *new_lst*. Although the operator is different and the exit edges point to the reverse direction, the two if-conditions represent the common idea of checking for an element in a list (represented by *containment relation of i and new_list*). At a high level, block 4 aims to insert an element into a list. One can perform the insertion in many ways, including invoke built-in functions such as *append, extend, and insert* with different arguments, or directly use the list concatenation operator "+" to insert elements to end of a list. The student program in Figure 4.1 concatenates *new_lst* with *i*, while we abstract the statement as *insert i into new_list*. We construct the reference concept graph using a similar strategy. Compared to the student solution that uses *list concatenation* with augmented

assignment "+=" at line 7, the reference program invokes the *append* method at line 5.

Concept Graph Matching and Grading Before matching student concept graph and reference concept graph, we build a bijective variable naming relation of the two programs to avoid mismatch caused by different variable names (e.g., $\{newlist : new\_lst, i : i, lst : lst\}$) using dynamic execution approach proposed in [47, 2, 99]. Given reference concept graph CG_{ref} and student concept graph CG_{stu}, ConceptGrader searches for their common subgraphs. We first find a mapping for concept nodes in CG_{ref} and CG_{stu} if they represent the same concept. In the motivating example, the concept node matching result is $\{1 \rightarrow 1, 2 \rightarrow 2, 3 \rightarrow 3, 4 \rightarrow 4, 5 \rightarrow 5\}$. If there exists an edge $e_{ref} = (n_i, n_j)$ in CG_{ref}, and $e_{stu} = (n_k, n_l)$ in CG_{stu}, where n_i matches n_k and n_j matches n_l, we consider e_{ref} and e_{stu} as common edges. We derive all common edges in CG_{ref} and CG_{stu}, and construct common subgraphs.

The bottom subfigure in Figure 4.2 shows the common subgraphs of the reference program and the incorrect student program. To improve the accuracy of auto-grading based on concept graph matching, we employ an auto-folding and unfolding mechanism of concept graph to detect the differences between the reference program and the incorrect student program. As seen in Figure 4.2, although the student implements the *declaration* concept correctly, the way of *new_lst* initialization in concept node 1 is incorrect. Refer to Section 4.4.2 where we describe how we penalize this mistake via automated concept unfolding.

The concept node 5 in Figure 4.2 is disconnected from the common concept subgraph because the reference concept graph does not contain any edge from node 4 to node 5. The difference between the two concept graphs (the top subfigure in Figure 4.2) in the corresponding edges of concept node 5 helps ConceptGrader to identify the mistake at line 8 of Figure 4.1(b). Considering the mismatched edges of concept node 5 and the incorrect initialization of concept node 1 (we consider this as a partially matched node), ConceptGrader assigns 45 points for the matching concept nodes (4.5 nodes are matched out of 5 concept nodes in total), and 42 points for the matching concept edge (5 edges are matched out of 6 concept edges in total), leading to a total score of 87. Compared to the CFG-based approach where only

the CFG nodes 2 and 5 are matched, ConceptGrader's score is more accurate.

4.3 Programming Concept Abstraction

Programming topics and mini-Python. Our main insight to model the input Python programs is that although different institutions and online learning platforms offer a great variety of CS-1 introductory programming courses [76, 28, 95, 29], the programming topics taught in these courses are often the same. Specifically, the common topics covered in these courses include *Expressions* (e.g., arithmetic expressions), *Variables*, *Simple statement* (e.g., assignment), *Conditional* (*if*-statement), *Loops* (*for*-statement and *while*-statement), *Functions*, *Lists*, and *Tuples*. Our goal is to design a concise representation of a Python program that models these programming topics. Our design is mainly based on: (1) the official Python 3 Abstract Syntax Grammar [33] that serves as a basis for our syntax rules, and (2) the key CS-1 programming topics that should be included in our concise representation. We select syntactic elements from the reference grammar that are also included in the common topics (i.e., we exclude advanced syntactic features such as lambda, yield, async and await expressions). Figure 4.3 shows the syntax for our Mini-Python grammar that supports programming topics studied in introductory Python courses. The grammar includes basic AST node types: Expression, Operators, and Statement. Each node type consists of multiple programming concepts.

Abstraction rules. Based on the mini-Python grammar in Figure 4.3, we derive a set of abstraction rules to translate the syntactic elements in the grammar. Table 4.1 shows our set of abstraction rules.

We follow two design principles for designing abstraction rules:

Human readable: We translate each syntactic element to natural language to generate human-readable feedback that can be used for explaining incorrect programming concepts. Figure 4.2(b) shows an example feedback generated by ConceptGrader.

Mitigating the program aliasing problem: We mitigate the program aliasing problem (i.e., semantically equivalent programs having several syntactically different forms [114]) by mapping several semantically equivalent syntactic elements to the

Table 4.1: Human-Readable Abstraction Rules

Rule Category	Sub-category	Example
Expression	BoolOp	x and $y \rightarrow$ logical relation of x and y
	BinOp	$x + y \rightarrow$ arithmetic relation of x and y
	UnaryOp	not $x \rightarrow$ not x
	Compare	$x == y \rightarrow$ equivalence relation of x and y
		$x > y \rightarrow$ relational relation of x and y
		x in $y \rightarrow$ containment relation of x and y
	Call	$len(i) \rightarrow$ call of len
		$x.append(i) \rightarrow$ insert i to x
	Subscript	$x[2] \rightarrow$ element of x
	Slice	$x[1:3] \rightarrow$ subrange of x
Simple Statement	Assign	$x = y \rightarrow$ declare x
		$x = [] \rightarrow$ reset x
		$x = x + 2 \rightarrow$ add x with constant
		$x = y + z \rightarrow$ update x
		$x = x + [i] \rightarrow$ insert i to x
	AugAssign	$x+ = 2 \rightarrow$ add x with constant
		$x+ = [i] \rightarrow$ insert i to x
	Return	*return* expr $\rightarrow$ return abstract(expr)
Control Statement	If	*if* expr $\rightarrow$ check abstract(expr)
	For	*for* i *in* lst $\rightarrow$ iterate i through lst
	While	*while* x < y $\rightarrow$ iterate compare relation

same translation. For example, we translate $x = x + [i]$ and $x+ = [i]$ into "insert i to x". Mitigating the program aliasing problem helps in increasing the accuracy in matching two semantically equivalent concept nodes.

$$
\begin{aligned}
\textit{Expression} \quad & e ::= e\ \textit{boolop}\ e \mid e\ \textit{op}\ e \mid \textit{uop}\ e \mid e\ \textit{cop}\ e \\
& \mid e\ (e, ..., e) \mid \textit{const} \mid \textit{id} \\
& \mid \{e : e, ..., e : e\} \mid \{e, ..., e\} \mid [e, ..., e] \mid (e, ..., e) \\
& \mid e[e] \mid e : e : e \\
\textit{Statement} \quad & s ::= e \mid s; s \mid e = e \mid e\ \textit{op} = e \\
& \mid \textbf{for}\ e\ \textbf{in}\ e : s \mid \textbf{while}\ e : s \mid \textbf{if}\ e : \textbf{else} : s \\
& \mid \textbf{continue} \mid \textbf{break} \mid \textbf{return}\ e \\
\textit{BoolOp}\ \textit{boolop} ::= & \ \textbf{and} \mid \textbf{or} \\
\textit{BinaryOp} \quad \textit{op} ::= & \ + \mid \textbf{-} \mid \textbf{*} \mid / \mid \% \mid \textbf{**} \\
\textit{UnaryOp} \quad \textit{uop} ::= & \ \sim \mid \textbf{not} \mid \textbf{UAdd} \mid \textbf{USub} \\
\textit{CompareOp} \quad \textit{cop} ::= & \ == \mid \textbf{!=} \mid < \mid <= \mid > \mid >= \\
& \mid \textbf{is} \mid \textbf{is not} \mid \textbf{in} \mid \textbf{not in}
\end{aligned}
$$

Figure 4.3: Syntax of mini-Python based on the abstract Python grammar [33]

Concept graph construction. We first define the notion of a concept graph for a program. The nodes of the concept graph are called *concept nodes* which are defined as follows:

Definition 5 (Concept Node). *A concept node* $cn=(c_1, c_2, \ldots, c_i)$ *in a program* p *is a set of programming concepts, where each concept node* cn *is abstracted from a basic block* $b = (s_1, s_2, \ldots, s_j)$ *in control-flow graph of* p *that* $\forall c \in cn, \exists s \in b(c \equiv f(s))$*, where* f *is one of the abstraction rules in Table 4.1.*

Definition 6 (Concept Edge). *A concept edge* $ce=(cn_1, cn_2)$ *in a program* p *is a transfer of control flow from a concept node* cn_1 *to* cn_2*. Each concept edge* ce *is abstracted from the corresponding edge* e *in the control-flow graph of* p*. Specifically, our abstraction preserves the control flow transitions of* e *but removes the true/false label from the conditional edges of* e*.*

Definition 7 (Concept Graph). *Let* $CG(p) = (N, E)$ *be the concept graph of* p*,* $CG(p)$ *is an abstracted graph of* $CFG(p)$*, where each node* $n \in N$ *represents a concept node, and each concept edge* $e = (n_i, n_j) \in E$ *corresponds to a possible transfer of control from concept* n_i *to concept* n_j*.*

Given a control-flow graph CFG of program p, we construct a concept graph CG of p by following Algorithm 2. For each edge $e = (b_{src}, b_{tgt}) \in CFG$, we abstract the source and target basic blocks separately, and re-construct a concept edge $ce = (c_{src}, c_{tgt})$. Given a basic block b as input, the *abstract(b)* procedure produces the concept node for b by traversing the AST of each statement to convert each statement to a programming concept. Starting from the parent node of each leaf node, ConceptGrader uses the corresponding abstraction rules from Table 4.1 (line 19 in Algorithm 2) for all non-leaf nodes via a bottom-up traversal until the AST node is the root node or it has been previously abstracted.

Abstracting concept edges For each edge $e = (b_{src}, b_{tgt}) \in CFG$, we abstract the source and target basic blocks separately, and re-construct a concept edge $ce = (c_{src}, c_{tgt})$. In a traditional control flow graph, the edges are usually annotated with a label representing the conditional branches (e.g., "True" and "False" in the CFGs in Figure 4.2). In contrast, in a concept graph CG, we abstract away the

true/false label but we still keep the actual predicates in a control flow edge e if the source node of e includes a conditional statement. This abstraction is based on our observation that students often implement conditional statements in various syntactically different but semantically equivalent ways.

Concept node folding Automated source code folding is a technique that automatically creates a code summary by hiding unimportant code elements in a program that are not useful and helps developers to get an overview idea of the program on first viewing. It has shown promising results in the context of source code summarization to optimize the similarity between the code summary and the source code [34]. Inspired by the idea of source code folding in code summarization, we introduce the idea of *concept node folding* where we temporarily ignore part of the complex expressions in a concept node. The folded concept nodes are unfolded until an automated repair engine detects that patches exist for the folded nodes when matching reference and student concept graphs (refer to Section 4.4.2 for the details). Given the AST of the expressions in a statement, we define *concept depth* as the number of times that abstraction rules are applied such that the AST tree depth is compressed to 1. Given a $CG(p)$, we say a concept node to be *foldable* if the concept node it represents has *concept node depth>2*. Specifically, ConceptGrader hides the content of a node if it is foldable (lines 20–21 in Algorithm 2). By hiding the content of a concept node with complex expressions, our approach is essentially excluding parts of a complex expression during concept graph matching. In this case, concept folding helps us to abstract away the irrelevant differences (i.e., different but correct implementation) between the reference program and the student program. The folded concept node will be unfolded only when our approach detects that a difference in the reference program and student program is related to a fix in the corresponding concept node in the student program (the fix is generated by an automated program repair engine [47]).

4.4 Graph Matching and Grading

Given concept graphs of reference solution and student solution CG_{ref} and CG_{stu}, we perform graph matching to assess how the intention of a student solution matches

Algorithm 2 Concept Graph Construction

Input: Control-flow graph of program p CFG
Output: Concept graph of program p CG

```
 1: procedure ConstructConceptGraph(CFG)
 2:     Let CG be the concept graph
 3:     for basic blocks (b_src, b_tgt) in CFG.edges() do
 4:         c_src = abstract(b_src)
 5:         c_tgt = abstract(b_tgt)
 6:         CG.addEdge(c_src, c_tgt)
 7:     end for
 8:     return CG
 9: end procedure
10: procedure abstract(b)
11:     Let cn be a concept node, visited be a list of abstracted AST nodes
12:     for stmt in basic block b do
13:         for l in getLeafASTNodes(stmt) do
            ▷ abstract non-leaf node
14:             pnode = getParent(l)
15:             c = abstractNode(pnode, visited)
16:             visited.add(pnode)
17:             cn.addConcept(c)
18:         end for
19:     end for
20:     return cn
21: end procedure
22: procedure abstractNode(n, visited)
23:     Let c be a programming concept
24:     c = abstractRules(n)                    ▷ abstract using rules in Table 4.1
25:     if foldable(c) then
26:         c = fold(c)                         ▷ fold by hiding the content of node
27:     end if
28:     if isRootNode(n) or n ∈ visited then
29:         return c
30:     end if
31:     return abstractNode(getParent(n),visited)
32: end procedure
```

the reference solution at the concept level. Finding the maximum common subgraphs between two graphs is a NP-complete problem [18]. However, students' programs in CS-1 education context are often small. We find common subgraphs by iterating all common edges and then connect edges together via connected components to get all subgraphs. The graph matching algorithm consists of two phases. First, we construct a set of subgraphs to represent the common concepts of CG_{ref} and CG_{stu} based on their common concept nodes and edges. Second, we introduce the idea

Algorithm 3 Concept Graph Matching

Input: Reference concept graph G_r, Student concept graph G_s, Variable mapping of reference program and student program vM
Output: The matched subgraphs *subgraphs*

```
1: procedure GraphMatching(G_r, G_s, vM)
   ▷ dict of matched concept nodes and list of edges
2:     nodeDict, edges = {}, []
3:     for n_s in G_s.nodes() do
4:         n_r = findNodeInGraph(G_r, n_s, vM)
           ▷ update dictionary for newly matched nodes
5:         if n_r ∉ nodeDict.val() then
6:             nodeDict[n_s] = n_r
7:         end if
8:     end for
9:     for (n_s, n_r) in nodeDict do
10:        N_s, N_r = listOfNeighbors(n_s), listOfNeighbors(n_r)
11:        N' = findMatchedNodes(N_s, N_r, nodeDict)
12:        edges.addEdges(n_s, N')
13:    end for
14:    subgraphs = merge(edges)
15:    return subgraphs
16: end procedure
```

of an automated concept unfolding approach to distinguish the minor difference between two matched concept nodes to improve match accuracy further.

4.4.1 Concept Graph Matching

The goal of concept graph matching phase is to find an initial concept-matching relation of reference and student solution at a high level. For each concept node in the student concept graph, we find a concept node from the reference concept graph that (1) represents the same programming concepts category and (2) involves mapped variables of the student concept node in the abstraction.

Then, ConceptGrader identifies the neighbor nodes N_s and N_r for matched concept node pairs $(n_s : n_r)$ in *nodeDict* and finds matched nodes of N_s and N_r by checking nodes $n_s^{'} \in N_s$, whether $nodeDict[n_s^{'}] == n_r^{'}$ and $n_r^{'} \in N_r$. For all nodes $n_s^{'} \in N_s$ that satisfy the condition, we consider $e = (n_s, n_s^{'})$ as a matched edge and add it into list of matched edges *edges* (lines 7–10 in Algorithm 3). For each pair of edges (i.e., e_1=(src_1, dst_1) and e_2=(src_2, dst_2)) in *edges*, ConceptGrader then merges e_1 and e_2 into a subgraph if $src_1 == dst_2$ or $dst_1 == src_2$ (line 11

in Algorithm 3). Note that standalone concept nodes (e.g., concept node 5 in Figure 4.2) and edges might exist, which eventually lead to a set of subgraphs.

4.4.2 Automated Concept Unfolding

As mentioned in Algorithm 2, we construct the concept graph at a high level of abstraction and fold concept nodes to avoid exposing details, which allows more flexible matching. However, when we match the student concept graph with the reference concept graph, we may need to unfold certain concept nodes on-the-fly during the matching. This is because for the concept nodes in student program which contain mistakes, the folding process may mask those mistakes by showing only high-level concept.

In the example of Figure 4.2, the reference and incorrect student programs have the same programming concept *declare newlist* in folded student concept node cn_{s1}, but the specific values assigned to the two *newlist* variables are different. In this case, the folded concept node fails to capture the differences in terms of the declared values, so ConceptGrader assigns an overestimated score to the student program.

To assess student programs more precisely, the details of concept nodes with mistakes need to be explored by unfolding. We leverage program repair engine to get patches for each incorrect student program. Our intuition is that if a patch of the incorrect student program exists within a folded concept node *cn*, then this indicates that *cn* contains a programming mistake that needs to be fixed, but the mistake was hidden because of folding.

