**UNIVERSIDADE DE LISBOA**

**INSTITUTO SUPERIOR TÉCNICO**

# MENTOR: Automated Feedback for Introductory Programming Exercises

**Pedro Miguel Orvalho Marques da Silva**

| | |
|---|---|
| **Supervisor**: | Doctor Vasco Miguel Gomes Nunes Manquinho |
| **Co-Supervisor:** | Doctor Mikoláš Janota |

Thesis approved in public session to obtain the PhD Degree in

**Computer Science and Engineering**

Jury final classification: **Pass with Distinction and Honour**

**2025**

TÉCNICO LISBOA

**UNIVERSIDADE DE LISBOA**

**INSTITUTO SUPERIOR TÉCNICO**

# MENTOR: Automated Feedback for Introductory Programming Exercises

**Pedro Miguel Orvalho Marques da Silva**

**Supervisor**: Doctor Vasco Miguel Gomes Nunes Manquinho
**Co-Supervisor**: Doctor Mikoláš Janota

Thesis approved in public session to obtain the PhD degree in

## Computer Science and Engineering

Jury final classification: **Pass with Distinction and Honour**

Jury

**Chairperson**: Doctor José Luís Brinquete Borbinha, Instituto Superior Técnico, Universidade de Lisboa

**Members Committee:**

Doctor Martin Monperrus, School of Electrical Engineering and Computer Science (EECS), KTH Royal Institute of Technology, Sweden

Doctor Daniel Le Berre, Computer Science Research Institute of Lens (CRIL-CNRS), Université d'Artois, France

Doctor José Carlos Alves Pereira Monteiro, Instituto Superior Técnico, Universidade de Lisboa

Doctor José Luís Brinquete Borbinha, Instituto Superior Técnico, Universidade de Lisboa

Doctor Vasco Miguel Gomes Nunes Manquinho, Instituto Superior Técnico, Universidade de Lisboa

Doctor João Ricardo Viegas da Costa Seco, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa

**Funding Institution**

FCT - Fundação para a Ciência e Tecnologia

**2025**

*“Instead of just considering programs composed of procedures which can recursively call themselves, why not get really sophisticated, and invent programs which can modify themselves – programs which can act on programs, extending them, improving them, generalizing them, fixing them, and so on? This kind of ’tangled recursion’ probably lies at the heart of intelligence”*

– Douglas R. Hofstadter. Gödel, Escher, Bach: an Eternal Golden Braid [1]. 1979.

2+2=4

# Acknowledgments

First and foremost, I would like to express my deepest gratitude to my doctoral advisors, Vasco Manquinho and Mikoláš Janota, for their unwavering support, invaluable insight, and exceptional guidance throughout these years. I am especially grateful for the opportunities they provided me to attend various conferences, which significantly enriched my PhD experience.

A special thanks to my mother, whose friendship and encouragement have been a constant source of strength. Her unwavering support and presence have been vital in helping me reach this point.

I am equally grateful to my father and Inês, for their continued support and invaluable guidance. Their belief in me has been a cornerstone throughout this journey.

I want to extend my thanks to my grandfather Alfredo. His love, support, and belief in me, while he was still with us, were a constant source of inspiration, and I will always carry his memory with me.

To David: thank you for always being there. Though were quite different in many ways, we've always been able to support each other, and that balance has made our friendship truly special.

Maria, thank you for your constant support, encouragement, and for being a source of joy throughout this journey. Your presence has made even the hardest moments easier to bear. I am also grateful to your mother, father, and the rest of your family for their kindness and support over the last year.

I would also like to extend my gratitude to my brother, Joana, my sisters, the rest of my family, and all of my friends and colleagues who have helped me in various ways during this journey. Thank you for your patience and for putting up with me all these years, it hasn't been an easy task, I am sure! ;)

My sincere thanks also go to the researchers at INESC-ID: João, Margarida, Daniel, Ricardo, Carolina, Henrique, Daniel, Martim, André, and all the others in the ARSR research group. I would also like to acknowledge the HR and administrative staff at INESC-ID, including Vanda, Ana, Manuela, and Sílvia, and Júlia from Técnico's graduation office, for their ongoing support.

A special note of gratitude to everyone at CIIRC, Czech Technical University in Prague, especially Josef Urban and Mikoláš Janota, for their friendship and support during my time there. You truly made my research stays in Prague memorable.

I also would like to thank my thesis evaluation committee for their time, thoughtful feedback, and valuable suggestions. In addition, I am grateful to all the anonymous reviewers who evaluated my research throughout this journey, their insights and critiques helped me improve the quality of my work.

Finally, I would like to acknowledge that everyone I met along this journey, in one way or another, has contributed to this thesis. Every interaction, whether large or small, has helped shape my thinking, challenged me to see things from different perspectives, and inspired new ideas. For that, I am grateful.

To each and every one of you, I extend my heartfelt thanks for your encouragement, guidance, and support throughout this long and challenging journey.

Pedro

I would also like to thank all the institutions that provided financial support during my PhD, as their contribution was essential in making this research possible. This doctoral thesis was supported by Fundação para a Ciência e Tecnologia (FCT) through grant SFRH/BD/07724/2020 (DOI: 10.54499/-2020.07724.BD). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not necessarily reflect the views of FCT. This work was also partially supported by FCT, under projects UIDB/50021/2020 (DOI: 10.54499/UIDB/50021/2020), PTDC/CCI-COM/2156/2021 (DOI: 10.54499/PTDC/CCI-COM/2156/2021) and 2022.03537.PTDC (DOI: 10.54499/-2022.03537.PTDC). PO acknowledges travel support from the EUs Horizon 2020 research and innovation programme under ELISE Grant Agreement No 951847. This work was also supported by European funds through COST Action CA2011; by the European Regional Development Fund under the Czech project AI&Reasoning no. CZ.02.1.01/0.0/0.0/15_003/0000466; by the MEYS within the program ERC CZ under the project POSTMAN no. LL1902 and co-funded by the EU under the project *ROBOPROX* (reg. no. CZ.02.01.01/00/22_008/0004590). This work is part of the RICAIP project funded by the EUs Horizon 2020 research and innovation program under grant agreement No 857306.

# Resumo

A crescente procura por programação deu origem a todos os tipos de cursos focados em exercícios de programação introdutórios (IPAs). Como consequência do elevado número de alunos inscritos, o desafio principal nestes cursos é fornecer feedback útil e personalizado aos estudantes. Esta tese apresenta o MENTOR, uma ferramenta de reparação automática de programas (APR) orientada para fornecer feedback automatizado para exercícios introdutórios de programação. O MENTOR aborda este desafio gerando possíveis reparações semânticas para programas dos alunos, permitindo reparações semânticas validadas através da execução de um conjunto de testes e indicando os segmentos defeituosos aos estudantes.

Ao contrário dos sistemas de reparação simbólica como o CLARA e o VERIFIX, que exigem implementações corretas com grafos de fluxo de controlo idênticos (CFGs), a abordagem do MENTOR baseada em Modelos de Linguagem de Grande Escala (LLM) permite reparações mais flexíveis, sem necessidade de alinhamento estrutural rigoroso. O MENTOR agrupa submissões corretas independentemente dos CFGs e emprega um módulo de alinhamento de variáveis baseado em Redes Neurais em Grafo (GNNs) para maior precisão. O módulo de localização de falhas do MENTOR, o CFAULTS, utiliza técnicas MaxSAT para identificar segmentos defeituosos com precisão. O módulo reparador do MENTOR integra Métodos Formais e LLMs através de um ciclo de Síntese Indutiva Guiada por Contrariedade (CEGIS), refinando iterativamente as reparações. Este trabalho propõe também um sistema de avaliação automática, GITSEED, que fornece feedback personalizado aos alunos sobre as submissões de código e integra com sucesso o CFAULTS para uma deteção eficaz de falhas no código dos estudantes. Resultados experimentais no C-PACK-IPAS demonstram que o MENTOR melhora significativamente as taxas de sucesso na reparação, alcançando 64,4% em reparações, comparado com apenas 6,3% para o VERIFIX e 34,6% para o CLARA.

**Palavras-chave:** Reparação Automática de Programas, Agrupamento de Programas, Análise de Programas, Localização de Falhas Baseada em Fórmulas, Reparação de Programas com Modelos de Linguagem de Grande Escala, Educação Assistida por Computador.

# Abstract

The increasing demand for programming education has given rise to all kinds of online evaluations such as Massive Open Online Courses (MOOCs) focused on introductory programming assignments (IPAs). As a consequence of a large number of enrolled students, one of the main challenges in these courses is to provide valuable and personalized feedback to students. This thesis presents MENTOR, a semantic automated program repair (APR) framework designed to provide Auto**m**ated F**e**edback for I**n**troduc**t**ory Pr**o**gramming Exe**r**cises. MENTOR addresses this challenge by generating possible repairs for faulty student programs, enabling semantic repairs validated through execution on a test suite and by highlighting these faulty statements to the students. Hence, in the context of this work, we provide scientific contributions in several areas, such as program clustering and analysis, automated fault localization and program repair. MENTOR advances the state of the art in the referred areas and provides an innovative practical framework to be deployed in educational environments.

Unlike symbolic repair tools like CLARA and VERIFIX, which require correct implementations with identical control flow graphs (CFGs), MENTOR's Large Language Model (LLM)-based approach enables flexible repairs without strict structural alignment. MENTOR clusters successful submissions regardless of CFGs and employs a Graph Neural Network (GNN)-based variable alignment module for enhanced accuracy. MENTOR's fault localization module, CFAULTS, leverages MaxSAT techniques to pinpoint buggy code segments precisely. MENTOR's program fixer integrates Formal Methods (FM) and LLMs through a Counterexample Guided Inductive Synthesis (CEGIS) loop, iteratively refining repairs. Furthermore, this work also proposes a language-agnostic automated assessment tool, GITSEED, that enhances student learning by providing personalized feedback on code submissions and successfully integrates CFAULTS for effective fault detection on student code. Experimental results on C-PACK-IPAS demonstrate that MENTOR significantly improves repair success rates, achieving 64.4%, compared to just 6.3% for VERIFIX and 34.6% for CLARA.

**Keywords:** Automated Program Repair, Program Clustering, Program Analysis, Formula-based Fault Localization, LLM-Driven Program Repair, Computer-Aided Education.

# Contents

# List of Tables

# List of Figures

# Glossary

**AAST** Anonymized Abstract Syntax Tree.

**AAT** Automated Assessment Tool.

**APR** Automated Program Repair.

**AST** Abstract Syntax Tree.

**CFG** Control Flow Graph.

**CS** Computer Science.

**FBFL** Formula-Based Fault Localization.

**FL** Fault Localization.

**FM** Formal Methods.

**GNNs** Graph Neural Networks.

**IPA** Introductory Programming Assignment.

**IPAs** Introductory Programming Assignments.

**LLM** Large Language Model.

**LLMC** Large Language Model for Coding.

**MaxSAT** Maximum Satisfiability Problem.

**MaxSMT** Maximum Satisfiability Modulo Theories.

**MBD** Model-Based Diagnosis.

**MOOCs** Massive Open Online Courses.

**SAT** Boolean Satisfiability Problem.

**SBFL** Spectrum-Based Fault Localization.

**SE** Software Engineering.

# 1

# Introduction

*“Computers are good at following instructions, but not at reading your mind.”*

– Donald E. Knuth.

## Contents

Nowadays, thousands of students enroll every year in programming-oriented Massive Open Online Courses (MOOCs) [2]. Additionally, due to the COVID-19 pandemic, even small programming courses are utilizing online automated assessment tools. Providing feedback to novice students on introductory programming assignments (IPAs) in these courses requires substantial effort and time from the faculty. Consequently, there is an increasing need for systems that can offer automated, comprehensive, and personalized feedback on incorrect programming assignments. Thus, automated program repair (APR) has become crucial for delivering automatic, personalized feedback to each novice programmer [3].

Typically, a programming assignment in Computer Science (CS) courses follows a pattern: the lecturer defines a computational problem; students program a solution; and each solution is submitted and checked for correctness using predefined tests. If a student's solution fails a given test, it is deemed incorrect without providing helpful feedback. When students encounter issues with their code, they often ask the lecturer for feedback on the unexpected behavior. If their program does not pass at least one predefined test, it indicates that their implementation is semantically incorrect. Unfortunately, personalized feedback from lecturers is often unfeasible due to the growing number of student enrollments. Therefore, semantic APR frameworks [2, 4–12] are ideal for offering hints on how students can repair their incorrect programming assignments. Another significant challenge for test-based evaluation in programming education is differentiating between a nearly semantically correct program with minor syntax errors (which does not compile) and a semantically incorrect program. In many cases, simply pointing out the location of syntactic errors (e.g., the line number) may be insufficient for students to understand what is wrong with their programs. Consequently, syntactic repair frameworks typically suggest concrete repairs to the code [3, 13–15].

Several APR systems aim to assist students and lecturers by providing automated personalized feedback in IPAs. APR tools provide various types of feedback, such as highlighting buggy statements in students' programs [16], generating a corrected version of an incorrect program [9], or explaining code errors in natural language [17]. These systems benefit both students and lecturers: for lecturers, they simplify assignment evaluation and enable programming courses to scale to larger student numbers by reducing the demand for personalized feedback. For students, APR technology enhances self-tutoring by providing prompt, personalized feedback on syntactic and semantic errors.

Compiler error messages do not always provide a precise location for syntax errors [15]. Additionally, these feedback messages can be too complex for novice programmers to comprehend. Over the past few years, various techniques have been proposed to address syntactic errors in IPAs, with the majority employing machine learning models. Several works [13, 14, 18, 19] approach the problem of syntactic repair as a machine translation task, converting buggy code into correct syntactic code. RLAssist [15] utilizes a reinforcement learning approach to fix compile errors in IPAs. DrRepair [3] employs a graph neural network to reason about syntax errors and compiler feedback messages in order to correct incorrect student solutions. More recently, PyDex [20] uses iterative querying with Codex, a Large Language Model (LLM) trained for coding tasks (LLMC), incorporating test-based few-shot selection and structure-based program chunking to repair both syntax and semantic errors in Python assignments.

Recently, Large Language Models (LLMs) have also been employed for APR [20–23], but they often

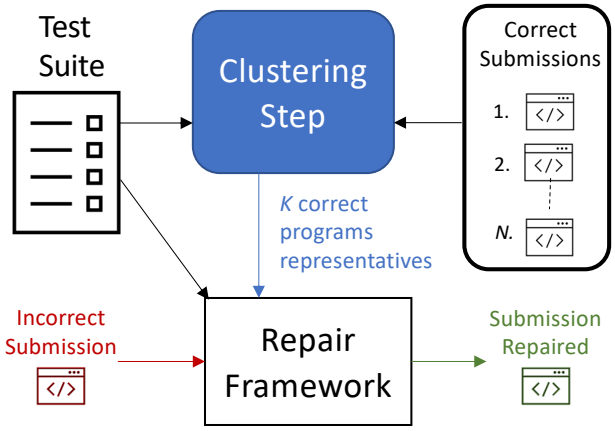



Figure 1.1: General framework for clustering-based Program Repair.

produce extensive rewrites instead of minimal adjustments. This tendency results in more invasive fixes, making it more difficult for students to learn from their mistakes.

The field of automated program repair [24–26] has primarily focused on fixing semantic errors [2, 10, 11, 19, 27–30], typically using program specifications such as test suites or reference solutions. However, most of these works assume that the students' programs compile successfully. Semantic automated repair techniques can be classified into two distinct methodologies: semantic-based and search-based (also known as generate-and-validate). Semantic-based repair approaches utilize constraint solving and symbolic execution of programs [31] to synthesize repairs based on semantic information. In contrast, generate-and-validate techniques [32, 33] apply a set of predefined mutation operators to modify the input program, generating multiple candidate solutions, which are then validated for correctness against a test suite.

Recent research has introduced semantic-based program repair frameworks [2, 10, 11, 34] for IPAs that leverage a large number of previously enrolled students in a programming course to obtain diverse correct implementations for each programming assignment. Having a similar correct implementation enables the computation of a smaller set of repairs to fix an incorrect program, rather than relying on the full set of repairs needed to make the submission semantically equivalent to a fixed reference implementation. However, analyzing all previous correct submissions for an IPA is often infeasible, as this set typically consists of thousands of programs. To tackle this problem, different program clustering approaches have been proposed to use in program repair tools which enable focusing only on the representatives of the clusters. Figure 1.1 shows the overall architecture of these clustering-based program repair tools. Given a set of $N$ correct submissions and a test suite, these tools compute $K$ clusters of programs ($N >> K$) and use only the set of $K$ clusters' representatives in the repair process. Given an incorrect student submission, these frameworks use clustering methods to find the most similar correct submission from the set of $K$ clusters' representatives to provide a minimal set of repairs to the student. CLARA [2] is a clustering-based program repair tool that clusters the correct programs based on their dynamic equivalence [35] and control flow, i.e., the order in which program statements, instructions, and function calls are executed. SARFGEN [10] computes program representations based

Table 1.1: Test-suite with three tests ($t_0$, $t_1$, and $t_2$). Given as input three numbers (num1, num2, and num3), each test should return the value in its respective output column.

| | Input | | | Output |
|---|---|---|---|---|
| | **num1** | **num2** | **num3** | |
| $t_0$ | 1 | 2 | 3 | 3 |
| $t_1$ | -1 | -2 | -3 | -1 |
| $t_2$ | 1 | 2 | 1 | 2 |

on each program's abstract syntax tree (AST) and clusters the programs based on their representations. SEMCLUSTER [34] utilizes both the control flow and the data flow of each program, where the data flow tracks the number of occurrences of consecutive values that a variable takes during its lifetime.

The problem of program equivalence, i.e., deciding if two programs are equivalent, is undecidable [36, 37]. On that account, finding an adequate representation for programs that performs well on program clustering is a challenging problem. The above-mentioned program representations used in the field of APR may be fragile, as we are going to show in Section 1.2.

Thus, this work presents MENTOR, a semantic program repair tool capable of providing Auto**m**ated F**e**edback for I**n**troduc**t**ory Pr**o**gramming Exe**r**cises. MENTOR takes advantages of previous students' submissions from past years to help repairing incorrect submissions. MENTOR, presented in Section 1.3, tries to overcome the current state-of-the-art program repair tools' drawbacks listed in Chapter 3. MENTOR has four main objectives: (1) to advance the current state-of-the-art in program clustering and program repair; (2) to improve the state-of-the-art in MaxSAT-based fault localization; (3) to enable the repair of a buggy program using a correct implementation whose control-flow graph (CFG) does not need to match the faulty program's CFG; and (4) to provide personalized and sound feedback to students.

## 1.1 Motivation

Imagine you are a lecturer in an introductory programming course with hundreds of students. For the first programming assignment, you ask your students to write a small program in the C programming language that finds and returns the maximum of three given numbers. A possible solution for this exercise is to initially assign the first number as the maximum and then check if either of the following two numbers is greater. If so, the maximum number is updated. Finally, the function should return the maximum number. Listing 1.1 presents a possible C implementation for this programming assignment, function `int max_three(int num1, int num2, int num3)`.

This is your students' first programming assignment, so many of them will need feedback on programming in C, as they may not be familiar with its syntax. Another concern is how to explain to each student the semantic errors in their code, specifically, what the problem is and why their program does not exhibit the expected behavior. For example, consider the following student submission, Listing 1.2, for the same exercise described in Listing 1.1:

**Listing 1.1:** Function `int max_three(int num1, int num2, int num3)` finds and returns the maximum number among `num1`, `num2` and `num3`.

```
1  int max_three(int num1, int num2, int num3){
2      int max = num1;
3      if(num2 > max){
4          max = num2;
5      }
6      if(num3 > max){
7          max = num3;
8      }
9      return max;
10 }
```

**Listing 1.2:** An erroneous implementation of the max_three function with syntactic and semantic errors.

```
1  int max_three(int n1, int n2, int n3){
2      int max = 0;
3      if(n1 > max){
4          max = n1;
5      }
6      if (n2 > max){
7          max = n2;
8      {
9      if (n3 > max){
10         max = n3;
11     }
12     return max /* Missing ; */
13 }
```

This student's submission contains both syntactic and semantic errors. Syntactic errors are highlighted in red, while semantic errors are highlighted in orange. Regarding syntax, the student has misplaced a brace { on line 8 and forgotten a semicolon after the return statement on line 12. Regarding the program's semantics, consider the test-suite presented in Table 1.1. After correcting the syntatic errors, the program provides the expected output for test $t_0$ and $t_2$, but it fails to pass test $t_1$ because the `max` variable is initialized as zero on line 3. Consequently, if all three numbers provided in the input are negative, this function will return zero.

With hundreds of students enrolled in your class, providing this kind of personalized syntactic and semantic feedback to each student becomes impractical. Hence, there has been a growing interest in automated program repair techniques in programming courses over the last decade, particularly due to the increasing number of students in programming-oriented Massive Open Online Courses (MOOCs) [15]. Therefore, students can benefit from having access to a system that can guide them in improving their IPAs.

## 1.2 Problem Description

In an introductory programming course, novice students typically develop a wide variety of solutions for the same introductory programming assignment (IPA). These differences can be syntactic (e.g., variables' names and structures used) or semantic (e.g., different data-flow). However, two significantly different programs, syntactically and semantically, can still be correct implementations for the same IPA.

**Example 1.** Consider the following two programs written in C. Both programs compute the sum of all the natural numbers from $1$ to a given number `n` i.e. $\sum_{i=1}^{n} i$.

**Listing 1.3:** Program that uses a while-loop to sum all the natural numbers from $1$ to $n$.

```c
int main(){
    int n;
    int sum=0, i;
    scanf("%d", &n);
    i = 1;

    while(i <= n) {
      sum = sum+i;
      i++;
    }

    printf("%d\n",sum);

    return 0;
}
```

**Listing 1.4:** Program that uses a for-loop and an auxiliary function to sum all the natural numbers from $1$ to $n$.

```c
int sum_for(int n){
    int j, s=0;
    for(j=1; j <= n; j++){
      s = j + s;
    }
    return s;
}

int main(){
    int n, s;
    scanf("%d", &n);
    s = sum_for(n);
    printf("%d\n", s);
    return 0;
}
```

Observe that the program on Listing 1.3 uses a while-loop that iterates over the natural numbers from $1$ to `n`. However, the program on the right, Listing 1.4, calls an auxiliary function (`int sum_for(int n)`) that uses a for-loop to iterate the set of natural numbers from $1$ to `n`. Hence, both programs are semantically equivalent since they produce the same result. However, if we create a representation of these programs using their syntax or abstract syntax trees (ASTs), the representations will differ significantly. In terms of syntax, the variable names (e.g., `i`, `j`, `s`, `sum`) and structures (e.g., `while`, `for`) are different. Additionally, the program on the right calls an auxiliary function, whereas the complete code of the first program is contained within the main function.

Current state-of-the-art automated program repair (APR) frameworks, such as CLARA [2] and VERIFIX [9], rely on strict repair techniques that require at least one correct implementation for the same IPA with an identical program structure (control flow and loop sequence) as the incorrect submission. If such implementation does not exist, these tools cannot repair the student's submission returning an error of *mismatch structure*. For example, consider the following program in Listing 1.5, which is another implementation for the same IPA described in Example 1. This program is semantically incorrect since the variable $j$ is assigned to the value of $9$ instead of $0$:

**Listing 1.5:** Program that uses a for-loop to sum all the natural numbers from $1$ to $n$.

```c
int main(){
    int n, j, s=0;
    scanf("%d", &n);
    for(j=9; j < n; j++){
      s = j + s;
    }
    printf("%d\n",s);
    return 0;
}
```

Current state-of-the-art APR tools cannot repair this program using the correct implementations presented in Example 1. These tools return a *mismatch structure* error because this program has a different loop structure compared to Listing 1.3 (while-cycle vs. for-cycle). Additionally, Listing 1.4 calls an auxiliary function (`int sum_for(int n)`) while this program does not.

Furthermore, Large Language Models (LLMs) have been used for APR [20–23, 38–41] but often make extensive rewrites instead of minimal adjustments. This tends to lead to more invasive fixes, making it harder for students to learn from their mistakes.

Thus, MENTOR aims to address the limitations of current state-of-the-art APR frameworks for IPAs by enabling the repair of a student's incorrect submission without requiring a correct implementation with the same program structure, while also minimizing changes to the student's code.

## 1.3 MENTOR

This work presents MENTOR, a clustering-based semantic program repair tool capable of providing Auto**m**ated F**e**edback for I**n**troduc**t**ory Pr**o**gramming Exe**r**cises. MENTOR takes advantages of previous students' submissions from past years to aid in repairing incorrect submissions.

As shown in Figure 1.2, MENTOR receives as input an incorrect submission for a given introductory programming assignment (IPA), a test suite, and a set of $N$ correct submissions for the same IPA. MENTOR is divided into five main modules, as depicted in Figure 1.2: (1) program clustering, (2) variable aligner, (3) fault localization, (4) program fixer, and (5) decider.

MENTOR starts by clustering, using INVAASTCLUSTER [42], all correct submissions using these program's sets of invariants and abstract syntax trees (ASTs). Next, MENTOR employs Graph Neural Networks (GNNs) to map the set of variables between each cluster's representative and the incorrect submission based on the ASTs of both programs. Mapping variables between two programs is crucial for a wide range of tasks, such as program equivalence, program analysis, and program repair.

For fault localization, MENTOR utilizes CFAULTS [16], a novel formula-based fault localization technique for C programs capable of addressing multiple faults. CFAULTS leverages Model-Based Diagnosis (MBD) with multiple observations, consolidating all failing test cases into a unified MaxSAT formula to ensure consistency in the fault localization process.

The program fixer module integrates all the features computed so far: (1) cluster representatives, (2) variable mappings, and (3) localized faulty statements. These features are incorporated into prompts for various Large Language Models (LLMs) to guide our LLM-driven counterexample guided program repair processes. Finally, in the decider module, MENTOR checks if the candidate program proposed by the program fixer is correct by evaluating it against the provided test suite. If the candidate program remains incorrect, the program fixer generates a new candidate. Otherwise, MENTOR returns personalized feedback to students, either by highlighting the buggy statements in their code or by providing the repaired program.

Experimental results on C-PACK-IPAS, our benchmark of IPAs, demonstrate that MENTOR's hybrid repair method, integrating FM-based fault localization with Large Language Models (LLMs), significantly

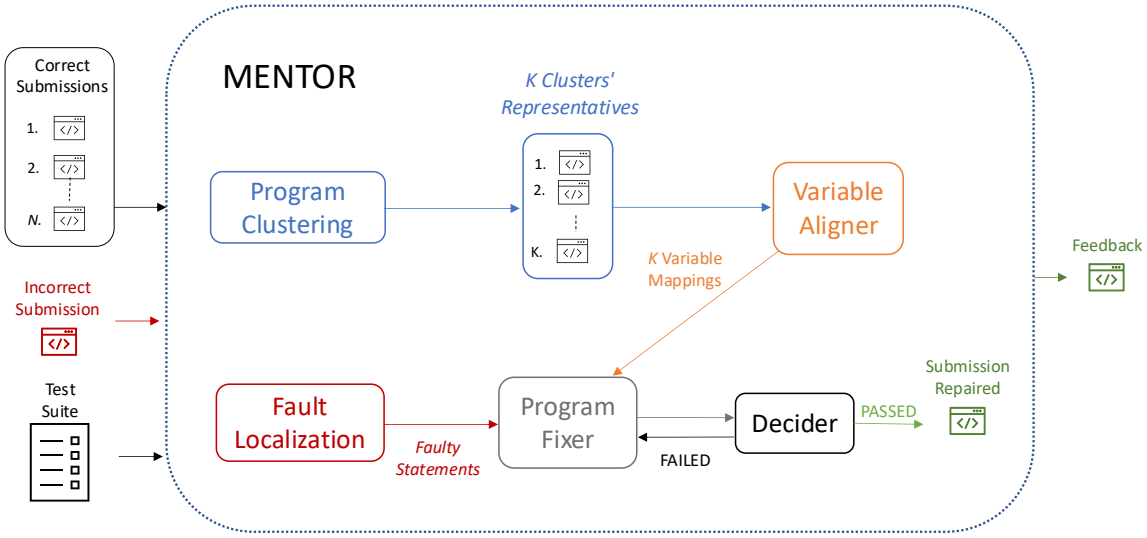



Figure 1.2: Overview of MENTOR.

enhances repair success rates while yielding smaller, more precise fixes. This innovative approach outperforms other existing repair strategies and state-of-the-art symbolic tools. For instance, VERIFIX repairs only 6.3% of the benchmarks, and CLARA achieves a repair rate of 34.6%. In contrast, depending on the prompt configuration and the specific LLM utilized, MENTOR achieves impressive repair rates ranging from 37.3% to 64.4% on C-PACK-IPAS.

Furthermore, this work also presents GITSEED, a language-agnostic automated assessment tool integrated with GITLAB, designed to enhance Programming Education and Software Engineering (SE). This tool not only facilitates students' learning of `git` fundamentals but also provides personalized feedback on their code submissions.

A distinctive aspect of MENTOR is its pedagogical approach. Rather than providing direct fixes, the tool highlights problematic areas in the student's code, encouraging independent problem-solving. MENTOR's fault localization module, CFAULTS, has been successfully integrated into GITSEED to accurately identify faults in students' programs. This teaching strategy, designed to better prepare students for future challenges, has proven effective since approximately 70% of Computer Science students at Instituto Superior Técnico found MENTOR's feedback beneficial to their learning. Furthermore, our evaluation shows that both GITSEED and MENTOR significantly enhance student engagement and learning outcomes.

Even though MENTOR employs this pedagogical approach of highlighting issues rather than providing direct fixes, it always repairs the given program. This allows for the possibility of offering alternative forms of feedback. For instance, the repaired program could be converted into a program sketch that outlines the expected solution structure. This would give students insight into the necessary program structure without revealing the precise fixes, encouraging independent problem-solving while still guiding them toward the correct solution.

As illustrated in Figure 1.3, MENTOR aims to support both students and lecturers by offering automated personalized feedback in IPAS. MENTOR provides benefits for both groups: for lecturers, it

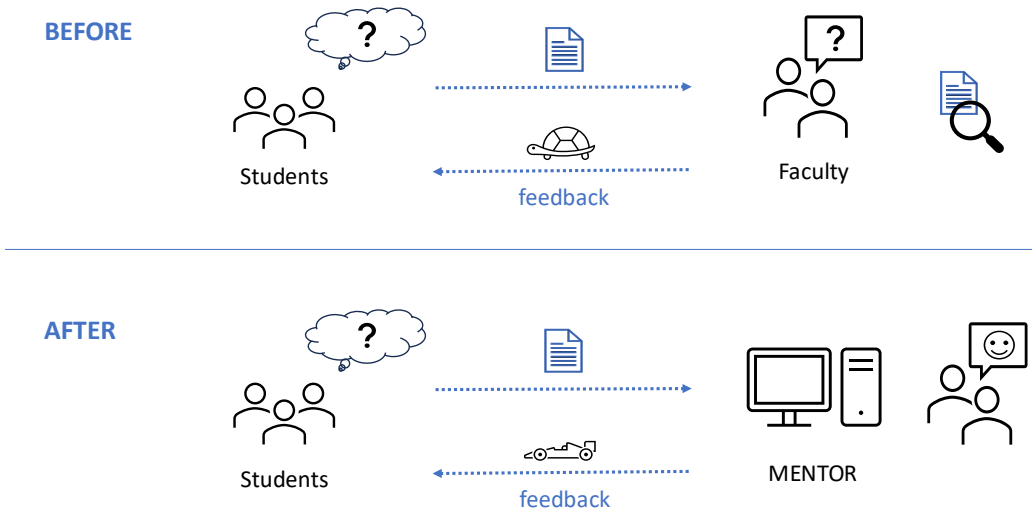



Figure 1.3: Benefits of using MENTOR.

simplifies the evaluation process and allows programming courses to scale to larger enrollments, as the need for direct personalized feedback from the lecturer is significantly reduced. For students, MENTOR improves their self-learning experience by providing targeted feedback on syntactic and semantic errors in a shorter time frame.

To summarize, this work makes the following contributions:

- **C-PACK-IPAS**: A benchmark consisting of C programs submitted for 25 different IPAs over three academic years. C-PACK-IPAS includes semantically correct, semantically incorrect, and syntactically incorrect programs, as well as a comprehensive test suite for each IPA (see Chapter 4);
- **INVAASTCLUSTER**: A novel and efficient clustering approach for submissions in introductory programming assignments (IPAs) using invariant sets and AAST representations (see Chapter 5);
- **MULTIPAS**: A program transformation tool capable of augmenting imperative program benchmarks through six different syntactic mutations and three semantic transformations (see Chapter 6);
- **Graph-based Program Representation**: A new program representation that abstracts variable names, using a representative variable node for each variable, connected to all occurrences in the program. This allows for leveraging Graph Neural Networks (GNNs) to map variables between programs, disregarding variable identifiers (see Chapter 7);
- **CFAULTS**: A MaxSAT-based fault localization tool for C programs, which considers multiple failing test cases and formulates fault localization as a unified optimization problem (see Chapter 8);
- **GITSEED**: An open-source, language-agnostic automated assessment tool for Software Engineering (SE) and Programming Education, integrated with GITLAB (see Chapter 9);
- **LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS)**: A novel approach to Automated Program Repair that employs MaxSAT-based fault localization to guide and refine Large Language Models' (LLMs) patches to buggy programs by providing bug-free program sketches (see Chapter 10).

## 1.4 Organization

This document is organized as follows. Chapter 2 presents the basic definitions and notation used in the following chapters. Afterwards, Chapter 3 provides some background on Program Repair.

Chapter 4 presents C-PACK-IPAS which is the benchmark of student program submissions for 25 distinct IPAS, used to evaluate this work. Next, Chapter 5 describes INVAASTCLUSTER, which is our program clustering tool. Afterwards, Chapter 6 presents MULTIPAS, our program transformation tool that can augment IPAS benchmarks by applying syntactic and semantic mutations to programs, introducing faults in the IPAS to enhance C-PACK-IPAS. In Chapter 7, we use graph neural networks (GNNs) to map the set of variables between two programs based on both programs' abstract syntax trees (ASTS).

Furthermore, Chapter 8 presents CFAULTS which is a novel formula-based fault localization (FBFL) approach for C programs with multiple faults. Next, GITSEED, our language-agnostic automated assessment tool (AAT) designed for Programming Education and Software Engineering (SE) and backed by GITLAB is presented in Chapter 9.

Chapter 10 proposes a novel approach that combines the strengths of MaxSAT-based fault localization, Large Language Models (LLMS), along with all the work described in the previous chapters, to enhance APR for IPAS. Finally, Chapter 11 presents the main conclusions.

# 2

# Preliminaries

*“A good notation has a subtlety and suggestiveness which at times make it almost seem like a live teacher.”*

– Bertrand Russell.

## Contents

This chapter provides some definitions that will be used throughout this manuscript. Some of the following definitions were adapted from other works [43–48]. First, Section 2.1 presents some basic definitions of propositional logic. Afterwards, Section 2.2 provides concepts on the field of programming languages, and Section 2.3 defines control and data flow graphs used to represent programs. Finally, Section 2.4 presents some well-known problems in Computer Science, such as Program Synthesis and Repair.

## 2.1 Logic

**Definition 1** (**Propositional Literal**)**.** A *propositional literal* is a Boolean variable $x$ (positive literal) or its negation $\neg x$ (negative literal).

**Definition 2** (**Clause**)**.** The disjunction of literals (e.g., $x \vee y$) is called a *clause*.

**Definition 3** (**Formula**)**.** A *propositional formula* $\phi$ in the *conjunctive normal form* (CNF) is a conjunction of clauses (e.g., $\phi = (x \vee y) \wedge (x \vee z)$).

**Definition 4** (**Interpretation**)**.** Let $c$ be a clause, $L$ be the set of $c$'s literals and $m$ be an interpretation, such that, $\forall l \in L, m \ : \ l \ \rightarrow \ \{0,1\}$. The clause $c$ is satisfied by $m$ if and only if at least one literal in $L$ is satisfied by $m$. A formula $\phi$ is satisfied by an interpretation $m$ if and only if all of $\phi$'s clauses are satisfied by $m$.

**Definition 5** (**Boolean Satisfiability Problem (SAT)**)**.** *SAT* is the decision problem for propositional logic, i.e., to decide if a given propositional formula $\phi$ has a satisfying interpretation or prove that such an interpretation does not exist [44].

SAT was the first problem proven to be NP-Complete in 1971 by Cook [49]. There are innumerous problems that can be modeled as a propositional formula. The satisfiability of such formulas is checked by logical engines called *solvers*.

**Definition 6** (**Solver**)**.** A *Solver* is a logic engine capable of deciding if a given propositional formula, $\phi$, is satisfiable. In this case, the solver produces an interpretation (attribution) of $\phi$ such that $\phi$ evaluates true. Otherwise, the solver returns that $\phi$ is unsatisfiable [50].

**Example 2.** Let $\phi_1 = (x_1 \vee \neg x_2) \wedge (x_1 \vee x_2)$ be a propositional logic formula and $\{x_1, x_2\}$ be the set of $\phi_1$'s variables. A SAT solver would produce one of two possible interpretations of $\phi_1$: $\{(x_1, 1), (x_2, 0)\}$ or $\{(x_1, 1), (x_2, 1)\}$.

Consider another propositional logic formula $\phi_2 = (x_1 \vee \neg x_2) \wedge \neg x_1 \wedge x_2$. There is no possible interpretation that makes $\phi_2$ satisfiable. Hence, a SAT solver would return "unsatisfiable" for this formula.

**Definition 7** (***Maximum Satisfiability (MaxSAT)***.)**.** The *Maximum Satisfiability* (MaxSAT) problem is an optimization version of SAT, i.e., the goal is to find an assignment that maximizes the number of satisfied clauses in a CNF formula. In partial MaxSAT, clauses in $\phi$ are split in hard $\phi_h$ and soft $\phi_s$. Given a formula $\phi = (\phi_h, \phi_s)$, the goal is to find an assignment that satisfies all hard clauses in $\phi_h$ while minimizing the

| | | | |
|---|---|---|---|
| Hard: | $h_1 : (v_1 \vee v_2)$ | $h_2 : (\neg v_2 \vee v_3)$ | $h_3 : (\neg v_1 \vee \neg v_3)$ |
| Soft: | $s_1 : (\neg v_1)$ | $s_2 : (\neg v_3)$ | |

Figure 2.1: Example of a MaxSAT formula

number of unsatisfied soft clauses in $\phi_s$. The partial MaxSAT problem can be further generalized to the weighted version, where each soft clause has an associated weight, and the optimization goal is to minimize the sum of the weights of the unsatisfied soft clauses [51, 52]. Finally, we assume that $\phi_h$ is satisfiable. Figure 2.1 presents an example of a partial MaxSAT formula of cost 1 since either $v_1$ or $v_3$ must be assigned to true.

**Definition 8** (**Minimal Correction Subset (MCS)**)**.** Let $\phi = (\phi_h, \phi_s)$ be a partial MaxSAT formula. A Minimal Correction Subset (MCS) $\mu$ of $\phi$ is a subset $\mu \subseteq \phi_s$ where $\phi_h \cup (\phi_s \setminus \mu)$ is satisfiable and, for all $c \in \mu$, $\phi_h \cup (\phi_s \setminus \mu) \cup \{c\}$ is unsatisfiable [51]. A dual concept of MCSes are *Minimal Unsatisfiable Subsets (MUSes)* [53, 54].

**Definition 9** (**Satisfiability Modulo Theories (SMT)**)**.** The *Satisfiability Modulo Theories (SMT)* problem is a generalization of the SAT problem. Given a decidable first-order theory $\mathcal{T}$, a $\mathcal{T}$-atom is a ground atomic formula in $\mathcal{T}$. A $\mathcal{T}$-literal is either a $\mathcal{T}$-atom $t$ or its complement $\neg t$. A $\mathcal{T}$-formula is similar to a propositional formula, but a $\mathcal{T}$-formula is composed of $\mathcal{T}$-literals instead of propositional literals. Given a $\mathcal{T}$-formula $\phi$, the SMT problem consists of deciding if there exists a complete assignment over the variables of $\phi$ such that $\phi$ is satisfied. Depending on the theory $\mathcal{T}$, the variables can be of type integer, real, Boolean, among others [45].

**Example 3.** Let $\phi_1 = (x_1 \geq 0) \wedge (x_1 \leq 4) \wedge (x_2 \leq 2) \wedge (x_1 + x_2 = 5)$ be an SMT formula where $\mathcal{T}$ is the Linear Integer Aridefinitionetic (LIA) theory. Clearly, $\phi_1$ is satisfiable and a possible solution would be $x_1 = 4, x_2 = 1$.
Let $\phi_2 = (x_1 \geq 0) \wedge (x_1 \leq 3) \wedge (x_2 \leq 1) \wedge (x_1 + x_2 = 5)$ be an SMT formula also in the LIA theory. In this case, $\phi_2$ is unsatisfiable since there is no assignment to the problem variables such that $\phi_2$ is evaluated to true.

## 2.2 Languages and Programs

**Definition 10** (**Onto Relation**)**.** A function/relation $f : A \to B$ is *onto* if $f(A) = B$, i.e., every $b \in B$ is the image of some $a \in A$ [55].

**Definition 11** (**One-to-one Relation**)**.** A function/relation $f : A \to B$ is *one-to-one* if $a \neq a'$ implies $f(a) \neq f(a')$, i.e., distinct points in $A$ have distinct images in $B$ [55].

**Definition 12** (**Bijective Relation**)**.** A function/relation $f : A \to B$ is a *bijection* iff $f$ is simultaneously an onto relation and an one-to-one relation [55].

Clearly, if $f : A \to B$ is a *bijective relation* (see Definition 12) for two finite sets $A$ and $B$ then these two sets have the same cardinality, i.e., $|A| = |B|$.

**Definition 13** (**Isomorphism**)**.** An *isomorphism* is an information preserving transformation. The word "isomorphism" applies when two complex structures (e.g., functions, graphs) can be mapped onto each other. With this mapping, each part of one structure has a corresponding part in the other structure, i.e., the two parts play similar roles in their respective structures [1].

Considering Defintion 13, we can clearly see that a relation $f : A \to B$ is an *isomorphism* if and only if $f$ is bijective relation [55].

**Definition 14** (**Context-free Grammar**)**.** A *context-free grammar* $G$ is a 4-tuple $(V, \Sigma, R, S)$, where $V$ is the set of non-terminals symbols, $\Sigma$ is the set of terminal symbols, $R$ is the set of rules and $S$ is the start symbol. A context-free grammar describes all the strings permitted in a particular formal language [46].

**Definition 15** (**Domain-Specific Language (DSL)**)**.** A *Domain-specific Language (DSL)* is a tuple $(G, Ops)$, where $G$ is a context-free grammar ($G = (V, \Sigma, R, S)$) and Ops is the semantics of DSL operators. The context-free grammar $G$ has the rules to generate all the programs in the DSL. The semantics of DSL operators is necessary to analyze conflicts and make deductions [47].

Each symbol $\sigma \in \Sigma$ corresponds to built-in DSL constructs (e.g., `if`, `while`, `return`), constants, variables or inputs of the system. Each production rule $p \in R$ has the form $p = (A \to \sigma(A_1, \ldots, A_m))$, where $\sigma \in \Sigma$ is a DSL construct and $A_1, \ldots, A_m \in \Sigma$ are symbols for the arguments of $\sigma$.

**Definition 16** (**Bag of Words (BoW)**)**.** A *Bag of Words (BoW)* representation [56] is a vector representation where a tokenized sentence is represented as a bag of its words in a vector. The vector representation contains information on the number of times each token in the language appears in the sentence. Note that this model does not consider the language's grammar and even word order. The tokenization step divides a string into $n$-grams, which are sub-sequences of the original string of $n$ items.

The following example presents a small illustration of a vector representation of a phrase using a BoW model.

**Example 4.** Let $B$ be a bag of words model computed using the following sentences: $\{'aa', 'e\ i', 'a\ e\ i\ o\ u', 'o\ i'\}$

Given the phrase $p =' a\ i\ a\ u'$, the vector representation of $p$ is, $B(p) = [0.5,\ 0.0,\ 0.25,\ 0.0,\ 0.25]$. The size of $B(p)$ is 5 since 5 is the size of the vocabulary of $B$. For each entry $s$ of $B(p)$, $B(p)[s]$ corresponds to the percentage of $p$ that is equal to $s$. For example, the symbol $a$ appears twice in a four-symbol phrase. Hence $B(p)[a] = 0.5$.

**Definition 17** (**Program**)**.** A *program* is considered sequential, comprising standard statements such as assignments, conditionals, loops, and function calls, each adhering to their conventional semantics in C. A program is deemed to contain a bug when an assertion violation occurs during its execution with input $I$. Conversely, if no assertion violation occurs, the program is considered correct for input $I$. In cases where a bug is detected for input $I$, it is possible to define an error trace, representing the sequence of statements executed by program $P$ on input $I$.

**Definition 18** (**Trace Formula (TF)**)**.** A *Trace Formula (TF)* is a propositional formula that is SAT iff there exists an execution of the program that terminates with a violation of an assert statement while satisfying all assume statements. For further information on TFs, interested readers are referred to [57, 58].

**Definition 19** (**Basic Block (BB)**)**.** A *basic block* is a linear sequence of program instructions having one entry point (the first instruction executed) and one exit point (the last instruction executed). It may, of course, have many predecessors and many successors and may even be its own successor. Program entry blocks might not have predecessors that are in the program; program terminating blocks never have successors in the program [48].

**Definition 20** (**Program Invariant**)**.** Program invariants are conditions that must always be true at a given point during a program's execution.

Dynamically generated program invariants are likely invariants observed during several program executions for a given program. The dynamically generated set of program invariants provides information about a program's behavior, i.e., its semantics. Program invariants are usually used to assert some assurances throughout a program (assertions).

**Example 5.** Let us examine the following C program that computes the sum of all the natural numbers from $1$ to a given number `n` i.e. $\sum_{i=1}^{n} i$.:

```
1  int main(){
2      int n, sum=0, i;
3      scanf("%d", &n);
4      i = 0;
5      while(i < n) {
6        i++;
7        sum = sum+i;
8      }
9      printf("%d\n",sum);
10
11     return 0;
12 }
```

There are three basic blocks (see Definition 19) in this program: `BB1` (lines $2-4$), `BB2` (lines $5-8$) and `BB3` (lines $9-11$). Consider that the variable `n` is always assigned to a natural number, $n > 0$. If a dynamic invariant detector (e.g. Daikon [59]) is used in this program, the following set of program invariants (see Definition 20) is observed at each iteration of the while-loop: $n > 0$; $sum \geq 0$; $0 \leq i \leq n$.

**Definition 21** (**Abstract Syntax Tree (AST)**)**.** An *abstract syntax tree (*AST*)* is a syntax tree in which each node represents an operation, and the children of the node represent the arguments of the operation for a given programming language described by a Context-free Grammar [46]. An AST depicts a program's grammatical structure [60]. Fig. 2.2a presents a small example of the AST representation for the declaration `int i`.

**Definition 22** (**Anonymized Abstract Syntax Tree (AAST)**)**.** An *anonymized abstract syntax tree (AAST)* is an AST in which nodes that have identifiers are anonymized, i.e., a node's identifier (name of


(a) AST representation.

(b) AAST representation.

Figure 2.2: Small example of an AST and an AAST for the declaration, `int i = 1`. An integer variable with identifier $i$.

a function or variable) is replaced by a unique token (*ID*). Fig. 2.2b shows the AAST representation for the same declaration presented previously, `int i`.

**Definition 23** (**Program Sketch.**)**.** A *program sketch* is a partially incomplete program where all buggy statements are replaced by placeholders, identified as "`@ HOLES @`". These placeholders indicate program parts that need to be synthesized to ensure the program complies with a given specification (e.g., a test suite).

**Definition 24** (**Formula-based Fault Localization (FBFL)**)**.** Given a faulty program and a test suite with failing test cases, *formula-based fault localization* (FBFL) methods encode the localization problem into an optimization problem to identify a minimal set of faulty statements (diagnoses) within a program. FBFL tools leverage MaxSAT and the theory of *Model-Based Diagnosis* (MBD) [16, 54, 61–63]. Moreover, these FBFL tools enumerate all diagnoses of a MaxSAT formula corresponding to bug locations.

## 2.3 Flow Representations

There are two types of dependencies in a program: (1) *data dependencies* and (2) *control dependencies*.

**Definition 25** (**Data dependence**)**.** A *data dependence* exists between two statements in a program whenever a variable that appears in one statement is assigned to an incorrect value if the two statements are reversed [64].

Let us look back at the program presented in Example 5. One can clear see that the $7^{th}$ line of code (`sum = sum+i`) depends on the execution of the $6^{th}$ line (`i++`), since executing the $7^{th}$ line before the $6^{th}$ would result in an incorrect value for the variable `sum`.

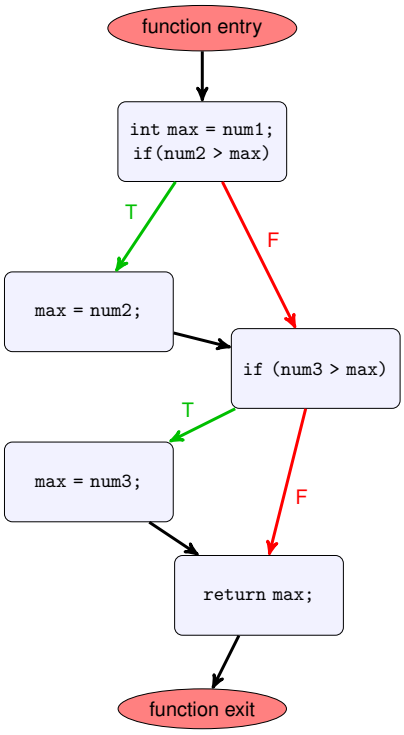
**Definition 26** (**Control dependence**)**.** A *control dependence* occurs between a statement and a control predicate whose value dictates if the statement is executed or not [64].

In Example 5, both the $6^{th}$ and $7^{th}$ lines of code depend on the predicate on line $4$ since its value determines whether the program executes the body of the while-loop or jumps to the next instruction in line $9$.

**Definition 27** (**Control flow path**)**.** A *control flow path* is defined by the order in which program statements, instructions, and function calls are executed.

```
1  int max_three(int num1, int num2, int num3){
2      int max = num1;
3      if(num2 > max)
4      {
5          max = num2;
6      }
7      if(num3 > max)
8      {
9          max = num3;
10     }
11     return max;
12 }
```

(a) Function `max_three`.




(b) Control flow graph of function `max_three`.

Figure 2.3: Function `int max_three(int num1, int num2, int num3)` which finds and returns the maximum number among `num1`, `num2` and `num3`, presented earlier in Listing 1.1 and its control flow graph (see Definition 29).

**Definition 28** (**Data flow path**)**.** A *data flow path* tracks the number of occurrences of consecutive values a variable takes during its lifetime.

**Definition 29** (**Control flow Graph (CFG)**)**.** A *control flow graph (CFG)* is a directed graph in which the nodes represent basic blocks, and the edges represent control flow paths [48].

Figure 2.3b shows the control flow graph of function `int max_three(int num1, int num2, int num3)` presented in Figure 2.3a. The nodes represent the set of basic blocks (see Definition 19) of the program. This graph represents every possible control flow path (see Defintion 27) of the function `max_three`. The CFG depicts the control dependencies (see Definition 26) in a given program.

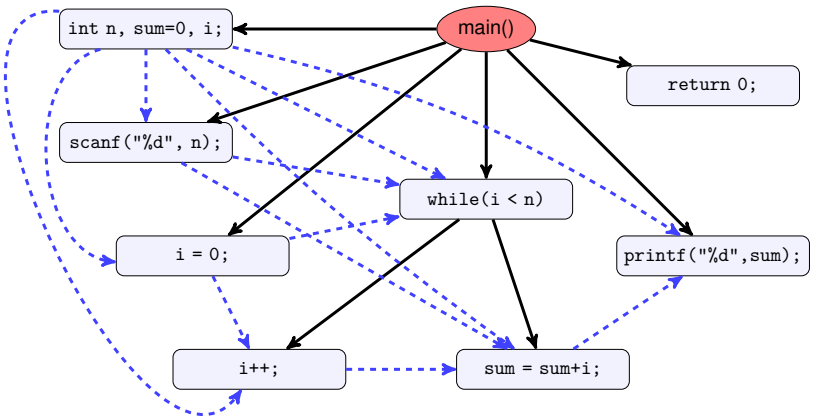
Given a program $P$, the set of all $P$'s dependencies may be viewed as inducing a partial ordering on the program's statements that must be followed to preserve $P$'s semantics [64], i.e., some permutations between $P$'s statements may result in a program with similar semantics or may break some dependence of $P$ which results in a program with different semantics.

**Definition 30** (**Program Dependence Graph (PDG)**)**.** A *program dependence graph (PDG)* is the graph representation of a program in which the nodes represent statements and predicate expressions, and the edges represent both the data dependencies and the control conditions on which the execution of the operations depends [64].

An example of a PDG can be found in Figure 2.4. Figure 2.4b shows the PDG representation of the program presented in Figure 2.4a. On this graph, each node corresponds to a program statement. The

```c
int main(){
    int n, sum=0, i;
    scanf("%d", &n);
    i = 0;
    while(i < n) {
        i++;
        sum = sum+i;
    }
    printf("%d\n",sum);
    return 0;
}
```

(a) Program that computes $\sum_{i=1}^{n} i$.




(b) PDG of the program on the left.

Figure 2.4: Program that computes $\sum_{i=1}^{n} i$, and its program dependence graph (PDG).

blue dashed arrows depict the data dependencies (see Definition 25) between two program statements. On the other hand, the black arrows represent the basic block that the program statement depends on.

## 2.4 Synthesis and Repair

**Definition 31** (**Synthesis Problem**)**.** Given $(S, G,$ Ops$)$, being $S$ a program's specification (e.g., input-output examples), $G$ a context-free grammar, and Ops the semantics for a particular DSL, the goal of *synthesis* is to infer a program $\mathcal{P}$ such that (1) the program is produced by $G$, (2) the program is correct concerning Ops and (3) $\mathcal{P}$ is consistent with $S$ [65].

Program Synthesis has even been considered the Holy Grail of Computer Science [66].

A *program synthesizer* is, according to Manna and Waldinger [67], "*a system that takes such a relational description (e.g., input-output examples) and tries to produce a program that is guaranteed to satisfy the relationship*". Program synthesizers search the space of programs described by a given domain-specific language (DSL).

**Definition 32** (**Counterexample Guided Inductive Synthesis (CEGIS)**)**.** CEGIS is an iterative algorithm commonly used in Program Synthesis and Formal Methods to construct programs or solutions that satisfy a given specification [68–70]. CEGIS consists of two steps: the synthesis step and the verification step. Given the specification of the desired program, the inductive synthesis procedure generates a candidate program. Next, the candidate program $P$ is passed to the verification step, which checks whether $P$ satisfies all possible inputs' specifications. Otherwise, the Checker produces a counterexample $c$ from the satisfying assignment, which is then added to the set of inputs passed to the synthesizer, and the loop repeats. The synthesis engine refines its hypothesis using this counterexample to avoid similar mistakes in subsequent iterations. This iterative loop (comprising candidate generation, verification, counterexample generation, and refinement) continues until a correct candidate is found that satisfies all given specifications and constraints.

**Definition 33** (**Programming By Example (PBE)**)**.** Given ($E, G$, Ops), being $E = (E_{in}, E_{out})$ a set of input-output examples, $G$ a grammar and Ops the semantics for a particular DSL, the goal of *Programming by Example* is to infer a program $\mathcal{P}$ such that (1) the program is consistent with $G$, (2) the program is consistent with Ops and (3) $\mathcal{P}(E_{in}) = E_{out}$ [65].

PBE is a special case of Program Synthesis, where the program specification is a set of input-output examples [45, 71–76].

**Definition 34** (**Semantic Program Repair.**)**.** Given $(T, G, O, P)$, let $T$ be a set of input-output examples (test suite), $G$ be a grammar, $O$ be the semantics for a particular Domain-specific language, and $P$ be a syntactically well-formed program (i.e. sets of statements, instructions, expressions) consistent with $G$ and $O$ but semantically erroneous for at least one of the input-output tests i.e., $\exists(t^i_{in}, t^i_{out}) \in T \ : P(t^i_{in}) \neq t^i_{out}$.

The goal of *Semantic Program Repair* is to find a program $P_f$ by semantically change a subset $S_1$ of $P$'s statements ($S_1 \subseteq P$) for another set of statements $S_2$ consistent with $G$ and $O$, such that,

$$P_f = ((P \setminus S_1) \cup S_2)$$

and

$$\forall(t^i_{in}, t^i_{out}) \in T \ : \ P_f(t^i_{in}) = t^i_{out}.$$

Semantic program repair can be performed using a set of program statements from one (resp. several) semantically correct program(s) in the case of implementation-based program repair (resp. clustering-based program repair). Moreover, semantic program repair tools can also perform program synthesis (see Definition 31 and 33) to synthesize the set of program statements $S_2$ needed to fix the erroneous program $P$.

# 3

# Background

*“Computer science is no more about computers than astronomy is about telescopes.”*

– Edsger Dijkstra.

## Contents

This chapter provides some background on the topic of automated program repair. Section 3.1 presents different types of program executions, while Section 3.2 focuses on fault localization techniques. Next, Section 3.3 presents current state-of-the-art on automated program repair, followed by Section 3.4 which covers related work on automated assessment tools. To conclude, Section 3.5 provides an overview of recent work on the field of providing feedback to students in introductory programming courses.

## 3.1 Program Executions

This section provides the notions of several types of program executions. First, Section 3.1.1 presents the traditional concept of concrete execution of a program. Section 3.1.2 introduces symbolic execution. Lastly, Section 3.1.3 shows concolic execution.

The following function `max_three`, presented earlier in Listing 1.1, is reintroduced for the reader's convenience since this function will be used as a running example to explain the different types of program executions.

**Listing 3.1:** max_three, a function that finds and returns the maximum number among three numbers.

```
1  int max_three(int num1, int num2, int num3)
2  {
3      int max = num1;
4      if(num2 > max){
5          max = num2;
6      }
7      if(num3 > max){
8          max = num3;
9      }
10     return max;
11 }
```

### 3.1.1 Concrete Execution

Each program has a sequence of instructions and conditions, i.e., its code. A concrete execution is the sequence of instructions that is executed for a given set of input values assigned to the program's variables. Different inputs may result in a different sequence of instructions.