When a program repair engine detects a patch exists for a student concept node, the unfolding mechanism is triggered to expand the previously folded content of both the student concept node and the matching reference concept node. Then, ConceptGrader deducts scores by performing detailed matching of each mistake made in the student concept node.

Consider the example in Figure 4.1 where the program repair engine generates a patch $new\_lst = [lst[0]] \rightarrow new\_lst = []$ for the concept node *declare new_list*. As a patch exists within the concept node of the incorrect student program, ConceptGrader unfolds the concept node of incorrect program into {“declare new_list”, “element of lst”}, while the corresponding concept node of the reference program still remains

unchanged. Consider another example with incorrect control-flow transition in our motivating example (Line 8 in Figure 4.1(b)). Unfolding is not triggered in this example because the patch is meant for fixing a concept edge, and our approach in Algorithm 3 is able to detect this discrepancy by producing a separate concept subgraph with only one concept node ("*return newlist*" in Figure 4.2).

4.4.3 Concept Based Grading

Our goal is to compute score for each matching graph $G_{matched}$ between the student concept graph G_s and the reference concept graph G_r, so as to compute the total score for the student program. We calculate the score of a matching graph $G_{matched}$ by comparing the concept node similarity and concept edge similarity between G_s and G_r. Algorithm 4 shows the overall grading workflow. ConceptGrader first constructs concept graphs G_s for student program P_s and G_r for reference program P_r, then it matches G_s and G_r with the help of a variable mapping relation of P_s and P_r to get the list of matched subgraphs *matchedList* by following Section 4.4.1 (Lines 2–5). *runAPR* invokes program repair engines to generate *patches* for the incorrect student program, which involves automated concept unfolding as described in Section 4.4.2.

For each $G_{matched}$ in *matchedList*, we traverse all concept node *cn* and extract the folded concept node pair (cn_s, cn_r) representing student concept node cn_s and reference concept node cn_r. Then, ConceptGrader unfolds cn_s, cn_r to get detailed content if the automated program repair engine has produced patches for the corresponding cn_s (Lines 13–15). By comparing all concepts in cn_s and cn_r, we collect a list of concepts *matchC* that exist both in cn_s and cn_r, and *totalC* that represents a list of all concepts in cn_r (Line 16).

Specifically, the score of a matching graph $G_{matched}$ consists of two parts: (1) average concept node similarity and (2) average concept edge similarity. Given $G_{matched}$, we define *average concept node similarity* of $G_{matched}$ as the average number of matching concept nodes in the reference concept graph G_r, calculated using the equation below:

$$conceptNodeSim(G_{matched}) = \frac{1}{nodeSize(G_r)} \sum_{i=1}^{n} \frac{matchC(cn_i)}{totalC(cn_i)}$$

where n denotes the number of matching concept nodes in $G_{matched}$, *matchC* repre-

sents concepts that exist in student concept node cn_s and reference concept node cn_r, $totalC$ denotes all concepts in cn_r, and nodeSize(G_r) denotes the number of nodes in G_r.

We define *average concept edge similarity* of $G_{matched}$ as the number of matching concept edge in the reference concept graph G_r, calculated using the equation below:

$$conceptEdgeSim(G_{matched}) = \frac{edgeSize(G_{matched})}{edgeSize(G_r)}$$

where $edgeSize(G)$ returns the number of edges in a graph G (e.g., edgeSize($G_{matched}$) denotes the number of edges in $G_{matched}$).

Finally, we compute the final score of the student program P_s as the sum of scores for all matched concept subgraphs $G_{matched}$ in between G_r and G_s. The equation is shown below:

$$score(P_s) = \frac{\alpha}{2} \times \sum_{i=1}^{m}(conceptNodeSim(G_i) + conceptEdgeSim(G_i))$$

In this equation, α represents the total score of the programming problem (usually determined by the instructor), m is the number of graphs in the list of matched subgraphs $matchedList$ and G_i is a matched concept subgraph in $matchedList$.

Feedback generation. ConceptGrader generates feedback by pointing out (1) missing concepts, and (2) problematic concepts ("...makes mistakes..." in Figure 4.2). Specifically, ConceptGrader identifies *missing concepts* by checking if (1) the concept nodes exist in reference concept graph, but (2) a matching node cannot be found in student concept graph. ConceptGrader considers the matched student concept nodes as *problematic concepts* if (1) concept nodes exist in reference concept graph and have matching concept nodes in student concept graph, but the unfolding mechanism indicates that a programming mistake exists, or the transfer relations of the matched concept nodes are different (e.g., concept node 5 in Figure 4.2 is shown as a mistake in the generated feedback). Instead of providing feedback via patches (prior study show that novice students may not know how to effectively utilize the generated patches as hints [107]), our feedback highlights the wrong concepts to promote active learning by asking "how can you fix the code here?".

Algorithm 4 Overall Grading Workflow

Input: Student program P_s, Reference program P_r, Test suite T, Total score α
Output: The final *score*

procedure GRADE(P_s, P_r, T, α)
  G_s = constructConceptGraph(P_s) ▷ Algorithm 2
  G_r = constructConceptGraph(P_r) ▷ Algorithm 2
  vM = variableMapping(P_s, P_r, T)
  $matchedList$ = graphMatching(G_r, G_s, vM) ▷ Algorithm 3
  $patches$ = runAPR(P_s, P_r, T)
  $score = 0$
  for $G_{matched}$in $matchedList$ **do**
    $score$ += computeScore($G_{matched}, G_r$)
  end for
  return $\alpha \times score$
end procedure
procedure COMPUTESCORE($G_{matched}, G_r$)
  for concept node cn in $G_{matched}.nodes$ **do**
    cn_s, cn_r = findConceptNode(cn)
    if hasPatches($patches$, cn_s) **then**
    ▷ unfold by expanding the content of node
      cn_s = unfold(cn_s)
      cn_r = unfold(cn_r)
    end if
    $matchC$, $totalC$ = compareConceptNode(cn_s, cn_r)
    $conceptNodeSim$ += $\frac{matchC}{nodeSize(G_r) \times totalC}$
  end for
  $conceptEdgeSim = edgeSize(G_{matched})/edgeSize(G_r)$
  return ($conceptNodeSim$ + $conceptEdgeSim$) / 2
end procedure

4.5 Evaluation

We evaluate ConceptGrader by addressing the following research questions:

RQ1: How does ConceptGrader perform in terms of grading accuracy, as compared to baseline approaches?

RQ2: How does test failure rate affect performance of ConceptGrader and baseline tools?

RQ3: What are the reasons for ConceptGrader's incorrect grading?

Implementation. We implemented the proposed approach in the tool ConceptGrader. We choose Refactory [47] as the automated program repair tool invoked during unfolding because it has shown promising results in fixing introductory assignments written in Python, particularly with respect to a reference correct solution. Similar to prior evaluations of approaches designed for programming assignments that sample additional reference solutions from correct students' submissions [41, 2, 47, 99], ConceptGrader follows the procedure of prior work [99] that selects five programs from correct students' submissions as additional reference solutions to mitigate the problem when students' implementation and reference's implementation use a different solving approach. To select five additional reference programs as representatives of most student programs, we (1) run Clara [41] to cluster correct student submissions, and (2) select one representative program from the top-5 clusters with most student programs. If an instructor's reference solution is the same as one of the five additional reference solutions, we select the instructor's reference solution and four other reference solutions. Then, ConceptGrader compares a student program against all reference solutions and selects the highest score as the final score. ConceptGrader currently supports programs with Python 3.10. We construct CFG using staticfg [24] and further customize it to build a concept graph.

Dataset. We evaluate ConceptGrader on five assignments from a CS-1 Python dataset used in the prior evaluation of introductory programming assignment [47]. For each programming assignment, the instructor prepared a reference solution and a test suite that evaluates students' correctness. Other datasets used in previous work [41, 99, 107] are either not publicly available or use different programming languages (e.g., C). Our concept abstraction rules currently do not support all programming language constructs (e.g., lambda expression). We exclude submissions with unsupported features, and trivial student programs without any real implementation (i.e., programs with less than three lines of code) from our evaluation. In total, we have 1540 incorrect submissions remaining.

Ground truth construction. The Refactory dataset [47] uses execution results of tests as feedback to students, and it does not have the *ground truth score* (the correct score to be assigned for an assignment) for each submission. We invited eight

senior Computer Science undergraduate students who have experience working as teaching assistants to be the annotators for grading those incorrect submissions. We provide the annotator with the problem description, instructors' reference solution, and instructors' test suite. We asked annotators to run test cases and grade the submissions by functionality. To provide freedom in grading, we did not mention any other steps (e.g., using the execution results of tests).

To mitigate potential grading bias, annotators are unaware of the existence of ConceptGrader, and each submission is graded by two annotators, and we asked each annotator to grade each submission out of the same total score of 100 (α=100 in the last equation in Section 4.4.3). If the scores given by the two annotators differ less than 10, we take their average as the ground truth score. Otherwise, a third annotator participates in the discussion until they reach a consensus. We use the same ground truth for evaluating RQ1–RQ3.

Baselines. We compare ConceptGrader against two baselines: (1) test-based approach [48, 7, 102, 96, 52, 46], and (2) CFG-based approach [97]. We compare with the test-based approach because it is the most widely used approach. To ensure fair comparison, we provide the same set of test cases to the test-based approach and the program repair engine used in our unfolding (described in Section 4.4.2). We do not compare ConceptGrader against the recent CFG-based automated grading technique [97] because (1) its implementation targets C programming assignments where ConceptGrader focuses and is evaluated on Python programming assignments, and (2) their approach can only be used to grade correct programs (passing all tests) as we confirmed with the authors, where ConceptGrader focuses more on evaluating incorrect programs. To ensure a fair comparison, we implemented a CFG-based baseline by removing concept abstraction, concept graph construction, and concept folding/unfolding from ConceptGrader (i.e., we keep the variable mapping to allow CFG-based baseline to handle different naming styles). Moreover, we do not compare with AutoGrader because it only returns a binary correct/incorrect based on path deviation [63].

The goal of automated grading is to automatically assign a score for a student program such that tutors can directly accept it or minimally adjust it. We use three metrics: (1) Cos-sim (cosine similarity), (2) RMSE (root means squared error), and

Table 4.2: Automated grading results of four approaches for incorrect student submissions on five assignments from the Refactory dataset[47]. The columns *Test* and *CFG* denote test-based grading and CFG-based grading, and *CG* and *CG'* show ConceptGrader and ConceptGrader without concept unfolding. The column "# of Inc. Sub." shows the number of incorrect submissions for each assignment, whereas the column "# of TC" denotes the number of test cases for each assignment, the column LoC represents the line of code. The columns "Cos-sim", "RMSE", and "MAE" represent the cosine similarity, root means squared error, and mean absolute error between automatically generated and ground truth scores. The columns "Average Time Taken (s)" denotes the average time taken in seconds to produce the score and feedback for a student submission in each assignment. We highlight the best result in bold.

Assig	# of	Cos-sim				RMSE				MAE				Average Time Taken (s)			
nment	Inc.	Test	CFG	CG'	CG	Test	CFG	CG'	CG	Test	CFG	CG'	CG	Test	CFG	CG'	CG
Week 1	544	**0.95**	0.79	0.93	0.94	**26.70**	56.91	32.19	30.26	**17.41**	51.03	25.03	23.62	4.18	**1.22**	1.23	24.84
Week 2	353	0.84	0.80	0.85	**0.87**	25.21	27.54	23.25	**22.53**	17.59	21.91	18.59	**16.84**	**6.91**	8.37	9.28	35.19
Week 3	272	0.69	0.80	0.85	**0.88**	64.41	50.91	38.73	**35.18**	60.39	43.33	30.97	**28.05**	**4.07**	7.85	7.87	37.52
Week 4	263	0.68	0.81	0.92	**0.93**	49.58	41.00	26.38	**23.12**	43.77	34.62	20.82	**18.35**	**3.97**	10.35	9.00	44.28
Week 5	108	0.62	0.74	**0.90**	**0.90**	63.41	53.69	34.92	**32.31**	58.70	46.66	28.96	**26.11**	**5.07**	18.58	17.69	63.70
Total	1540	0.81	0.79	0.91	**0.92**	42.96	51.73	32.14	**30.41**	32.56	44.82	25.86	**22.93**	**4.89**	6.31	5.94	35.49

(3) MAE (mean absolute error) to evaluate the distance between auto-generated scores and tutors' ground truth scores. Given the auto-generated and ground-truth scores for all incorrect student programs, Cos-sim evaluates their cosine similarity in the normalized vector space.

Given the ground truth score y_i, the auto-generated score $\hat{y}_i$, and N samples, their equations are: $Cos-sim(y,\hat{y}) = \frac{y \cdot \hat{y}}{|y||\hat{y}|}$, $RMSE(y,\hat{y}) = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2}$, and $MAE(y,\hat{y}) = \sum_{i=1}^{N}|y_i - \hat{y}_i|$

RMSE and MAE are often used to evaluate the differences between values predicted by a model [103]. They represent the absolute closeness of auto-generated scores and ground-truth scores by computing their standard deviation and absolute distance. Lower RMSE and MAE values indicate better performance.

4.5.1 RQ1: Overall Grading Accuracy

Table 4.2 shows the results of all incorrect student programs from the five selected assignments [47]. On average, ConceptGrader outperforms all baseline approaches by achieving Cos-sim at 0.92, whereas test-based approach and CFG-based approach achieve 0.81 and 0.79 for Cos-sim, respectively. Compared to the two baseline approaches, ConceptGrader produces the lowest RMSE (30.41) and MAE (22.93) values, which improves the result of test-based approach by 30% and 29% and the

result of CFG-based approach by 41% and 49%, in terms of RMSE and MAE. The low average RMSE and MAE values indicate that ConceptGrader is effective in predicting the ground truth scores for programming assignments.

It is worthwhile to mention that the grading accuracy of test-based approach in *Sequential search* is better than the other four assignments. We analyzed the reason for the higher accuracy of test-based approach in "Sequential search" task. Table 4.3 shows that around 59.3% (323/544) student programs pass more than 75% of test cases in "Sequential search", whereas the ratio on the other four assignments is 19.0% on average. Passing more test cases often indicates better quality of a program. When a program passes majority of test cases, tutors also tend to assign relatively high scores. However, if a program fails majority of test cases, it does not necessarily mean the program is completely incorrect because even a subtle mistake can cause different behavior.

Average time taken. In terms of the average time taken to generate a score for a student program, test-based grading is the fastest as it only requires running the student program against all test cases. ConceptGrader is slower than other approaches because it may need to invoke program repair engine several times to generate patches for concept unfolding in the final grading process. Overall, the average time taken 35.49s is acceptable as prior study shows that human tutors often take 100 seconds to grade one student submission [107].

Table 4.3: The test failure rate distribution for evaluated submissions.

Assignment	# Incorrect Submissions	# of TC	Test failure rate (%) 0–25	25–50	50–75	75–100
Sequential search	544	11	323	91	59	71
Unique dates/months	353	17	159	54	29	111
Duplicate elimination	272	4	14	9	29	220
Sorting tuples	263	6	10	23	82	148
Top-k elements	108	5	6	3	6	93
Total/Average	1540	9	512	180	205	643

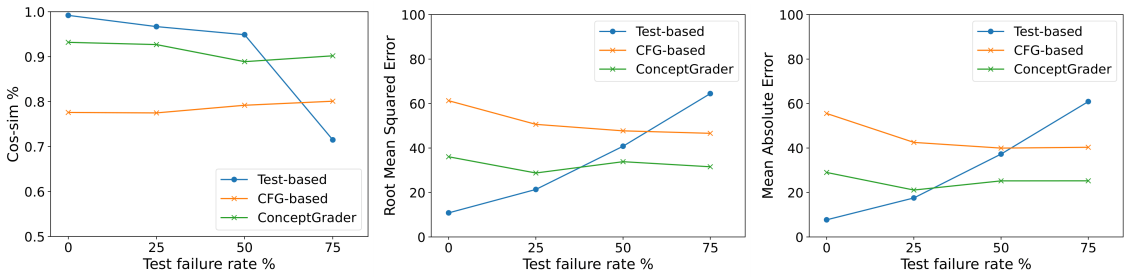
Effectiveness of concept abstraction and concept unfolding. Although ConceptGrader shows better grading accuracy compared to the two baseline approaches, it is worthwhile to investigate the effectiveness of each component in ConceptGrader. We implemented another version of ConceptGrader denoted as *CG-wo-f* by removing

concepts unfolding (described in Section 4.4.2). We first compare *CFG* and *CG-wo-f* to show the impact of automated concept abstraction. The difference between *CFG* and *CG-wo-f* is that *CFG* matches CFG of student's program and CFG of reference program by comparing the source code in basic blocks, whereas *CG-wo-f* first applies the abstraction rules in Table 4.1 for basic blocks in CFG of student's program and CFG of reference program to construct corresponding concept graphs, then matches nodes in student concept graph and reference concept graph. Table 4.2 shows that with concept abstraction and concept graphs, *CG-wo-f* improves Cos-sim over CFG-based approach by 0.12, and reduces RMSE and MAE over CFG-based approach by 19.59 (38.9%) and 18.96 (42.3%) respectively. In addition, *CG-wo-f* has almost no overhead regarding time taken to grade a student program (average time taken is 5.94s).