Consider the program presented in Listing 3.1. Depending on the values assigned to the variables `num1`, `num2` and `num3`, different sequences of this program's instructions may be executed. For example, consider the first input test from the test suite presented in Table 10.1: {`num1=1`, `num2=2`, `num3=3`}. Running the program with these values, all lines of the function `max_three` are executed since $num1 < num2 < num3$. However, if we consider the second test in Table 10.1, {`num1=-1`, `num2=-2`, `num3=-3`}, both if-conditions on lines $4$ and $7$ fail then lines $5$ and $8$ are not executed on this concrete execution.

### 3.1.2 Symbolic Execution

Symbolic execution tools assign symbolic values for inputs instead of assigning the actual inputs as a concrete execution of the program would. Symbolic execution [31, 77, 78] is a technique for program analysis that explores all possible execution paths of a program. Instead of using concrete values for inputs, symbolic execution tools assign symbolic values to input variables, allowing the exploration of multiple execution paths simultaneously. Then the program is executed symbolically on these symbolic inputs. A symbolic execution tool builds a first-order formula by collecting symbolic path constraints and using a theorem prover to verify if a given branch is feasible. The first-order formula is built by connecting: (1) the variables' symbolic values with the expressions the program executes and (2) the conditional branches' constraints that represent the program's several outcomes.

Each time a conditional branch is encountered, the symbolic execution tool forks the current symbolic path into two. The first path corresponds to the conditional expression's then-block, represented by the conjunction of the current symbolic formula and the conditional expression. The second path corresponds to the else-block, represented by the conjunction of the symbolic formula and the negation of the conditional expression.

For example, consider again the function `max_three` presented in Listing 3.1. A symbolic execution framework would find four different symbolic paths for this function as presented by the symbolic tree in Figure 3.1. Each path is defined by its own symbolic formula as follows:

- P1 $\longleftarrow$ `(num1 < num2)` $\wedge$ `(num2 < num3)`;
- P2 $\longleftarrow$ `(num1 < num2)` $\wedge \neg$`(num2 < num3)`;
- P3 $\longleftarrow \neg$`(num1 < num2)` $\wedge$ `(num2 < num3)`;
- P4 $\longleftarrow \neg$`(num1 < num2)` $\wedge \neg$`(num2 < num3)`;

A few examples of symbolic execution engines are KLEE [79] for the C language, Symbolic PathFinder [80, 81] for the Java language and Cosette [82] for the JavaScript language. Symbolic execution has various applications, from the automatic generation of high-coverage tests [79] to program verification [83]. However, there are two significant issues with symbolic execution. First, it does not scale for large programs and cannot solve all the constraints that may be generated.

### 3.1.3 Concolic Execution

Concolic execution (a portmanteau of **conc**rete and symb**olic**) is a hybrid approach of a symbolic execution using actual inputs as concrete execution techniques instead of using symbolic values like symbolic execution approaches. The idea is to explore all execution paths of the program being analyzed one by one. Firstly, the concolic execution engine randomly chooses input values for the input variables. Afterward, executes the program concretely on those values and saves the path constraints taken by this concrete execution. Then, the concolic engine builds a first-order formula with these path constraints, negates this formula and asks a solver for another interpretation for the program's variables. If such

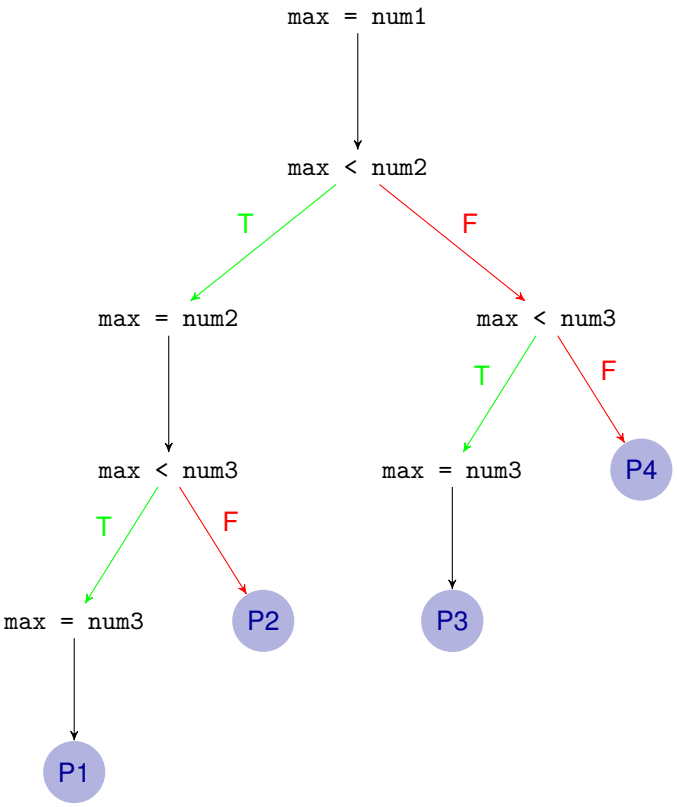



Figure 3.1: Symbolic execution tree for function `max_three(int num1, int num2, int num3)`, presented in Listing 3.1.

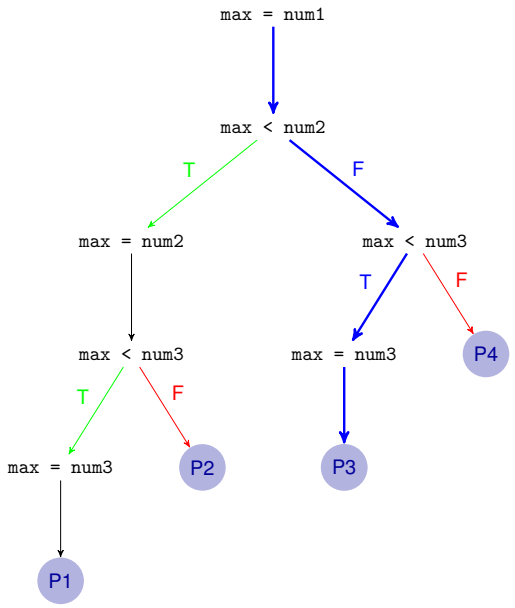
interpretation exists, then it necessarily forces the program to explore a new execution path different from the ones taken so far, described by the first-order formula.

For example, consider the function `max_three` presented in Listing 3.1. Assume that a concolic engine randomly assigns `num1 = 7`, `num2 = 5` and `num3 = 10`. Then, as Figure 3.2a shows, the concrete execution using these values takes program through the execution path `P3`. Afterward, the concolic engine construct the following formula for this path:

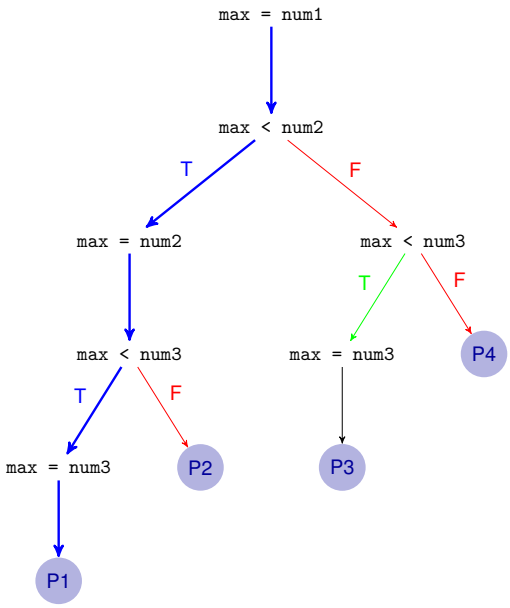
$$P3 = \neg(num1 < num2) \wedge (num1 < num3) \tag{3.1}$$

To explore another execution path, the concolic engine asks the solver a model for the negation of this formula, i.e., $\neg P3$. Let `num1 = 4`, `num2 = 5` and `num3 = 10` be the second model returned by the solver. Figure 3.2b shows that if we concretely execute function `max_three` using these input values, a different execution path is explored which is `P1`. This interaction with the solver continues until every execution path is explored. Thus, the different collection of input tests generated by a concolic execution engine, one for each execution path, is a high-coverage set of tests [79].

There are two main reasons for the rise of concolic execution. Firstly, every time the symbolic execution tool hits a conditional expression, it has to interact with a solver to decide if both branches (then-block and else-block) are feasible paths. Secondly, symbolic execution calls for implementing a symbolic interpreter for a programming language. A few examples of well-known concolic execution engines are CREST [84, 85] (formerly known as CUTE [86]), jCUTE [87], DART [88] and SAGE [89].




(a) If `num1 = 7, num2 = 5` and `num3 = 10`. The execution path is P3 (highlighted in blue).




(b) If `num1 = 4, num2 = 5` and `num3 = 10`. Now the execution path is P1 (highlighted in blue).

Figure 3.2: Concolic execution (two iterations) for function `max_three(int num1, int num2, int num3)`, presented in Listing 3.1.

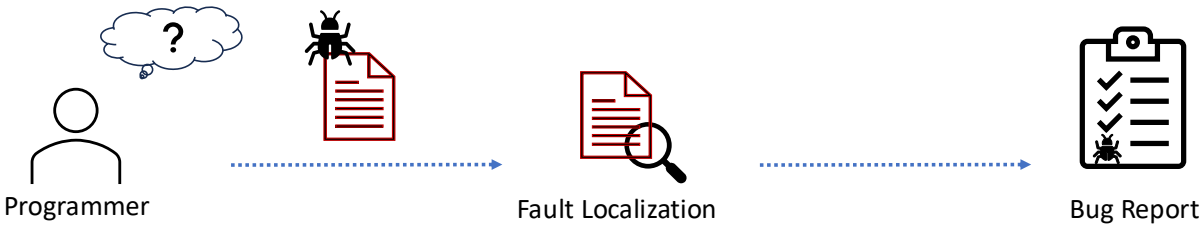



Figure 3.3: Fault Localization (FL).

## 3.2 Fault Localization (FL)

Localizing system faults has always been one of the most time-consuming and expensive tasks in software development. Given a buggy program, *fault localization* (FL) involves identifying locations in the program that could cause a faulty behaviour (bug), as depicted in Figure 3.3. Fault localization (FL) techniques typically fall into two main families: *spectrum-based (SBFL)* and *formula-based (FBFL)*. The following sections present each of these families. Additionally, *program slicing* [90–92] has also emerged as a technique for localizing faults within programs.

### 3.2.1 Spectrum-Based Fault Localization (SBFL)

*Spectrum-Based Fault Localization* (SBFL) methods [93–98] estimate the likelihood of a statement being faulty based on test coverage information from both passing and failing test executions. While SBFL techniques are generally fast, they may lack precision, as not all identified statements are likely to be the cause of failures [91, 99].

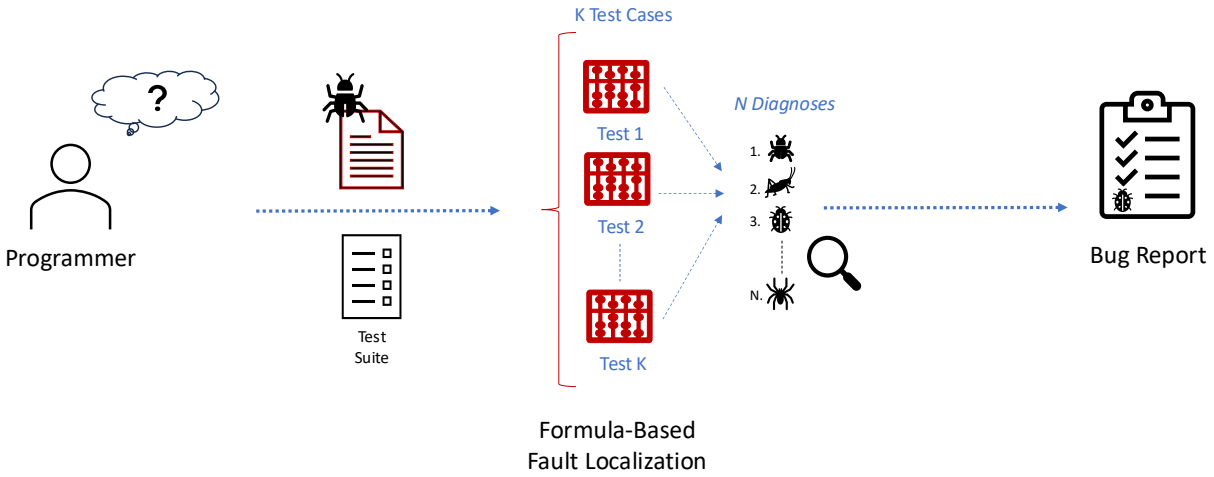



Figure 3.4: Formula-Based Fault Localization (FBFL).

### 3.2.2 Formula-Based Fault Localization (FBFL)

*Formula-Based Fault Localization (FBFL)* approaches [62, 100–107] are considered exact. As shown in Figure 3.4, FBFL methods encode the fault localization problem into several optimization problems aimed at identifying the minimum number of faulty statements within a program. Typically, these methods perform a MaxSAT call for each failing test, allowing them to individually identify a minimal set of faults (diagnoses) for each failing test case rather than simultaneously addressing all failing test cases. Then, FBFL methods apply their own aggregation techniques to all enumerated faults to determine the minimal set of faults so they can report it back to the users. For instance, BUGASSIST [62] prioritizes faults based on their occurrence frequency, while SNIPER [101] computes the Cartesian product of all faults and then sorts the resulting sets of aggregated faults.

A more syntactic FBFL approach [91] is to use program slicing to enumerate all minimal sets of repairs for a given faulty program. Another method for identifying the causes of faulty program behaviour involves analyzing the variances between various versions of the software [92]. *Refinement* has a long-standing tradition in verification; particularly for refining abstractions of reachable states [108–110]. In that sense, our form of refinement is different because it enables us to more precisely pinpoint faults of the user, at the sub-expression level.

Typically, FBFL approaches encode the localization problem as a *Model-Based Diagnosis* (MBD) problem using a MaxSAT encoding.

**Model-Based Diagnosis (MBD).** The following definitions are commonly used in the *MBD* theory [16, 54, 61, 63]. A system description $\mathcal{P}$ is composed of a set of components $\mathcal{C} = \{c_1, \ldots, c_n\}$. Each component in $\mathcal{C}$ can be declared healthy or unhealthy. For each component $c \in \mathcal{C}$, $h(c) = 0$ if $c$ is unhealthy, otherwise, $h(c) = 1$. As in prior works [54, 111], $\mathcal{P}$ is described by a CNF formula, where $\mathcal{F}_c$ denotes the encoding of component $c$:

$$\mathcal{P} \triangleq \bigwedge\nolimits_{c \in \mathcal{C}} (\neg h(c) \vee \mathcal{F}_c) \tag{3.2}$$

Observations represent deviations from the expected system behaviour. An observation, denoted as $o$, is a finite set of first-order sentences [54, 61], which is assumed to be encodable in CNF as a set of unit clauses. In this work, the failing test cases represent the set of observations.

A system $\mathcal{P}$ is considered faulty if there exists an inconsistency with a given observation $o$ when all components are declared healthy. The problem of model-based diagnosis (MBD) aims to identify a set of components which, if declared unhealthy, restore consistency. This problem is represented by the 3-tuple $\langle \mathcal{P}, \mathcal{C}, o \rangle$, and can be encoded as a CNF formula:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C}} h(c) \models \bot \tag{3.3}$$

For a given MBD problem $\langle \mathcal{P}, \mathcal{C}, o \rangle$, a set of system components $\Delta \subseteq \mathcal{C}$ is a diagnosis iff:

$$\mathcal{P} \wedge o \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{3.4}$$

A diagnosis $\Delta$ is minimal iff no subset of $\Delta$, $\Delta' \subsetneq \Delta$, is a diagnosis, and $\Delta$ is of minimal cardinality if there is no other diagnosis $\Delta'' \subseteq \mathcal{C}$ with $|\Delta''| < |\Delta|$.

A diagnosis is redundant if it is not subset-minimal [54].

To encode the Model-Based Diagnosis problem with one observation with partial MaxSAT, the set of clauses that encode $\mathcal{P}$ (3.2) represents the set of hard clauses. The soft clauses consists of unit clauses that aim to maximize the set of healthy components, i.e., $\bigwedge_{c \in \mathcal{C}} h(c)$ [63, 112]. This MaxSAT encoding of MBD enables enumerating minimum cardinality diagnoses and subset minimal diagnoses, considering a single observation. Furthermore, a minimal diagnosis is a minimal correction subset (MCS) of the MaxSAT formula (see Definition 8). Given an inconsistent formula that encodes the MDB problem (3.3), a minimal diagnosis $\Delta$ satisfies (3.4), thereby making $\Delta$ an MCS of the MaxSAT formula. BUGASSIST [62], SNIPER [101], and other model-based diagnosis (MBD) tools for fault localization in circuits [54, 63, 112] encode the localization problem with partial MaxSAT.

More recently, the MaxSAT encoding for MBD [54] has been generalized to multiple inconsistent observations. Let $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$ be a set of observations. Each observation is associated with a replica $\mathcal{P}_i$ of the system $\mathcal{P}$. The system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. For a given observation $o_i$, a diagnosis is given by the following:

$$\mathcal{P}_i \wedge o_i \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \not\models \bot \tag{3.5}$$

The goal is to find a minimal diagnosis $\Delta \subseteq \mathcal{C}$, such that $\Delta$ is a minimal set of components when deactivated the system becomes consistent with all observations $\mathcal{O} = \{o_1, \ \ldots \ o_m\}$. Moreover, when considering multiple observations, an aggregated diagnosis is a subset of components that includes one possible diagnosis for each given observation.

## 3.3 Automated Program Repair (APR)

A theme of research since the 80's [113, 114], one of the main challenges in *Automated Program Repair* for IPAs is to provide valuable and personalized feedback to students. Usually, this personalized feedback is provided as a list of possible repairs to a student's program. Automated program repair can be of two different kinds: syntactic repair and semantic repair. Syntactic repairs provide students with possible fixes to improve their programs that do not compile i.e., syntax errors. Semantic repairs are fixes that improve a program's behavior by checking the program's performance on a set of tests provided by the lecturer or by comparing the student's program against a reference solution using program analysis.

The following sections present current state-of-the-art syntactic and semantic automated program repair techniques and how these approaches can be used to provide feedback to students in introductory programming exercises.

### 3.3.1 Syntactic Program Repair

Several syntactic errors arise because of the programmer's inexperience or lack of attention to detail, which causes compilation errors. These errors typically include missing declarations, missing delimiters (e.g., braces), and type errors [3], which is usually the first issue in introductory programming courses. The novice programmers struggle to comprehend the compiler errors messages [115]. In this section, we will present some of the most recent work on syntactic program repair.

#### 3.3.1.1 Sequence-to-Sequence Models

DEEPFIX [13] was developed by leveraging the structural similarities between natural languages and programming languages. Both exhibit rich syntactic structures, and just as grammatical errors occur in natural language, syntactic errors arise in programming languages. DEEPFIX approaches the problem of syntactic program repair in the same way as machine translation, treating it as a sequence-to-sequence learning problem where an erroneous program is transformed into a corrected one.

DEEPFIX employs a multi-layered, sequence-to-sequence neural network with attention mechanisms to fix common syntactic errors in the C programming language. The model consists of an encoder recurrent neural network (RNN) to process the input (an erroneous program) and a decoder RNN with attention to generate the output (a syntactically correct program). As shown in Figure 3.5, DEEPFIX operates as an end-to-end solution: given a program that fails to compile, it returns a version that compiles successfully, without any external frameworks to assist in error detection and correction. The only external tool used is a compiler, acting as an oracle to validate the final programs correctness and ensure it compiles without errors.

Other works also use sequence-to-sequence models to repair syntax errors on programming assignments [18, 19, 116]. TRACER [116] is similar to DEEPFIX but focuses only on single-line errors, while DEEPFIX can fix multiple-line errors. SYNFIX [18] and SK_P [19] provide repairs for syntactic errors for particular programming assignments. SK_P is capable of generating syntactic and semantic repairs for

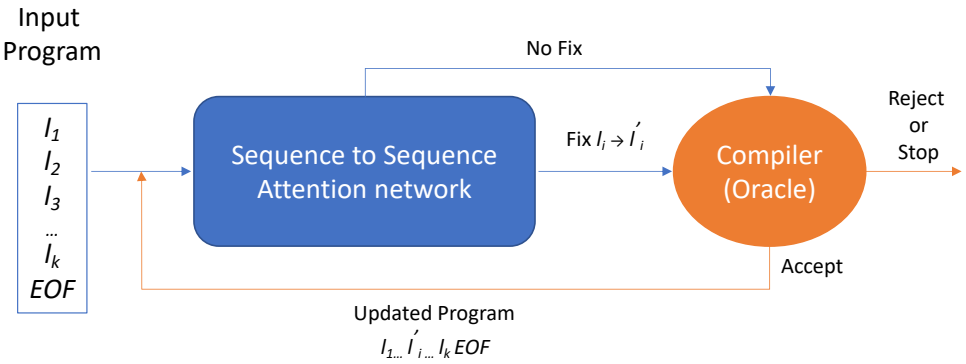



Figure 3.5: The iterative repair strategy of DEEPFIX [13].

a given program. The authors of SK_P try to learn task-specific patterns for a specific programming assignment by training SK_P on correct student solutions for the same assignment. SYNFIX combines a sequence-to-sequence model with a constraint-based synthesis step for syntactic and semantic program repair. Given a set of correct students' solutions for a particular programming assignment, SYNFIX, like DEEPFIX, is trained to learn an output program, a syntactically-fixed program. Afterward, SYNFIX uses a well-known constraint-based synthesis, Sketch [66], to find the minimal set of semantic repairs needed to repair the program semantically. These frameworks, SK_P and SYNFIX are trained using correct students' submissions of the same problem. Therefore, these techniques cannot be used for a new programming assignment without any correct submissions since there is no training data.

TEGCER [117] is an automated feedback tool for novice programmers. This tool uses supervised learning to match *compilation errors* in new code submissions with pre-existing errors submitted by previously enrolled students. TEGCER only works for syntactic errors. TEGCER also performs variable renaming to each program. This tool renames the variables with their generic types using the LLVM [118], a standard static analysis tool. Moreover, CLACER [119] is a neural network model designed to classify compilation errors by proposing categories based on program tokens, aiming to improve localization effectiveness and prediction performance, thereby enhancing the students' learning process.

**Large Language Models (LLMs).** Large Language Models (LLMS) trained on code (LLMCS) have demonstrated significant effectiveness in generating program fixes [21, 23, 38–41]. For instance, RING [21] is a multilingual repair engine powered by an LLMC that uses fault localization information from error messages and leverages the few-shot capabilities of LLMCS for code transformation. In the context of Automated Program Repair (APR) for programming education, several works have explored the use of LLMS for coding tasks [17, 20, 22]. PyDex [20], for example, employs iterative querying with CODEX, an LLMC version of ChatGPT, using test-based few-shot selection and structure-based program chunking to repair syntax and semantic errors in Python assignments. Similarly, CODEHELP [17] utilizes OpenAI's LLMS to provide textual feedback to students on their assignments. However, to the best of our knowledge, no existing work has explored the use of LLMS guided by formula-based fault localization.

#### 3.3.1.2 Abstract Syntax Tree Differences

Mesbah et al. [14] also tackled the problem of syntactic program repair with a Neural Machine Translation approach like DEEPFIX. Their technique, DEEPDELTA, focuses on learning Abstract Syntax Tree (AST) changes instead of learning whole fixed programs. DEEPDELTA's authors designed a new domain-specific language called Delta that encodes changes to a program's AST. DEEPDELTA receives as input the compiler feedback information about the erroneous program and returns as output the set of changes to the program's AST, expressed in Delta. These Delta changes are the repairs needed to fix the syntactic problems identified by the compiler.

DEEPDELTA is trained using developer data on compilation errors. The authors collected examples of erroneous programs and the human-authored code changes that cause those failing programs to correct syntactic programs at Google. Afterward, they generate the difference between the ASTs of erroneous and fixed programs. Therefore, DEEPDELTA is trained using as input the compiler feedback message and as output this difference between the erroneous and correct programs ASTs, written in DEEPDELTA's language Delta.

#### 3.3.1.3 Reinforcement Learning

More recently, Gupta et al. [15] proposed RLASSIST, a Deep Reinforcement Learning agent that fixes syntactic errors in introductory programming assignments. Usually, a programmer goes through the program code in order to find the location of a given error when faced with one. When she finds the bug location, she makes the necessary edits to the code to fix the bug.

RLASSIST is a reinforcement learning agent that mimics an experienced programmer's actions by accessing and modifying the program code. As demonstrated in Figure 3.6, given the incorrect program presented in Listing 1.2, RLASSIST goes through the program code. When it finds a possible location for the bug, RLASSISTs modifies that code segment. After each modification, RLASSIST asks a compiler, used as a black box, to validate the fixes proposed, i.e., checks if the number of error messages decreased. The reward function used to train this reinforcement learning agent is a function that tries to minimize the number of compiler error messages.

#### 3.3.1.4 Graph Neural Networks

DRREPAIR [3] is a graph-based program repair framework that uses a graph neural network to fix a syntactically incorrect program. Given an erroneous program and a compiler error message, DRREPAIR builds a program-feedback graph to localize the buggy line in the program. As shown in Figure 3.7, this graph connects a program to the diagnostic feedback provided by a compiler (given as input). It takes as nodes all identifiers from the erroneous program code and the symbols that appear in the compiler message and, to encode semantic correspondence, connects instances of the same symbol. Afterward, Yasunaga and Liang [3] design a graph neural network using this program-feedback graph that connects symbols relevant to program repair in the code and compiler feedback. This neural network models the reasoning process.

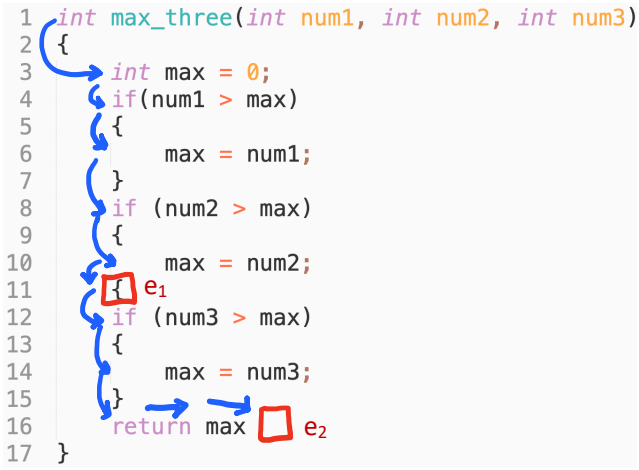



Figure 3.6: The erroneous program presented in Listing 1.2 and the sequence of actions taken by a trained RLAssist [15] agent to fix it: The error locations are highlighted in the red color. The arrows show how the agent navigates over the program text. The edit actions are marked by $e_1$ and $e_2$.

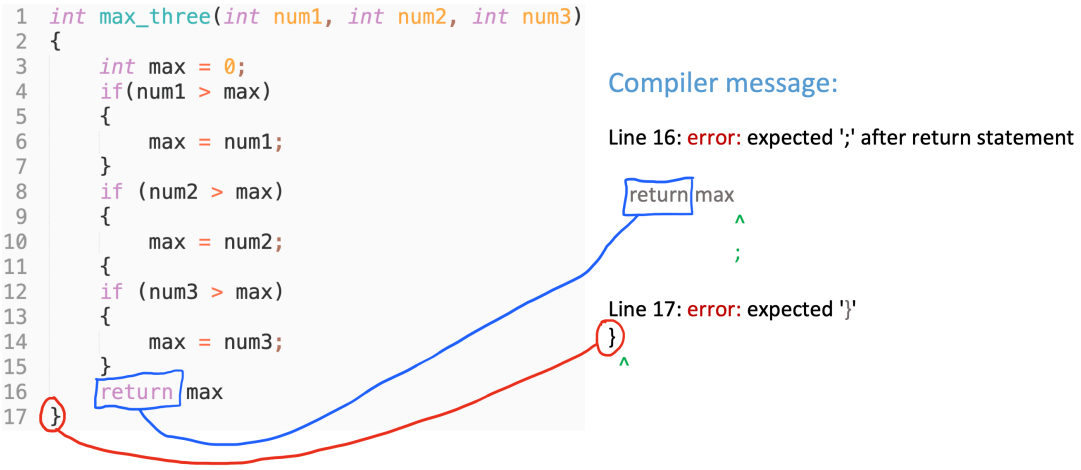



Figure 3.7: Given a broken program and diagnostic feedback (compiler error message), the goal of DrRepair [3] is to localize an erroneous line and generate a repaired line. On the left is the erroneous program presented in Listing 1.2, and on the right the compiler error message returned by gcc.

Table 3.1: Syntactic Tools

| Tools | Task Specific | Multi-line Debug | Approach | Programming Language | Code Available |
|---|---|---|---|---|---|
| SK_P [19] (2016) | Yes | No | Seq-2-Seq | Python | No |
| DeepFix [13] (2017) | No | Yes | Seq-2-Seq | C | Yes |
| TRACER [116] (2018) | No | No | Seq-2-Seq | C | No |
| SynFix [18] (2018) | Yes | No | Seq-2-Seq | Python | No |
| DeepDelta [14] (2019) | No | Yes | Seq-2-Seq | Java | No |
| RLAssist [15] (2019) | No | Yes | RL | C | Yes |
| DrRepair [3] (2020) | No | Yes | GNNs | C | Yes |

#### 3.3.1.5 Discussion

This section discusses the different aspects and drawbacks of various techniques presented in this document for syntactic automated program repair.