Based on *CG-wo-f*, *CG* takes advantage of patches generated by automated program repair engine (Refactory) as hints to identify students' mistakes that have been abstracted in automated concept folding process, and unfolds those students' concepts to compare in detail to capture those minor mistakes. The comparison between *CG* and *CG-wo-f* shows the impact of automated concept unfolding. Overall, the result of Cos-sim does not change much. This is because ConceptGrader with folding and unfolding is a fine-tuning procedure. When a folded concept node in student concept graph finds a matching in reference concept graph but Refactory reveals that a patch is required for the concept node, ConceptGrader still assigns partial scores based on the coverage of matching unfolded concepts in the concept node. The usefulness of folding and unfolding is shown by the lower RMSE and MAE values (i.e., the values improved by 5.4% and 11.3%, respectively). This means that *CG*'s grading has less discrepancy with respect to the tutors' ground truth compared to *CG-wo-f*.

4.5.2 RQ2: Relation with Test Failure Rate

Our intuition of designing ConceptGrader is that in introductory programming assignments, even a simple mistake could fail many tests within the test suite, which leads to a test-based grading approach that may underestimate students' understanding and effort. As Table 4.2 shows the overall grading accuracy of each



(a) Average Cos-sim **(b) Average RMSE** **(c) Average MAE**

Figure 4.4: Average grading performance of all incorrect student submissions across different test failure rates.

approach on all incorrect student programs, we are also interested in investigating the effect of *test failure rate* (percentage of failing tests in the entire test suite for an assignment) on grading accuracy. We divide all incorrect student submissions (we consider a student submission as incorrect if test failure rate > 0) into four groups based on their test failure rate (0%–25%, 25%–50%, 50%–75%, and 75%–100%). Figure 4.4 shows the grading accuracy of all four approaches regarding different test failure rates.

For all the evaluated metrics (Figures 4.4(a), 4.4(b), and 4.4(c)), our results show that when test failure rate is low (0–50%), test-based grading tends to be more effective. However, as test failure rate increases, the performance of test-based grading downgrades. In contrast, the performance of ConceptGrader tends to be stable as the test failure rate increase. This result indicates that ConceptGrader preserves the ability to capture students' misunderstanding, not affected by the changes in test failure rate. Considering the fact that most students' programs fail half of test cases, it illustrates the importance of a concept-based grading approach.

4.5.3 RQ3: Limitations of ConceptGrader

To understand the limitations of ConceptGrader, we manually analyzed (1) the cases where ConceptGrader performs worse than test-based approach, (2) the quality of reference solutions to ConceptGrader.

Analyzing Unreasonable Scores: To reduce the manual effort in analyzing cases where the differences between the scores given by a test-based s_{test} and those assigned by ConceptGrader $s_{concept}$ are minor, we only analyze cases where the scores

given by ConceptGrader is *unreasonable* (i.e., the difference between $s_{concept}$ and s_{test} is greater than 5 points). In total, we observed 565/1540 (36.7%) scores to be unreasonable.

Our manual analysis of the 565 unreasonable scores shows that unreasonable scores occur due to: (1) syntactically different student implementations, and (2) inaccurate variable mapping. Specifically, although we design the abstraction rules to mitigate the program aliasing problem by translating different programs to the same representation, our rules are not exhaustive so ConceptGrader fails to match correctly when the incorrect student programs are substantially different from the reference solutions, especially for cases where the test failure rate is low. When the students' programs use sub-optimal algorithms or syntactically different implementations, ConceptGrader could not match the concept nodes and edges accurately, resulting in a lower score assigned to the incorrect student programs. Meanwhile, as ConceptGrader relies on the variable mapping mechanism of Refactory [47], we observe that ConceptGrader may produce inaccurate scores when the student programs use too many temporary variables which increase the number of un-mapped variables in the variable mapping.

In the future, a hybrid automated grading tool that combines ConceptGrader and test-based approach may be interesting to be explored. Using the AST edit distance between student program and reference program as estimator of the quality of student program, ConceptGrader suggests scores when AST edit distance is small but test failure rate is high, while test-based approach can still be used when test failure rate is low, but AST edit distance is high (indicating there is no good reference solution for the student program).

Impact of Different Numbers of Reference Solutions: In previous sections, we conducted experiments for ConceptGrader using multiple reference solutions. Although using multiple reference solutions from correct student solutions is a recent trend in other relevant work [99, 41, 47, 2], there may not be sufficient high-quality reference solutions available for each programming assignment in practice. To address this concern, we analyze the impact of different numbers of reference solutions to ConceptGrader by grading with fewer reference solutions. Given the five reference solutions crafted in Section 4.5, we gradually remove the reference solutions being used by ConceptGrader, starting from the most less popular reference solution, until

there is only instructors' provided reference solution. Table 4.4 shows the average results for the five programming assignments as we reduce the number of reference solutions used in ConceptGrader. From Table 4.2 and Table 4.4, we can observe that using only two reference solutions, ConceptGrader already performs better than test-based approach. Compared to only one reference solution, the performance of ConceptGrader with three reference solutions increases by 7% for Cos-sim, 21.1% for RMSE, and 23.6% for MAE, which reaches a comparative level of the default configuration, using all reference solutions (# of Ref. Solutions=5).

Table 4.4: The impact of different number of reference solutions in ConceptGrader (CG) and ConceptGrader without unfolding (CG-wo-f).

# of Ref.	Cos-sim		RMSE		MAE	
Solutions	CG	CG-wo-f	CG	CG-wo-f	CG	CG-wo-f
5	0.92	0.91	30.41	32.14	22.93	25.86
4	0.91	0.90	32.05	33.82	23.76	27.05
3	0.91	0.88	33.28	34.97	26.82	28.53
2	0.89	0.85	36.42	37.51	30.46	31.68
1	0.85	0.82	42.23	45.83	35.11	37.32

4.6 User Survey

User Survey Setup. To obtain qualitative data for demonstrating the effectiveness of ConceptGrader, we conducted a survey among 29 tutors from two semesters of a large CS-1 introductory programming course. The tutors include both lab instructors who taught lab sessions and graders who grade programming assignments. All tutors are undergraduates who have taken the course in previous semesters from the Computer Science department. Among the 29 tutors with whom we have shared the survey, we received 16 replies. Participation in the survey is voluntary, and the authors do not have any personal connection with the participants. To reduce bias due to personal preference towards a particular approach, we anonymize the name of each approach. The survey aims to collect tutors' opinions on the grade and feedback generated by three automated approaches. In total, it contains ten incorrect student submissions drawn from five assignments.

User Study Questions. Each tutor answers a question about prior teaching experience. On average, the 16 tutors have served as tutors 2.1 times. We randomly sampled two incorrect submissions for each assignment from the evaluated dataset (in Section 4.5). For each incorrect submission, we provide (1) the instructor's reference solution, (2) an assignment description, (3) test cases, and (4) the questions below:

Q1. Rate the quality of the automated mark in terms of assessing students' understanding and effort.

Q2. Rate the usefulness of automated feedback to students in terms of improving their learning outcome, based on your previous learning experience.

Q3. To what extent will the automated feedback be preferable or as good as the feedback that you would manually give?

The first question (Q1) aims to assess the quality of the generated score, whereas Q2 and Q3 are designed to assess the quality of the generated feedback (Figure 4.2(b) shows an example of the generated feedback). For each incorrect student submission, participants need to rate each item based on a five-point Likert scale (with 1 being very low and 5 being very high). We allocate 30 minutes for each tutor to complete the survey.

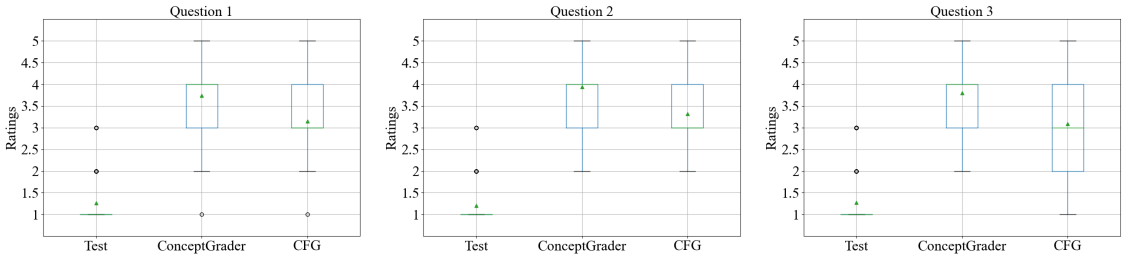


Figure 4.5: The boxplot of average rating of all user study questions. Green triangle represents the mean value. The whiskers denote the minimum/maximum value, and the rectangle denotes the first/third quartile.

User Study Results. Figure 4.5 presents the results of the 16 tutors' ratings for all user study questions. Overall, we observe that the tutors show a positive attitude of ConceptGrader for all questions (Q1 – Q3) with a mean rating of 3.8. Tutors rate highest (average rating of 3.7) for the quality of ConceptGrader's generated scores (Q1), compared to test-based and CFG-based approaches (average rating of 1.3 and

3.2, respectively). As those who are invited for the ground truth constructions are different from tutors for the user study, this further confirms our grading accuracy experiment in Section 4.5. For the usefulness of automated feedback in terms of improving students' learning outcomes (Q2), tutors think that ConceptGrader is the most useful among all approaches (average rating of 3.9). This indicates that our approach provides better support for convergent formative assessment. Compared to the baseline approaches, tutors prefer the quality of the feedback generated by ConceptGrader (average rating for Q3 is 3.8). This shows that the feedback constructed via our human-readable abstraction rules may benefit tutors in designing personalized feedback for students.

Significance of study result. To validate the significance of our study result, we performed a two-tailed T-test for the difference between the results for CFG-based approach and ConceptGrader. The result shows that our study has a p-value $<$ 0.001 for Q1 to Q3, indicating that the difference between CFG-based approach and ConceptGrader in Figure 4.5 is statistically significant. Moreover, the standard deviation of CFG-based approach and ConceptGrader for Q1 to Q3 is (0.84, 0.77, 0.89) and (0.93, 0.76, 0.86) respectively.

4.7 Threats to Validity

External. Our findings of programming concepts focus on Python introductory programming courses. Hence, our experiments may not be exhaustive and generalize to other languages. We evaluate and implement ConceptGrader within the scope of mini-python, ConceptGrader may produce inaccurate scores if programming assignments include language features beyond mini-python. We left the extension of more advanced programming features in Python as future work. (i.e., it does not currently support advanced programming topics such as lambda expression). ConceptGrader may produce inaccurate concept matching if the student program and reference solution solve a programming problem with different algorithms. We mitigate this by following previous work [47, 2, 41, 99] to include correct students' programs (i.e., student submissions that pass all test cases) as additional reference

solutions.

Internal. Our code and automated scripts may have bugs that can affect our reported results. To mitigate this threat, we have made our tool and data publicly available. Our implementation of the CFG-based approach [97] may not be as effective as the original implementation for C programs. Nevertheless, as our concept graph uses the same CFG as basis for abstraction, our evaluation that compares the CFG approach and the abstracted CFG (our concept graph) ensures fair comparison of the two approaches.

4.8 Conclusion

We propose ConceptGrader, an automated grading approach for programming assignments to assess students' understanding via programming concepts. We derive programming concepts from common programming topics in first-year programming courses, and design a concept graph that abstracts incorrect student program and reference solution. Such an abstract representation allows us to identify students' misunderstanding of a specific problem, so as to generate reasonable scores to reduce tutors' workload and improve students' learning outcomes through convergent formative assessment. Compared to test-based automated grading and CFG-based automated grading, our evaluation shows that the scores generated by ConceptGrader are more accurate in terms of cosine similarity, RMSE, and MAE. Our user study among tutors also shows that the automated generated scores and feedback can help tutors in constructing their manual feedback that eventually assists students in rectifying their mistakes. In the future, we plan to extend ConceptGrader to handle more advanced programming features. We also plan to integrate ConceptGrader into an intelligent tutoring system and deploy it for live interactive programming teaching.

Chapter 5

Design of Intelligent Tutoring System for Programming

5.1 Introduction

In Computer Science (CS) education, we face the challenge of increasing student enrollments over the past few years [86]. Consequently, it has become increasingly difficult to provide high-quality and individualized learning support, particularly for novice students [111, 71]. Mirhosseini et al. [71] recently conducted an interview study with CS instructors to identify their biggest *pain points*. Among other issues, they found that CS instructors struggle with limited or no Teaching Assistant (TA) support and the generally time-consuming task of providing student feedback and grading assignments. Thus, CS instructors would greatly benefit from automating tutoring activities to support TAs in their responsibilities.

Despite prior research [107, 2, 47, 99, 41, 31] from automated program repair and synthesis demonstrating the potential of automated feedback and grading in programming courses, these systems are not yet widely adopted in CS education. The main reasons include their prototype nature, difficulty of use, and lack of evolution. Additionally, this line of work inevitably exposes too many details (i.e., direct fixes of errors) to students, which may hinder the learning process. Conversely, recent advancements in large language models have also advanced automated feedback systems in computer science education [12, 66, 59, 92, 19, 44, 54, 56, 62]. These systems typically focus on prompting large language models (LLMs) for specific teaching scenarios, such as question-answering for lecture topics. However, they heavily rely on LLM output, which is known to be prone to hallucinations and is

not always reliable [12, 44]. Such randomness in the generated feedback can lead to confusion and frustration among students.

In this chapter, we report our design principles and the architecture of an Intelligent Tutoring System (ITS) for programming education that synergizes the strengths of both approaches. The ITS first searches for precise bug-fixing patches with a hybrid program repair engine, and then ITS invokes LLM to use those patches as guidance to pinpoint students' conceptual misunderstandings and provide more reliable feedback. The key is to bridge the gap between accurate low-level fixing by program analysis and knowledgeable high-level explanations by LLMs. Figure 5.1 illustrates the general concept of the Intelligent Tutoring System. The lecturer provides reference programs and test cases as specifications of a programming assignment, and students submit their solutions to the ITS. The ITS then automatically fixes the student's code if it is incorrect and elaborates on the fixes to provide high-level feedback as hints, gradually guiding the student to understand the foundational reason for the error. Additionally, the ITS provides a grading support system that automatically grades student submissions for lecturers.

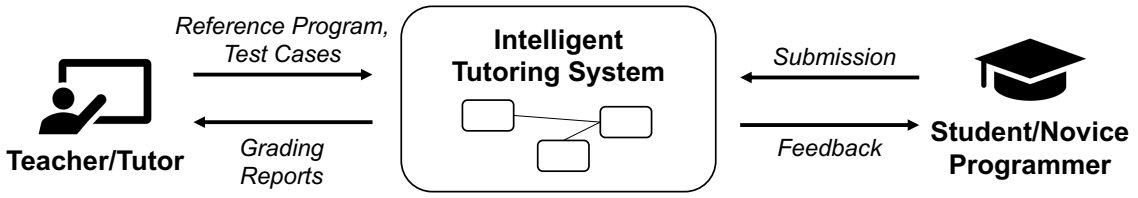


Figure 5.1: General idea of an ITS that supports students and tutors in CS-1 programming courses.

We have integrated the Intelligent Tutoring System into the Coursemology teaching platform of the CS department at the National University of Singapore. We conducted user studies with 15 students from CS1010S Programming Methodology to evaluate the ITS's effectiveness and usability before a live deployment with 571 students. Our user studies and deployment revealed that the current ITS can help students by generating clear feedback regarding precise error location and easy-to-understand conceptual hints.

In summary, we make the following core contributions:

- We present our approach to designing and building an extensible automated feedback system for computer science education.

- We highlight the pathway of linking LLMs with program analysis-based techniques to provide reliable feedback for CS education.
- We share our experience of large-scale deployment of the ITS in a CS-1 programming course with 571 students.

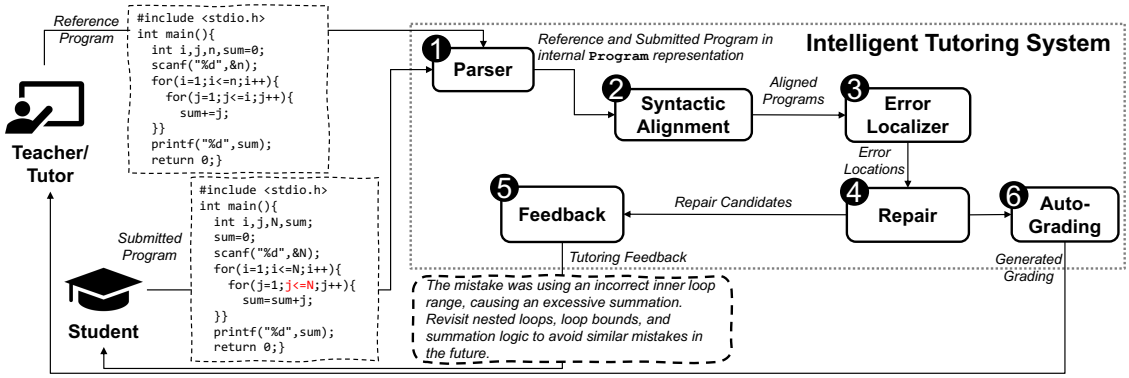


Figure 5.2: Illustrates the general workflow of the Intelligent Tutoring System.