Table 3.1 shows several aspects of recent syntactic repair tools. SK_P, SynFix, and TRACER

are sequence-to-sequence models that can fix individual bugs, i.e., single-line bugs. Therefore, these tools only work on programs that are almost syntactically correct. DEEPFIX and DEEPDELTA are also sequence-to-sequence models, although these tools can fix multi-line bugs. SK_P and SYNFIX are task-specific, i.e., are trained for a specific introductory programming assignment and cannot be used to provide feedback for other types of programming assignments. Between these five sequence-to-sequence tools, only DEEPFIX is an open-source tool available on Bitbucket[1]. RLASSIST is also available online on Bitbucket[2]. However, without a GPU, DEEPFIX and RLASSIST require a lot of training hours to work since DEEPFIX's and RLASSIST's authors do not provide the learned weights of the neural networks used for these frameworks' papers.

Syntactic program repair can be used in introductory programming assignments to grade syntactical submissions and not to simply provide feedback to students. REFAZER [12] is a tool that learns syntactic program transformations. The authors of REFAZER performed a study in an introductory programming course. REFAZER learns program transformations using previous students' incorrect and correct submissions and then uses these transformations to correct new submissions.

### 3.3.2 Semantic Program Repair

Several semantic program repair techniques have been proposed to check if a student's program is semantically correct, i.e., if a given implementation has the behaviour expected by a lecturer. There are several approaches to semantic program repair: semantic-based [4, 5, 7], solution-driven [2, 9–11, 120, 121], semantic code search [6, 122–126], static analysis violations [27, 28] and generate-and-validate (a.k.a heuristic-based) techniques [32, 33, 127].

The typical way to provide feedback to students on their programming assignments is to provide them with a failing test case, i.e., a test case on which their program returns an erroneous output. This type of feedback is quite helpful since it mimics how experienced programmers develop their code [2]. This section presents test-driven approaches for semantic program repair: semantic-based, solution-driven, and semantic code search approaches. Section 3.3.2.1 presents semantic-based methods that verify a program's behaviour considering only a collection of tests provided by the lecturer. Then, these techniques try to synthesize a set of patches (repairs) that fix a program's behaviour considering the set of tests. Section 3.3.2.2 presents solution-driven methods that compare a program's control flow with one or several solutions provided by the lecturer, propose patches based on the difference between the control flow of both programs, and check those repairs on a set of tests. Lastly, Section 3.3.2.3 presents semantic code search techniques that use a code snippets database. These methods query the database for code based on a particular specification of desired behaviour. Given an input-output example, these techniques search for a code fragment that produces the expected output on the given input and fix the student's program with this code fragment.

The interested reader is referred to the literature [24, 25] for more details about generate-and-validate and static analysis violations techniques.

[1] https://bitbucket.org/iiscseal/deepfix
[2] https://bitbucket.org/iiscseal/rlassist

#### 3.3.2.1 Semantic-based Methods

This section presents semantic-based approaches for semantic software repair such as SemFix [4], DirectFix [5] and Angelix [7]. These are techniques developed for real-world software and not for IPAs. At the end of this section, a study [8] on applying these techniques to an educational setting is discussed. The general idea of these techniques is to find the likely buggy region of a program given a test suite and synthesize a repair.

**SemFix [4] and DirectFix [5]** SemFix uses fault localization frameworks to rank the program's statements by suspiciousness. Then, SemFix tries to generate a fix by replacing this statement with a symbolic expression [31] that represents a generic value (see Section 3.1.2). SemFix runs the program with this symbolic expression on the provided test-suite. The whole program is executed concretely until the modified expression, and then it is executed symbolically [31]. With these executions, one for each available test on the test suite, SemFix can generate a set of constraints representing the conditions that the symbolic expression must satisfy to pass all tests. Although, SemFix is only able to fix single-line bugs. DirectFix makes use of a MaxSMT formula to synthesize the minimal set of repairs required to make a program pass all tests on a given test suite. In addition, DirectFix encodes a trace formula of the whole program to fix more than one line at once, if necessary. However, this ability to fix multi-line bugs makes DirectFix significantly less scalable than SemFix [7].

**Angelix [7]** tries to fix multi-line bugs without giving up scalability. This is possible using angelic values, angelic states, angelic paths, and angelic forests. An angelic value is the expected value an expression should return to pass a given test. An angelic state is the set of variables that are visible in the scope of the expression being analyzed. An angelic path is encoded as a triple containing the faulty expression and its respective angelic value and state, these paths can be achieved by symbolic execution of programs [31]. Finally, an angelic forest is the set of angelic paths that encode a repair problem. Angelix uses these forests to synthesize multi-line fixes.

**Automated Software Repair Techniques in Introductory Programming Assignments** Yi et al. [8] did a feasibility study on using automated program repair techniques, used to fix bugs on large real-world software, in introductory programming assignments. In student solutions that only failed a small number of tests, automated software repair tools such as Angelix [7] were applied. However, with this study, they realized that directly applying a technique from the real world in an educational setting would not be as good as one might think. Yi et al. [8] argue that the main reason is that the student program is normally more incorrect and fails more tests on a given test suite compared to experienced programmers debugging real-world software. Usually, significant changes are needed to fix a student program [8].

Yi et al. [8] achieved good results applying these software repair techniques on programming assignments by generating hints for students for each failing test and not trying to repair the whole program at once. Yi et al. [8] did this by changing these frameworks' repair policies. For a given erroneous program, a set of repairs are suggested to the student if (1) the repaired program passes at least one more test

and (2) all the tests passed by the student program are also passed by the repaired program. Therefore, this is a good way of applying software repair techniques in introductory programming assignments. Gao et al. [128] also perform a study with students using industrial automated testing on a framework capable of providing feedback to students and also achieved good results.

#### 3.3.2.2 Solution-driven Methods

Semantic-based program repair techniques may be overfitted to the test suite unavoidably providing incorrect patches to a program [27, 129]. Another problem with semantic-based approaches is that building a comprehensive collection of representative tests may be impracticable to obtain and may also not completely express the behavior expected by the lecturer to the reference solution.

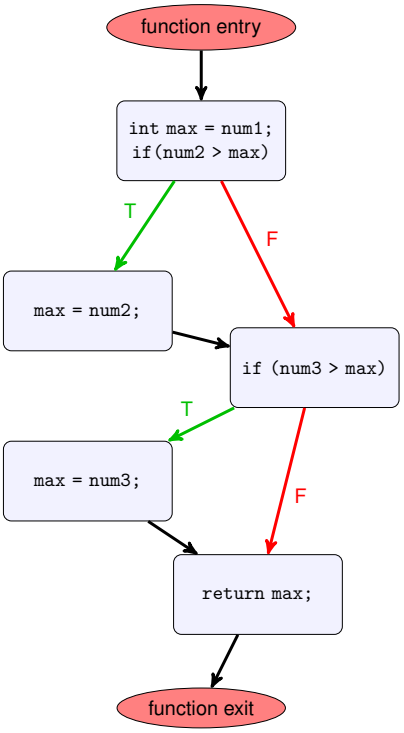
This section presents solution-driven approaches for semantic program repair. These approaches fix a given incorrect solution by making its control flow similar to a given correct solution for the same programming assignment. Solution-driven program repair techniques can be divided into two leading families: implementation-based and clustering-based. The following sections will present both families of approaches.

**Implementation-based techniques**

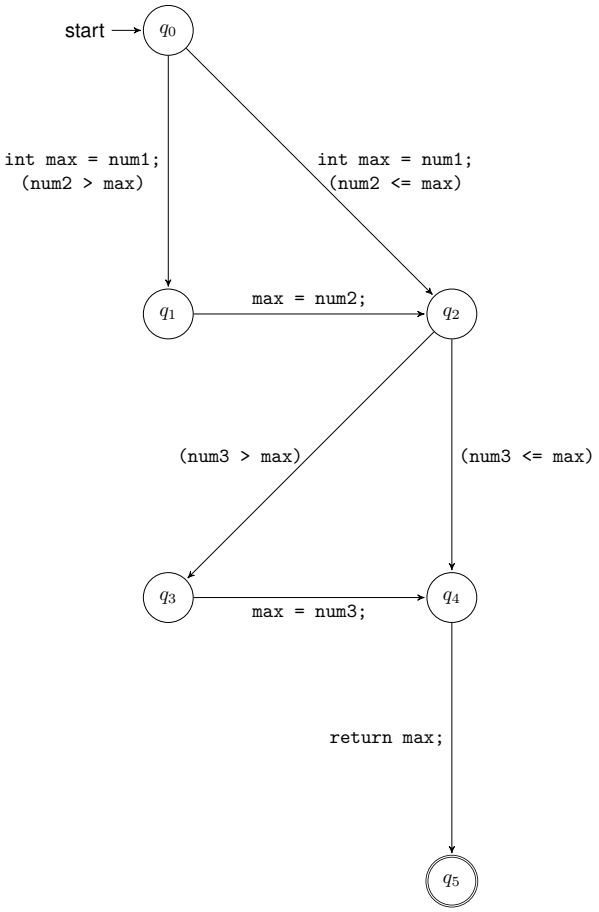
Some program repair tools [9, 121] use a single reference implementation provided by the lecturer to repair a student's program. These tools usually are only able to use one correct implementation to repair each program.

**Verifix [9]** is an implementation-based method that aligns an incorrect program with the lecturer's reference solution into an automaton. Using this alignment relation and MaxSMT solving, Verifix suggests fixes to the incorrect submission. Ahmed et al. [9] stated in their paper that clustering-based program repair tools take advantage of other students' correct submissions for repairing incorrect submissions since it is challenging to generate feedback when the control-flow graph (see Definition 29) of the student program is different from the instructor's reference program. Furthermore, the authors also express that other repair tools, when collecting correct students' submissions, only check if those submissions pass a given set of input-output tests and do not verify their correctness against the lecturer's implementation. Hence, Verifix proposes to overcome these drawbacks with verified repair, i.e., constantly repairing a given incorrect student submission with a single correct reference implementation provided by the lecturer by making the student's submission semantically equivalent to the reference solution.

Verifix starts by modeling the reference implementation provided by the instructor and the incorrect student submission as a *Control Flow Automata (CFA)* as Figure 3.8b shows which is essentially a control-flow graph (Figure 3.8a) with code statements labeling the edges of the graph. Hence, a CFA is a graph where the nodes represent control-flow decisions, and the edges represent guarded code commands. Afterward, Verifix aligns both CFA using a syntax-based alignment technique, i.e., aligns control entry nodes, control exit nodes, loop entry/exit nodes. For this syntax-based alignment, Verifix




(a) Control flow graph of function `max_three`.




(b) Control flow automata of function `max_three`.

Figure 3.8: CFG and CFA representations of function `int max_three(int num1, int num2, int num3)` presented earlier in Listing 3.1.

requires a bijective relation between the set of variables of both programs. Hence, the incorrect program must have the same number of variables as the lecturer's implementation. There must also be a bijective relation between the types of both programs' sets of variables. Having such an aligned automaton, the verification phase is to check if each pair of aligned edges are similar, i.e., are semantically equivalent. If not, then there is no semantic equivalence between both programs, and a repair is needed.

Verifix's repair process is based on the aligned CFA of both programs. The verification step to check if a pair of aligned edges are equivalent is translated into an SMT formula. If this SMT formula is satisfiable, then the generated satisfying assignment corresponds to a counter-example why both edges are not equivalent. Verifix performs a repair based on program synthesis using these counter-examples, more concretely a counter-example-guided inductive synthesis (CEGIS) [130]. This program synthesis strategy synthesizes a fix for the incorrect program's node that rules out the counter-example generated previously. Verifix searches for a minimal repair for each aligned edge under consideration, i.e., the repair modifies the minimum number of the program's expressions. This is done by formulating the problem as a MaxSMT problem. This way, the minimal repair preserves the maximum number of original expressions of the incorrect student submission. Given a likely buggy program scope (an edge of the CFA), Verifix starts by replacing this scope's expressions with unique holes to synthesize, using

CEGIS patches for each expression. The program space, i.e., space of possible patch candidates to replace each expression, contains the original expression. Verifix tries to keep the original expressions for as many holes as possible by giving a higher weight penalty to use a new synthesized expression by the CEGIS over the original expression of the incorrect student's submission. This assignment of weights guarantees that the MaxSMT solver prefers the original expressions in the incorrect submission.

Verifix's main drawbacks are:

- Verifix always aligns a given incorrect submission with the same reference implementation provided by the lecturer. Therefore, the set of repairs proposed by Verifix is not guaranteed to be minimal. Perhaps using a correct implementation submitted by another student, the set of fixes required to fix the incorrect program would be smaller and more straightforward.
- The syntax-based alignment technique used by Verifix requires a bijective relation between the set of variables of both programs. Thus, if a student uses more or fewer variables than the instructor's implementation, Verifix cannot repair the student's program.
- To perform verified repair, Verifix translates code into SMT logical formulae, although only certain features of the C language are implemented. Hence, Verifix is only able to repair simple introductory imperative C programs and cannot be used to repair other languages or more complex C programs (e.g., programs with multi-dimensional arrays).
- One of Verifix's main objectives is to overcome the main drawback of other repair tools that make the same control-flow assumption, i.e., assume that there is always a correct implementation with the same control-flow graph as the incorrect submission. Verifix supposedly can repair submissions whose control-flow graph is different from the lecturer's program. However, as explained in Section 1.2, Verifix cannot repair programs with a different number of auxiliary functions than the reference implementation.

**AutoGrader [121]** is another implementation-based program repair tool that automatically compares symbolic executions of a student's submission against the executions of a reference solution on a given test suite to grade students' assignments. The novelty of AutoGrader is that when these executions are semantically different, AutoGrader finds potential path differences between the executions of a student's submission and a reference implementation. Afterward, AutoGrader provides feedback to students in the form of counter-examples for each path difference found.

#### Clustering-based techniques

Figure 3.9 shows the generic idea of clustering-based program repair frameworks [2, 10, 34, 131]. These frameworks receive an incorrect student submission, a test suite, and a collection of $N$ correct student submissions for the same IPA. For scalability concerns, these frameworks eliminate dynamically equivalent correct programs with the provided test suite and through clustering techniques, i.e., semantically equivalent solutions. Furthermore, those clustering approaches aggregate the set of $N$ correct

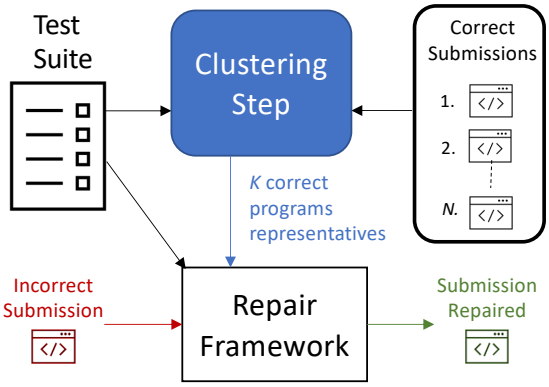



Figure 3.9: Clustering-based Program Repair.

solutions into $K$ semantically different clusters ($N \gg K$). Finally, the repair tool uses these $K$ clusters' representatives to repair the incorrect student submission.

*Program clustering* has also been used to find different semantic solutions for a given programming exercise [131–133]. PaCon [133] clusters programming assignments based on their symbolic analysis. PaCon clusters two submissions together if their path conditions are equivalent. PaCon only takes into consideration a program's semantics. Overcode [132] lets the user visualize and explore different implementations for the same exercise. SEMCLUSTER [34] clusters programs based on their control and data flow features. SEMCLUSTER creates vector representations, *program features vectors* (PFV), for each program using a test suite. This PFV takes into account control flow features as well as data flow features. For each program, SEMCLUSTER counts the number of times each control flow path is used in each test and builds a vector with this data. Afterward, SEMCLUSTER builds another vector containing the data flow features, i.e., the number of occurrences of consecutive values a variable takes during its lifetime. Finally, SEMCLUSTER merges these two feature vectors into a single vector, the PFV.

**CLARA [2]** takes advantage of existing correct students' solutions for a given programming assignment from past years' submissions to present a new student with possible semantic repairs for an erroneous program. Hence, Gulwani et al. [2] use the *wisdom of the crowd* to repair semantically a given erroneous program for a particular introductory programming exercise.

CLARA works as presented in Figure 3.10. For a particular programming assignment, CLARA starts by clustering the set of available correct student submissions from past editions of the programming course. Gulwani et al. [2] cluster these correct programs based on their dynamic equivalence [35] and their control-flow. Afterward, given an incorrect submission, CLARA finds, for each cluster, a set of repairs that makes the incorrect program dynamic equivalent to the cluster of programs. CLARA chooses the minimal set of repairs from all repair candidates (one for each cluster) and returns it to the student. By doing this, Gulwani et al. [2] try to find the set of most similar correct student solutions. Then they present the student with the set of modifications needed to get the student from an incorrect to a valid solution.

Regarding Clara's clustering method, Clara puts two programs into the same cluster if they have

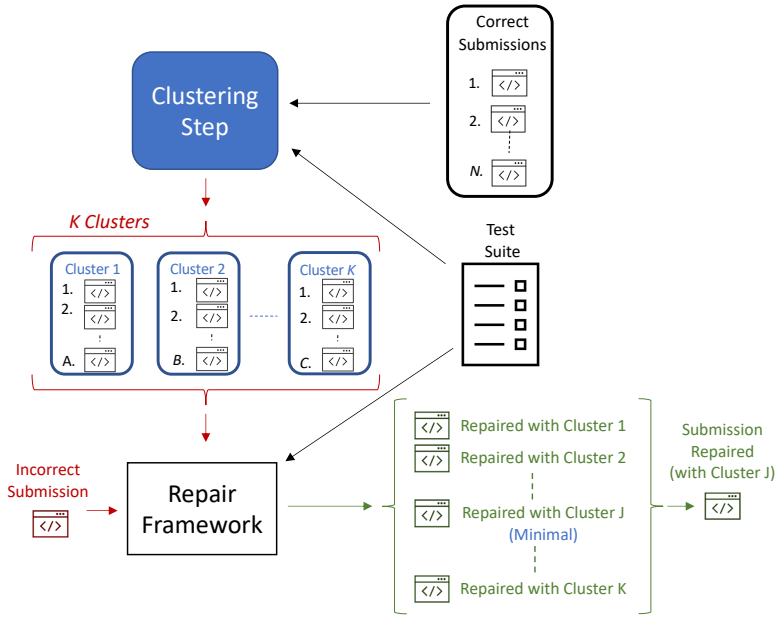



Figure 3.10: Overview of CLARA [2].

the same control flow structure and if there exists a mapping between their variables. Clara requires a perfect match between the two programs' control flow graphs (i.e. branches, loops, functions) and a bijective relation between both programs' variables. Otherwise, Clara returns a structural mismatch error, and those programs are not clustered together. For each computed cluster, Clara keeps all the programs' information (e.g., expressions, variables) that belong to that cluster.

Clara receives one or more correct programs to repair a given incorrect program. This set of input programs can correspond to a set of correct programs provided by the user or to clusters' representatives generated by Clara. If Clara receives a set of clusters' representatives, then Clara should also receive all the information about the programs that are in each cluster to help with the repair process. Clara generates a set of repairs considering each cluster separately. A repair is a program modification that makes an expression of the incorrect program match with some expression of a program in the cluster. Note however that the repairs proposed by Clara only suggest modifications to a program's expressions and cannot modify the program's control-flow. Each repair has a specific cost which is equal to the syntactic difference (tree edit distance [134]) between the incorrect program's AST (see Definition 21) and the repaired program's AST. Clara matches each variable of the incorrect program with a variable of the cluster's representative and generates a set of repairs considering this mapping between these variables. Consequently, considering all the possible repairs between the incorrect program and a cluster's representative, Clara takes advantage of constraint-optimization methods to find a consistent subset of repairs, with the smallest cost, that fixes the incorrect programs and makes it compliant with the test suite.

During Clara's repair process, if none of the correct programs provided has an exact match with the incorrect submission's control flow/looping structure, then Clara is not able to repair the program and returns a *Structural Mismatch* error. Otherwise, Clara gathers the set of repairs using each correct program and returns the minimal set among them.

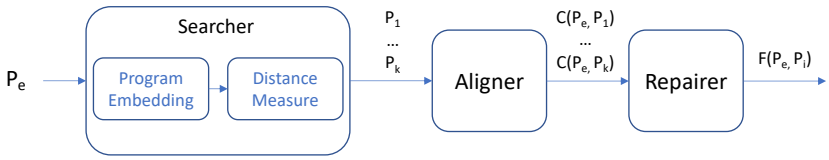



Figure 3.11: Overview of SARFGEN [10].

More recently, Contractor and Rivero [135], Chowdhury et al. [136] improved CLARA's matching algorithm to a new graph matching algorithm that is more relaxed in terms of control flow restrictions. However, this new graph-matching algorithm only works for Python programs.

**SARFGEN [10]** is another clustering-based semantic repair technique. For a given erroneous program, it searches for the most similar correct solution among all correct solutions, aligns both programs, and returns the minimal set of repairs needed to make the erroneous program semantically equivalent to this solution.

SARFGEN, used for C# programs, creates programs embeddings based on the programs' ASTs [137]. Then, given an incorrect program, finds the closest correct submission using those embeddings and tries to repair the program by aligning the variables in both programs. As presented in Figure 3.11, first, SARFGEN identifies similar correct student submissions by embedding a given erroneous program and measures the distance between the generated embedding and the set of embeddings for the available correct students' solutions. Then in the Aligner step, SARFGEN aligns each code segment in the erroneous program with the segments on the correct submissions to suggest possible corrections to the student program. Finally, in the Repairer step, given the set of corrections from the previous step, SARFGEN generates and minimizes the set of fixes needed to perform on the student erroneous program to make it semantically valid.

**Refactory [11]** uses existing correct student solutions, like CLARA [2] and SARFGEN [10], to fix incorrect students' attempts. Although unlike previous approaches, Refactory does not require a significantly large and diverse set of correct student submissions. It still works using only a reference solution provided by the lecturer to a specific programming assignment.

As presented in Figure 3.12, Refactory can be divided into three steps: (1) Refactoring, (2) Structure Alignment, and (3) Block Repair. Firstly using refactoring rules, Hu et al. [11] refactor the set of correct programs to generate additional correct programs with a slightly different control flow but semantically equivalent. These new programs are semantically equivalent to some previous correct program with some mutations on its control-flow graph (see Definition 29) using pre-defined refactoring rules. Figure 3.13 shows an example of a refactored program in Python.

In the second step, Refactory finds which correct program has the same control flow as the erroneous program provided by the student. If there is no correct program with the same control flow, Refactory assumes there is a bug on the student's code control flow and performs a mutation to the student program. This mutation causes modifications to the program's control flow to make it identical to the closest refactored correct program using a tree edit distance.

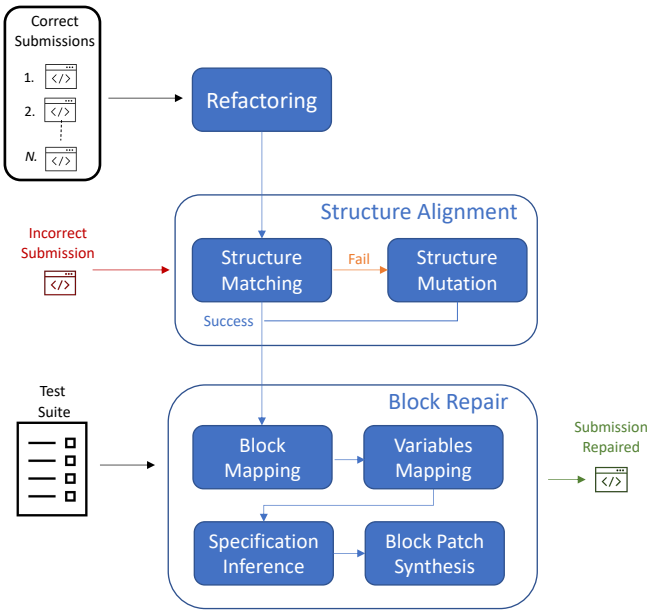



Figure 3.12: Overview of Refactory [11].

```
def sum_until_k(n, k):
    s=0
    for j in range(n):
      if s < k:
        s = j + s


    return s
```

```
def sum_until_k(n, k):
    s=0
    for j in range(n):
      if s < k:
        s = j + s
      else:
        pass
    return s
```

Figure 3.13: Example of a correct program refactored.

In the last step, Refactory maps the refactored correct programs' basic blocks to the erroneous program's basic blocks based on the control-flow graph's isomorphism. With this mapping of blocks and block variables, Refactory can infer input-output specifications of the student program from the refactored programs executions. Refactory then compares each basic block from the incorrect submission with the aligned basic block from the correct program. If the buggy program satisfies the correct program's input-output specification, it is considered correct. If it does not satisfy such specification, then that basic block is deemed to be incorrect. Finally, using search-based synthesis, Refactory modifies the incorrect basic blocks of the student program with a patch capable of making the program pass the given test-suite provided as input.

#### 3.3.2.3 Code Search Methods

Code search methods [6, 122–124, 126, 138, 139] are another family of semantic program repair techniques. Code search uses a specification (e.g., input-output tests) to find code in a large repository that satisfies that specification. To repair a given incorrect program, semantic code search methods do not find the closest correct implementation for the same IPA or use a reference implementation provided by the lecturer. Instead, these methods search for code fragments of other correct programs to repair a given incorrect submission.

**SearchRepair [6]** was proposed to take advantage of the significant corpus of existing open-source code (e.g., GitHub, Bitbucket) to find possible repairs. To this end, SearchRepair uses semantic code search [125], guided by a test suite, over these open-source code or a collection of previous correct students' submissions for a given IPA, to find potential fixes for buggy regions.

SearchRepair constructs a database of various code fragments taken from open-source projects or a collection of correct submissions from a previous edition of an introductory programming course. On this database, these code snippets are encoded, based on the input-output behavior, as SMT constraints for repairing semantically erroneous programs. The code fragments' SMT encoding is the disjunction of the execution paths' constraints.

SearchRepair uses Tarantula [140], a fault localization engine, to find the likely buggy code segments for a given failing program. For each buggy code segment, SearchRepair encodes as an SMT formula the input-output constraint that defines this segment using the failing and passed tests. Then, SearchRepair searches for compatible input-output SMT formulas on its database, i.e., these formulas encode repairs that satisfy the input-output constraint of the buggy segment. Finally, SearchRepair, after applying these repairs, checks if the repaired program passes all tests.

The input-output constraints are represented by the program state (values of the program's variables) before and after each buggy code fragment. Since the authors believe that negative examples/failed tests would encode faulty behavior, only passing tests are used to encode input-output constraints to search for potential patches. SearchRepair searches for a code snippet compliant with these constraints that can serve as a potential patch. If such a patch is found, SearchRepair renames the variables by mapping the incorrect program's variable to the code snippet and replaces the buggy region with this patch.

Regarding the granularity of the code fragments present in the database, SearchRepair assumes that higher-granularity patches are likely to lead to higher-quality patches. Thus, this tool gathers entire blocks of instructions (basic blocks) and sequences of 1 to 5 statements of code.

SearchRepair has three significant drawbacks:

- To find a consistent mapping between the variables of the incorrect program and the variables of the code fragment, SearchRepair includes constraints encoding all possible mappings between these two sets of variables;
- SearchRepair assumes a bijective relation between the program's set of variables and the candidate repair (code fragment). Therefore, only patches with the same number of variables can be found, making it impossible to find repairs that remove or introduce new variables to the program;
- SearchRepair disregards the negative example behavior (failed tests from the test suite) when generating input-output constraints to search the database;

**SOSRepair [122]** is built on the ideas of SearchRepair [6]. The major difference between these two frameworks is that SOSRepair has a more scalable search query encoding and can query large databases of multi-million-line C projects [122].

SOSRepair constructs a database of code snippets. It identifies candidate code fragments from the C blocks taken from the programs' AST (see Definition 21). Afzal et al. [122] performed a case study and found out that snippets of the length of 3-7 lines achieve patches with higher quality. Hence, SOSRepair has a database of code snippets encoded as constraints. These constraints have information about the snippet's variables and the static path constraints generated by each input-output test. These path constraints are obtained by symbolically executing (see Section 3.1.2) the code snippet using KLEE [79].

Given an incorrect program and a test suite, SOSRepair, like SearchRepair, looks for a potential patch on a code snippets database based on some specifications of desired behavior provided by the test suite. SOSRepair uses this test suite to construct input-output constraints of the potential buggy region and searches over the database code fragments with similar behavior. SOSRepair, unlike SearchRepair, takes advantage of the negative example behavior (failed tests from the test suite) when generating input-output constraints to search the database. SOSRepair's query encoding allows the search for patches that insert, delete and replace code. SOSRepair executes the program on the test suite, saving the variables' values in the buggy region, before (entry values) and after (exit values) the region. To insert new code, the encoding enforces the potential patch to keep the same entry values on the passing tests but to change the entry values on the failing tests. To replace buggy code, the encoding enforces that the values of the variables after the code snippet (exit values) must remain the same on the passing tests but have to change on the failing tests.

For each buggy code segment, SOSRepair encodes the buggy region input-output constraints. Then, first, it tries deleting the buggy region. Next, SOSRepair attempts to replace the buggy region by searching the database. Finally, if no code replacement was found, SOSRepair searches the database for a code fragment to insert right before the buggy region. Finally, when a viable patch is found, SOSRepair renames the variables by mapping the incorrect program's variable to the code snippet and inserting the patch right before the buggy region in the case of a code insertion or replaces the buggy region with this patch in the case of code replacement.

SOSRepair has two significant drawbacks:

- SOSRepair considers a higher granularity of code fragments since the authors believe that replacing entire loops/if-statements with similar statements is more likely to fix the code. However, when considering an educational setting, this does not guarantee or even considers the generation of the minimal set of repairs in order to help the novice programmers;

- To find a consistent mapping between the variables of the incorrect program and the variables of the code fragment, SOSRepair, like SearchRepair, includes constraints encoding all possible mappings between these two sets of variables;

#### 3.3.2.4 Discussion

This section will discuss the different aspects and drawbacks of various techniques presented in this document for semantic automated program repair.

Table 3.2 shows several characteristics of recent test-driven, solution-driven, and clustering-based semantic program repair tools. The repairs generated by test-driven approaches [4–7, 29] may not be generalized to test cases that are not present in the test-suite, i.e., the repairs generated by these approaches may be overfitted to the test-suite [129]. Using a reference implementation to check for correctness reduces the overfitting in the test-suite [141].

Machine learning techniques, like sk_p [19] and SynFix [18], and clustering-based approaches, like CLARA [2], require several student correct submissions to generate good quality repairs. Therefore, machine learning techniques require considerable time to train students' correct submissions. Afterward, these approaches generate repairs faster [10]. However, the generated repairs are frequently imprecise and not minimal [10]. The set of repairs proposed by Verifix [9] is not guaranteed to be minimal since it always aligns a given incorrect submission with the same reference implementation provided by the lecturer. Verifix also requires a bijective relation between the set of variables of both programs. Thus, if a student uses more or fewer variables than the instructor's implementation, Verifix cannot repair the student's program. Additionally, Verifix can supposedly repair submissions whose control-flow graph is different from the lecturer's program. However, as explained in Section 3.3.2.2, Verifix cannot repair programs with a different number of auxiliary functions than the reference implementation.

Section 3.3.2.2 presented clustering-based program repair frameworks. CLARA [2] and SARFGEN [10] assume that there are several and diverse correct student submissions from past years. These approaches need a substantial number of previous correct submissions to have diverse clusters to provide valuable repair. Hence, for a novel programming assignment, these techniques do not work. Refactory [11] and AutoGrader [121] deal with this problem using just one reference implementation. Furthermore, Clara requires a perfect match between the two programs' control flow graphs (i.e., branches, loops, functions) and a bijective relation between both programs' variables. Otherwise, Clara returns a structural mismatch error, and those programs are not clustered together. During Clara's repair process, if none of the correct programs provided has an exact match with the incorrect submission's control flow, then Clara is not able to repair the program and returns a *Structural Mismatch* error. Lastly, Refactory [11] relies on hand-crafted program refactoring rules currently implemented only for Python programs.

As explained in Section 3.3.2.3, code search techniques, like SearchRepair [6] and SOSRepair [122], use code snippets and have no knowledge of the program's structure where that code fragment came from. Therefore, there is no guarantee that the set of repairs proposed by a code search tool is the minimal set required to fix a student's program. SearchRepair includes constraints encoding all possible mappings between these two programs' sets of variables and assumes a bijective relation between the programs' variables. Therefore, only patches with the same number of variables can be found. Moreover, SearchRepair disregards the negative example behavior when generating input-output constraints to search the database. SOSRepair considers a higher granularity of code fragments which does not guarantee the generation of the minimal set of repairs to help the novice programmers. To find a consistent mapping between the variables of the incorrect program and the variables of the code fragment, SOSRepair, like SearchRepair, includes constraints encoding all possible mappings between these two

Table 3.2: Semantic Tools

| Tools | Test-suite | Multi-line Debug | Approach | Programming Language | Code Available |
|---|---|---|---|---|---|
| SemFix [4] (2013) | Yes | No | Symbolic Execution | C | No |
| DirectFix [5] (2015) | Yes | Yes | Symbolic Execution; MaxSMT | C | No |
| SearchRepair [6] (2015) | Yes | No | SMT | C | Yes |
| Angelix [7] (2016) | Yes | Yes | Angelic Forests | C | Yes |
| CLARA [2] (2018) | Yes | Yes | Clustering; Dynamic Analysis | Python/C | Yes |
| SARFGEN [10] (2018) | No | Yes | Program Analysis | C# | No |
| Refactory [11] (2019) | Yes | Yes | Refactoring | Python | Yes |
| AutoGrader [121] (2019) | Yes | Yes | Symbolic Execution | Python | Yes |
| SOSRepair [122] (2019) | Yes | Yes | Semantic Code Search | C | Yes |
| Verifix [9] (2021) | Yes | Yes | Verified Repair | C | No |

sets of variables.

Lastly, Large Language Models (LLMs) are used for semantic program repair but often make extensive rewrites instead of minimal adjustments. This tends to lead to more invasive fixes, making it harder for students to learn from their mistakes.

## 3.4 Automated Assessment Tools (AATs)

**Automated Assessment Tools (AATs)** Over the past decades, there has been a growing interest in the automated evaluation of Software Engineering (SE) and Computer Science (CS) students [142]. Typically, AATs assess programming tasks using input/output (IO) tests predefined by the course's faculty. There exists a substantial number of AATs that function as web-based Integrated Development Environments (IDEs) for evaluating students' code using IO tests. Examples include CODEOCEAN [143], MOOSHAK [144], and WEB-CAT [145]. Furthermore, AUTOLAB [146] and SUBMITTY [147] are open-source web-based course management platforms that automatically grades students' code. AUTOLAB maintains scoreboards for each evaluation element in order to motivate the students, while SUBMITTY provides an interface for Teaching Assistants (TAs) to manually grade assignments. CHECK50 [148] is a Python command line tool that automatically assesses students code using IO tests, works for C and Python programs.

Additionally, Codeboard.io [149] is a web-based IDE to teach programming tasks. Faculty members can share programming exercises with the students. These exercises are assessed using a result string or a set of predefined unit tests. The set of programming languages available is limited and Codeboard.io is difficult to tailor in order to get more personalized feedback for the students. Drop Project [150] is a web-based IDE where students drop their Java/Kotlin projects to check for correctness and quality using a set of predefined unit tests and other software evaluation metrics. GRADESTYLE [151] serves as a code style marker tool that provides feedback for assignments in Java by opening a GitHub issue in each students repository. Moreover, GRADESCOPE [152] is another online tool to administer and grade programming assignments as well as other kinds of assessments (e.g., exams). However, GRADESCOPE only has paid licenses for education. Recently, Coleman and Sommer [153] proposed GIT-KEEPER, a git-based assignment management system that automates the assessment of student submissions for

programming assignments and automatically emails students their evaluation report. Lastly, GITHUB CLASSROOM [154] is an AAT tool available on GITHUB that allows faculty to create and manage digital classrooms and assignments. GITHUB CLASSROOM uses the same mechanism of a CI runner (GitHub Actions) to process student code and report on quality aspects.

**Competitive Programming Contests (CPCs)** CPCs are online platforms that host programming contests. In these websites, students and CS/SE professionals, engage in solving computational problems under time/memory constraints. These contests serve as platforms to assess and enhance problem-solving skills, algorithmic efficiency, and programming proficiency. Typically, CPCs assess contestants code using an IO test suite. LEETCODE [155], TOPCODER [156], CODEFORCES [157], and REPLIT [158] are among the most famous CPCs. Both CODEFORCES [157] and REPLIT [158] offer features for programming education. However, the evaluation process relies solely on IO tests.

## 3.5 Providing Feedback

Feedback generation for students on programming exercises has received a fair amount of attention from the scientific community over the last decade. Gulwani et al. [159], Singh et al. [160] study the synthesis of solutions for geometric assignments enabling them to provide hints to students. Ahmed et al. [161] generate problems and solutions in Natural Deduction with an offline computation of proofs. Pex4fun [162] is an automated website for students to solve programming exercises; the website uses advanced techniques from Formal Methods technology based on SMT to identify failing inputs of students solutions. Solution correction for program assignments is related to Program Sketching or Program Synthesis [163, 164]. Gulwani et al. [165] propose a small language extension that allows a lecturer to specify an algorithmic strategy using key values that should appear during the execution of an implementation for a given IPA. Zimmerman and Rupakheti [166] proposed a tool capable of recommending a reliable series of AST edits to the student program to transform it into the lecturer's target solution.

CODERASSIST [167] is a counter-example guided feedback generation tool that provides feedback on student implementations of *dynamic programming algorithms*. CODERASSIST starts by clustering both correct and incorrect programs based on these dynamic programs' syntactic features. Afterward, CODERASSIST generates feedback for a buggy program by calling an SMT solver, using a counterexample obtained from an equivalence check against a correct implementation in the same cluster. However, CODERASSIST only works for dynamic programs since the implemented feature extraction procedure searches for syntactic features of dynamic programming algorithms. Thus, this tool cannot be used for clustering our dataset of IPAS. Rocha et al. [168] introduce a framework to improve how teaching assistants provide feedback in introductory programming courses, helping them understand different feedback approaches and resources. This framework offers structured guidance for pedagogical decision-making regarding adaptive feedback. Furthermore, CODEHOUND [169] is a system that automatically tracks pedagogical code dependencies by using static analysis to detect function introductions and reuse throughout an entire course. It offers instructors assistance in creating new content, collabo-

rating on content refactoring, and estimating future course change costs. Furthermore, ODS [170] is a system for detecting overfitting patches in automatic program repair. ODS utilizes supervised learning with AST level code features and patch correctness labels to automatically learn a probabilistic model, which can then classify new program repair patches.

# 4

# C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments

*"C is quirky, flawed, and an enormous success."*

– Dennis Ritchie.

## Contents

This chapter presents C-PACK-IPAS, a publicly available benchmark comprising student-program submissions for 25 distinct IPAS. C-PACK-IPAS contains semantically correct, semantically incorrect, and syntactically incorrect programs, along with a dedicated test suite for each IPA. Hence, C-PACK-IPAS serves as a valuable resource for evaluating the progress of novel automated program repair frameworks, addressing both semantic and syntactic aspects, with a specific focus on providing feedback to novice programmers. Notably, some semantically incorrect programs in C-PACK-IPAS have been manually fixed and annotated with diverse program features, enhancing their utility for the development of various program analysis frameworks. Moreover, this chapter presents evaluations on C-PACK-IPAS using two leading semantic program repair tools tailored for IPAS, CLARA and VERIFIX.

This chapter has been published as a workshop paper at the 5th International Workshop on Automated Program Repair, APR 2024 [171].

## 4.1 Introduction

Typically, in Computer Science courses, programming assignments follow a familiar pattern: the lecturer outlines a computational problem, students devise solutions, and each solution undergoes evaluation for correctness using predetermined tests. If the students' tentative solutions do not pass a given test, they are deemed incorrect without helpful feedback. In instances where students' programs fail a subset of the predefined tests, seeking feedback from the lecturer becomes a common practice to understand the reasons behind the unexpected behavior. When a program fails to pass even a single predefined test, it signifies a semantic error in the implementation. Unfortunately, due to the increasing number of student enrollments, personalized feedback from the faculty may not always be feasible. Therefore, automated semantic program repair frameworks [2, 4–12] are ideal for providing hints on how students should repair their incorrect programming assignments.

This chapter presents C-PACK-IPAS, a **C**90 **P**rogr**a**m ben**c**hmar**k** of introductory programming assignments (**IPAs**). C-PACK-IPAS comprises students' programs submitted for 25 different IPAS along with the respective test suite used for each assignment. The details of the IPAS are provided in Section 4.3. For each IPA, C-PACK-IPAS includes sets of both semantically correct and incorrect implementations. Additionally, C-PACK-IPAS encompasses a collection of syntactically faulty programs submitted for each IPA. The primary objective of this chapter is to present C-PACK-IPAS, a resource featuring semantically and syntactically incorrect student implementations. This benchmark is intended to facilitate the evaluation of novel automated program repair frameworks, addressing both semantic and syntactic aspects, with a focus on assisting novice programmers.

C-PACK-IPAS includes a subset of semantically incorrect programs, manually annotated with various features such as the number of faults in each program, the location of these faults, and the type of each fault. Additionally, C-PACK-IPAS also comprises corrected versions of these programs, which were manually fixed. The availability of annotated and fixed programs serves as a valuable resource, allowing developers to validate their results during the development of novel program analysis tools, particularly for tasks such as program repair and fault localization.

Table 4.1: High-level Description of C-PACK-IPAS Benchmark.

| Labs | #IPAs | #Correct Submissions | #Semantically Incorrect Submissions | #Syntactically Incorrect Submissions |
|---|---|---|---|---|
| **Lab02** | 10 | 799 | 486 | 223 |
| **Lab03** | 7 | 351 | 699 | 106 |
| **Lab04** | 8 | 465 | 246 | 119 |
| **Total** | **25** | **1615** | **1431** | **448** |

The main contributions of this work are:

- C-PACK-IPAS, a benchmark consisting of C programs submitted for 25 different IPAS, during three academic years. C-PACK-IPAS contains semantically correct, semantically incorrect, and syntactically incorrect programs plus a test suite for each IPA.
- A subset of C-PACK-IPAS's semantically incorrect programs has been manually fixed and annotated, incorporating several program features such as the number of faults and the locations of these faults.
- The evaluation of C-PACK-IPAS was conducted using two state-of-the-art program repair tools tailored for IPAS: CLARA and VERIFIX;
- C-PACK-IPAS is publicly available on GitHub: `https://github.com/pmorvalho/C-Pack-IPAs`.

The structure of the remainder of this chapter is as follows. Section 4.2 presents C-PACK-IPAS. Next, Section 4.3 presents a brief description of the set of programming exercises. Section 4.4 presents the experimental evaluation where C-PACK-IPAS was evaluated using state-of-the-art program repair tools. Lastly, Section 4.5 describes related work, and the chapter concludes in Section 4.6.

## 4.2 C-PACK-IPAS

C-PACK-IPAS is a pack of student programs developed during an introductory programming course in the C programming language. These programs were collected over three distinct practical classes for 25 different IPAS at Instituto Superior Técnico, throughout three academic years. The set of submissions was split into three groups: semantically correct, semantically incorrect, and syntactically incorrect submissions. The students' submissions that satisfied the set of input-output test cases for each IPA were considered semantically correct. The submissions that failed at least one input-output test but successfully compiled were considered semantically incorrect implementations. Lastly, the students' submissions that did not successfully compile were considered syntactically incorrect.

Table 4.1 presents the number of submissions gathered. For 25 different programming exercises, this benchmark contains 1615 different correct programs, 1431 semantically incorrect submissions, and 448 syntactically incorrect implementations. C-PACK-IPAS is organized chronologically. This arrangement proves valuable for training and evaluating new program analysis tools. For instance, the programs from the first academic year can serve as training data, those from the second year as the validation set,

Table 4.2: The number of faulty programs in each exercise of Lab02, with different types of program faults.

| | Lab 02 | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| **Fault Type** | **Ex 01** | **Ex 02** | **Ex 03** | **Ex 04** | **Ex 05** | **Ex 06** | **Ex 07** | **Ex 08** | **Ex 09** | **Ex 10** |
| **Incomplete Binary Operation** | 1 | | | 2 | | | | | | |
| **Incorrect Data Type** | | | | | | 1 | | 3 | | |
| **Incorrect Input** | 4 | 4 | 3 | | | 1 | | | | |
| **Incorrect Output** | 63 | 18 | 11 | 14 | 3 | 3 | 3 | 6 | 17 | 3 |
| **Misplaced Expression** | | | | | | 1 | | | | |
| **Misplaced Loop Decrement** | | | | | | 1 | | | | |
| **Missing Expression** | | | | | | 2 | 6 | | | |
| **Missing Instruction** | 1 | | 2 | 8 | | | | | | |
| **Missing Instructions** | | | | 5 | | | | | | |
| **Missing Loop Decrement** | | | | | | 2 | | | | |
| **Missing Loop Increment** | | | | | | 1 | | 1 | | |
| **Missing Output** | 1 | | | | | | | | | |
| **Missing Variable** | | | | | 1 | 2 | 6 | | | |
| **Non-zero Return** | 1 | | | | | | | | | |
| **Presentation Error** | 44 | 26 | 22 | 27 | 2 | 1 | 6 | | 3 | 2 |
| **Uninitialized Variable** | 11 | | | 12 | 1 | 3 | 1 | 5 | | |
| **Variable Misuse** | 2 | 1 | 3 | 5 | 1 | | 1 | | 2 | |
| **Wrong Binary Operation** | | | | 4 | | 2 | 1 | | | 1 |
| **Wrong Comparison Operator** | 3 | 12 | | 3 | | | | 1 | | 1 |
| **Wrong Exercise** | 8 | 2 | | 2 | 1 | | | | | |
| **Wrong Expression** | | | | 7 | 1 | 2 | 6 | 1 | 3 | 1 |
| **Wrong Initialization** | 1 | | | | | | 1 | | | |
| **Wrong Instruction** | 5 | | | | | | | | | |
| **Wrong Literal** | | | 1 | | 1 | 2 | | 2 | 3 | |
| **Wrong Parameter** | | | | | | | | 1 | | |
| **Average Number of Faults** | 1.8 | 1.56 | 1.28 | 2.06 | 1.33 | 3.56 | 2.36 | 1.91 | 1.58 | 1.43 |

and the programs from the third year as the evaluation set. Appendix A.2 presents three tables with the number of student submissions (correct, semantically incorrect and syntactically incorrect) received for each of the 25 different programming assignments.

Furthermore, C-PACK-IPAS only contains students' submissions that gave their permission to use their programs for academic purposes. Each student's identification was anonymized for privacy reasons, and all the comments were removed from their programs. A unique identifier was assigned to each student. These identifiers are consistent among different IPAS and different years of the programming course. For example, if the identifier `stu_15` appears in more than one programming exercise, it corresponds to the same student. If some students take the course more than once, they are always assigned to the same anonymized identifier. Currently, C-PACK-IPAS contains submissions from 102 different students.

**Annotated Programs** A benchmark of programs should be enriched with informative annotations, enabling developers to validate their results while developing novel program analysis tools for tasks such as program repair and fault localization. With this objective in mind, the Lab 02 submissions in C-PACK-IPAS have been meticulously annotated with various program features. These annotations serve as valuable resources for developers and facilitate the training and evaluation of machine learning models. Each semantically incorrect program in C-PACK-IPAS's Lab 02 submissions has been annotated with the following features::

- *#Variables* : indicates the number of different variables present in each program;
- *Program Features* : encompasses various program features, including uninitialized variables, seg-

mentation faults, etc., providing valuable information for analysis;

- *#Passed Tests* : represents the count of passed tests from the test suite;
- *#Failed Tests* : specifies the number of failed tests from the test suite;
- *IO tests' output* : presents the output obtained by running the incorrect program with the test suite;
- *#Faults* : indicates the total number of faults present in the program;
- *Faults* : enumerates the list of faulty instructions/expressions within the program;
- *Faulty Lines* : lists the program lines containing faults;
- *Faults' Types* : specifies the types of each fault in the program;
- *Repair Actions* : enumerates the required repair actions to fix the faulty instructions/expressions, with possible actions including Insert, Replace, Remove, or Move;
- *Suggested Repairs* : provides a list of suggested repairs for each identified faults in the program;
- *Next Correct Submission* : indicates the path to the subsequent correct submission by the same student, if available.

Table 4.2 presents the various types of faults used to annotate programs in C-PACK-IPAS. For each manually annotated fault type, the table presents the count of faulty programs exhibiting that specific fault, categorized by each exercise of Lab 02. The most prevalent fault types include *Presentation Error* (differences only in white spaces), *Incorrect Output* (output is incorrect), and *Uninitialized Variable*. Additionally, at the bottom of Table 4.2, the average number of faults in programs for each exercise of Lab 02 is provided. All annotations are stored in two formats: as plain text (`txt`) files and within an `sqlite3` database.

**Organization** C-PACK-IPAS is structured by lab and exercise, organized chronologically by academic year. Each program is stored in its respective folder, along with the program's output results for the test suite and the previously described handmade annotations.

## 4.3 IPAS Description

The set of IPAS corresponds to three different lab classes of the introductory programming course to the C programming language. Each lab class focuses on a different topic of the C programming language. In Lab02, the students learn how to program with integers, floats, IO operations (mainly `printf` and `scanf`), conditionals (if-statements), and simple loops (for and while-loops). In Lab03, the students learn how to program with loops, nested loops, auxiliary functions, and chars. Finally, in Lab04, the students learn how to program with integer arrays and strings. The textual description of each programming

Table 4.3: The number of programs repaired by VERIFIX, CLARA without using clusters and CLARA using clusters.

| Lab 02 | | | | | |
|---|---|---|---|---|---|
| | | Unsuccessful | | | |
| **Repair Method** | **% Fixed** | **% Structural Mismatch** | **% Unsupported Features** | **% Other Errors/Exceptions** | **% Timeouts (600s)** |
| **Verifix** | 93 (19.14%) | 92 (18.93%) | 55 (11.32%) | 246 (50.62%) | 0 (0.0%) |
| **Clara (No Clusters)** | 275 (56.58%) | 210 (43.21%) | 0 (0.0%) | 1 (0.21%) | 0 (0.0%) |
| **Clara (Clusters)** | 346 (71.19%) | 12 (2.47%) | 0 (0.0%) | 33 (6.79%) | 95 (19.55%) |

| Lab 03 | | | | | |
|---|---|---|---|---|---|
| | | Unsuccessful | | | |
| **Repair Method** | **% Fixed** | **% Structural Mismatch** | **% Unsupported Features** | **% Other Errors/Exceptions** | **% Timeouts (600s)** |
| **Verifix** | 0 (0.0%) | 0 (0.0%) | 699 (100.0%) | 0 (0.0%) | 0 (0.0%) |
| **Clara (No Clusters)** | 11 (1.57%) | 511 (73.1%) | 138 (19.74%) | 39 (5.58%) | 0 (0.0%) |
| **Clara (Clusters)** | 168 (24.03%) | 65 (9.3%) | 138 (19.74%) | 328 (46.92%) | 0 (0.0%) |

| Lab 04 | | | | | |
|---|---|---|---|---|---|
| | | Unsuccessful | | | |
| **Repair Method** | **% Fixed** | **% Structural Mismatch** | **% Unsupported Features** | **% Other Errors/Exceptions** | **% Timeouts (600s)** |
| **Verifix** | 0 (0.0%) | 6 (2.44%) | 237 (96.34%) | 3 (1.22%) | 0 (0.0%) |
| **Clara (No Clusters)** | 0 (0.0%) | 107 (43.5%) | 138 (56.1%) | 1 (0.41%) | 0 (0.0%) |
| **Clara (Clusters)** | 36 (14.63%) | 18 (7.32%) | 138 (56.1%) | 54 (21.95%) | 0 (0.0%) |

assignment can be found in the public GitHub repository, and the input/output tests used to evaluate semantically the set of students' submissions. Moreover, there is also a reference implementation for each IPA in the public git repository that can be used by program repair frameworks that only accept a single reference implementation to repair incorrect programs. Appendix A.1 presents the entire list of IPAS.

## 4.4 Experimental Results

To evaluate C-PACK-IPAS, we used two publicly available state-of-the-art program repair tools for fixing introductory programming assignments (IPAS): CLARA [2] and VERIFIX [9]. We simply focused on the set of semantically incorrect programs which is composed by 1434 programs as presented in Table 4.1.

**CLARA and VERIFIX** VERIFIX [9] aligns the control flow graph (CFG) of an incorrect program with the reference solution's CFG. Then, using that alignment relation and MAXSMT solving, VERIFIX proposes fixes to the incorrect program. VERIFIX also requires a compatible control flow graph between the incorrect and the correct program. On the other hand, to repair an incorrect program, CLARA [2] receives either one or a set of correct programs. This set of programs corresponds to clusters' representatives produced by CLARA. During CLARA's repair process, if none of the correct programs provided has an exact match with the incorrect submission's control flow, then CLARA is not able to repair the program and returns a *Structural Mismatch* error. Otherwise, CLARA gathers the set of repairs using each correct program and returns the minimal one. Since CLARA can take advantage of several correct implementations from previous years for a given IPA, we fed CLARA all correct programs, from the three different academic years, to generate clusters for each IPA in our benchmark. Furthermore, we also run CLARA using the faculty's reference implementation for each IPA, i.e., CLARA uses a single program and not a set of clusters' representatives. Moreover, we run VERIFIX using also the reference implementation since VERIFIX can only accept a single correct program as input.

**Experimental Setup** All the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 32GB and a timeout of 600 seconds.

**Results** Table 4.3 presents the number of programs repaired by VERIFIX, CLARA (utilizing a single reference implementation), and CLARA (utilizing its own clusters). The results indicate that VERIFIX demonstrates success primarily in Lab 02, repairing approximately 19% of the programs. However, VERIFIX encounters challenges in repairing 80% of Lab 02 and the entire set of programs from Lab 03 and Lab 04. The primary factor influencing VERIFIX's performance is its limited support for certain C Library functions utilized in several exercises. For instance, nearly all exercises in Lab 03 involve the use of C Library functions such as `putchar` or `getchar`, which are not supported by VERIFIX.

As previously mentioned, if none of the correct programs provided matches the control flow of the incorrect submission exactly, CLARA issues a *structural mismatch* error. Table 4.3 reveals that CLARA, when employing a reference implementation, exhibits a notably higher percentage of structural mismatch errors compared to CLARA utilizing clusters. This difference arises because clusters, with multiple programs, offer various control flow options, leading to a reduced rate of structural mismatches. Additionally, CLARA generates a set of repairs for each cluster's representative. Therefore, a higher number of clusters corresponds to more time spent in the repair process. This constitutes one of the primary reasons for the increased occurrence of timeouts observed with CLARA when utilizing clusters as opposed to not using clusters.

Figure 4.1 presents a cactus plot illustrating the CPU time allocated for repairing each program (on the $x$-axis) in relation with the number of successfully repaired programs (on the $y$-axis) across the three different repair techniques. The legend is organized in descending order based on the count of programs successfully repaired. Notably, VERIFIX, within the 60-seconds, repairs 90 out of 1434 programs (approximately 6.5%). In comparison, CLARA, utilizing a reference implementation, repairs 286 programs (20%), while employing its own clusters allows CLARA to repair around 500 programs within the same time limit (approximately 35%).

Figure 4.2 illustrates a scatter plot comparing the CPU time spent using CLARA's clusters against running CLARA with the faculty's reference implementation (no clusters). Each data point in the plot represents a program, where the $x$-value (corresponding to using CLARA's clusters) and $y$-value (representing no clusters) denote the CPU time spent on repairing that program.

If a point falls below the diagonal, it indicates that using a reference implementation outperformed using clustering. In this scenario, CLARA, with its own clusters, repairs each program more slowly than with a single correct program. Consequently, when considering programs repaired by both clustering methods, not using clusters is faster, although repairs a significantly smaller number of programs, as presented in Table 4.3. Additionally, Figure 5.8 highlights that the set of programs repaired by CLARA differs when using a reference implementation compared to clusters.

In short, we evaluated two state-of-the-art semantic program repair tools tailored for IPAS. CLARA was the clear winner, repairing 550 programs equivalent to 38.4% of C-PACK-IPAS. This outcome indicates ample room for improvement. Given that C-PACK-IPAS encompasses 25 distinct IPAS of varying

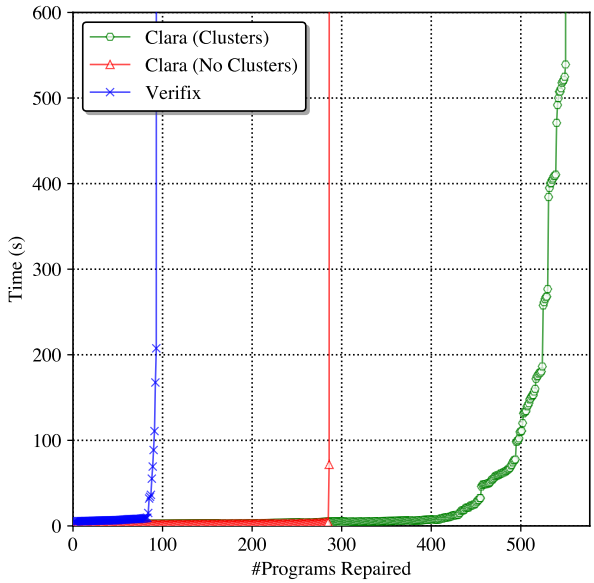



Figure 4.1: Cactus plot - The time spent by each method repairing each semantically incorrect submission of C-PACK-IPAS, using a timeout of 600 seconds.

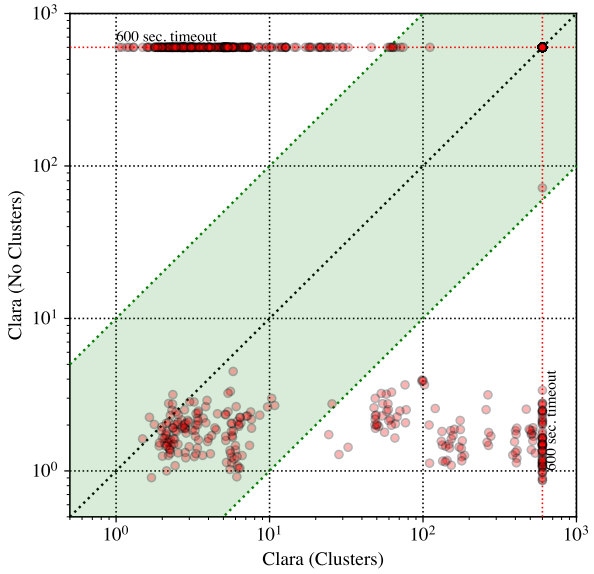



Figure 4.2: Scatter plot - Time Performance (600s) - CLARA (using clusters) VS CLARA (reference implementation).

complexities, it stands as a valuable resource for the development of advanced program repair tools capable of addressing more intricate IPAS. Moreover, C-PACK-IPAS [171] has also proven successful in evaluating various works across program analysis [172, 173], program transformation [174], and program clustering [42].