5.2 Intelligent Tutoring System (ITS)

In this section, we introduce the design principles of our Intelligent Tutoring System (ITS) for programming assignments, followed by a detailed presentation of its architecture, key components, and workflow for practitioner adaptation.

5.2.1 Design Principles

To build a practical and up-to-date ITS that can be widely adopted, we adhere to the following three design principles:

- **Language-Independent:** The ITS must be capable of processing multiple programming languages to fit the needs of various CS-1 programming courses. Developing and maintaining a separate ITS for each language is both costly and impractical. To achieve language independence, the ITS should be designed with clean interfaces allowing language-specific plugins or adapters. These plugins handle language-specific syntax and semantics, while the core system manages the general logic of tutoring and feedback. This principle ensures that the ITS can be utilized across diverse programming courses, easyily adapting to curriculum changes.

- **Modular and Extensible:** The ITS needs to be modular to incorporate the unique benefits of various research tools, facilitating maintenance and upgrades. The architecture should feature well-defined interfaces between modules, enabling the addition of new components or the replacement of existing ones without disrupting the overall system. This principle ensures the system's ability to evolve by integrating the latest research findings. For instance, the core repair engine can incorporate new repair strategies, and the feedback generator can be enhanced through new interactions with large language models, such as LLM agent collaboration, making the ITS a future-oriented solution.
- **Scalable:** The ITS needs to be scalable to handle a large number of student submissions and provide feedback in a timely manner without sacrificing usability. Our design of independently operating modules allows the dynamic deployment of scaling methods like load balancing for all components.

Inspired by prior research in program repair [41, 99, 2, 47], we have identified several foundational components essential for the ITS. Figure 5.2 illustrates the detailed architecture and workflow of the ITS. The figure also includes a sample code submission with an incorrect loop condition and the corresponding generated feedback. All components are provided via interfaces, allowing for independent implementation. In the following sections, we introduce each component in detail and explain the workflow.

5.2.2 Language Parser

To support multiple programming languages, we designed an internal intermediate program representation capable of expressing the majority of first-year-level syntax and semantics, such as variable declarations, control structures, and basic data types. This intermediate representation ensures that the core functionalities of the ITS can operate independently of the programming language used. For example, it enables lightweight program analyses, such as control flow, variable usage, and data dependency analysis. As the first step in the workflow, the ITS runs a grammar checker to identify the programming language of the current feedback request. Next, the parser component processes the source code of both the reference program and the student's submission. It invokes the corresponding language-specific parser

to generate the intermediate representation of the programs. This representation standardizes the code into a common format used by other components, which allows the ITS to function consistently across different languages. Currently, the parser component includes specific parsers for C, Java, and Python.

5.2.3 Syntactic Alignment

One key difference between general program repair for large software and program repair for educational purposes is the availability of an expected program specification in the form of a reference implementation. The *Syntactic Alignment* component is designed to align the reference program with the student's submission. It processes the intermediate representations of both the reference and student programs to identify matching basic blocks and map the existing variables for each function within the programs. The alignment algorithms [41, 47, 2] are based on the similarity of control flow and variable usage, specifically using Def-Use Analysis, to compare the reference and student programs. The results of this alignment can then be used to pinpoint the locations where the reference and submitted programs diverge in behavior. Furthermore, this information is instrumental in attempting to repair the submitted program by leveraging the data from the reference program. Note that, the ITS takes in multiple reference solutions with different solving strategies as input which increases the alignment success rate like existing APR tools [2, 47].

5.2.4 Error Localizer and Interpreter

Error localization is a crucial step in APR systems that aims to identify the buggy locations within the software. In the context of computer programming education, error localization identifies specific basic blocks or expressions that violate the expected specifications. The *Error Localizer* component employs several dynamic execution-enabled localization algorithms to trigger erroneous behavior in the student's program. These algorithms include trace-based localization and statistical fault localization [105]. The dynamic program execution is facilitated by an *Interpreter* component. This interpreter allows the execution of a program in its intermediate CFG-based representation without the need for compilation or execution on the actual system. It generates an execution trace with the sequence

of executed basic blocks and a memory object, which holds the variable values at specific locations. The *Error Localizer* component utilizes the *Interpreter* to execute test cases while observing the variable values at specific locations. This process enables the system to detect semantic differences between the reference and submitted programs, thereby pinpointing the precise locations of errors.

5.2.5 Repair Engines

Given the reference programs, student submissions, and the identified error locations as input, the *Repair* component attempts to fix the submitted programs by generating edits that transform the student's program to be semantically equivalent to the reference program. The *Repair* component acts as an engine that can utilize various repair strategies, such as optimization-based repair [41], synthesis-based repair [47, 2], and LLM-based repair [111]. Upon receiving a repair request from the previous components, these repair strategies are invoked in parallel to search for potential repair candidates. The repair engine then selects the optimal repair candidate that minimally alters the student's submission. This approach aims to guide students in correcting their mistakes while preserving their original intentions as much as possible. Note that, the repair candidate is represented at the level of the intermediate representation of the program, and we convert it back to the source code before proceeding to the feedback generation phase.

5.2.6 Feedback Generator

With the collected information from previous components, the *Feedback* component generates natural language explanations to guide students in correcting their mistakes without revealing the direct answer. This component incorporates a common front-end prompt interface with various LLM backends, allowing flexible switching between different LLMs and easy integration of new LLMs. Currently, it supports both commercial LLMs like GPT and Claude series, as well as open-source LLMs like LLaMA [94] from Meta. We use GPT-3.5 as the default LLM backend to balance performance and cost. Our prompt template consists of (1) a task description, (2) the student submission, and (3) program patches from the repair engine annotated with error locations:

You are a teaching assistant for an introductory programming course.

You will be given (1) text description of a programming task (2) a wrong student submission (3) sample fixes to the wrong submission

Based on the sample fixes, please explain to the student conceptually why the mistake exists in this task, and what programming concepts should the students revisit.

Description of the programming task: {description}

Wrong student submission: {student code}

Fixes to the wrong submission: {patches from repair engine}

These prompt ingredients can be seen as a precise hint which replaces the reasoning step in popular Chain-of-Thought [100] prompting. The prompt also instructs the LLM to generate feedback that highlights both assignment-specific mistakes and related general programming concepts. This dual focus helps students understand the underlying issues more comprehensively.

5.2.7 AutoGrader

Test-suite based automated grading suffers from the problem that a small mistake by the student can cause many test cases to fail. To provide better support for tutors, we integrate an auto-grading capability in the ITS, which aims to test the *conceptual* understanding of the student and awards grades accordingly [31]. This is achieved by constructing a concept graph from the student's attempt and comparing it with the concept graph of the instructor's reference solution. The aim is to automatically determine which of the ingredient concepts being tested by the programming assignment are correctly understood by the student. Given the instructor-provided reference solutions and students' incorrect solutions, we apply the abstraction rules to convert students' concrete implementation to conceptual understandings and compare them against the conceptual requirements in reference solutions. Based on the result, the *Auto-Grading* component generates a grading report for the tutor. It assesses the student's submission by their missing or improperly used programming concepts to address the over-penalty issue [31] in the

conventional test-based assessment.

5.3 Pre-Deployment in CS-1 Teaching

We conducted an IRB-approved pre-deployment control experiment and study with 15 students from CS1010S before launching the ITS in a live setting. The self-assessed programming experience of the participants is shown in Figure 5.3. All participants were compensated with a small amount of cash as an incentive for their involvement. The user study was conducted anonymously.

5.3.1 Study Methodology

The students were divided into two groups with balanced levels of programming experience: one group had access to the ITS (group A), while the other did not (group B). User feedback was collected from both groups to assess the potential impact of the ITS on students' learning experiences. Participants were instructed to solve programming tasks using an institution-internal submission system that allowed them to run provided test cases. Each task had a time limit of 20 minutes, and students were allowed to make an unlimited number of submission attempts. Overall, the study was structured in three parts: (1) a background survey, (2) the programming tasks, and (3) a feedback survey. We provided group B with a brief introduction to the ITS *after* they solved their programming tasks, so they can also provide feedback on ITS.

5.3.1.1 Programming Tasks

We have chosen four entry-level programming tasks covering various programming topics. Table 5.1 shows the details of each task and their respective topics. We selected these programming tasks from past mid-term exams of the CS-1 course, which represent the practical challenges students may face.

Table 5.1: Subjects of programming tasks in our experiments

Tasks	**Description**	**Topic**
Remove Extras	Remove duplicates from tuple	For loop, Tuple manipulation
Reverse String I	Iteratively reverse a string	For loop, String manipulation
Reverse String II	Recursively reverse a string	Recursion
Reverse Numbers	Iteratively reverse an integer	While loop

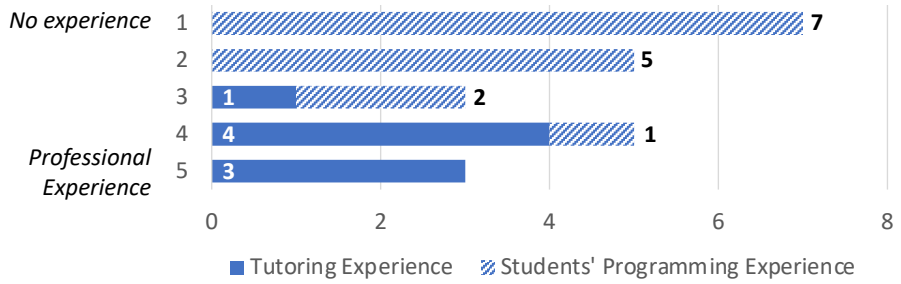


Figure 5.3: Participants' Self-Assessed Experience

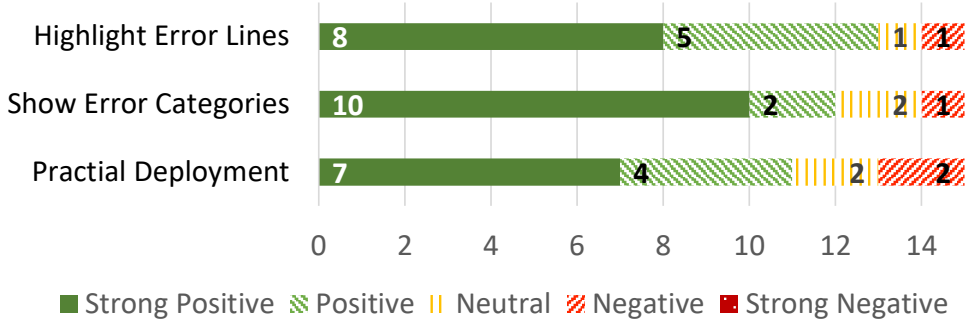


Figure 5.4: Students' feedback of ITS

5.3.2 Result Analysis for Students

We recorded the submitted solutions and their timestamps for each programming task of non-duplicate students' attempts. Students were considered to have *solved* a task if their attempts passed all test cases. In total, we received 128 attempts for the four programming tasks; 65 by Group A and 63 by Group B. For all open-ended questions, we conducted a qualitative content analysis coding [84] that summarizes the themes and opinions.

5.3.2.1 Students' Expectations

Based on the Background (Part 1) survey, we identified the main challenges for novice programmers and their expectations for an ITS. Their general underlying difficulties in learning programming are (1) understanding programming tasks and starting to program, (2) debugging the code and rectifying identified errors, (3) translating their solution strategy into actual code. In addition, we asked the students more specifically about the difficulties the ITS can address. Generally, they confirmed that their main difficulties are with (1) figuring out what goes wrong in the program and (2) finding the error location.

5.3.2.2 Students' Performance

Table 5.2 presents the quantitative results of the students' performances in the two controlled groups. Specifically, we focus on students who failed on their first attempt. The second column represents the average number of students' attempts for each task if their first attempt failed. The third column represents the rectification rate (X/Y) of students who failed to solve a particular task on the first attempt; X represents the number of students who eventually rectified their solutions, and Y represents the number of students who failed to solve a task on the first attempt. The column "*Avg Rectifying Time*" indicates the duration taken by a student to correct an incorrect solution for a programming task.

Table 5.2: The average number of failed attempts, rectification rates, average rectifying time of failed attempts in minutes.

Tasks	**Avg # Failed Attempts**		**Rectification Rate**		**Avg Rect. Time (mins)**	
	A	B	A	B	A	B
Task 1	4.8	4	4/5	0/2	7	-
Task 2	1.9	5.5	7/7	3/4	9.2	9.3
Task 3	2.3	2.8	5/5	2/4	4.6	2.5
Task 4	2.3	3.1	5/6	5/7	4.5	11.3
Total	2.7	3.7	21/23	10/17	6.7	8.9

5.3.2.3 Fewer attempts, higher accuracy

As shown in Table 5.2, students who received assistance from ITS (Group A) solved more programming tasks with fewer attempts compared to students without ITS (Group B). Although Group A made more attempts than Group B for Task 1, it is important to note that the two students in Group B who failed Task 1 could not rectify their solution. Therefore, the fewer average attempts made by Group B may be due to a lack of knowledge on how to fix their solutions after a few attempts, resulting in giving up on the task. On average, Group A made 2.7 failed attempts compared to 3.7 for Group B, indicating that Group A submitted slightly fewer attempts during the experiment. Even though the difference in attempts is not very significant, Group A had a higher success rate in rectifying their solutions; they successfully fixed 21 (91.3%) out of the 23 failed attempts. While Group B had a higher success rate on their first attempt, they struggled more when they failed on their first attempt, only succeeding in fixing 10/17 (58.8%), demonstrating the effectiveness of ITS guidance.

Regarding rectifying time, Group A was faster, with an average of 6.7 minutes to fix one incorrect solution, compared to Group B's average of 8.9 minutes. The average rectifying time for task 1 in Group B is unavailable since no student could rectify their incorrect attempts. Moreover, the average rectifying time for Group B is significantly lower for Task 3 (2.5 minutes) because the two incorrect solutions were almost correct (e.g., typos).

5.3.2.4 Usefulness of ITS

Figure 5.4 shows feedback survey results for students, where we queried their satisfaction with the ITS regarding the usefulness of the features, such as highlighting the potential error lines in the code editor and showing hints about error categories for their mistakes. The results of the questions indicated that most Group A students found the ITS helpful and were satisfied with its feedback and current shape. For example, over 80% of the students responded positively to the usefulness of highlighted lines and mistake categories for their code. Furthermore, over 73% of the students would like the ITS deployed in their programming course. However, we found that one student showed negative feedback toward all questions. This student failed to solve any tasks with correct syntax and struggled to find proper solution strategies. As a result, the ITS could not generate any feedback, as it could not explain the student's intuition at this stage. While this particular experience highlights the limitations of the ITS, the overall positive feedback from the other students supports the potential of ITS in enhancing CS-1 programming education.

5.4 Deployment Experience

In addition to our user studies, we share our experience of deploying the ITS in CS1010S during the fall semester of the 2023-2024 academic year, which involved 571 students. CS1010S covers topics such as recursion, higher-order functions, abstract data types, basic data structures, sorting algorithms, and object-oriented programming through nine weekly assignments. In CS1010S, students submit their solutions into Coursemology, which automatically runs pre-defined test cases for programming assignments. They can make multiple attempts to revise their solutions before the deadline. After the deadline, tutors manually review incorrect submissions

and write personalized feedback to students who submit incorrect solutions.

We integrated our ITS with Coursemology and deployed it for 30 programming tasks spanning six weeks of assignments. We excluded the first two introductory weeks and the final week, which involves OOP that the current ITS does not support. In addition to the traditional assignment workflow, Coursemology invokes the ITS to automatically generate feedback when a student submits a solution that fails test cases. As a first step towards a complete deployment, the current deployment generates feedback visible only to the human tutors to support their manual feedback and grading efforts. Throughout the semester, we kept track of all students' submissions, and eventually, a total of 571 students submitted 3,117 incorrect solutions that failed the test cases of programming assignments in Coursemology. The deployed ITS successfully generated semantically correct patches for 1,758 (56.4%) incorrect submissions by its repair engine. A patch is deemed correct if it makes the original submission pass all test cases for a particular programming assignment. Then the feedback component generated corresponding natural language comments for these submissions. The remaining 43.6% of incorrect submissions that failed to be fixed consist of two main reasons.

Lack of Substantial Content: Some students either submitted empty files or attempted to brute-force the public tests without finding a correct solution strategy. These incomplete submissions lacked the necessary content for the repair engine to generate meaningful repairs to represent students' thinking.

Non-functional Restrictions: Some submitted solutions passed all test cases but violated non-functional requirements not covered by the tests. For example, in an abstract data types assignment, students were required to reuse a specific abstract function. These violations needed manual checking by tutors. Failing to do so resulted in non-functional penalties that the current ITS could not fix.

In the current deployment, we chose to not generate feedback on incorrect submissions that our ITS cannot fix to avoid misleading or incorrect guidance to students. However, failures by the repair engine often imply the student requires additional guidance. It can serve as an indicator to promptly alert tutors about students who need help. Nevertheless, we acknowledge these limitations and discuss possible solutions as future work in Section 5.5. Additionally, we randomly sampled 10% of ITS-generated feedback to manually evaluate the quality of ITS's automated

feedback by assessing whether they were semantically equivalent to the corresponding tutors' feedback. Our manual analysis shows that 136 (77.2%) of ITS-produced feedback is semantically equivalent to tutors' feedback, illustrating its capabilities to assist human tutors. Figure 5.5 shows an example of two feedbacks given by ITS and the tutor for a student's mistake of not handling an edge case in a task that simulates DNA transcript.

```
 1 def find_transcription_region(dna_strand):
 2   if not "TATA" in dna_strand or not "CGCG" in dna_strand:
 3     return None
 4   else:
 5     tata_box_index = dna_strand.find("TATA")
 6     stop_sequence_index = dna_strand.find("CGCG")
 7 +++ if tata_box_index > stop_sequence_index:
 8 +++     return None
 9     return ...
10 -------
11 #ITS's Feedback:
12 Your program does not handle the case where the TATA box appears after the stop sequence CGCG
13 #Tutor's Feedback:
14 [-1] doesn't check if TATA is before CGCG
```

Figure 5.5: Example of two semantic equivalent feedback given by ITS and tutor for missing the edge case at lines 7–8.

5.5 Lessons Learned and Prospects

Despite the promising results of ITS deployment in CS1010S, human tutors remain essential to address specific limitations of the ITS. We acknowledge these limitations and offer insights for future research.

First, the ITS is not capable of generating step-by-step feedback for students who are stuck at the beginning or middle of a task. This is because the APR techniques employed in the repair engine are designed to and cannot fix a partial solution. To address this limitation, we plan to develop APR techniques that can handle finer-grained code changes within incomplete programs. Fortunately, the modular design of the ITS allows us to easily integrate such new APR techniques into the system once available.

Second, the ITS currently does not support feedback for non-functional requirements such as code style, readability, and efficiency. To address this, we consider integrating code style checkers and code quality analyzers into the ITS as additional

components. These components can deliver their analysis results to the feedback engine, which can then utilize the built-in LLM to generate feedback on non-functional requirements.

Finally, the feedback engine in the ITS uses LLMs in a zero-shot manner, which may not fully utilize their capabilities. Recently, LLM-based code agents have shown promising performance in solving real-world software development tasks. We consider exploring computer science education-specific LLM agents to provide more accurate feedback as a future research direction. The program patches generated by the repair engine can still guide these LLM agents, and all other components of the ITS can remain the same.

5.6 Conclusion

In this work, we present the idea of synergizing the strengths of program repair and large language models to create a precise and knowledgeable Intelligent Tutoring System for programming education. We systematically illustrate the architecture and workflow of the ITS to engage practitioners in adapting the system. Our modular design ensures that the ITS can be widely adopted and continue evolving with the latest research breakthroughs. Indeed, the ITS has been delivered to Monash University and also IIT Kanpur. Moreover, our deployment results in CS1010S indicate that the ITS is proficient at providing high-quality instructional feedback similar to human tutors, with 77.2% of the sampled feedback being semantically equivalent to that of human tutors. This demonstrates the potential of the ITS in supporting CS instructors to provide individualized feedback.

The rapid advancements in automated programming and program repair techniques are gradually shifting from manual programming to AI-assisted programming, which may reshape programming education. Consequently, advanced foundational CS courses will become more important, and we envision that future automated tutoring systems need to cover advanced topics such as data structures, algorithms, and database management. We believe that the ITS represents a well-suited platform for future research while serving as a practical tool for current programming education.

Chapter 6

Linking Software Engineering Teaching with Programming Teaching

6.1 Introduction

The remarkable capability in generative AI and computing has attracted unprecedented interest and enrollments in computer science and also interdisciplinary students to learn programming. This poses a significant demand on teaching staff, who must maintain high teaching quality through tasks such as tutorials, recitations, assignment comments, and grading. On the other hand, another typical problem in CS education is the provision of Software Engineering (SE) projects. Software engineering is typically a compulsory course in the university's curriculum for computer science students, and it is often followed or accompanied by development projects, in which students can collect hands-on experience in software development in a team going beyond a programming exercise. Such projects come with inherent difficulties like acquiring industry partners and the dilemma that such software projects are often under- or over-specified. Additionally, such projects are often one-time efforts within one team or one course, and students cannot experience the evolution of a software system.

Recent research [27, 30, 113] has shown that existing LLMs can already help to build a fully autonomous workflow in software development of real-world software projects [113] (e.g., bug-fixing, feature addition), and simple prompt engineering strategies to LLMs can achieve promising results in solving and fixing introductory programming assignments. As automatic programming is gradually becoming a reality, we wonder that in the future, the first-year CS programming teaching might

be replaced, and the educational focus will shift to cultivating a deep understanding of foundational knowledge in advanced computer science courses such as data structure and algorithms, operating systems, database management, and software engineering will become more important than ever. Unfortunately, in contrast to the fast-evolving research advancements in generative AI and the increased public interest, the pace of adjusting computer science education curriculums to fit the GenAI technologies remains slow.

We tackle these two problems (1) scaling programming teaching, (2) innovating SE projects in CS education by building an *Intelligent Tutoring System* (ITS) *with* and *for* students. In chapter 5, we already discussed the design and principles of our *Intelligent Tutoring System* that can provide automated and individual feedback for student code submissions and grading support for tutors and lecturers. In this chapter, we discuss how we assist the education of both novice and senior CS students in the generative AI era through a long-running capstone software engineering project. This foundational software engineering course provides an opportunity for senior CS students to dive deeply into the foundational SE skills with hands-on project experience. On the other hand, the outcome of the long-running project serves as a platform to facilitate the teaching of various computer science courses. As a multi-year research and teaching effort, we combine third-year SE teaching and programming teaching via a long-term, practical, self-sustained software system.

Figure 5.1 in Chapter 5 describes the high-level idea of our ITS that provides automated and individual feedback for student code submissions and grading support for tutors and lecturers. Further, we involve third-year undergraduate students in the incremental development of such a system. We offer various SE projects for the students in our CS3213 - Foundations of Software Engineering course. In CS3213, the students can choose from a wide range of projects, which essentially represent the development or extension of ITS components.

Based on the nature of the overall Intelligent Tutoring System project, we can conduct requirements engineering activities (e.g., surveys, interviews, and user studies) in-house because the various stakeholders are available in the university context. Each student project has the chance to contribute to the overall long-running SE project and eventually impact the learning experience of hundreds of other CS students. In our experience, this creates additional motivation because the

effort is not lost, and they can relate to the users because they (at some point in their studies) also faced the challenges of learning programming.

Based on our experience with around 125 undergraduate students who helped develop the system throughout two years of teaching, the students enjoyed the course project. In particular, they liked the potential reuse of their implementation in the real deployment of the ITS. Additionally, they enjoyed the fact that there is already a system, which they have to extend including the added complexity in understanding the already existing architecture, design, and codebase.

Our course not only impacts the programming courses at the National University of Singapore but also has the potential to impact other universities which adopt a similar teaching concept linking the teaching of software engineering with the teaching of programming such as Monash University. In the future, we plan to conduct more user studies to explore learning success across university boundaries.

We demonstrate the effectiveness of this workflow in a second-year data structure and algorithm course (CS2040S) at the National University of Singapore.

6.2 Design of Software Engineering Course

To achieve our vision of linking software engineering and programming teaching. We collaborated with a new variant of software engineering course, "CS3213 - Foundations of Software Engineering" at the National University of Singapore that provides the students with foundational knowledge and understanding of different aspects of software engineering.

6.2.1 Teaching Concept

Unlike traditional software engineering (SE) courses that teach students basic SE practices through creating similar small-scale one-time effort projects. The teaching concept of CS3213 instead highlights the "*foundation*". The foundation is rooted in two key aspects. The foundation research in SE and the foundation principles of software development. Our goal is to deepen students' understanding of software engineering and practice the already learned principles in a realistic environment of developing an in-house Intelligent Tutoring System. By integrating cutting-edge research with core development practices, we aim to provide senior

SE students with (1) exposure to frontier ideas from the research community, such as fuzzing, debugging, static analysis, and program repair, and (2) an immersive software development environment of contributing to an existing, functional, in-use codebase that allows the students to go beyond programming-in-the-small in the course project.

The development of "Intelligent Tutoring System for Computing Education" especially for programming tasks to handle the increasing enrollments provides a well-framed and particularly interesting scenario for software engineering projects in the university context. This is because (1) the demand for programming tutoring support will exist for a long time, and we can continuously collect feedback from users to curate new requirements. The user feedback can serve as project topics for the next iteration of CS3213. (2) the SE students who contribute to ITS can relate to the end users since they once had to learn programming, and (3) all the related stakeholders are available in the university, which makes the requirement elicitation and milestone discussion become possible.

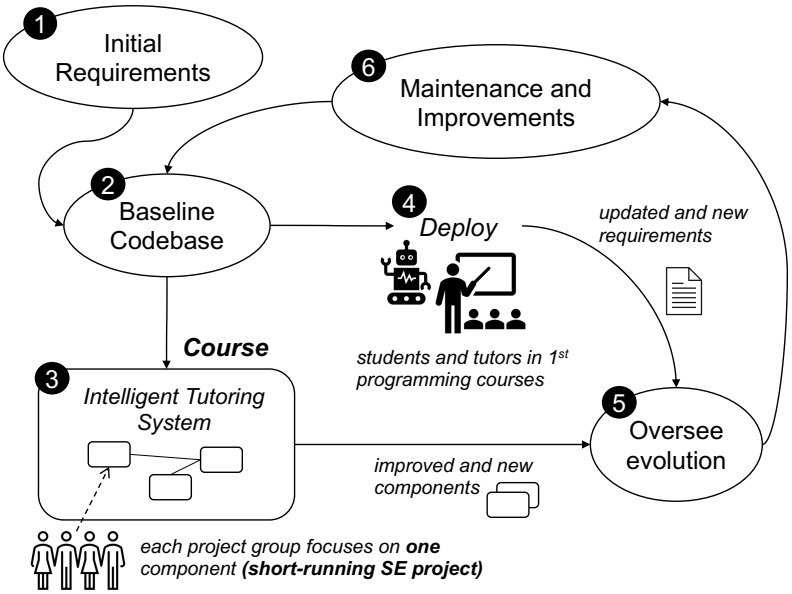


Figure 6.1: Concept of a long-running software engineering project that is incrementally improved by short-running projects inside a teaching environment.

6.2.2 Overview of Long-running Project

Figure 6.1 illustrates the overall diagram of the Intelligent Tutoring System as a self-sustained long-running SE project that evolved over multiple years in CS3213.

Initial Requirement Collections. Every software development project starts with a high-level coarse-grain requirement drafting phase, which is then refined and detailed over time. In our case, before we started any development activities on ITS, we collected initial requirements based on our research experience of automated program repair [2, 31, 47] in educational scenarios and also from a few discussions with the lecturers of some first-year programming courses. At a high level, the initial requirements are oriented to tutors. They mainly consist of highlighting suspicious error code snippets, producing precise code patches, and drafting high-level feedback explanations on behalf of tutors.

Architecture Design. After the initial requirements, we designed the architecture of the Intelligent Tutoring System which is discussed in detail in chapter 5. We also developed an initial codebase, which included the interface definition between each component. Having an initial baseline codebase provides the students with additional requirements like the existing architecture, which should not be changed. On the other hand, it also provides them with existing functionalities similar to a real-world long-running SE project.

Course Projects Setup. For the CS3213 course project (step 3), we design multiple short-running SE projects based on the feedback from first year course instructor in the requirement elicitation session of our course, and these projects essentially representing the implementation variants of existing or new components. For example, in the first year, we mainly had projects to build program analysis capabilities. We further designed projects to extend core features like *Automated Feedback*, *Automated Grading*, *Automated Repair* in the second year. We further designed projects to extend core features like *Automated Feedback* and *Automated Grading* in the second year. based on the existing program analysis artifacts. We discuss those short-running projects in detail in Chapter 6.2.3.

Deployment and Refinement. After each CS3213, our teaching team evaluates all projects and integrates the best contributions of each project topic into our baseline implementation (step 5). Therefore, over the years, the baseline will grow and improve. At the same time, we also deploy the increments of the system in real-world programming courses (e.g., CS1010S, CS2040S), and collect additional feedback and requirements from students and tutors (step 4). To keep the implementation standards high and to ensure that our architecture and design can cope with the increasing codebase and the possibly new and changing requirements, we constantly maintain and improve the implementation (step 6).

Overall, our Intelligent Tutoring System as a long-running SE project course is structured so that the teaching of SE projects is accomplished over multiple years via a real-life SE project. Over the years, the Intelligent Tutoring System became more and more robust, and varied, with the continuous effort of each year's student's contribution. It eventually became a full-fledged functioning automated tutoring system that is being and ready to be widely deployed.

6.2.3 Overview of CS3213 Course Management

6.2.3.1 Course Curriculum

The course curriculum focuses on the main activities in SE. Furthermore, we introduce selected relevant SE topics for our project, e.g., automated program repair, static analysis, and fault localization. Each lecture is separated into two parts: (a) the teaching of foundations in the aforementioned areas, and (b) the teaching of project-specific knowledge and corresponding applications.

Requirements Analysis and Modeling The course starts with a focus on requirements engineering, their elicitation, and modeling. Therefore, we invite stakeholders like lecturers and teaching assistants from the first-year programming courses to an interview session with the third-year students. This interview session is prepared with corresponding assignments about question design and followed up with requirements modeling exercises using UML Use Cases. We also teach other requirements modeling, e.g., with finite state machines and sequence diagrams.

Software Architecture and Design Afterwards, we introduce general principles for software architecture design and modeling. The project-specific part of the lecture introduces the existing architecture and its components, including the available interfaces, which need to be used by the students in their own implementations. We further discuss architecture variants of the existing architecture to discuss pro and contra of the made design decisions.

Our baseline Java implementation already provides the students with elementary classes and functionalities, which they can and need to reuse. To illustrate the fine-grained design, we first introduce relevant design principles and patterns that occur in our implementation. We do not give a comprehensive introduction to design patterns because there is another dedicated software design course in our institution. Instead, we only introduce the most relevant design aspects to enable the students to work on the projects.

Project Planning and Implementation As part of the assignments, the students have to submit a project plan. Therefore, we also introduce the basics of project planning, work package design, and milestone and resource planning, including necessary models like Gantt-Charts. The coding itself is a major part of the project and is mostly supported by the mentors in project-specific meetings. The lecture introduces general principles like Clean Code and testing and debugging techniques meant to help the students in their concrete implementation efforts.

Testing, Debugging, and Integration As automated testing and debugging is a major part of an intelligent tutoring system, we also introduce several validation concepts and debugging techniques. In particular, we teach foundations in test-suite estimation, functional testing, whitebox testing, structural testing, dataflow testing, and mutation testing. To this end, we also introduce the basics of static analysis like control-flow graphs (CFGs) and Define-Use Analysis (DUA). Furthermore, we discuss the basics of debugging with the TRAFFIC principle and delta debugging and dive deeper into the basics of static and dynamic slicing and statistical fault localization. Towards the end of the curriculum, we also discuss integration testing strategies and the related challenges.

Project-Specific Topics In addition to the foundations in general software engineering, we teach the background in automated program repair and provide an overview of existing solutions for ITS components. Depending on the advertised projects, we also discuss more specialized topics like taint analysis and Worst-Case Execution Time (WCET) analysis to ensure the students have the relevant background and material to work on their projects.

Labs and Assignments Each week in our curriculum is accompanied by a lecture and a lab session. The labs are used to meet in smaller groups of students and discuss their assignments. The assignments track the major milestones in the student's projects. We share the course assignment and major milestone in Table 6.1 for practitioners.

Table 6.1: Course assignments that accompanies the major project milestones.

ID	Topic	Details
1	Requirements Analysis & Elicitation	Preparations and questions for the interview session with the customer.
2	Requirements Modeling	Requirement modeling with UML Use Case and Activity diagrams.
3	Architectural Drivers and Architecture Variants	Discussion of architecture variants and the requirements that influence architectural design.
4	Strategy and Project Planning	Project-specific planning including a Gantt-Chart and a resource plan.
5	Detailed Design	Structural and behavioral design of the students' implementation with UML models.
6	Intermediate Deliverable	Towards the middle of the course, we ask the students to submit a minimal project implementation and a report with their project plans and various models.
7	Validation (i.e., Unit Testing)	Test case design and test report.
8	Presentation & Final Artifact	At the end of the course, all teams need to present their project and submit their code.
9	Final Report	After the presentation, the students additionally need to submit a final report, including a retrospective of their project and design decisions.