## 4.5 Related Work

The number of publicly available benchmarks to help develop and evaluate new program repair tools is significantly small [174]. The ITSP dataset [8] has been used by other automated software repair

tools [8, 9] that use only one reference implementation. This dataset is also a collection of C programs although it is well balanced, i.e., the number of correct submissions is closer to the number of incorrect submissions in this dataset. The INTROCLASS dataset [175] is a collection of C programs submitted to six different IPAS and has the information about the number of defects in each program and the total number of unique defects for each IPA. CODEFLAWS [176] is a dataset of programs submitted for programming competitions on the Codeforces website. Lastly, BUGSC++ [177] is a benchmark that compiles real-world bugs gathered from 22 open-source C/C++ projects.

In the context of the *fault localization* problem, TCAS [178] stands out as a well-known program benchmark extensively utilized in the literature. This benchmark comprises a C program and multiple versions of it with intentionally introduced faults, with known positions and types of these faults. More program benchmarks are available for other languages than the C programming language. For example, the dataset of Python programs used to evaluate REFACTORY [11] is also publicly available. More datasets for automated program repair applied to industry software are also available [1].

## 4.6 Conclusion

C-PACK-IPAS, a **C**90 **P**rogr**a**m ben**c**hmar**k** of introductory programming assignments (**IPAs**), is a publicly available benchmark of students' submissions for 25 different programming assignments. C-PACK-IPAS has a set of semantically correct and incorrect implementations as well as syntactically faulty programs submitted for each IPA. To the best of our knowledge, C-PACK-IPAS is one of the few, if not the only, benchmark of IPAS written in the C programming language that contains both semantically and syntactically incorrect students' implementations and diverse correct implementations for the same IPA. Thus, C-PACK-IPAS can help evaluate novel semantic, as well as syntactic, automated program repair frameworks whose goal is to assist novice programmers in introductory programming courses. We have also manually fixed and annotated some of C-PACK-IPAS's semantically incorrect programs with several program features to help developing all sort of program analysis frameworks. Additionally, we evaluated C-PACK-IPAS using two state-of-the-art semantic program repair tools tailored for IPAS, CLARA and VERIFIX.

[1] https://program-repair.org/benchmarks.html

# 5

# INVAASTCLUSTER: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments

*"Good problems and mushrooms of certain kinds have something in common; they grow in clusters."*

– G. Polya, How To Solve It [179].

## Contents

Typically, automated program repair (APR) techniques focused on introductory programming assignments (IPAs) use program clustering to take advantage of previous correct student implementations to repair a new incorrect submission. These repair techniques use clustering methods since analyzing all available correct submissions to repair a program is not feasible. However, conventional clustering methods rely on program representations based on features such as abstract syntax trees (ASTs), syntax, control flow, and data flow.

This chapter proposes INVAASTCLUSTER, a novel approach for program clustering that uses dynamically generated program invariants to cluster semantically equivalent IPAs. INVAASTCLUSTER s program representation uses a combination of the program's semantics, through its invariants, and its structure through its anonymized abstract syntax tree (AASTs). Invariants denote conditions that must remain true during program execution, while AASTs are ASTs devoid of variable and function names, retaining only their types. Our experiments show that the proposed program representation outperforms syntax-based representations when clustering a set of correct IPAs. Furthermore, we integrate INVAASTCLUSTER into a state-of-the-art clustering-based program repair tool. Our results show that INVAASTCLUSTER advances the current state-of-the-art when used by clustering-based repair tools by repairing around 13% more students' programs, in a shorter amount of time.

The work presented in this chapter is currently under review.

## 5.1 Introduction

Over the last few years, several program repair tools [2, 10, 11, 34] have appeared that use a large number of diverse correct implementations submitted for each IPA by previously enrolled students. Given an incorrect student submission, these frameworks use clustering methods to find the most similar correct submission from previous years to provide a minimal set of repairs to the student. Using the same reference implementation to fix all incorrect programs can potentially generate a large set of repairs. On the other hand, having a similar correct implementation allows computing a smaller set of repairs. However, comparing on the fly all previous correct student submissions against a given incorrect submission is not feasible.

To tackle this problem, different program clustering approaches have been used in program repair tools, which enable focusing only on the representatives of each cluster. CLARA [2] clusters the correct programs based on their dynamic equivalence [35] and control flow, i.e., the order in which program statements, instructions, and function calls are executed. SARFGEN [10] computes program representations based on each program's abstract syntax tree. SEMCLUSTER [34] uses each program's control and data flow. A program's data flow tracks the number of occurrences of consecutive values a variable takes during its lifetime.

The problem of program equivalence, i.e., deciding if two programs are equivalent, is undecidable [36, 37]. On that account, finding an adequate representation for programs that performs well on program clustering is a challenging problem. The previously-mentioned program representations used in the field of program repair may be brittle. For instance, consider two programs that calculate the sum of natural

numbers from 1 to a given number, one using a while-loop and the other a for-loop. Despite producing the same result, their syntactic and structural differences pose challenges for conventional program representations to recognize their semantic equivalence, as we will show in Section 5.2. To address this problem, we propose to use dynamically-generated program invariants to cluster semantically equivalent programs, overcoming some of the identified weaknesses. A program invariant is a condition that must always be true at a given step of the program during its execution (see Section 5.3). Program invariants are usually used to assert assurances throughout a program (assertions).

This chapter proposes to leverage the information of a program's structure using its *abstract syntax tree* (AST) together with semantic information provided by its invariants. Previous research has been conducted regarding using invariants to promote patch diversity (i.e., diversity in the set of possible repairs to a given incorrect program) on search-based program repair [180–182]. These works use DAIKON [59] to generate invariant sets for each possible patch. DAIKON is a system that infers likely dynamically generated invariants observed over several program executions. Therefore, these invariants depend on the program executions. Nevertheless, previous work [180] showed promising results in using invariants to semantically cluster patches to provide the user with a semantic reason for similar patches.

This chapter presents a novel approach for clustering introductory programming assignments (IPAS) leveraging their sets of invariants. Our approach for clustering IPAS also takes into account each program's code and *anonymized abstract syntax tree (AAST)*. AASTs are essentially ASTS stripped of variable and function identifiers, preserving only their respective types (see Section 5.4.2). The main contribution of this work is a vector representation of programs based on their invariants and AASTS, bringing together their semantic and syntactic features. The proposed clustering technique has been implemented in a framework INVAASTCLUSTER. This tool has been designed as an independent clustering tool. Therefore, it can be used to help evaluate students' submissions for IPAS by clustering semantically equivalent solutions for programming exercises. However, INVAASTCLUSTER can also be easily integrated into any clustering-based program repair tool for IPAS. Furthermore, INVAASTCLUSTER can even be used in a plagiarism detection tool, like MOSS [183].

Figure 5.1 shows the generic architecture of clustering-based program repair frameworks [2, 10, 34]. These frameworks receive an incorrect student submission, a test suite, and a collection of $N$ correct student submissions for the same IPA. For scalability concerns, these frameworks eliminate, through clustering techniques, semantically equivalent solutions, i.e., dynamically equivalent correct programs, given the provided input-output test suite. Those clustering approaches try aggregating the set of $N$ correct solutions into $K$ semantically different clusters ($N \gg K$). Finally, the repair tool uses these $K$ clusters' representatives to repair the provided incorrect student submission. As Figure 5.1 shows, INVAASTCLUSTER can be used as the clustering technique of those clustering-based program repair tools. However, some program repair tools [9, 121] use a single reference implementation provided by the lecturer to repair a student's program. Typically, these tools can only use one correct implementation to repair each program. Therefore, INVAASTCLUSTER was designed to be also capable of finding on a set of correct student submissions which submission is the closest correct solution to the incorrect program. Thus, INVAASTCLUSTER can suggest a specific reference implementation for each incorrect

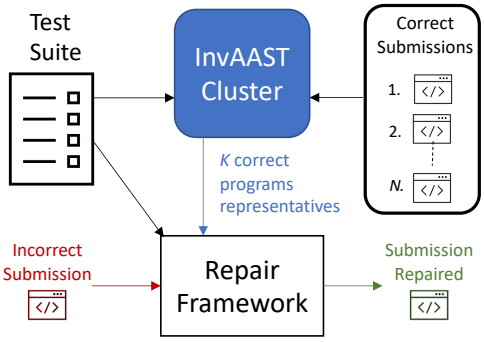



Figure 5.1: Clustering-based Program Repair.

submission that may require fewer changes to fix the program.

We evaluate INVAASTCLUSTER on C-PACK-IPAS [171], a real-world student programs developed during a university introductory programming course. Experimental results show that the proposed invariant-based representation improves upon syntax-based representations when performing program clustering. Additionally, we integrate INVAASTCLUSTER into CLARA, a clustering-based program repair tool, in order to compare our clustering technique against CLARA's clustering method, which is the current publicly available state-of-the-art method for clustering IPAS.

To summarize, this chapter makes the following contributions:

- We propose a novel and efficient approach for clustering submissions for introductory programming assignments (IPAS) based on the submissions' sets of invariants and AASTS representations.

- We present a study showing the results of using our program clustering tool, INVAASTCLUSTER, on a set of 1620 real-world IPAS correct submissions to show the effectiveness of invariant-based program clustering.

- We compare INVAASTCLUSTER with the clustering method used by the currently available state-of-the-art program repair tools. Experimental results show that INVAASTCLUSTER outperforms state-of-the-art clustering methods, allowing clustering-based program repair tools to fix around 13% more IPAS in a shorter amount of time.

- The INVAASTCLUSTER framework is publicly available on GitHub at https://github.com/pmorvalho/InvAASTCluster.

The structure of the remainder of this chapter is as follows. First, Section 5.2 illustrates the strengths of using invariants for program representation. Section 5.3 describes how to gather and represent sets of program invariants. Section 5.4 discusses several program representations, including a new invariant-based program representation. Section 5.5 discusses the implementation of INVAASTCLUSTER. Section 5.7 presents the experimental evaluation that supports our claim that invariant-based program representations are beneficial to cluster programming assignments semantically. Finally, Section 5.8 presents the related work, and the chapter concludes in Section 5.9.

## 5.2 Motivation

Current program representations for repairing students' programming assignments leverage certain program features, such as code syntax [13], abstract syntax tree [10], control flow [2], and data flow [34], to encode each program into a vector representation. However, all of these features have some weaknesses when we want to cluster programs based on their semantics.

**Example 6.** Consider the following two programs written in C, that compute the sum of all the natural numbers from $1$ to a given number `n`, i.e., $\sum_{i=1}^{n} i$.

```
1 int n, sum = 0, i;
2 scanf("%d", &n);
3 i = 0;
4 while(i < n) {
5   i++;
6   sum = sum + i;
7 }
8 printf("%d\n",sum);
```

```
1 int j, n, s = 0;
2 scanf("%d", &n);
3
4 for(j = n; j > 0; j--)
5 {
6   s = j + s;
7 }
8 printf("%d\n", s);
```

Observe that the program on the left, in Example 6, uses a while-loop that iterates over the natural numbers from $0$ to `n`. The program on the right uses a for-loop that iterates from `n` to $0$ in decreasing order. However, both programs are semantically equivalent since both have the same result. Nevertheless, if we build a program representation using the programs' syntax or abstract syntax trees, both programs will have very different representations. In terms of syntax, the names of the used variables (e.g. `i, j, s, sum`) and structures (e.g. `while`, `for`) are different. Additionally, in terms of data flow and dynamic equivalence, both programs are also different since, for example, the values assigned to the variable `i` go from $0$ to `n` in the first program while in the other the variable `j` is assigned the same values but in decreasing order.

Consider that the variable `n` is always assigned to a natural number, $n > 0$. If a dynamic invariant detector (e.g. DAIKON [59]) is used, the following set of invariants is observed:

- In the first program, at each iteration of the while-loop: $n > 0$; $sum \geq 0$; $0 \leq i \leq n$.
- In the second program, at each iteration of the for-loop: $n > 0; s \geq 0; 0 \leq j \leq n$.

Therefore, after renaming some variables ($sum \rightarrow s; i \rightarrow j$), these two sets of invariants would be considered equivalent. Hence, using sets of invariants allows finding semantically equivalent programs that can differ in their syntax and/or data flow.

Hence, this chapter aims to improve the semantic representation of programming exercises using their sets of invariants. These invariants are dynamically detected by DAIKON [59], at the beginning and at the end of each scope, over several program executions using a predefined set of test cases for each programming assignment. In addition to the set of invariants, which provides semantic information about a program, we also leverage the information of a program's structure to its anonymized abstract syntax tree (AAST), i.e., an AST after removing all the variables' names.

**CLARA.** Furthermore, in this chapter, we compare our clustering approach against CLARA's clustering method since, to the best of our knowledge, CLARA [2] stands as the sole publicly accessible state-of-the-art clustering-based repair tool for repairing IPAS. CLARA leverages correct solutions from past years' submissions for a programming assignment to suggest potential semantic repairs for an erroneous program submitted by a new student.

**CLARA's Clustering Approach** CLARA assigns two programs to the same cluster if they share identical control flow structures and possess a bijective mapping between their variables [135]. However, if there is any deviation in the control flow graphs or a lack of bijective relation between variables, CLARA returns a 'structural mismatch error', resulting in these programs not being clustered together. For each cluster generated, CLARA maintains a `json` file containing all relevant information about the programs, including expressions and variables.

Revisiting the programs illustrated in Example 6, CLARA does not detect them as matching. Furthermore, when clustering both programs using CLARA's method, they are assigned to separate clusters.

**CLARA's Repairing Process.** To repair an incorrect program, CLARA receives either a single or multiple correct programs, which can be representative of clusters produced by CLARA itself. In such cases, CLARA also necessitates access to all pertinent information regarding the programs contained within each cluster, stored in individual `json` files. CLARA proceeds to generate a series of repairs for each cluster autonomously, wherein a repair entails adjusting a program to align an expression from the incorrect submission with an expression from a program within the cluster. It is important to emphasize that CLARA's repair suggestions are confined to modifying program expressions and do not involve altering control flow. If CLARA encounters an incorrect program whose control flow does not precisely match any of the correct programs provided, it flags a "Structural Mismatch" error, indicating an inability to repair the program. Moreover, CLARA is unable to use the programs outlined in Example 6 for repairs, as it does not identify them as matching.

In this chapter, our focus is not on enhancing CLARA's repair procedure. Instead, we aim to compare our clustering approach against CLARA's clustering method, and evaluate CLARA's repair performance when utilizing its own clusters versus INVAASTCLUSTER's clusters of programs.

## 5.3 Program Invariants

Program invariants are conditions that must always be true at a given point during a programs execution. Dynamically generated program invariants are *likely invariants* observed during several program executions for a given program. The dynamically generated set of program invariants provides information about a programs behavior, i.e., its semantics. If two programs share the same program invariants, they are likely semantically equivalent. Hence, an invariant-based representation of programs should allow us to find out which student submissions in a given programming assignment have the same or similar behavior.

In order to compare two sets of program invariants, a relation between the variables in both sets is required. We propose to rename all the variables in a program based on the variables type and usage. All the variables are renamed the first time they are assigned to some value in a program. The variables new name is a concatenation between its type and a counter for how many variables have already been renamed in the program. With this technique of variable renaming, two programs sets of invariants can be easily compared. This method is very simple and fragile, although IPAs are usually relatively small and simple imperative programs. Therefore, this naive approach should work for IPAs. Example 7 shows two programs whose variables were renamed using this variable renaming method.

**Example 7.** Consider again the programs presented in Example 6, it is important to note that all variables are renamed based on their usage. Specifically, each variable is renamed the first time it is assigned a value in the program. For instance, in the first program, `n` is renamed to $int_2$ since it is the second variable to be used, in the `scanf` (line 2). After renaming all variables based on their usage, the following mapping of variables for the first program is obtained: $\{sum \rightarrow int_0; n \rightarrow int_1; i \rightarrow int_2\}$. Applying the same procedure to the second program yields the mapping $\{s \rightarrow int_0; n \rightarrow int_1; j \rightarrow int_2\}$. Thus, the two programs after renaming are as follows:

```
1  int int1, int0 = 0, int2;
2  scanf("%d", &int1);
3  int2 = 0;
4  while(int2 < int1){
5    int2++;
6    int0 = int0 + int2;
7  }
8  printf("%d\n",int0);
```

```
1  int int2,int1,int0 = 0;
2  scanf("%d", &int1);
3
4  for(int2 = int1; int2 >= 0; int2--)
5  {
6    int0 = int2 + int0;
7  }
8  printf("%d\n", int0);
```

Hence, the set of invariants of both cycles (for and while) is the same: $\{int_1 > 0; int_0 \geq 0; 0 \leq int_2 \leq int_1\}$.

In this work, we use DAIKON [59] to compute dynamically-generated likely invariants observed across multiple program executions for each student submission, employing a predefined set of input-output tests for each programming assignment. DAIKON's default format for program invariants combines Java and mathematical logic, aiming to convey meaning concisely to programmers.

To adapt DAIKON for small imperative C programs, we initially apply a method for variable renaming to all student submissions. Next, we inject empty functions into each scope and pass the variables of the respective scope as parameters. A scope is defined as each block of statements without branching, and in cases of nested scopes, we include all variables available in the parent scopes as parameters as well. We adhere to conventional methods of modeling control flows, such as, computing invariants before a loop, before a loop guard, inside a loop guard, inside the loop, and after the loop.

Finally, DAIKON is executed using all input tests for each programming assignment. The dynamically-generated invariants produced by DAIKON are stored for each program's structure or scope (e.g., `if statements`, `loops`, `blocks`). We do not specifically ask DAIKON to generate any type of invariant. The only type of invariants we turned off is the OneOf invariants (e.g., x is OneOf {1,2}) that may cause

overfitting to the test suite. Example 8 presents the two programs, previously presented in Example 7, after being injected with empty functions in each program scope.

**Example 8.** For Daikon [59] to work on small imperative programs, we have to call empty functions at the beginning of each scope and pass all the visible variables, in that scope, as parameters.

Consider again the programs presented in Example 7, after renaming all the variables based on their usage. These programs, after being injected with empty functions, would look like the following programs:

```
1  scope_1();
2  int int1, int0 = 0, int2;
3  scanf("%d", &int1);
4  int2 = 0;
5  while(int2 < int1,
6  while_1(int0, int1, int2)){
7    scope_2(int0, int1, int2);
8    int2++;
9    int0 = int0 + int2;
10 }
11 scope_3(int0, int1, int2);
12 printf("%d\n",int0);
```

```
1  scope_1();
2  int int2, int1, int0 = 0;
3  scanf("%d", &int1);
4
5  for(int2 = int1; int2 >= 0,
6  for_1(int0, int1, int2); int2--)
7  {
8    scope_2(int0, int1, int2);
9    int0 = int2 + int0;
10 }
11 scope_3(int0, int1, int2);
12 printf("%d\n", int0);
```

## 5.4 Program Representations

As the primary objective of this work is to cluster programs based on semantic and syntactic features, each program is represented as a feature vector. In particular, we propose to use a *bag of words (BoW)* model (see Definition 16). Using BoW models, we generate vector representations for each student submission based on several features. These features may include the Abstract Syntax Tree (AST), set of invariants, or even the program code. It is also possible to combine several of these features. Next, all the vector representations used in this work are described.

### 5.4.1 Syntax Vectors

The syntax vector program representation is the simplest to compute since it is based solely on the program syntax (code). In the interest of comparing the syntax of programs independently of the variables' names, first, all the programming solutions are renamed using the method described in Section 5.3. Next, all the student submissions are tokenized, and a vocabulary with all the available tokens is obtained. Then, vectors for each student submission are created, where the $i^{th}$ entry is the number of times the $i^{th}$ word of the vocabulary appears in the program. Finally, the numbers of occurrences in these vectors are normalized.

### 5.4.2 Anonymized Abstract Syntax Tree Vectors

An alternative representation is to compute a bag of words using the strings of the abstract syntax trees (ASTs) of all submissions for a given programming assignment. This representation has already been used in program clustering [10, 137]. However, we represent each AST as a string and remove all names of variables and functions, keeping only their respective types in the AST. Thus, for each submission, we have an *anonymized abstract syntax tree (*AAST*)* (see Definition 22). With these AASTs, we keep the information about a program's structure, ignoring the name of its variables. The information about a program's structure is kept since an AAST contains all the non-terminal symbols of the language's grammar. Next, a vocabulary is built with the tokens present in all submissions, and a normalized vector representation for each AAST is computed.

### 5.4.3 Invariant Vectors

Another approach is to use an invariant-based vector representation. In this case, we apply the bag of words model to the set of invariants of the programs. We gather all program invariants as described in Section 5.3. Previous work on the use of invariants to detect semantic similarity between possible patches to a program [180] showed that using string distance measure between invariant sets had similar results and was more efficient than computing the logical similarity between their corresponding sets of program invariants. Therefore, we represent our invariants in the form of strings. However, instead of using a string distance measure between invariant sets (e.g., Levenshtein edit distance [184]), we create a bag of words model with those sets of invariants.

### 5.4.4 Combination of Program Features

Finally, observe that these vector representations (*Syntax,* AAST*, Invariants*) can be combined, thus taking advantage of using several types of features. For example, we use a bag of words in our work using the program's AST and the sets of invariants. In this case, first, we build two BoW representations independently, one based on AASTs and another one based on invariants. Then, we concatenate, for each submission, the submission's vector representations using the two BoWs, achieving a vector representation based on the program's AAST and set of invariants. Both vectors, from the Invariants BoW and the AAST BoW, are normalized before concatenating. Therefore, the invariants and the AASTs have an equal contribution to the AAST + Invariants BoW. The program syntax was not included in this last representation since the BoW based on syntax has a large vocabulary that generates vectors that are too sparse.

## 5.5 Implementation

This section presents the implementation of our program clustering technique. We implemented the proposed approach in the tool INVAASTCLUSTER (**Inv**ariants and **AAST** Program **Cluster**ing). INV-

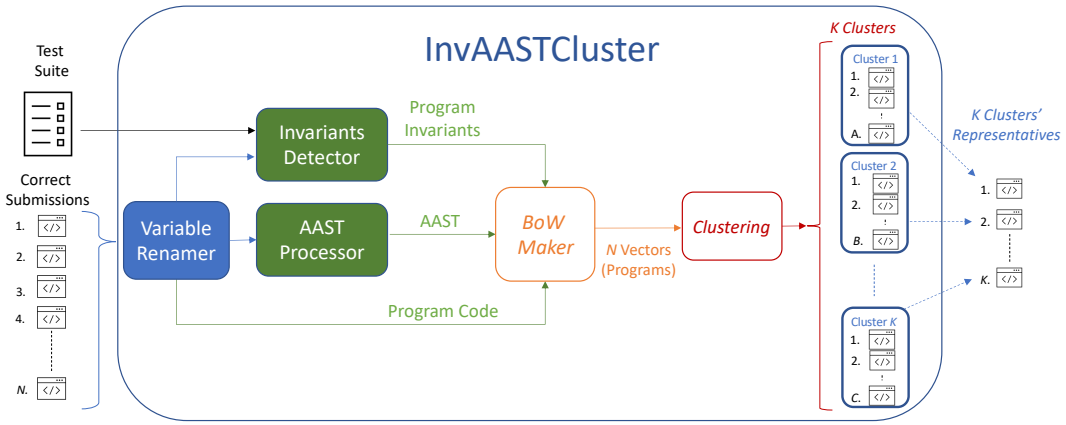



Figure 5.2: The high-level overview of InvAASTCluster.

AASTCluster is publicly available on GitHub at https://github.com/pmorvalho/InvAASTCluster. Figure 5.2 shows the overall architecture of InvAASTCluster. Given a set of $N$ correct submissions and a test suite, InvAASTCluster computes $K$ clusters of programs ($N \geq K$) and returns the set of $K$ clusters' representatives, i.e., the set of correct programs that are closest to the center of each one of the $K$ clusters. InvAASTCluster is divided into six main modules: variable renamer, invariants detector, AASTs processor, bag of words (BoW) maker, clustering procedure, and the selection of each cluster representative.

**Variable Renamer.** In this module, InvAASTCluster renames all variables of each one of the $N$ given correct submissions. All variables are renamed based on their usage in each program, as explained in Section 5.3. InvAASTCluster uses `pycparser`[1] to find all variables in a program. Then, when a variable is first used in the program (e.g., assignment) that variable receives a new name considering the variable's type.

**Invariants Detector.** InvAASTCluster uses Daikon [59] to compute dynamically-generated invariants for a given test suite (see Section 5.3). After all the variables have been renamed, this module produces a set of invariants for each program's scope using the provided test suite. All these sets of invariants are then sent to the BoW maker module.

**AAST Processor.** In this step, InvAASTCluster also uses `pycparser` to compute a program's abstract syntax tree (AST). Additionally, InvAASTCluster removes all the variables' and functions' identifiers from the AST to transform the program's AST into an anonymized abstract syntax tree (AAST), conserving only the program's structure.

**Bag of Words (BoW) Maker.** This module receives three sets as input: (1) the set of correct program submissions with all their variables renamed from the Variable Renamer module; (2) all the program's AASTs from the AAST processor, and (3) the set of the programs' dynamically-generated invariants.

[1] https://github.com/eliben/pycparser

The *BoW Maker* computes the bag of words (BoW) model (see Definition 16) that is going to be used to generate vector representations for each program.

Depending on this module's parameterization, the BoW maker can compute four different bags of words: (1) based on the programs' code (syntax), (2) based on the programs' AASTs (structure), (3) using the set of programs' invariants (semantics) and (4) joining the programs' AASTs and their sets of invariants (structure + semantics). To compute these BoW models, INVAASTCLUSTER uses scikit-learn package, `feature_extraction`[2].

INVAASTCLUSTER tokenizes the input strings into tokens of size $n$ ($n$-grams) to build a vocabulary with all the submissions' information, i.e., invariants, syntax, or AASTs. In our case, we define $n = 3$ ($3$-grams) for this parameter of the BoW maker. Afterward, once a vocabulary has been collected, INVAASTCLUSTER computes a vector representation for each program by counting the number of times each token appears in the program's information string (invariants, syntax, or AASTs) normalizing the vector by the length of the BoW's vocabulary.

**Clustering.** The main goal of INVAASTCLUSTER is to reduce the vast number of correct submissions, $N$, into a significantly smaller number of program representatives, $K$, to help program repair frameworks to become more scalable (if $N \gg K$). Therefore, INVAASTCLUSTER accepts as parameter the number of desired clusters $K$, which is by default $10\%$ of $N$. The BoW maker module passes the set of vector representations for each one of the $N$ correct submissions to the clustering procedure. Then, INVAASTCLUSTER uses the *KMeans* algorithm to cluster these submissions into $K$ different clusters. The *KMeans* algorithm receives as a parameter the number of clusters it should return ($K$). The KMeans algorithm divides the set of observations, in our case, students' programs, into $K$ clusters, where each program is assigned to the cluster with the nearest mean [185]. INVAASTCLUSTER uses KMeans but other clustering algorithms can be applied. INVAASTCLUSTER provides users with the option to select from various clustering algorithms offered in scikit-learn [3]: Affinity Propagation, MeanShift, MiniBatch KMeans, Agglomerative Clustering, Ward, Spectral Clustering, DBSCAN, OPTICS, BIRCH, and Gaussian Mixture.

**Clusters' representatives selection.** In this last module, INVAASTCLUSTER chooses a program representative for each cluster. For each one of the $K$ clusters, INVAASTCLUSTER computes the program closest to the center of the cluster using the Euclidean distance. Afterward, INVAASTCLUSTER returns a set of $K$ clusters' representatives.

*Easy upgradability.* INVAASTCLUSTER was designed with modularity in mind. On that account, one can easily remove, add or modify any module of INVAASTCLUSTER. For example, one can use other models instead of the bag of words model, only needing to replace that specific procedure.

[2] sklearn.feature_extraction.text.TfidfVectorizer
[3] https://scikit-learn.org/stable/modules/clustering.html

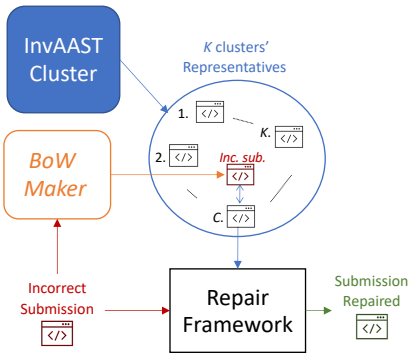



Figure 5.3: Finding the closest correct program, i.e., the closest correct program representative to the incorrect submission vector representation. This approach passes only one program to the repair tool instead of $K$ programs.

Several repair tools [9, 11, 121] are implementation-based program repair tools, i.e., they receive a single correct program to act as a reference implementation for repairing any given incorrect program. Hence, these frameworks are not designed to take advantage of a vast number of semantically different correct submissions. These frameworks can be run in parallel. However, they typically do not have a procedure to choose which among several possible repairs is the best (minimal). These tools compute the set of repairs based on the reference implementation. INVAASTCLUSTER cannot be used on these frameworks since its output is a set of $K$ clusters' representatives. Consequently, in order to allow these non-clustering-based frameworks to take advantage of INVAASTCLUSTER, we developed an additional module that finds the closest correct program representative to an incorrect program. Our motivation is that with the closest correct submission from previous years, these tools can provide the student with a minimal set of repairs.