6.2.3.2 Overview of Short-running Projects

Project Preparation As the key part of the CS3213 course, we carefully curated a set of short-running projects before each semester started. Those short-running projects are not only engineering efforts but also cover different research topics in the Software Engineering community. Table 6.2 shows a few examples of short-running projects that were provided in the first CS3213 course. The topics range from program

structure understanding, static analysis practice, replication of error localization, and automated program repair techniques. In addition, we also prepared a specific testing project that helps students gain fuzzing and mutation testing experience. These short-running projects are inherently different from traditional SE courses that merely focus on development activities. The additional context on SE research exposed students to the fundamental techniques behind software artifacts.

Table 6.2: Example of Short-running Projects provided in the first CS3213

ID	Project	Trained Skills
1	Parsing	Develop an understanding of abstract syntax tree and control flow graph and how a parser in compiler works.
2	Program Alignment	Develop skills in static analysis (e.g., Def-Use) and practice them in aligning two programs.
3	Error Localization	Understand and implement frontier error localization research (e.,g execution-based, statistical-based fault localization.)
4	Repairing Programs	Understand and implement frontier various APR research [2, 41, 99, 47], the approaches include program verification, synthesis, and ILP optimization.
5	Tests Generation	Using fuzzing or mutation testing to generate incorrect programs that test the capability of the whole ITS.

Team Management and Project Guidance To reduce students' workload, we ask the students to form groups of 3-4 people to work on the project. We allow them to search for their team members instead of a random assignment by the teaching team. We prepare an ungraded *Assignment 0* for the project selection, which provides an overview and additional references for all available projects for the specific year. Each team can bid for three projects, while the teaching team allocates the final project.

Additionally, we assigned each team a graduate-level mentor who are familiar with the project topics to help students get started on their project smoothly. Each team was required to meet and discuss weekly with their mentor focusing on the team's planning, design, and implementation progress. Interestingly, those graduate mentors all had experience in tutoring programming courses. So the deep involvement of graduate mentors also works like discussion meetings with stakeholders.

Through the course and project organization, the students advance their skills in software development, grasp a deeper understanding of fundamental SE concepts,

and expose them to frontier SE research. All student projects eventually contribute to an intelligent tutoring system, whose details were discussed in Chapter 5.

Our teaching materials and detailed short-running course project description can be found at: `https://nus-its.github.io/courses/cs3213/`

6.3 Experience of ITS in Data Structures

To demonstrate the effectiveness of the long-running ITS project outcome in practical usage. We present one complex data structure example from the Leetcode platform and our latest experience through an anonymous control experiment with 30 students from CS2040S - data structure and algorithm. Prior experience with ITS in the first-year programming course CS1010S was already discussed in Chapter 5.4.

6.3.1 Demonstration

Figure 6.2(a) presents a data structure task of "Construct Binary Tree from Preorder and Inorder Traversal". Unlike most CS-1 tasks that illustrate a straightforward programming goal. This task is a variation application of teaching the tree traversal strategies. This problem examines a set of data structure definitions and algorithm usage. To solve this problem, students at least need to understand (1) the concept of binary tree and its traversal strategies, (2) the unique tree properties preserved after preorder and inorder tree traversal, and (3) the recursive approach to building a tree.

Figure 6.2(b) shows an example of an incorrect submission from a student who has finished a data structure course. Despite this submission looking correct at first glance, it has an issue with the "build" helper function at lines 14 – 15. Because of the additional condition checker, the code skips the recursion for the root node and leads to an incorrect tree construction. The mistake is subtle in terms of fixing, but it reflects an inadequate understanding of the recursive termination condition. Interestingly, ChatGPT is difficult to confidently detect and explain this mistake, possibly because the code logic is mostly correct.

Repair and Feedback Generation. Driven by the automated program repair engines, the ITS easily identifies this issue and provides a correct patch that fixes

the problem. Moreover, the patch is post-processed by the integrated LLMs to elaborate a detailed explanation as shown in Figure 6.2(c).

Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

(a) Problem Statement of the Tree Traversal Task

```
1  public TreeNode buildTree(int[] preorder, int[] inorder) {
2      if (preorder.length <= 0 || inorder.length <= 0)
3          return null;
4      if (preorder.length != inorder.length)
5          return null;
6      return build(preorder, inorder, 0, inorder.length - 1);
7  }
8
9  public int index = 0;
10 private TreeNode build(int[] preorder, int[] inorder, int low, int high) {
11     if (high < low)
12         return null;
13     TreeNode root = new TreeNode(preorder[index]);
14 -   if (high == low)
15 -       return root;
16
17     int mid = -1;
18     for (int i = low; i <= high; i++) {
19         if (inorder[i] == preorder[index]) {
20             mid = i;
21             break;
22         }
23     }
24     index++;
25     TreeNode left = build(preorder, inorder, low, mid - 1);
26     TreeNode right = build(preorder, inorder, mid + 1, high);
27     root.left = left;
28     root.right = right;
29
30     return root;
31 }
```

(b) One Example of Incorrect Submission

Feedback from ITS

Mistake on Termination Condition: This submission prematurely skips the recursive calls and returns the root node before the subtrees are properly constructed. By returning prematurely, the build function does not explore the entire preorder or inorder arrays, leading to an incomplete tree construction.

(c) Example Feedback for Figure 6.2(b) Curated by ITS

Figure 6.2: Example of Data Structure Example from Leetcode.

6.3.2 User Study in CS2040S

CS2040S Data Structures and Algorithms is a follow-up course to the first-year programming course (CS1010S) at the National University of Singapore (NUS) School of Computing. The course introduces students to the fundamental concepts of data structures and algorithms using Java. CS2040S includes topics such as basic data structures (e.g., arrays, linked lists, stacks, queues, trees, graphs), sorting and searching algorithms, and algorithm analysis. We conducted a controlled experiment of solving four LeetCode tasks among 30 students at the end of the semester to evaluate ITS's impact on their learning experience. We selected two programming tasks relevant to Tree and two programming tasks relevant to Graph, which are two very important data structure topics, both tasks are taken from LeetCode with similar difficulty and the required programming concepts are closely related to weekly problem sets. The detailed four tasks are presented in Figure 6.3.

In the control experiment, we equally divided the 30 student participants into two groups based on their programming expertise. Group A participants have access to ITS feedback for Task 1 and 3 (after finalizing submissions), whereas Group B participants do not have access to ITS feedback for all four tasks. The goal is to confirm our hypothesis that the students in Group A should perform better than students in Group B for Task 2 and Task 4 even without ITS. *If the feedback they have received on Task 1 and Task 3 has strengthened their conceptual understanding of the topic.* The students were given 25 minutes to solve each task, after which they should move to the next task.

Tree Topic
We assess students' understanding of **Binary Search Tree definition and Pre/In/Post Tree Traversal**, which is covered in Problem Set 4 of CS2040S. Task 1: "Construct Binary Tree from Preorder and Inorder Traversal" Task 2: "Construct Binary Tree from Inorder and Postorder Traversal"

Graph Topic:
We assess students' understanding of the **Shortest-Path Problem in Graph**, which is covered in Problem Set 8 of CS2040S. Task 3: "Minimum Cost to Reach City With Discounts" Task 4: "Minimum Cost to Buy Apples"

Figure 6.3: Overview of Data Structure Tasks used in Control Experiment.

Experiment Result Figure 6.4 reports the number of final correct submissions for each task. For Task 1 and Task 3, group A and Group B both do not have access to ITS feedback at the beginning, and the number of correct submissions is thus similar in both groups (Task 1: 6 vs 5, Task 3: 3 vs 4). During the gap between Task 1, 2 and Task 3, 4, we presented the ITS feedback to group A students whose final submissions were incorrect. This led to a substantial increase in the number of correct submissions for Task 2 and Task 4 in Group A compared to Group B. For Group A, 4 out of 9 students who failed Task 1 successfully passed Task 2, and 5 out of 12 students who failed Task 3 successfully passed Task 4. For Group B, only 2 out of 10 students who failed Task 1 successfully passed Task 2, and only 2 out of 11 students who failed Task 3 successfully passed Task 4.

Despite there might be some students solved follow-up tasks because they got additional time to work on a similar task, the improvement difference between Group A and Group B indicates the positive impact of ITS feedback on students' learning experience.

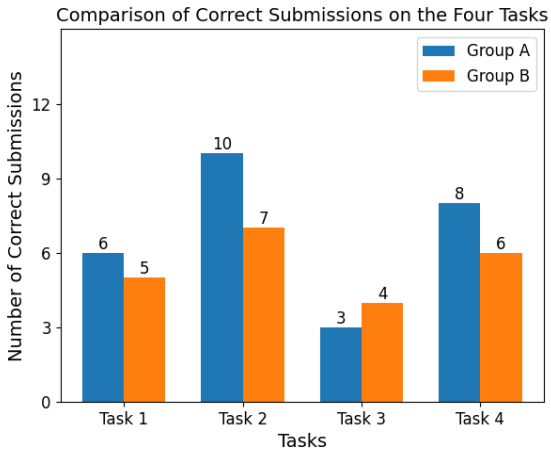


Figure 6.4: Comparison of Correct Submissions on the Four Tasks.

Case Study. We present one case study of a Group B student who was almost correct but still failed Task 1 and Task 2. The student's submission for Task 1 is explained in Figure 6.2(b), and we show the student's submissions for Task 2 in Figure 6.5. The student submitted 8 attempts for task 1 and reached an almost correct solution which only needs a last-mile repair to remove the redundant condition if (high == low) in the build helper method. Ideally, the student is expected to solve

task 2 in the additional 25 minutes since the two tasks are isomorphic. However, the student started with a task 1 identical solution in task 2 and later submitted 13 attempts, and none of them touched the erroneous condition. This example further demonstrates how struggling students can be even if they are close to the correct solution and highlights the importance of immediate feedback. Interestingly, after we presented the feedback in Figure 6.2(c), the student immediately realized his mistake and successfully rectified the error in both Task 1 and Task 2.

```
1 # Example Student Submission for Task 2 (Incorrect)
2 public TreeNode buildTree(int[] inorder, int[] postorder) {
3     if (inorder.length <= 0 || postorder.length <= 0 || inorder.length != postorder.length) {
4         return null;
5     }
6
7     if (inorder.length == 1)
8         return new TreeNode(inorder[0]);
9
10    return build(postorder, inorder, 0, inorder.length - 1);
11 }
12
13 public int index = 0;
14 private TreeNode build(int[] postorder, int[] inorder, int low, int high) {
15    if (high < low || curr < 0)
16        return null;
17
18    TreeNode root = new TreeNode(postorder[index]);
19
20    if (high == low)
21        return root;
22
23    int mid = -1;
24    for (int i = low; i <= high; i++) {
25        if (inorder[i] == postorder[index]) {
26            mid = i;
27            break;
28        }
29    }
30    index++;
31    TreeNode left = build(postorder, inorder, mid + 1, high);
32    TreeNode right = build(postorder, inorder, low, mid - 1);
33    root.left = left;
34    root.right = right;
35
36    return root;
37 }
```

Figure 6.5: Case Study of an Example Incorrect Submission from Group B Participant for Task 2.

6.4 Challenges & Lessons Learned

To further share our experience with our combined research and teaching effort, we report the challenges we faced and the lessons learned concerning the teaching of

CS3213 - Foundations of Software Engineering.

6.4.1 Incentives for Stakeholders

We have three main user groups: the students who receive feedback, the tutors who can use the ITS to better understand the students' errors and get grading support, and the lecturers who provide the inputs like assignments and reference implementations. *Lecturers* are naturally concerned about deploying more tools, including the potential negative effects on the learning outcome caused by inaccurate output. To gradually convince the lecturers, we decided to first focus on a targeted deployment for tutors. For tutors, an imperfect output is less critical and still can provide helpful guidance to them and help us to get feedback continuously. In contrast to the lecturers, the *tutors* have a generally more positive attitude regarding the ITS; they are willing to join longer interviews to share their experience in the tutoring process and their requirements. As a result, we have been able to successfully invite tutors to our requirements elicitation sessions as well as to our user studies. To engage with *first-year students*, we designed a user study that not only has a monetary reimbursement but also provides additional programming training and an extra tutorial after the user study to explain the programming tasks to them individually. The *third-year students* who develop the components in our course showed great interest in our project because it is (or will be) deployed in a real context and because they like working on a larger project with existing parts. Overall, it is a valuable experience for them, as shown by the following student quotes about the question of what they liked the most in the course:

"As the module is new, its content to be taught may change but I'm certain the ITS project is here to stay. (iterative building of the system)."

"The project component – It's really interesting, and I like that it will actually be used. I think that makes it one of the most interesting modules I've taken so far. It's very cool to understand the reasoning for design details with the teaching team that actually built it."

"Participation in an actual to–be–deployed software project is exciting and makes your effort somewhat worthwhile."

6.4.2 Project Preferences

In the first instance of our course, we allowed students to pick projects on their own. Therefore, we ended up with an imbalanced selection of projects. Students tended to prefer a project with more explicit requirements, e.g., a *Parser* component, instead of a *Repair* project that involves more research. In the second instance, we therefore only allowed bidding on projects while the teaching team made the final decision.

6.4.3 Managing Software Evolution

Overall, we experienced that our general approach is feasible and helps both the third-year and the first-year students. However, we have also seen that we must invest significant time from our side in managing the software evolution. This includes selecting and integrating the best projects, maintaining the code base, updating the design to cater to new requirements, and implementing new components to check their feasibility before we can offer them as a project in the course.

6.5 Impact and Vision for the future

In this chapter, we presented our concept for linking the teaching of software engineering projects with the teaching of programming and introduced our intelligent tutoring system (ITS). Further, we discussed our experiences using the ITS in CS2040S through a controlled experiment. In the following two sections, we discuss the observed *impact* of our work and provide a concluding *outlook* for intelligent tutoring in the AI era.

6.5.1 Impact: Teachers, Students, Research

Based on our experience, the presented ITS impacts several aspects of programming. With our long-running teaching effort, we incrementally develop and improve the ITS into a usable product. We change how first-year *students* learn programming and support *teachers* in the introductive CS courses. Furthermore, we provide the platform for senior *students* to practice software engineering in a realistic scenario. Additionally, they are encouraged to work on research-oriented topics by selecting

the corresponding projects. Overall, we received positive feedback in our user studies: from the 15 students and tutors, more than 78% would like to see the ITS deployed in their next programming course. Moreover, the ITS helps to integrate the latest *research* in educational APR and related topics. Our teaching innovation can also impact students from other universities as they adopt our concept and join the ITS development team. In fact, we have already exported the teaching concept to another university.

6.5.2 Intelligent Tutoring in AI Era

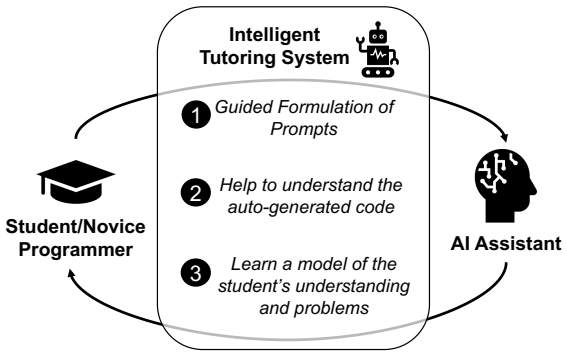


Figure 6.6: Envisioned three-way interaction between Student, ITS, and AI Assistant.

With the shift from manual programming to AI-assisted programming, CS education must also be innovated. We think the ITS represents a well-suited platform to help students learn an effective way of using AI-based code generation tools like GitHub Copilot and ChatGPT. Therefore, instead of exposing the student directly to the AI assistant, the ITS can *moderate* the prompts and explain the generated code, achieving a three-way interaction between the student, ITS, and AI assistant (see Figure 6.6). Based on the student's performance, mistakes, and interaction with the AI assistant, the ITS can learn a model of the student's current mental model. This can be achieved by mapping the student's mistakes and questions to the underlying programming concepts.

Chapter 7

Related Work

This chapter discusses the existing work related to this thesis in the research area of program repair, programming feedback generation, program equivalence checking, and capstone software engineering projects.

7.1 Automated Program Repair

Automated Program Repair (APR) [38, 72] is an enabling technology which allows for the automated fixing of observable program errors thereby relieving the burden of the programmers. General purpose APR techniques such as GenProg [58], SemFix [75], Prophet [64] and Angelix [69], require an incomplete correctness specification typically in the form of a test-suite.