**Closest correct program finder.** The overall idea of this module is presented in Figure 5.3. Given a student's incorrect submission, INVAASTCLUSTER finds which of the $K$ clusters' representatives, returned by INVAASTCLUSTER's selection module, is the closest program to the incorrect submission. This is done by identifying the smallest Euclidean distance between the vector representation of each one of the clusters' representatives and the incorrect submission. Hence, we can identify one correct program that is most likely the reference implementation to use for repairing a specific student's program. In the example of Figure 5.3, INVAASTCLUSTER would return only the program $C$ to the repair framework since it is the closest program to the incorrect submission.

## 5.6 Introductory Programming Assignments (IPAS) Datasets

**C-PACK-IPAS.** In order to evaluate the program representations described in Section 5.4, we used C-PACK-IPAS described in Chapter 4.

Since this work focuses only on program semantics, only submissions that compile without any errors were selected. The set of submissions was split into two sets: correct submissions and incorrect submissions. The students' submissions that satisfied a set of input-output test cases for each IPA

Table 5.1: Description of our dataset of IPAs.

| Labs | #IPAs | #Correct Submissions | #Incorrect Submissions | #IPAs (CLARA) | #Correct Submissions (CLARA) | #Incorrect Submissions (CLARA) |
|---|---|---|---|---|---|---|
| **Lab02** | 10 | 789 | 486 | 10 | 738 | 118 |
| **Lab03** | 7 | 363 | 699 | 5 | 244 | 35 |
| **Lab04** | 8 | 468 | 246 | 5 | 159 | 43 |
| **Total** | **25** | **1620** | **1431** | **20** | **1141** | **196** |

Table 5.2: Description of ITSP [8] dataset. Correct programs that our approach and CLARA do not support were removed.

| ITSP Dataset | #IPAs | #Correct Submissions | #Incorrect Submissions |
|---|---|---|---|
| **Lab3** | 4 | 45 | 63 |
| **Lab4** | 6 | 74 | 75 |
| **Lab5** | 7 | 64 | 62 |
| **Lab6** | 6 | 19 | 24 |
| **Total** | **23** | **202** | **224** |

were considered correct and selected as benchmark instances. The submissions that failed at least one input-output test were considered incorrect.

Table 5.1 presents the number of submissions gathered for C-PACK-IPAS [186]. For 25 different programming exercises, this dataset contains 1620 different correct and 1431 incorrect submissions. CLARA's clustering method does not support all the features present in the correct submissions collected. Hence, as shown in Table 5.1, after removing the set of exercises and programs that CLARA does not support, we achieved a final set of 1141 correct submissions and 196 incorrect submissions for 20 IPAS.

**ITSP.** The ITSP dataset has been used by other automated program analysis tools [8, 9, 174]. This dataset is also a collection of C programs although it is well balanced, i.e., the number of correct submissions is closer to the number of incorrect submissions in this dataset. Table 5.2 presents the number of programs in the ITSP dataset after we removed the programs that CLARA and our variable renamer module do not support.

## 5.7 Experiments

The experimental results presented in this section aim to support our claims that the proposed novel program representation based on a program's AAST and its set of program invariants help (1) to efficiently cluster semantically equivalent small imperative programs submitted in IPAS, and (2) to repair faster and significantly more IPAS' incorrect submissions in current state-of-the-art clustering-based program repair tools, such as CLARA [2].

The goal of our experiments was to answer the following research questions:

**RQ1.** How does invariant-based program clustering compare against AAST and syntax-based clustering on a set of correct submissions? (Section 5.7.1)

Table 5.3: The values for the cluster accuracy using four different clustering algorithms on each program representation after ten different runs, each run using a different seed.

| Clustering Algorithm | Program Representation | Average | Median | Variance | Standard deviation |
|---|---|---|---|---|---|
| **KMEANS** | AAST+Invariants | **81.44%** | **80.65%** | 0.02% | 1.35% |
| | AAST | 73.63% | **73.70%** | 0.03% | 1.69% |
| | Invariants | **78.69%** | **78.73%** | 0.01% | 0.88% |
| | Syntax | 58.05% | 58.15% | 0.01% | 1.22% |
| **MiniBatch KMEANS** | AAST+Invariants | 79.09% | 79.63% | 0.05% | 2.23% |
| | AAST | 72.33% | 72.96% | 0.10% | 3.12% |
| | Invariants | 75.83% | 76.23% | 0.03% | 1.68% |
| | Syntax | 58.46% | 58.21% | 0.03% | 1.81% |
| **Birch** | AAST+Invariants | 80.10% | 80.09% | 0.00% | 0.51% |
| | AAST | **73.92%** | 73.55% | 0.08% | 2.81% |
| | Invariants | 77.99% | 77.90% | 0.01% | 0.86% |
| | Syntax | **59.05%** | **58.98%** | 0.01% | 0.94% |
| **Gaussian Mixture** | AAST+Invariants | 78.89% | 79.60% | 0.05% | 2.26% |
| | AAST | 70.97% | 71.70% | 0.09% | 2.92% |
| | Invariants | 75.59% | 74.81% | 0.07% | 2.63% |
| | Syntax | 57.70% | 57.35% | 0.02% | 1.58% |

**RQ2.** Does CLARA repair more programs using INVAASTCLUSTER's closest correct submission or its set of KMEANS clusters' representatives? (Section 5.7.2)

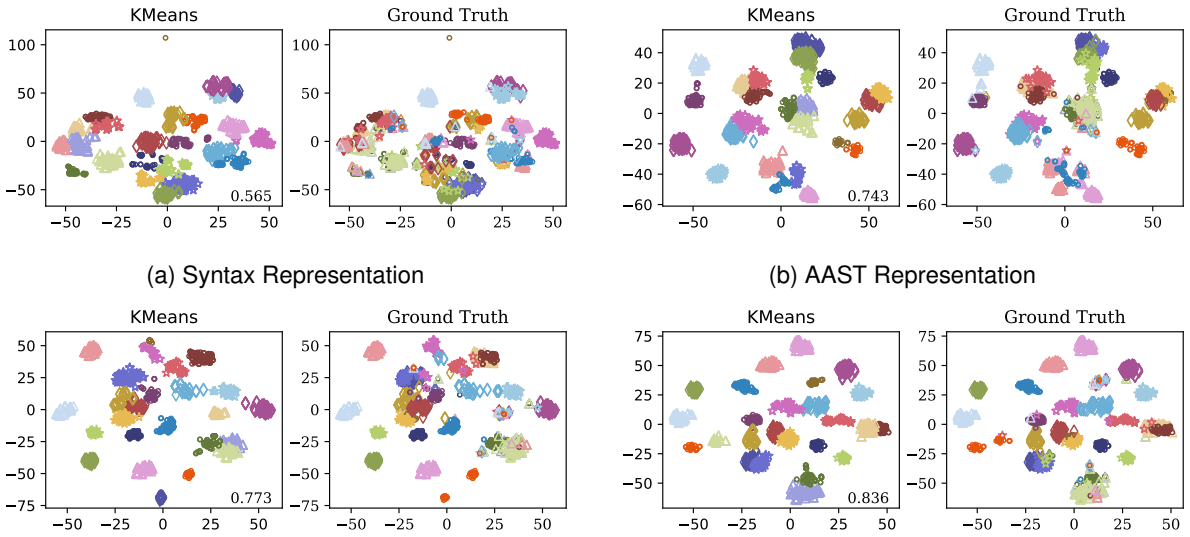
To answer these research questions, we evaluate INVAASTCLUSTER in two different use cases: (1) clustering IPAS (Section 5.7.1), and (2) repairing IPAS (Section 5.7.2). For this evaluation, we have gathered a set of IPAS, previously described in Section 5.6, developed during an introductory programming university course in C language. Section 5.7.1 presents the first use case where we perform clustering on the students' submissions for different IPAS and evaluate its accuracy on different program representations. Afterward, Section 5.7.2 shows the second use case where we integrate our program representations into a state-of-the-art program repair tool, CLARA [2], to evaluate if our clustering technique is able to outperform CLARA's clustering method, which is the only current publicly available state-of-the-art clustering method for repairing IPAS. We are going to give our program clusters for each IPA to CLARA and use CLARA's repairing process. The idea is to evaluate our clustering approach being integrated into a clustering-based program repair tool. Program repair is only one of the several possible applications for our program clustering method.

All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 64GB.

### 5.7.1 Use Case 1: Clustering IPAS

A study was performed to evaluate different program representations by applying program clustering to the set of correct programs described in Table 5.1. The main idea of this experiment was to evaluate if program invariants help identify different IPAS' submissions.

INVAASTCLUSTER was used to cluster the 1620 correct submissions, to different IPAS, into 25 distinct clusters since our dataset has 25 different programming exercises. Other works [34] that perform program clustering on IPAS perform an equivalent study on clustering submissions for different exercises. The main reason to cluster programs from 25 different exercises is that we know the ground truth




(a) Syntax Representation

(b) AAST Representation

(c) Invariant-Based Representation

(d) AAST + Invariants Representation

Figure 5.4: Comparison between the ground truth (on the right) and the clusters and cluster accuracy obtained using the KMEANS algorithm (on the left) for each type of program representation.

label for each program since we know for which specific IPA the students submit their assignments. Otherwise, we would have to manually choose semantically different implementations for the same IPA and assign labels, which might be subjective.

INVAASTCLUSTER, as explained in Section 5.5, starts by renaming all the variables in the student submissions. Then uses DAIKON [59] to collect the student submissions' dynamically generated invariants sets as described in Section 5.3. Lastly, it uses the python library, `pycparser`[4], to compute all the anonymized abstract syntax trees (AAST) (see Section 5.4.2). Using all these program features, we computed four different bags of words models. One model for each program representation (syntax, AAST, and invariants) and one additional model using a combination of a program's AAST and its invariants set. The program syntax is not included in this last representation since the bag of words based on program syntax has a large vocabulary that generates vectors that are too sparse.

The following clustering algorithms available in scikit-learn[5] were applied to each program representation: KMEANS, MiniBatch KMEANS, BIRCH and Gaussian Mixture. The *KMeans* algorithm divides the set of observations, in our case students' programs, into $n$ clusters where each program is assigned to the cluster with the nearest mean. We used the Euclidean distance as the similarity measurement for KMEANS. MiniBatch KMEANS is similar to KMeans, but instead of using the entire dataset to update the cluster centers at each iteration, this algorithm uses randomly selected subsets. On the other hand, BIRCH is a hierarchical clustering algorithm that builds a tree structure to represent the data. It incrementally clusters data points by recursively splitting clusters into subclusters. Finally, a Gaussian Mixture Model is a weighted sum of $n$ component Gaussian densities where $n$ is the number of clusters [187]. Our discussion centers around these clustering algorithms, as they yielded the most favorable outcomes in our experiments.

[4]https://github.com/eliben/pycparser
[5]https://scikit-learn.org/stable/modules/clustering.html

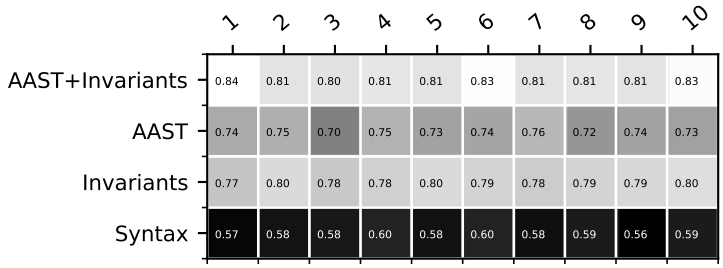



Figure 5.5: The values for cluster accuracy using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

Since our dataset of IPAS has 25 different programming exercises, the ground truth has 25 different clusters. Each student program is a submission to a specific programming exercise (label) that we know. Consequently, the *cluster accuracy* metric can be used to evaluate the obtained clusters. With this metric, each cluster is assigned the label (exercise) which is most frequent in the cluster. Afterward, the accuracy of this assignment is measured by counting the number of correctly assigned student submissions and dividing by the number of total submissions. This metric is also known as *purity* [188].

Table 5.3 presents the average, median, variance, and standard deviation values for the cluster accuracy for each clustering algorithm on four different program representations after ten different runs. Each run uses a different seed. Entries highlighted in bold correspond to the highest average/median accuracy values for each different program representation for all clustering algorithms. One can see that the AAST + Invariants representation has the best performance considering all the clustering algorithms. The Invariants BoW has the second highest accuracy, followed by the AAST BoW. Lastly, the Syntax BoW presents the poorest performance of all. From now on, we focus the discussion on the KMEANS results since this clustering algorithm achieved the best results (see Table 5.3).

The KMEANS clustering algorithm divides the set of observations, in our case, students' programs, into $n$ clusters where each program is assigned to the cluster with the nearest mean [185]. The KMEANS algorithm receives as a parameter the number of clusters it should return, i.e., it always returns 25 different clusters of programs. Figure 5.5 shows a matrix with the different cluster accuracy values using the KMEANS algorithm on each program representation for ten different seeds. Each entry is highlighted accordingly to its value. The lowest value is highlighted in black, and the highest is highlighted in white. Intermediate values are highlighted in different shades of grey, depending on how far they are from the lowest value. Matrices with values of the cluster accuracy for the other clustering algorithms can be found in Appendix B.1.1.

Furthermore, Figure 5.4 presents the results of applying the KMEANS model to each one of the four program representations being analyzed. To present these results graphically, we used a method for visualizing high-dimensional data in a $2-$dimension map, called T-SNE [189]. Each subfigure corresponds to a different type of representation. The left side of each subfigure shows the clustering results and the value of the cluster accuracy (right-bottom corner). The right side presents the real clusters of each programming exercise, i.e., the ground truth represented using each program representation. Figure 5.4a shows the results after clustering all the student submissions using a syntax representa-

tion, which resulted in a cluster accuracy of almost $57\%$. The AAST representation achieved a cluster accuracy of $74.3\%$ as presented in Figure 5.4b.

Regarding the use of program invariants, Figure 5.4c and Table 5.3 support the idea that program invariants improve program clustering since this representation obtained a cluster accuracy of $77.3\%$. Lastly, Figure 5.4d presents the representation that uses the combination AASTs and invariants sets, which also shows an improvement compared to the invariant-based representation. This representation outperforms all the other representations with an accuracy of $83.6\%$. Furthermore, Table 5.3 shows that this representation based on AAST and invariants achieved the best cluster accuracy for all clustering algorithms. Another advantage of this representation is that it best separates all the students' submissions in different regions of the space, i.e., the majority of the clusters are visibly separated from each other.

Other evaluation metrics for the KMEANS algorithm can be found in Appendix B.1.2 for the interested reader's convenience. Clustering metrics such as the *Rand index*, the *adjusted Rand index*, the *normalized mutual information*, the *adjusted mutual information*, the *FowlkesMallows index*, the *completeness score*, the *homogeneity score*, and the *V measure*.

### 5.7.2 Use Case 2: Repairing IPAs

This section presents the results of integrating INVAASTCLUSTER as the clustering approach for CLARA [2], a publicly available state-of-the-art clustering-based program repair tool. Since our set of IPAS, described in Table 5.1, has a small number of incorrect submissions, only 196, for this evaluation, we have also considered the ITSP dataset [8] described in Section 5.6. Thus, overall we have a total of 420 incorrect submissions (196 from our dataset plus 224 from the ITSP dataset) and 1343 correct submissions (1141 from our dataset plus 202 from the ITSP dataset) for 43 different IPAS (20 from our dataset plus 23 from the ITSP dataset). To fully evaluate our clustering technique for repairing IPAS, we are going to compare INVAASTCLUSTER's results against CLARA's in terms of: (1) the number of student submissions repaired; (2) the number of clusters produced by each clustering approach for each IPA; and (3) the time spent to repair each incorrect submission.

We would like to point out that in this experiment, we are not trying to improve CLARA's repair process. Instead, we are comparing the performance of CLARA's repair process when using its own or INVAASTCLUSTER's clusters of programs.

Different procedures for program clustering using INVAASTCLUSTER (see Table 5.4) are evaluated:

- KMEANS - *BoW*: Uses KMEANS and four different bag of words (BoW) based on AAST, syntax, and invariants (lines 2–5 in Table 5.4);
- *Closest Program (*KMEANS*)* - AAST *+ Invs*: Uses the closest program (see Section 5.5) using the AAST + Invs BoW, from a set of clusters' representatives using KMEANS (line 6);

Table 5.4 presents the overall repair evaluation on 319 incorrect submissions since CLARA's repair algorithm does not support the C implementation of 101 incorrect submissions (24.05% of the instances). Entries in bold correspond to the highest rate of submissions repaired, the lowest percentage of struc-

Table 5.4: This table presents the percentage of submissions repaired (success), structural mismatch errors, and timeouts (failure) for each clustering approach. The total number of submissions is 319.

| | | %Success | %Failure | |
|---|---|---|---|---|
| | **Clustering Method** | **%Submissions Repaired** | **%Structural Mismatch** | **%Timeouts (600s)** |
| **1** | CLARA | 71.79% | 7.52% | **20.69%** |
| **2** | KMEANS - Invs | 82.45% | **11.91%** | 5.64% |
| **3** | KMEANS - Syntax | 84.01% | 10.97% | 5.02% |
| **4** | KMEANS - AAST | 84.64% | 9.72% | 5.64% |
| **5** | KMEANS - AAST + Invs | **84.95%** | 10.03% | 5.02% |
| **6** | Closest Program (KMEANS) - AAST + Invs | 84.33% | 10.66% | 5.02% |

tural mismatch errors, or the lowest rate of timeouts, i.e., executions that did not repair a program using a timeout of 10 minutes (600s).

One can see in line 1 (Table 5.4) that CLARA, using its own clusters, can only repair 229 (around 72%) of the incorrect submissions and shows the largest percentage of instances that were not repaired due to timeout (20.69%). Secondly, the configuration using INVAASTCLUSTER's KMEANS and the BoW based on AAST and Invariants achieved the highest score, repairing 84.95% of the incorrect submissions. Furthermore, the BoW based only on invariants has the highest percentage of structural mismatch (11.91%), which may be explained by CLARA's inability to use a program with a different control flow in the repair process. Using only invariants on a vector representation helps clustering programs with similar semantics, although it does not take into account the programs' structure (control flow). Hence, a higher rate of structural mismatch is observed.

Since the BoW based on AASTS and program invariants achieved the best results both in the program clustering experiment (see Section 5.7.1) as well as when repairing submissions (lines 2-5 in Table 5.4), we opted to use only this BoW when finding the closest correct program (line 6, Table 5.4). Regarding the use of just one correct solution to fix an incorrect submission, the *Closest Program (*KMEANS*)* approach did not achieve better results than using the set of clusters' representatives.

We have also analyzed the closest program technique using all submissions, i.e., use the closest program among all submissions (no clustering step). This approach, the *Closest Program (All Submissions)*, was able to repair 86.5% of the submissions. The number of timeouts in this approach and using KMEANS was similar. Once again, this high rate of repaired programs (86.5%) may be explained by CLARA's strict requirements for both programs, the program being repaired and the correct program used by the repair process, to have the same control flow. Therefore, when INVAASTCLUSTER finds the closest program among all submissions instead of using clusters, INVAASTCLUSTER has a more diverse collection of programs' structures. Consequentially, the *Closest Program (All Submissions)* approach also achieved the lowest score of structural mismatch errors (only 9.4%). Although we would like to draw the reader's attention to the difference between the number of submissions repaired using KMEANS (85%) or using the closest correct program (86.5%), which is less than 2%. Furthermore, the computation to find the closest correct program among all correct submissions can only be done online since it requires the student's incorrect program. On the other hand, the computation of the KMEANS clusters can be

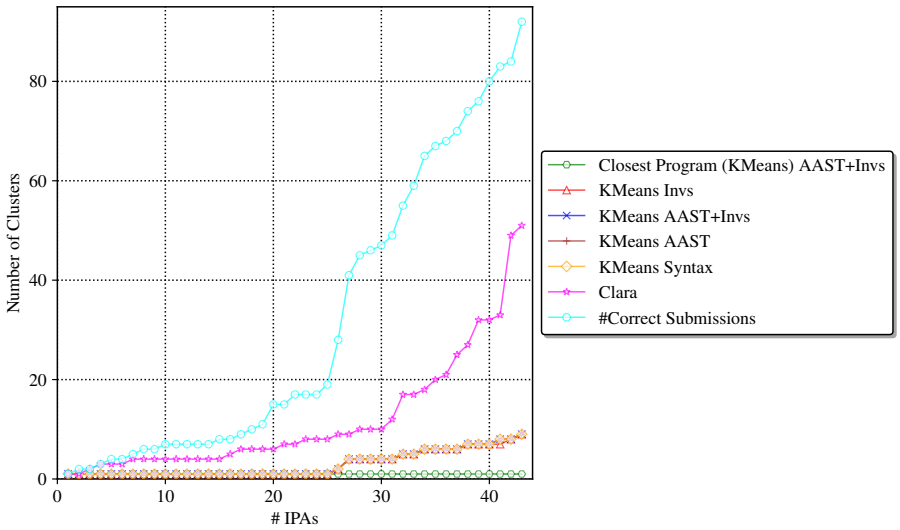



Figure 5.6: Cactus plot - The number of clusters generated by each clustering technique for 43 different IPAs.

done offline since it only requires past students' correct submissions. In this evaluation, this is not a concern since each IPA has at most a hundred correct submissions. However, in a large-scale MOOC with thousands of correct submissions per exercise, the process of finding the closest correct program among all of the submissions may become impractical to compute in a short period of time.

#### 5.7.2.1 Number of Clusters

Figure 5.6 illustrates a cactus plot detailing the number of clusters generated by each clustering technique for each of the 43 different IPAs used. One can see that CLARA generates an enormous quantity of clusters, almost half of the correct submissions of each IPA. This large number of clusters is explained by CLARA's strict clustering method, which does not allow two programs to be in the same cluster if there is no exact match between both programs' control flows. Furthermore, even if two programs are semantically equivalent and share the same control flow but one of them uses a for-loop and the other uses a while-loop, then CLARA assigns these two programs to different clusters.

INVAASTCLUSTER produces $K$ clusters, which in this experiment is always set to 10% of the number of correct submissions of each exercise. The technique that uses the closest correct program has only a single cluster which is the closest correct program. This evaluation of the number of clusters used by each approach allows us to observe that CLARA produces a large number of clusters, resulting in a detriment of performance. In contrast, our approach can generate fewer clusters resulting in a more effective repairing process.

**Example 9.** The following two programs are correct implementations for the IPA where students are asked to print the maximum value among three given numbers. These programs are clustered together using INVAASTCLUSTER (AASTs + Invariants) because their AASTs are quite similar, and their sets of program invariants are identical. In contrast, CLARA assigns these two programs to different clusters because they have a different number of variables. Therefore, because CLARA is highly strict in comparing

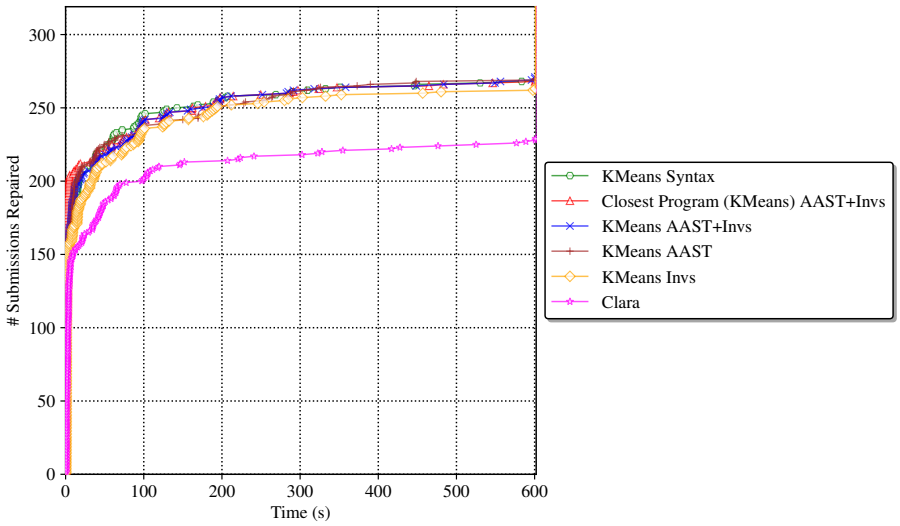



Figure 5.7: Cactus plot - Time Performance (timeout=600s).

programs, it generates an excessive number of program clusters.

```c
int main(){
    int n1, n2, n3, max;
    scanf("%d%d%d", &n1, &n2, &n3);
    max = n1;
    if(n2 > max){
        max = n2;
    }
    if(n3 > max){
        max = n3;
    }

    printf("%d\n", max);
    return 0;
}
```

```c
int main(){
    int res,b,c;
    scanf("%d%d%d", &res, &b, &c);

    if (b > res){
        res = b;
    }
    if (c > res){
        res = c;
    }

    printf("%d\n", res);
    return 0;
}
```

#### 5.7.2.2 CPU time

Regarding the time performance of each clustering technique, Figure 5.7 shows a a cactus plot that presents the CPU time spent on repairing each program ($x$-axis) against the number of repaired programs ($y$-axis) using different clustering techniques. The legend in this plot is not sorted.

One can clearly see a gap between CLARA's time performance and INVAASTCLUSTER's (considering any clustering approach). For example, after 100 seconds CLARA using its own clusters, can only repair around 200 programs while using our clusters, it can repair around 230/250 programs. Furthermore, Figure 5.8 presents a scatter plot comparing the CPU time spent using CLARA's clusters against the KMEANS AAST + Invariants clusters. Each point in this plot represents a program where the $x$-value (resp. $y$-value) is the CPU time spent to repair the program using the KMEANS AAST + Invariants Clusters (resp. CLARA's clusters). If a point is above the diagonal, then it means that our clusters outperformed CLARA's clusters since using our clusters CLARA is able to repair each program above the

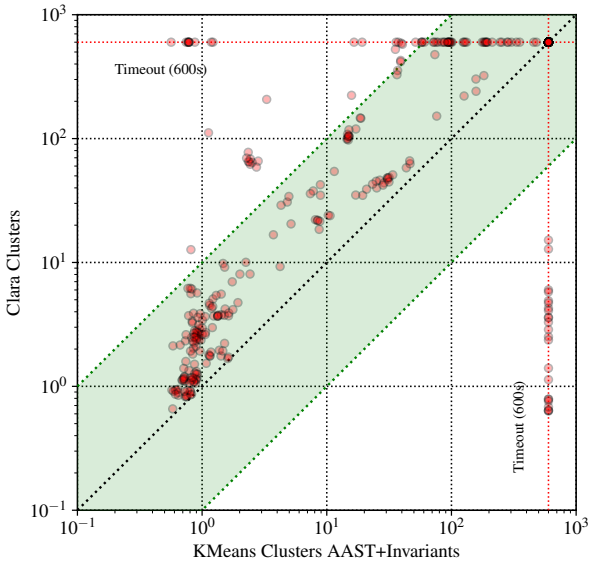



Figure 5.8: Scatter plot - Time Performance (timeout=600s) - CLARA VS INVAASTCLUSTER (KMEANS w/ AAST + Invariants)

diagonal faster than using its own clusters. Thus, if we consider the programs repaired by both clustering methods, using our clusters is always faster than using CLARA's.

There are two main reasons for CLARA's time performance. Firstly, CLARA generates a significantly larger number of clusters compared to INVAASTCLUSTER (see Figure 5.6). Consequently, CLARA needs to compute a set of repairs for each cluster's representative, resulting in more time spent in the repair process due to the larger number of clusters. Secondly, as explained in Section 5.2, CLARA maintains a `json` file containing data (e.g., expressions) of all the programs belonging to the cluster. During its repair algorithm, CLARA takes advantage of all this data to repair a given submission, leading to more time spent on the repair process.

The main goal of educational program repair frameworks is to provide real-time feedback to students on how they should repair their submissions. In this evaluation, we used a timeout of 10 minutes. However, a student expects a faster result. Therefore, Figure 5.9 shows another cactus plot that shows the time performance of the clustering approaches, although with a timeout of 10 seconds. One can see that after 10 seconds, CLARA using its own clusters, can only repair around 150 submissions (47%). On the other hand, using INVAASTCLUSTER, CLARA can repair around 200 submissions (63%). Furthermore, one can also verify that after 2 seconds CLARA can only repair around 75 programs using its own cluster while using our clusters CLARA is able to repair around 150 programs.