7.1.1 Test-based Program Repair

Test-suite based automated program repair (APR) has attracted significant attention in the last decade [38]. These techniques aim to generate a patch for a buggy program to pass a given test-suite. APR techniques typically include search-based, semantic-based, and pattern/learning-based APR. Search-based APR techniques like GenProg [58] take a buggy program and generate patches using predefined code mutation operators, or search for a patch over the patch space that passes the given test suite. Semantics-based APR techniques [75, 69] generate patches by formulating a repair constraint that needs to be satisfied based on a given test suite specification, and then solving the repair constraint to generate patches. Learning-based APR techniques [65, 50, 115] often train a deep learning model with large code repositories and are guided by a specific representation of code syntax and semantics to predict the next tokens that are most likely to be correct patch.

APR techniques have been successfully deployed in industries for domain-specific bug fixing [104, 67, 11]. However, these techniques achieve low repair success rate on student programs that suffer from multiple mistakes, since they can scale to large programs but not necessarily to large repair search spaces [107]. As student programs are substantially incorrect, the search space for repairs is typically large.

ITSP [107] reports positive results on deploying general APR tools for grading purpose by expert programmers, and negative result when used by novice programmers for feedback. Their low repair success rate and reliance on test-cases (overfitting) can be seen as a motivator for our work.

S3 [57] synthesizes a program using generic grammar and user-defined test-cases. Semgraft [68] uses simultaneous symbolic execution on a buggy program and a reference program to find a repair, which makes the two program equivalent for a group of test inputs; this class of test inputs is captured by a user-provided input condition. Our work shifts away from test inputs and instead constructs verification guided repair. Furthermore, for our application domain of pedagogy, we seek to build minimal repairs by retaining as much of the buggy program as possible.

7.1.2 Program Repair of Programming Assignments

Autograder [87] is one of the early approaches in this domain. In Autograder, the correctness of generated patches is verified only in bounded domains (e.g., the size of a list in the program is bounded to a constant number), and thus the verification result is generally unsound. Autograder also requires instructors to manually provide an error model that specifies common correction patterns of student mistakes, which is not needed in Verifix.

Clara [41] also performs bounded unsound verification. Clara checks whether each concrete execution trace of the student program matches that of the reference program, and performs a repair on mismatch. Since a concrete execution trace is obtained from test execution, the correctness of a generated patch cannot be guaranteed. We have provided a detailed experimental comparison with Clara. Clara assumes the availability of multiple correct student submissions with matching control-flow to the incorrect submissions, limiting their applicability, unlike Verifix. Sarfgen [99] generates patches based on a lightweight syntax-based approach and

assumes the availability of previous student submissions. Both Clara and SarfGen require strict Control-Flow Graph (CFG) similarity between the student and reference program. In comparison, Verifix requires a matching function and loop structure between the student and reference program. Unlike Clara and SarfGen, Verifix can recover from differences in return/break/continue edge transitions due to its usage of Control-Flow Automata (CFA) based abstraction.

Refactory [47] handles the CFG differences by mutating the CFG of the student program to that of the reference program by using a limited set of semantics-preserving refactoring rules, designed manually. For example, refactoring a while-loop by replacing it with a for-loop structure. Note that Verifix, unlike Refactory, keeps the original CFG of the student program as much as possible, as shown in Fig 3.1. Our goal is to produce small feedback of high quality. We cannot experimentally compare with Refactory since its implementation targets Python programming assignments.

CoderAssist [53], to the best of our knowledge, is the only APR approach that can generate verified feedback. CoderAssist clusters submissions based on their solution strategy followed by manual identification (or creation) of correct reference solutions in each cluster. After the clustering phase, CoderAssist undertakes repair at the contract granularity rather than expression granularity — that is, while CoderAssist can suggest which pre-/post-condition should be met for a code block, CoderAssist does not have the capacity to suggest a concrete expression-level patch. CoderAssist repair algorithm and evaluation results focus on dynamic programming assignments. In contrast, Verifix is designed and evaluated as a general-purpose APR.

There have been several attempts to use neural networks [15, 98, 42, 79, 21, 3] for program repair. These approaches typically target syntactic/compilation errors, and the repair rate for semantic/logical errors is low [79]. Such machine learning based techniques do not offer any relative completeness guarantees, and the repair is evaluated against incomplete specification (e.g. tests).

There has been prior work on live deployment of APR tools for repairing student programs [107, 4]. The work of ITSP [107] shows negative results on providing semantic repair feedback to students on their programs. At the same time, the work of Tracer [4] demonstrates positive results for repair based feedback, albeit on

simpler (compilation) errors. In this thesis, we present an approach for repairing complex logical errors in student programs. Our tool Verifix can generate verified feedback for 58.4% of incorrect student submissions from 28 diverse assignments, collected from an actual CS-1 course offering. The human acceptability of our verified feedback can be further investigated via future user-studies.

7.1.3 Program Equivalence Verification

Verifix performs program equivalence verification which itself is a separate long-standing research area [13, 110, 22, 74]. In program equivalence verification, it is proved whether given two programs are semantically equivalent to each other. Program equivalence verification is usually performed by first constructing a product program (similar to our aligned CFA) where the loops of the two programs are aligned with each other [13, 110]. Aligning loops is considered as one of the major challenges in program equivalence verification [22]. In the traditional application areas of program equivalence verification such as optimized-code verification [74, 23], the original code and its optimized code often have different program structures, and thus alignment is challenging in those programs. This problem is much less severe in introductory programming assignments, as shown in our experiments, where Verifix fails to obtain a repair due to structural mismatch between the student and reference program in 27.2% of our dataset. The main difference of our work from program equivalence verification is that we add a CEGIS (counter-example-guided inductive synthesis) loop inside the verification procedure, so that repair and verification can take place hand in hand.

7.2 Automated Grading.

Many approaches have been proposed for automatic assessment of programming assignments [7, 102, 96, 52, 46, 63]. These systems rely either on (1) test cases [48, 7, 102, 96, 52, 46], (2) formal semantics [63], or (3) syntactic differences (e.g., in the form of CFG) between reference solution and student solution [70, 6, 97] to grade introductory programming assignments. Test-based grading approaches (e.g., AutoGrader [7]) assign scores to programming assignments by relying on program's outputs on a set of test inputs [102, 96, 52]. These test inputs can either

be manually designed by course instructors or automatically generated [36, 63]. These test-based approaches have several limitations, including: (1) they cannot reflect the students' effort and mastery of knowledge because a minor mistake could fail many test cases which lead to a large portion of marks being deducted, (2) students may struggle to identify their mistake using only the failing test cases as feedback. Different from these approaches, ConceptGrader grades a student's submission using programming concepts, and generates intuitive feedback which points out the missing or wrongly used programming concepts. Another popular approach to automated grading is calculating the similarity between different program representations (e.g., control flow graph) of a student's submission and corresponding reference implementation [70, 6, 97]. However, these approaches do not support convergent formative assessment as they only grade student's submissions without providing feedback. Liu et al. [63] proposed an approach based on formal semantics for automated grading of programming assignments. They use symbolic execution techniques to explore the semantic difference between instructor's reference solution and students solution in the form of path deviations. However, their approach only produces a binary correct or incorrect result, while our approach gives a quantitative evaluation of student submissions.

Prior work focuses on fixing introductory programming assignments using automated program repair techniques, and providing the automatically patches as feedback [107, 88, 41, 99, 47, 53]. Although ConceptGrader uses patches generated by a program repair engine (Refactory [47]) to trigger the unfolding mechanism to improve the accuracy of the score calculation for incorrect student submissions, ConceptGrader does not use the automatically generated patches directly as feedback. As we design the abstraction rules to be human readable, the feedback generated by ConceptGrader can provide explanation of student mistakes to support convergent formative assessment. Different from existing repair approaches, ConceptGrader is designed for automated grading of introductory programming assignments.

7.3 Capstone Software Engineering Projects

Project-based software engineering courses are essential for students to get training for professional software development skills like architecture design, team

management, software maintenance, etc. Students are often required to work as a team to develop software either from industrial partners or simulated real-world topics via semester-long projects [51, 90, 17, 93, 26, 37]. However, there exist certain barriers and challenges to this teaching setting. For example, continuously collecting project topics from industry partners and establishing an efficient communication channel between stakeholders (students and company clients) are challenging tasks for the instructor. More importantly, the students work on different project topics each year, which means they usually do not have a general picture of the entire system, therefore they cannot experience the evolution of a software system.

In this thesis, our focus is presenting the idea of having an in-house, long-running, sustainable software engineering project in the university context. This kind of long-running SE project shares characteristics with other community-driven course concepts [10]. Our proposed teaching concept is however novel in the sense that it links the teaching of software engineering courses and the teaching of introductory programming courses. This is done by developing an intelligent tutoring system. Students not only get training for software development but also gain exposure to the latest research in the software engineering community.

Chapter 8

Conclusion

Automated programming tutoring system aims to provide constructive suggestions on students' programming mistakes and precisely assess students' effort in an automated way. To achieve this vision, prior work utilized program repair / synthesis and symbolic analysis to automatically fix students' assignments and use the corresponding patch as the feedback returned to students. Although these techniques have shown high accuracy in fixing incorrect student submissions, they use test suites as incomplete specifications and suffer from overfitting issues and the assessment does not always reflect student's conceptual understanding. Moreover, these works are not being used or maintained because of the nature of the research prototype. In this thesis, we propose a series of techniques to scale the intelligent tutoring of programming. Those proposed techniques all revolve around the topic of automated program repair. Overall, we assist teaching staff and students in the Computer Science department by providing guaranteed repair for programming assignments, conceptual level automated grading, building a full-fledged Intelligent Tutoring System (ITS), and a self-sustained Software Engineering course that continuously evolves the ITS. The contributions of this thesis are summarized as follows.

8.1 Summary of Contributions

- We propose Verifix, which uses program equivalence checking techniques along with pMaxSMT-based automated program repair to provide a trustworthy guarantee as well as high-quality, minimal program patches as feedback for students' programming assignments. Our evaluation of 341 real-world incorrect solutions from students shows significant improvement compared to prior work. (Chapter 3)

- We propose ConceptGrader, which provides conceptual-level feedback and precise assessment for programming assignments. ConceptGrader is based on concept graph, an abstracted CFG that highlights programming concepts in submissions of introductory programming assignments. The concept graph contains expressions translated into natural language to enhance readability, and make it more suitable as hints to provide feedback to students. To allow more abstract matching of programs, we introduce concept node fold- ing where we temporarily hide complex expressions in concept nodes for fuzzy concept matching, and unfold (unhide) the expressions for precise concept matching whenever we detect a likely programming mistake within the folded concept node. (Chapter 4)

- We designed and implemented a full-fledged, ready-to-use Intelligent Tutoring System for Programming, and deployed it in the programming courses (e.g., CS1010S and CS2040S) at NUS. This development process involves the iteration of improving the teaching of software engineering by providing a realistic, self-sustained software engineering project in the university context, and also an improved ITS is ready to provide personalized feedback that facilitates students' learning of programming. (Chapter 5 and Chapter 6)

8.2 Future Work

Although program repair has proved its effectiveness in educational settings, especially for first-year programming courses. There is a long journey toward widely adopting those techniques in everyday teaching. We believe there are still plenty of opportunities and challenges for aiding computing education. Reflecting on our findings and observations throughout this thesis, we envision the following research directions.

Practical Assessment on Learning Outcomes. The ultimate goal of having an intelligent tutoring system is not to merely repair students' mistakes but to enhance student's learning outcomes. In our preliminary investigation through both user study and live deployment, we have seen our Intelligent Tutoring System is

effective, but the evidence we have shown is mainly technical perspective. To push a wider integration of the Intelligent Tutoring System, we believe further study is required to fairly assess the Intelligent Tutoring System's long-term impact on student learning outcomes from a pedagogy perspective. For instance, establishing a controlled experiment environment to observe student's behavior and performance with/without ITS can help us gain further insights into the both positive and potential negative impacts in real-pedagogy scenarios.

Early-staged Assisting Mechanism Even though my thesis shows promising results in providing last-mile repair as feedback to students, there are still many students who cannot benefit. One of the other key challenges for students who struggle with learning programming is how to correctly interpret the problem and start the thinking process. We believe a proactive repair approach that accompanies step-by-step problem-solving guidance from scratch is desired for those relatively weaker students to get familiar with programming fast. This would enable ITS to help more characteristics of student's difficulties and increase its applicability.

Application in Advanced Computing Courses. The rapid development of generative AI powered automatic programming might reduce the importance of programming activities in those entry-level programming courses. These paradigm changes may indicate that the fundamental concept in advanced computing courses will always stay and become more important for Computer Science students in the GenAI era. We believe that the idea of program repair can be adapted with the knowledge within large language models to better facilitate advanced computing topics such as algorithms, databases, operating systems, etc.

In conclusion, this thesis not only attempts to address the trustworthy issue in program assignment repair and conceptual assessment, but also opens many opportunities for future research works. The Intelligent Tutoring System built in this thesis also serves as a well-established platform for conducting and integrating future research to enrich its trustworthiness, generalizability, and applicability for wider populations in learning computing.

Publication Appeared

[1] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–31, 2022.

[2] Z. Fan, S. H. Tan, and A. Roychoudhury, "Concept-based automated grading of cs-1 programming assignments", in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 199–210.

[3] Z. Fan, Y. Noller, A. Dandekar, and A. Roychoudhury, "Intelligent tutoring system: Experience of linking software engineering and programming teaching", *arXiv preprint arXiv:2310.05472*, 2023.

[4] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models", in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 1469–1481.

[5] Z. Fan, H. Ruan, S. Mechtaev, and A. Roychoudhury, "Oracle-guided program selection from large language models", in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 628–640.

Note that, the research presented in paper [1] was done as a joint first-author with Umair Z. Ahmed, and the CS3213 teaching activities presented in [3] were organized by Yannic Noller.

Bibliography

[1] R. Abreu, P. Zoeteweij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization", in *Testing: Academic and industrial conference practice and research techniques-MUTATION (TAICPART-MUTATION 2007)*, IEEE, 2007, pp. 89–98.

[2] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury, "Verifix: Verified repair of programming assignments", *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 4, pp. 1–31, 2022.

[3] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani, "Compilation error repair: For the student programs, from the student programs", in *Proceedings of the 40th International Conference on Software Engineering: Software Engineering Education and Training*, 2018, pp. 78–87.

[4] U. Z. Ahmed, N. Srivastava, R. Sindhgatta, and A. Karkare, "Characterizing the pedagogical benefits of adaptive feedback for compilation errors by novice programmers", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering Education and Training (ICSE)*, 2020, pp. 139–150.

[5] T. Ahoniemi and T. Reinikainen, "Aloha-a grading tool for semi-automatic assessment of mass programming courses", in *Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006*, 2006, pp. 139–140.

[6] K. Ala-Mutka, T. Uimonen, and H.-M. Jarvinen, "Supporting students in c++ programming courses with automatic program style assessment", *Journal of Information Technology Education: Research*, vol. 3, no. 1, pp. 245–262, 2004.

[7] C. S. for ALL Students, "Autogradr", `https://www.csforall.org/members/autogradr_automated_grading_for_programming_assignments/`, Accessed: 2020-10-06, 2022.

[8] R. Alur, R. Bodik, G. Juniwal, M. M. K. Martin, M. Raghothaman, S. A. Seshia, R. Singh, A. Solar-Lezama, E. Torlak, and A. Udupa, "Syntax-guided synthesis", in *Proceedings of the International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, Oct. 2013, pp. 1–17.

[9] N. d. C. Alves, C. G. von Wangenheim, J. C. R. Hauck, and A. F. Borgatto, "A large-scale evaluation of a rubric for the automatic assessment of algorithms and programming concepts", in *Proceedings of the 51st ACM technical symposium on computer science education*, 2020, pp. 556–562.

[10] B. Anderson, M. Henz, and K.-L. Low, "Community-driven course and tool development for cs1", in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023, Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 834–840, ISBN: 9781450394314. [Online]. Available: https://doi.org/10.1145/3545945.3569740.

[11] J. Bader, A. Scott, M. Pradel, and S. Chandra, "Getafix: Learning to fix bugs automatically", *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–27, 2019.

[12] R. Balse, B. Valaboju, S. Singhal, J. M. Warriem, and P. Prasad, "Investigating the potential of gpt-3 in providing feedback for programming assessments", in *Proceedings of the 2023 Conference on Innovation and Technology in Computer Science Education V. 1*, 2023, pp. 292–298.

[13] G. Barthe, J. M. Crespo, and C. Kunz, "Relational verification using product programs", in *International Symposium on Formal Methods*, Springer, 2011, pp. 200–214.

[14] B. Bell, N. Bell, and B. Cowie, "Formative assessment and science education", Springer Science & Business Media, 2001, vol. 12.

[15] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments", in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, IEEE, 2018, pp. 60–70.

[16] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain", in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1032–1043.

[17] B. Bruegge, S. Krusche, and L. Alperowitz, "Software engineering project courses with industrial clients", *ACM Transactions on Computing Education (TOCE)*, vol. 15, no. 4, pp. 1–31, 2015.

[18] H. Bunke, P. Foggia, C. Guidobaldi, C. Sansone, and M. Vento, "A comparison of algorithms for maximum common subgraph on randomly connected graphs", in *Joint IAPR International Workshops on Statistical Techniques in Pattern Recognition (SPR) and Structural and Syntactic Pattern Recognition (SSPR)*, Springer, 2002, pp. 123–132.

[19] D. Cambaz and X. Zhang, "Use of ai-driven code generation models in teaching and learning programming: A systematic literature review", in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 172–178.

[20] Z. Chen, S. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "Sequencer: Sequence-to-sequence learning for end-to-end program repair", *IEEE Transactions on Software Engineering*, vol. 47, no. 9, pp. 1943–1959, 2019.

[21] D. Chhatbar, U. Z. Ahmed, and P. Kar, "Macer: A modular framework for accelerated compilation error repair", in *International Conference on Artificial Intelligence in Education*, Springer, 2020, pp. 106–117.

[22] B. Churchill, O. Padon, R. Sharma, and A. Aiken, "Semantic program alignment for equivalence checking", in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2019, pp. 1027–1040.

[23] B. Churchill, R. Sharma, J. Bastien, and A. Aiken, "Sound loop superoptimization for google native client", *ACM SIGPLAN Notices*, vol. 52, no. 4, pp. 313–326, 2017.

[24] coetaur0, "Python3 control flow graph generator", `https://github.com/coetaur0/staticfg`, Accessed: 2022-10-09, 2019.

[25] S. A. Cook, "Soundness and completeness of an axiom system for program verification", *SIAM Journal on Computing*, vol. 7, no. 1, pp. 70–90, 1978.

[26] D. Delgado, A. Velasco, J. Aponte, and A. Marcus, "Evolving a project-based software engineering course: A case study", in *2017 IEEE 30th Conference on Software Engineering Education and Training (CSEE&T)*, IEEE, 2017, pp. 77–86.

[27] P. Denny, J. Prather, B. A. Becker, J. Finnie-Ansley, A. Hellas, J. Leinonen, A. Luxton-Reilly, B. N. Reeves, E. A. Santos, and S. Sarsa, "Computing education in the era of generative ai", *Commun. ACM*, vol. 67, no. 2, pp. 56–67, Jan. 2024, ISSN: 0001-0782. [Online]. Available: https://doi.org/10.1145/3624720.

[28] B. EECS, "Cs 9h: Python for programmers", https://selfpaced.bitbucket.io/#/python/calendar.

[29] S. Engineering, "Cs106a - programming methodology", https://web.stanford.edu/class/archive/cs/cs106a/cs106a.1206/schedule.html.

[30] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan, "Automated repair of programs from large language models", in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, IEEE, 2023, pp. 1469–1481.

[31] Z. Fan, S. H. Tan, and A. Roychoudhury, "Concept-based automated grading of cs-1 programming assignments", in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023, pp. 199–210.

[32] P. Fontaine, M. Ogawa, T. Sturm, X. T. Vu, *et al.*, "Wrapping computer algebra is surprisingly successful for non-linear SMT", in *SC-square 2018-Third International Workshop on Satisfiability Checking and Symbolic Computation*, 2018.

[33] P. S. Foundation, "Abstract syntax trees", https://docs.python.org/3/library/ast.html, 2022.

[34] J. Fowkes, P. Chanthirasegaran, R. Ranca, M. Allamanis, M. Lapata, and C. Sutton, "Autofolding for source code summarization", *IEEE Transactions on Software Engineering*, vol. 43, no. 12, pp. 1095–1109, 2017.

[35] X. Gao, S. Mechtaev, and A. Roychoudhury, "Crash-avoiding program repair", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 8–18.

[36] L. Gong, "Auto-grading dynamic programming language assignments", *University of California, Berkeley, Tech. Rep*, 2014.

[37] A. Goold and P. Horan, "Foundation software engineering practices for capstone projects and beyond", in *Proceedings 15th Conference on Software Engineering Education and Training (CSEE&T 2002)*, 2002, pp. 140–146.

[38] C. L. Goues, M. Pradel, and A. Roychoudhury, "Automated program repair", *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[39] S. Gulwani, V. A. Korthikanti, and A. Tiwari, "Synthesizing geometry constructions", *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 50–61, 2011.

[40] S. Gulwani, I. Radicek, and F. Zuleger, "Feedback generation for performance problems in introductory programming assignments", in *FSE*, 2014, pp. 41–51.

[41] S. Gulwani, I. Radicek, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments", in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 465–480.

[42] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning", in *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, 2017, pp. 1345–1351.

[43] D. C. Halbert, "Programming by example", University of California, Berkeley, 1984.

[44] A. Hellas, J. Leinonen, S. Sarsa, C. Koutcheme, L. Kujanpää, and J. Sorva, "Exploring the responses of large language models to beginner programmers' help requests", in *Proceedings of the 2023 ACM Conference on International Computing Education Research-Volume 1*, 2023, pp. 93–105.

[45] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy abstraction", in *ACM SIGPLAN SIGACT Symposium on Principles of Programming Languages (POPL)*, 2002.

[46] J. B. Hext and J. Winings, "An automatic grading scheme for simple programming exercises", *Communications of the ACM*, vol. 12, no. 5, pp. 272–275, 1969.

[47] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Refactoring based program repair applied to programming assignments", in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2019, pp. 388–398.

[48] P. Ihantola, T. Ahoniemi, V. Karavirta, and O. Seppälä, "Review of recent systems for automatic assessment of programming assignments", in *Proceedings of the 10th Koli calling international conference on computing education research*, 2010, pp. 86–93.

[49] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis", in *ICSE*, 2010, pp. 215–224.

[50] N. Jiang, T. Lutellier, and L. Tan, "Cure: Code-aware neural machine translation for automatic program repair." In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, IEEE, IEEE, May 2021, pp. 1161–1173. [Online]. Available: `https://arxiv.org/pdf/2103.00073`.

[51] L. Johns-Boast and S. Flint, "Simulating industry: An innovative software engineering capstone design course", in *2013 IEEE Frontiers in Education Conference (FIE)*, IEEE, 2013, pp. 1782–1788.

[52] M. Joy, N. Griffiths, and R. Boyatt, "The boss online submission and assessment system", *Journal on Educational Resources in Computing (JERIC)*, vol. 5, no. 3, 2–es, 2005.

[53] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani, "Semi-supervised verified feedback generation", in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2016, pp. 739–750.

[54] M. Kazemitabaar, R. Ye, X. Wang, A. Z. Henley, P. Denny, M. Craig, and T. Grossman, "Codeaid: Evaluating a classroom deployment of an llm-based programming assistant that balances student and educator needs", in *Proceedings of the CHI Conference on Human Factors in Computing Systems*, 2024, pp. 1–20.

[55] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches", in *ICSE*, 2013, pp. 802–811.

[56] C. Koutcheme, N. Dainese, S. Sarsa, A. Hellas, J. Leinonen, and P. Denny, "Open source language models can provide feedback: Evaluating llms' ability to help students using gpt-4-as-a-judge", *arXiv preprint arXiv:2405.05253*, 2024.

[57] X.-B. D. Le, D.-H. Chu, D. Lo, C. Le Goues, and W. Visser, "S3: Syntax- and semantic-guided repair synthesis via programming by examples", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering (FSE)*, 2017, pp. 593–604.

[58] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair", *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[59] J. Leinonen, A. Hellas, S. Sarsa, B. Reeves, P. Denny, J. Prather, and B. A. Becker, "Using large language models to enhance programming error messages", in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 563–569.

[60] Y. Li, S. Wang, and T. N. Nguyen, "Dlfix: Context-based code transformation learning for automated program repair", in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 602–614.

[61] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé, "Tbar: Revisiting template-based automated program repair", in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019, pp. 31–42.

[62] M. Liu and F. M'hiri, "Beyond traditional teaching: Large language models as simulated teaching assistants in computer science", in *Proceedings of the*

55th ACM Technical Symposium on Computer Science Education V. 1, 2024, pp. 743–749.

[63] X. Liu, S. Wang, P. Wang, and D. Wu, "Automatic grading of programming assignments: An approach based on formal semantics", in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*, IEEE, 2019, pp. 126–137.

[64] F. Long and M. Rinard, "Automatic patch generation by learning correct code", in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, 2016, pp. 298–312.

[65] T. Lutellier, H. V. Pham, L. Pang, Y. Li, M. Wei, and L. Tan, "Coconut: Combining context-aware neural translation models using ensemble for program repair." In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*, S. Khurshid and C. S. Pasareanu, Eds., ACM, Jul. 2020, pp. 101–114.

[66] S. MacNeil, A. Tran, A. Hellas, J. Kim, S. Sarsa, P. Denny, S. Bernstein, and J. Leinonen, "Experiences from using code explanations generated by large language models in a web software development e-book", in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, 2023, pp. 931–937.

[67] A. Marginean, J. Bader, S. Chandra, M. Harman, Y. Jia, K. Mao, A. Mols, and A. Scott, "Sapfix: Automated end-to-end repair at scale", in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, IEEE, 2019, pp. 269–278.

[68] S. Mechtaev, M.-D. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury, "Semantic program repair using a reference implementation", in *Proceedings of the 40th International Conference on Software Engineering (ICSE)*, 2018.

[69] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis", in *Proceedings of the 38th International Conference on Software Engineering (ICSE)*, 2016, pp. 691–701.

[70] G. Michaelson, "Automatic analysis of functional program style", in *Software Engineering Conference, Australian*, IEEE Computer Society, 1996, pp. 38–38.

[71] S. Mirhosseini, A. Z. Henley, and C. Parnin, "What is your biggest pain point? an investigation of cs instructor obstacles, workarounds, and desires", in *Proceedings of the 54th ACM Technical Symposium on Computer Science Education V. 1*, ser. SIGCSE 2023, Toronto ON, Canada: Association for Computing Machinery, 2023, pp. 291–297, ISBN: 9781450394314. [Online]. Available: https://doi.org/10.1145/3545945.3569816.

[72] M. Monperrus, "Automatic software repair: A bibliography", *ACM Computing Surveys*, vol. 51, 1 2018.

[73] L. Moura and N. Bjørner, "Z3: An efficient SMT solver", in *TACAS*, 2008, pp. 337–340.

[74] G. C. Necula, "Translation validation for an optimizing compiler", in *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, 2000, pp. 83–94.

[75] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "Semfix: Program repair via semantic analysis", in *2013 35th International Conference on Software Engineering (ICSE)*, IEEE, 2013, pp. 772–781.

[76] M. OpenCourseWare, "6.0001 introduction to computer science and programming in python", https://ocw.mit.edu/courses/.

[77] T. Phung, J. Cambronero, S. Gulwani, T. Kohn, R. Majumdar, A. Singla, and G. Soares, "Generating high-precision feedback for programming syntax errors using large language models", *arXiv preprint arXiv:2302.04662*, 2023.

[78] J. Pryor and B. Crossouard, "A socio-cultural theorisation of formative assessment", *Oxford review of Education*, vol. 34, no. 1, pp. 1–20, 2008.

[79] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay, "Sk_p: A neural program corrector for moocs", in *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity*, 2016, pp. 39–40.

[80] Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang, "The strength of random search on automated program repair", in *Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 254–265.

[81] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems", in *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA)*, 2015.

[82] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples", in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, IEEE, 2017, pp. 404–415.

[83] T. J. Ryan, G. M. Alarcon, C. Walter, R. Gamble, S. A. Jessup, A. Capiola, and M. D. Pfahler, "Trust in automated software repair", in *International Conference on Human-Computer Interaction*, Springer, 2019, pp. 452–470.

[84] M. Schreier, "Qualitative content analysis in practice", Sage publications, 2012.

[85] N. U. of Singapore, "Coursemology, gamified online education platform", `https://coursemology.org/`, Accessed: 2022-10-06, 2022.

[86] N. Singer, "The Hard Part of Computer Science? Getting Into Class", T. N. Y. Times, Ed., Accessed: 16-March-2023, Jan. 2019. [Online]. Available: `https://www.nytimes.com/2019/01/24/technology/computer-science-courses-college.html`.

[87] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments", in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation (PLDI)*, 2013, pp. 15–26.

[88] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments", in *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation*, 2013, pp. 15–26.

[89] A. Solar-Lezama, L. Tancau, R. Bodik, S. Seshia, and V. Saraswat, "Combinatorial sketching for finite programs", in *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, 2006, pp. 404–415.

[90] M. Spichkova, "Industry-oriented project-based learning of software engineering", in *2019 24th International conference on engineering of complex computer systems (ICECCS)*, IEEE, 2019, pp. 51–60.

[91] S. H. Tan, H. Yoshida, M. R. Prasad, and A. Roychoudhury, "Anti-patterns in search-based program repair", in *FSE*, 2016, pp. 727–738.

[92] A. Taylor, A. Vassar, J. Renzella, and H. Pearce, "Dcc–help: Transforming the role of the compiler by generating context-aware error explanations with large language models", in *Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1*, 2024, pp. 1314–1320.

[93] S. Tenhunen, T. Männistö, M. Luukkainen, and P. Ihantola, "A systematic literature review of capstone courses in software engineering", *Information and Software Technology*, p. 107 191, 2023.

[94] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, *et al.*, "Llama: Open and efficient foundation language models", *arXiv preprint arXiv:2302.13971*, 2023.

[95] C. M. University, "Cmu 15-122 fundamentals of programming and computer science", `https://www.cs.cmu.edu/~112/schedule.html`.

[96] U. Von Matt, "Kassandra: The automatic grading system", *ACM SIGCUE Outlook*, vol. 22, no. 1, pp. 26–40, 1994.

[97] M. Vujošević-Janičić, M. Nikolić, D. Tošić, and V. Kuncak, "Software verification and graph similarity for automated evaluation of students' assignments", *Information and Software Technology*, vol. 55, no. 6, pp. 1004–1016, 2013.

[98] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embedding for program repair", *arXiv preprint arXiv:1711.07163*, 2017.

[99] K. Wang, R. Singh, and Z. Su, "Search, align, and repair: Data-driven feedback generation for introductory programming exercises", in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2018, pp. 481–495.

[100] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, *et al.*, "Chain-of-thought prompting elicits reasoning in large language models", *Advances in neural information processing systems*, vol. 35, pp. 24 824–24 837, 2022.

[101] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection", in *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*, IEEE, 2016, pp. 87–98.

[102] M. Wick, D. Stevenson, and P. Wagner, "Using testing and junit across the curriculum", *ACM SIGCSE Bulletin*, vol. 37, no. 1, pp. 236–240, 2005.

[103] Wikipedia contributors, "Root-mean-square deviation — Wikipedia, the free encyclopedia", [Online; accessed 10-November-2022], 2022. [Online]. Available: `https://en.wikipedia.org/w/index.php?title=Root-mean-square_deviation&oldid=1117272661`.

[104] D. Williams, J. Callan, S. Kirbas, S. Mechtaev, J. Petke, T. Prideaux-Ghee, and F. Sarro, "User-centric deployment of automated program repair at bloomberg", *arXiv preprint arXiv:2311.10516*, 2023.

[105] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization", *IEEE Transactions on Software Engineering*, vol. 42, no. 8, pp. 707–740, 2016.

[106] Y. Xiong, X. Liu, M. Zeng, L. Zhang, and G. Huang, "Identifying patch correctness in test-based program repair", in *Proceedings of the 40th international conference on software engineering*, 2018, pp. 789–799.

[107] J. Yi, U. Z. Ahmed, A. Karkare, S. H. Tan, and A. Roychoudhury, "A feasibility study of using automated program repair for introductory programming assignments", in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 740–751.

[108] J. Yi, S. H. Tan, S. Mechtaev, M. Böhme, and A. Roychoudhury, "A correlation study between automated program repair and test-suite metrics", *Empirical Software Engineering*, vol. 23, no. 5, pp. 2948–2979, 2018.

[109] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming", *IEEE Transactions on software engineering*, vol. 46, no. 10, pp. 1040–1067, 2018.

[110] A. Zaks and A. Pnueli, "Covac: Compiler validation by program analysis of the cross-product", in *International Symposium on Formal Methods*, Springer, 2008, pp. 35–51.

[111] J. Zhang, J. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen, "Repairing bugs in python assignments using large language models", *arXiv preprint arXiv:2209.14876*, 2022.

[112] K. Zhang and D. Shasha, "Simple fast algorithms for the editing distance between trees and related problems", *SIAM journal on computing*, vol. 18, no. 6, pp. 1245–1262, 1989.

[113] Y. Zhang, H. Ruan, Z. Fan, and A. Roychoudhury, "Autocoderover: Autonomous program improvement", in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 1592–1604.

[114] Z. Zhong, J. Guo, W. Yang, J. Peng, T. Xie, J.-G. Lou, T. Liu, and D. Zhang, "Semregex: A semantics-based approach for generating regular expressions from natural language specifications", in *Proceedings of the 2018 conference on empirical methods in natural language processing*, 2018.

[115] Q. Zhu, Z. Sun, Y.-a. Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang, "A syntax-guided edit decoder for neural program repair." In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, Eds., ACM, Aug. 2021, pp. 341–353. [Online]. Available: `https://arxiv.org/pdf/2106.08253`.