#### 5.7.2.3 Program invariants of incorrect submissions

We have also tried the method *Closest Program (*KMEANS*)* with INVAASTCLUSTER not taking into account incorrect submissions' invariants to check if incorrect submissions' sets of invariants had a negative impact on program representations. First, INVAASTCLUSTER clustered all the correct programs considering their AASTs and their sets of invariants. Secondly, to find the closest correct program to an incorrect submission INVAASTCLUSTER used only the AAST BoW. However, this combined approach

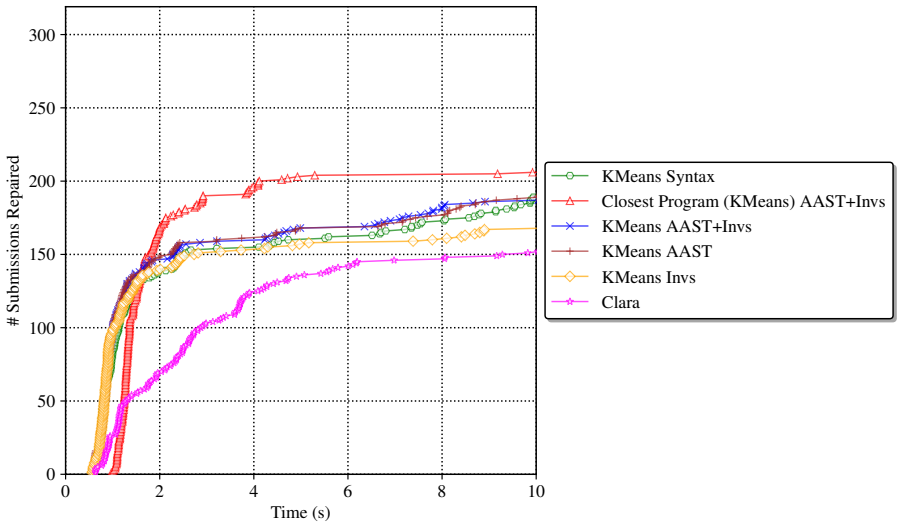



Figure 5.9: Cactus plot - Time Performance (timeout=10s).

of clustering with one BoW (AASTs + invariants) and calculating the programs' distances with another (AASTs only) was only able to repair 269 submissions (84%). Thus, according to this experiment, incorrect submissions' sets of program invariants do not cause negative effects on program representations.

To summarize, in Section 5.7.1, we used InvAASTCluster to cluster different IPAs correct students' submissions. The obtained results support that this work's novel program representation based on a program's AAST and invariants performs better when compared to representations solely based on a program's code, AST, or set of program invariants.

Furthermore, in Section 5.7.2, we integrated InvAASTCluster into Clara to evaluate our tool's performance when integrated into a clustering-based framework for repairing IPAs. This study shows that InvAASTCluster significantly increases the performance of the state-of-the-art clustering technique by allowing Clara to repair more student submissions and doing so notably faster.

### 5.7.3 Threats to Validity

This work relies on Daikon to compute dynamically-generated likely invariants. Employing a different tool for invariant detection could yield different results. Moreover, the quality of dynamically-generated invariants are highly dependent on the input-output test suite used for each programming exercise. Thus, using a different test suite may lead to alternative sets of invariants across student submissions.

Our evaluation of InvAASTCluster is limited to small imperative programs typically found in IPAs. While we defer evaluation on more complex programs to future work, we do not foresee major scalability issues when applying InvAASTCluster to larger datasets or more sophisticated code. The number of clusters produced by InvAASTCluster tends to scale with the diversity of both semantic and syntactic implementations in each IPA. Although greater implementation diversity is expected with more complex exercises, the scalability of InvAASTCluster should remain unaffected, as it does not impose limits on the number of supported IPAs.

Although our experiments exclusively considered C programs, the underlying clustering methodology of INVAASTCLUSTER is language-agnostic. Supporting other languages would primarily involve adapting (1) the variable renaming module and (2) the invariant detection componentreplacing DAIKON with a suitable alternative for the target language.

Similarly, employing a different program repair tool could lead to different outcomes. For instance, alternative tools might support C features that CLARA does not handle, or vice versa.

INVAASTCLUSTER also has potential applications beyond repair, such as detecting similarities or possible plagiarism among student submissions via AASTS and program invariants. However, realizing this would require adapting existing plagiarism detection tools. Notably, popular tools like MOSS are not open-source, which presents an additional limitation.

In this chapter, we compared INVAASTCLUSTER against CLARA's clustering approach, as CLARA is, to our knowledge, the only publicly available state-of-the-art clustering-based repair tool for IPAS. Evaluating INVAASTCLUSTER in conjunction with other recent tools would be worthwhile. Unfortunately, we could not find publicly available implementations or datasets for other relevant tools [10, 34, 120, 133], aside from the ITSP dataset [8].

## 5.8 INVAASTCLUSTER vs SEMCLUSTER

Similar to INVAASTCLUSTER, SEMCLUSTER, described in Section 3.3.2.2, uses the KMEANS clustering algorithm. However, unlike INVAASTCLUSTER, which uses each program's AAST, SEMCLUSTER does not account for any syntactic features. Furthermore, SEMCLUSTER tries to capture the semantics of a program by counting the number of different values each variable takes. On the other hand, INVAASTCLUSTER considers the program's set of invariants which can be more robust and independent of the test suite used. CODEBERT [190], CODE2SEQ [191] and other deep learning models [192] build vector representations of programs by training machine learning models using the programs' code and ASTs. However, unlike INVAASTCLUSTER, these techniques only consider the programs' syntax, not their semantics. Some research has been conducted regarding *the use of invariants to promote patch diversity* or to help with patch selection on a search-based program repair [180–182].

## 5.9 Conclusions

In the context of introductory programming assignments (IPAS) in university courses or MOOCs, it is possible to collect a large number of correct implementations for the proposed IPAS. Hence, when a student submits an incorrect program, one can take advantage of previously correct submissions to automatically suggest repairs that help the student. However, it is not feasible to analyze all possible previous correct submissions to find an appropriate reference implementation for the repair tool. Therefore, clustering is often used to identify similar program implementations. Afterward, the automated repair tool analyses a single reference program from each cluster to find the most suitable correction to the student's incorrect submission.

This chapter proposes INVAASTCLUSTER, a novel approach for program clustering based on their semantic and syntactic features. INVAASTCLUSTER uses AASTS and invariant-based program representations to distinguish small imperative programs according to their semantics (invariants) and structure (AAST). Results show that the proposed AASTS and invariant-based representation improve upon syntax-based representations when performing program clustering on several correct student submissions for different programming exercises. Additionally, given an incorrect student submission and a set of correct students' submissions, INVAASTCLUSTER can also find the closest correct submission to the faulty program using INVAASTCLUSTER's program vector representations.

Furthermore, INVAASTCLUSTER has also been integrated into a state-of-the-art clustering-based program repair framework to evaluate the proposed clustering techniques for repairing IPAS. This evaluation showed that INVAASTCLUSTER outperforms the current state-of-the-art clustering method used in clustering-based program repair. Using INVAASTCLUSTER, the automated repair tool CLARA can repair significantly more IPAS, around 13%, with a better time performance and with a smaller number of program clusters.

To conclude, INVAASTCLUSTER is a program clustering tool based on programs' invariants and AASTS. INVAASTCLUSTER can be used: (1) to cluster semantically equivalent implementations for programming exercises; (2) by any clustering-based program repair tool; and (3) by any program repair framework that requires a single correct implementation (INVAASTCLUSTER's closest correct program).

In Chapter 10, INVAASTCLUSTER is integrated into a Large Language Model (LLM)-driven program repair tool to assess the impact of INVAASTCLUSTER in assisting the repair process of LLMS.

# 6

# MULTIPAS: Applying Program Transformations to Introductory Programming Assignments for Data Augmentation

*"A computer program can modify itself but it cannot violate its own instructions  it can at best change some parts of itself by obeying its own instructions."*

– Douglas R. Hofstadter. Gödel, Escher, Bach: an Eternal Golden Braid [1]. 1979.

## Contents

Datasets of IPAs publicly available tend to be small and with no valuable annotations about the defects of each program. Small datasets are not very useful for program repair tools that rely on machine learning models. Furthermore, a large diversity of correct implementations allows computing a smaller set of repairs to fix a given incorrect program rather than always using the same set of correct implementations for a given IPA. For these reasons, there has been an increasing demand for the task of augmenting IPAs benchmarks.

This chapter presents MULTIPAS, a program transformation tool that can augment IPAs benchmarks by: (1) applying six syntactic mutations that conserve the program's semantics and (2) applying three semantic mutilations that introduce faults in the IPAs. Moreover, this chapter demonstrates the usefulness of MULTIPAS by augmenting with millions of programs two publicly available benchmarks of programs written in the C language, and also by generating an extensive benchmark of semantically incorrect programs.

This chapter has been published as a conference paper at the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022 [174].

## 6.1 Introduction

An increasing body of research has focused on applying machine learning (ML) models to automated program repair [3, 13, 15, 42, 172, 193–198]. These ML-based tools depend greatly on the existence of many correct/incorrect programs to train their repair models. However, in most cases, the publicly available benchmark sets [171, 176] of students' submissions for IPAs are small, i.e., only hundreds of submissions. Hence, these benchmarks might not be sufficient to effectively train an ML model. Additionally, another problem with some real-world IPAs datasets is that sometimes there is no knowledge about the number and types of defects present in each incorrect student program, which can also negatively impact the training of ML-based program repair tools.

Hence, there is an increasing demand for data augmentation of program benchmarks to (1) achieve minimal sets of program patches by having a more diverse collection of syntactically different correct solutions and (2) have a more representative dataset of programs to train ML-based program repair tools with labelled incorrect programs. This data augmentation task aims to enlarge the real-world datasets of students' programs with more semantically correct implementations for each IPA by syntactically mutating existent correct students' submissions and to create a labelled dataset of incorrect programs with the information about the number and the type of the bugs present in each incorrect program.

Thus, this chapter presents MULTIPAS, a tool that performs data augmentation by syntactically mutating and/or semantically mutilating IPAs written in the C programming language. The main goal of MULTIPAS is to augment IPAs benchmarks with: (1) more semantically correct implementations by applying six different syntactic mutations to pre-existent correct implementations and (2) new semantically incorrect programs by mutilating pre-existent correct implementations. MULTIPAS stores the variable mapping between the original correct implementation and the new mutated/mutilated program. More-

over, for the newly generated set of incorrect programs, MULTIPAS also stores the information about the bugs in these programs. Later, these bug(s) annotations can be used to train ML models.

Experimental results show that MULTIPAS can augment small-sized publicly available benchmarks of IPAS, ITSP [8] and C-PACK-IPAS [186], generating millions of mutated/mutilated programs. To summarize, this chapter makes the following contributions:

- It presents MULTIPAS, a program transformation framework capable of augmenting small imperative program benchmarks by performing six different syntactic program mutations and three semantic program mutilations;
- MULTIPAS is publicly available on GitHub: https://github.com/pmorvalho/MultIPAs, with a demo video at https://arsr.inesc-id.pt/ pmorvalho/MultIPAs-demo.html.
- MULTIPAS keeps the information about the types and the number of bugs present in each generated incorrect program, which can be used to train ML-based program repair frameworks;
- MULTIPAS produces a variable mapping between the original program given as input and the mutated/mutilated program.

## 6.2 MULTIPAS

This section presents MULTIPAS, a new tool capable of augmenting IPAS benchmark sets by applying syntactic mutations and semantic mutilations to C programs. MULTIPAS is divided into two modules: program mutator and program mutilator. The C programs are parsed and the changes (program mutations and mutilations) happen at the AST level. Section 6.2.1 presents the six different syntactic program mutations that MULTIPAS can perform to change a program syntax while preserving its semantics. Afterwards, Section 6.2.2 explains three different semantic program mutilations that introduce semantic bugs in an IPA. Finally, Section 6.2.3 explains briefly the variable mappings produced by MULTIPAS. MULTIPAS is publicly available on GitHub: https://github.com/pmorvalho/MultIPAs and there is also a demonstration video available at https://arsr.inesc-id.pt/ pmorvalho/MultIPAs-demo.html.

### 6.2.1 Program Mutator

The goal of automated program repair when applied to IPAS is to achieve the best possible set of repairs (i.e., program patches) to fix a given student's incorrect submission for a programming exercise. The best repair is usually described as a minimal set of fixes required to make the student's program compliant with the test suite that describes the desired semantic behaviour for that specific IPA. To this end, many program repair tools, such as Verifix [2, 9], try to align the student's submission's control flow graph with another correct submission's control flow graph. Next, these frameworks propose a set of syntactic patches to fix the incorrect program. Hence, applying program mutations to an IPAS benchmark increases the number of different syntactic structures and allows program repair tools to achieve smaller sets of repairs. For that reason, MULTIPAS can perform syntactic mutations to a program

such that it preserves the program semantics, i.e., both programs, the original and the mutated, have the same behaviour.

The six syntactic program mutations available on MULTIPAS are:

- *M1 - Comparison Expression Mirroring (CEM)*: MULTIPAS mirrors one or several comparison expressions e.g., $a \geq b$ becomes $b \leq a$;

- *M2 - If-else-statements Swapping (IES)*: MULTIPAS swaps the if-branch and the else-branch and negates the if-condition. This operation is done only for simple if-else-statements, i.e., there are no additional if-statements inside the else-branch;

- *M3 - Increment/Decrement Operators Mirroring (IOM)*: MULTIPAS mirrors the two increment (and decrement) operators in the C programming language (e.g., `c++` and `++c`), only when the return value of the expression that contains the increment/decrement operator is discarded, e.g., the increment step of a for-loop;

- *M4 - Variable Declarations Reordering (VDR)*: MULTIPAS reorders the variables' declarations present in each code block. For this, MULTIPAS takes into account the dependencies between the variables' declarations, i.e. if a variable declaration depends on other variables, this is done by computing all possible topological orders of the variables' declarations;

- *M5 - For-2-While Translation (F2W)*: MULTIPAS translates for-loops into while-loops. Just in cases of for-loops that do not contain any continue instructions;

- *M6 - Variable Addition (VA)*: MULTIPAS introduces a new dummy variable declaration in the program. The mutated program does not have the same set of variables as the original program.

**Example 10.** Consider the two programs in Listings 6.1 and 6.2 in the C programming language. Both programs are semantically equivalent since both programs sum all the natural numbers from $1$ to $n$ and print the current accumulated value in each iteration. The program in Listing 6.2, the mutated program, is the result of applying all the program mutations available on MULTIPAS to the program in Listing 6.1, the original program. Note that the comparison expression in the for-loop condition was mirrored (mutation *(1)*). Mutation *(2)* is not applicable since there is no if-else-statement. Regarding mutation *(3)*, one can see that the increment step of the for-loop was also mirrored, line 6 (resp. 9) in the original (resp. mutated) program. Furthermore, the mutated program has a different variable declaration order than the original program (mutation *(4)*). Moreover, the for-loop was translated into a similar while-loop corresponding to mutation *(5)*. Lastly, a dummy variable `y` was introduced in line 3 of the mutated program (mutation *(6)*).

**Listing 6.1:** Original program.

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5    s=0;
6    for(i=1; i<=n; i++){
7      s = s+i;
8      printf("%d\n",s);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

**Listing 6.2:** Mutated program.

```
1  int main(){
2    int n, s, i, y;
3    scanf("%d", &n);
4    s=0;
5    i = 1;
6    while(n>=i){
7      s = s+i;
8      printf("%d\n",s);
9      ++i;
10   }
11   printf("%d\n",s);
12   return 0;
13 }
```

### 6.2.2 Program Mutilator

In the development of program repair tools, there are two main concerns on using incorrect programs of IPAs datasets: (1) usually, there is no knowledge of how many errors are present in each buggy student program; and (2) since the number of semantic errors on each program is unknown, repair framework's developers cannot divide the set of the incorrect programs into subsets of programs with a specific number of semantic errors (e.g., a subset for programs with one semantic error, another subset for programs with two semantic errors, etc.). Furthermore, dividing the dataset of incorrect IPAs into subsets of different numbers or types of bugs can be important to train ML-based program repair tools [193–196]. Therefore, having a program mutilator that creates a dataset of programs with a specific number of semantic bugs and only certain kinds of bugs becomes crucial. This way, developers of program repair tools' can evaluate the scalability of their frameworks in terms of the number of semantic errors present in each program and train their tools to repair specific families of bugs.

Thus MULTIPAS also contains a program mutilator module. This program mutilator takes a set of students' submissions for a given IPA and alters each program to introduce $n$ errors, $n$ being passed as a parameter by the user. The errors introduced by MULTIPAS are semantic mutilation which modifies the programs' semantics. The following three program mutilations (bugs) are available on MULTIPAS:

- *B1 - Wrong Comparison Operator (WCO)*: MULTIPAS swaps an expression's comparison operators for some syntactically similar operator. For example, swaps the operator < for <=. MULTIPAS can also swap > for >=, <= for <, >= for >, == for =, and != for ==;

- *B2 - Variable Misuse (VM)*: MULTIPAS swaps a variable in the program by another variable of the same type. The resulting mutilated program can be compiled successfully since MULTIPAS ensures that both variables are of the same type;

- *B3 - Assignment Deletion (AD)*: MULTIPAS deletes an assignment expression in the program.

**Listing 6.3:** Original program.

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5    s=0;
6    for(i=1; i<=n; i++){
7      s = s+i;
8      printf("%d\n",s);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

**Listing 6.4:** Mutilated program.

```
1  int main(){
2    int n;
3    int i, s;
4    scanf("%d", &n);
5
6    for(i=1; i<n; i++){
7      s = s+i;
8      printf("%d\n",i);
9    }
10
11   printf("%d\n",s);
12   return 0;
13 }
```

**Example 11.** Consider the two programs in Listings 6.3 and 6.4 written in the C programming language. The program in Listing 6.3, hereafter the original program, has been already presented in Listing 6.1. The program in Listing 6.4, hereafter the mutilated program, is the result of applying all the program mutilations available in MULTIPAS. The first mutilation is located in line 6 of the mutilated program where the operator <= was swapped by the operator <. Furthermore, the *variable misuse* mutilation was performed in line 8. Lastly, MULTIPAS removed the assignment of value zero to variable `s` (line 5).

The first class of bugs, *wrong comparison operator*, is common among novice programmers [176] and has been used to evaluate ML-based program repair tools [194, 197]. The second family of bugs, *variable misuse*, is also common among students as well as among experienced programmers [199, 200]. This specific task has received a lot of attention from the ML research community [192–195]. Lastly, the *assignment deletion* bug is also common among students [176]. In the previous example, it is likely for a novice student to forget to initialize variable `s`.

**Bug mapping.** For each mutilated program generated, MULTIPAS stores the information about the location and the types of the bugs introduced in the program. This information can help train ML-based program repair frameworks.

### 6.2.3 Variable Mapping

Typically, semantic program repair tools [2, 9] repair an incorrect program using a correct implementation for the same IPA. In order to compare two programs, it is required a relation between both sets of variables. For example, consider the programs presented in Listings 6.3 and 6.4. In this case, having a mapping between both programs' variables lets the repair framework reason about which program modifications it should perform to fix the faulty program. Program modifications include the same variable being used in a different comparison expression, the variable being initialized in one program but not in the other one, etc. Moreover, the variable mapping can also be helpful for the task of *code adaption*

Table 6.1: Number of programs that can be generated by MULTIPAS using each different mutation or mutilation for two different small datasets of IPAS: ITSP [8] and C-PACK-IPAS [186].

| | | | Mutations | | | | | | | Mutilations (Bugs) | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **ITSP Dataset [8]** | **#IPAs** | **#Correct Submissions** | **M1 (CEM)** | **M2 (IES)** | **M3 (IOM)** | **M4 (VDR)** | **M5 (F2W)** | **M6 (VA)** | **All Mutations** | **B1 (WCO)** | **B2 (VM)** | **B3 (AD)** | **All Bugs** |
| **Lab3** | 4 | 67 | 1.25E+03 | 9.90E+01 | 6.70E+01 | 4.74E+05 | 6.70E+01 | 1.34E+02 | 6.03E+06 | 1.86E+02 | 4.51E+03 | 1.51E+02 | 4.71E+04 |
| **Lab4** | 8 | 125 | 3.99E+04 | 2.22E+02 | 3.15E+02 | 8.02E+05 | 2.41E+02 | 2.30E+02 | 1.90E+11 | 9.93E+02 | 1.20E+04 | 4.16E+02 | 7.12E+05 |
| **Lab5** | 8 | 90 | 1.06E+04 | 1.59E+02 | 5.13E+02 | 3.78E+05 | 4.45E+02 | 1.80E+02 | 3.07E+12 | 5.24E+02 | 4.43E+03 | 3.78E+02 | 1.52E+05 |
| **Lab6** | 8 | 87 | 1.94E+04 | 1.29E+02 | 5.36E+03 | 1.12E+06 | 1.45E+03 | 1.74E+02 | 9.75E+13 | 5.52E+02 | 6.32E+03 | 5.70E+02 | 4.02E+05 |
| **Total** | **28** | **369** | **7.12E+04** | **6.09E+02** | **6.25E+03** | **2.77E+06** | **2.20E+03** | **7.18E+02** | **1.01E+14** | **2.26E+03** | **2.73E+04** | **1.52E+03** | **1.31E+06** |
| **Cx-Pack-IPAs Dataset [186]** | **#IPAs** | **#Correct Submissions** | **M1 (CEM)** | **M2 (IES)** | **M3 (IOM)** | **M4 (VDR)** | **M5 (F2W)** | **M6 (VA)** | **All Mutations** | **B1 (WCO)** | **B2 (VM)** | **B3 (AD)** | **All Bugs** |
| **Lab02** | 10 | 316 | 1.04E+04 | 4.49E+02 | 4.88E+02 | 3.64E+06 | 4.07E+02 | 6.32E+02 | 1.72E+07 | 9.68E+02 | 9.71E+03 | 1.11E+03 | 2.93E+05 |
| **Lab03** | 7 | 145 | 3.21E+05 | 3.20E+02 | 1.09E+03 | 4.93E+04 | 6.07E+02 | 2.90E+02 | 1.48E+10 | 1.02E+03 | 4.94E+03 | 8.07E+02 | 3.94E+05 |
| **Lab04** | 8 | 192 | 2.83E+04 | 2.85E+02 | 1.97E+03 | 8.72E+03 | 1.28E+03 | 3.80E+02 | 1.93E+11 | 1.07E+03 | 5.58E+03 | 1.08E+03 | 2.39E+05 |
| **Total** | **25** | **653** | **3.59E+05** | **1.05E+03** | **3.54E+03** | **3.70E+06** | **2.29E+03** | **1.30E+03** | **2.08E+11** | **3.06E+03** | **2.02E+04** | **2.99E+03** | **9.26E+05** |

where the repair framework tries to adapt all the variable names in a pasted snippet of code, copied from another program or a Stack Overflow post to the surrounding preexisting code [196].

Thus, every time MULTIPAS mutates or mutilates a program, a mapping between the original program's set of variables and the mutated/mutilated program's sets of variables is generated. This variable mapping can help a program repair framework [2, 196] to find a minimal repair.

**Example 12.** MULTIPAS would produce the following variable mapping between the set of variables of the programs in Listings 6.1 and 6.2: `{int i: int i; int n : int n; int s : int s; int y : UNK_VAR}`. Moreover, for the program in Listings 6.3 and 6.4 the variable mapping would be: `{int i: int i; int n : int n; int s : int s}`.

## 6.3 Evaluation

The experimental results presented in this Section show the evaluation of MULTIPAS on two publicly available small-sized datasets of IPAS: ITSP [8] and C-PACK-IPAS [186]. The evaluation consists of using MULTIPAS to augment both benchmark sets by mutating or mutilating all correct programs. Table 6.1 shows the overall results of our evaluation. The first two columns in Table 6.1 show, for each dataset and for each lab class, the number of IPAS (**#IPAs** column) and the number of correct students' submissions (**#Correct Submissions** column). All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 64GB.

**Mutating Programs.** Table 6.1 shows the number of programs that can be generated by MULTIPAS when applying each individual mutation described in Section 6.2.1. One can see that all the program mutations are able to augment at least 100% of both benchmarks. Furthermore, both mutations *M1 (CEM)*, comparison expression mirroring, and *M4 (VDR)*, variable declarations reordering, are able to augment both benchmarks with thousands of programs. These program mutations produce so many programs since the IPAS in both benchmarks use more than one variable, and in several IPAS, the students need to compare the values of different variables (comparison expressions). Hence, MULTIPAS computes all possible mirroring combinations of the comparison expressions and all possible re-orderings of the variable declarations that are valid. Lastly, if the user asks MULTIPAS to perform all six program mutations on both benchmarks, the number of mutated programs reaches several billions of programs.

**Mutilating Programs.** Regarding the program mutilations, the right-hand side of Table 6.1 shows the number of programs MULTIPAS can generate using each different mutilation described in Section 6.2.2, or all of them together. All the program mutilations are able to generate a dataset with several thousands of incorrect programs. Mutilation *B2 (VM)*, variable misuse, is the mutilation that is able to generate more incorrect programs since typically there are many possibilities when MULTIPAS is changing a variable occurrence for another variable.

**User Configuration.** The number of programs that can be generated by MULTIPAS can reach several million. Therefore, MULTIPAS has three flags available related to the total number of programs that can be generated. By default, MULTIPAS generates only 20% of those programs. The user can choose a different percentage of programs to be generated using the flag `-p | -percentage_total_progs`. Instead of generating all the programs, MULTIPAS chooses a sample of size `p`. The user can ask MULTIPAS, with flag `-info`, to print the total number of mutated/mutilated programs for a given configuration of program mutations/mutilations. MULTIPAS only outputs the number of programs without generating them. If the user desires all the programs and has the time and memory to generate all the possible mutated/mutilated programs, this can be done using the flag `-ea | -enumerate_all`.

## 6.4 Related Work

In the last few years, there has been a growing interest in data augmentation by program transformation. Yu et al. [201] proposed to apply several program transformations for big code data augmentation based on a pre-defined set of syntax-based rules to mutate programs written in Java. Liu and Zhong [202] proposed to extract Java code samples from Stack Overflow, and mine repair patterns from the extracted code samples. DEEPBUGS [197] also uses rule-based mutations to build, and not to augment, a dataset of programs from scratch to train its ML-based program repair tool. BUGLAB [194] is a Python program repair framework that learns how to detect and fix minor bugs. In order to train BUGLAB, Allamanis et al. [194] applied four program mutations and four program mutilations, different than MULTIPAS's program mutations and mutilations, in order to augment their benchmark of Python programs.

## 6.5 Conclusion

This chapter introduces MULTIPAS, an open-source framework for augmenting benchmarks of introductory programming assignments (IPAS). MULTIPAS can generate semantically equivalent programs by applying up to six different syntax mutations to a given program. Furthermore, MULTIPAS can also produce semantically incorrect programs using three semantic program mutilations. Moreover, MULTIPAS saves the variable mappings between the original program and the mutated/mutilated one and the information about the bugs introduced in each program. Experiments on two publicly available datasets of IPAS show that MULTIPAS can augment with millions of programs small-sized benchmarks of IPAS.

In Chapter 7, MULTIPAS is used to augment C-PACK-IPAS (see Chapter 4) and to generate training data for Graph Neural Networks (GNNS).

# 7

# Graph Neural Networks For Mapping Variables Between Programs

*"One of the essential skills in computer programming is to perceive when two processes are the same in this extended sense, for that leads to modularization- the breaking-up of a task into natural subtasks."*

– Douglas R. Hofstadter. Gödel, Escher, Bach: an Eternal Golden Braid [1]. 1979.

## Contents

Typically, in order to compare two programs, a relation between both programs' sets of variables is required. Thus, mapping variables between two programs is useful for a panoply of tasks such as program equivalence, program analysis, program repair, and clone detection. In this work, we propose using graph neural networks (GNNs) to map the set of variables between two programs based on both programs' abstract syntax trees (ASTs). To demonstrate the strength of variable mappings, we present three use-cases of these mappings on the task of *program repair* to fix well-studied and recurrent bugs among novice programmers in introductory programming assignments (IPAs). Experimental results on a dataset of 4166 pairs of incorrect/correct programs show that our approach correctly maps 83% of the evaluation dataset. Moreover, our experiments show that the current state-of-the-art on program repair, greatly dependent on the programs' structure, can only repair about 72% of the incorrect programs. In contrast, our approach, which is solely based on variable mappings, can repair around 88.5%.

This chapter has been published as a conference paper at the 26th European Conference on Artificial Intelligence, ECAI 2023 [173].

## 7.1 Introduction

The problem of program equivalence, i.e., deciding if two programs are equivalent, is undecidable [36, 37]. On that account, the problem of repairing an incorrect program based on a correct implementation is very challenging. In order to compare both programs, i.e., the correct and the faulty implementation, program repair tools first need to find a relation between both programs' sets of variables. Besides *program repair* [9], the task of mapping variables between programs is also important for *program analysis* [203], *program equivalence* [132], *program clustering* [42, 204], *program synthesis* [74], *clone detection* [137], and *plagiarism detection* [183].

Due to a large number of student enrollments every year in programming courses, providing feedback to novice students in *introductory programming assignments* (IPAs) requires substantial time and effort by the faculty [3]. Hence, there is an increasing need for systems capable of providing automated, comprehensive, and personalized feedback to students in programming assignments [2, 9, 13, 15]. *Semantic program repair* has become crucial to provide feedback to each novice programmer by checking their IPAs submissions using a pre-defined test suite. Semantic program repair frameworks use a correct implementation, provided by the lecturer or submitted by a previously enrolled student, to repair a new incorrect student's submission. However, the current state-of-the-art tools on semantic program repair [2, 9] for IPAs have two main drawbacks: (1) require a perfect match between the control flow graphs (loops, functions) of both programs, the correct and the incorrect one; and (2) require a bijective relation between both programs' sets of variables. Hence, if one of these requirements is not satisfied, then, these tools cannot fix the incorrect program with the correct one.

For example, consider the two programs presented in Figure 7.1. These programs are students' submissions for the IPA of printing all the natural numbers from $1$ to a given number $n$. The program in Listing 7.1 is a semantically correct implementation that uses a for-loop to iterate all the natural numbers until $n$. The program in Listing 7.2 uses a while-loop and an auxiliary function. This program is

**Listing 7.1:** A semantically correct student's implementation.

```
int main(){
    int n, i;
    scanf("%d", &n);
    for(i = 1; i <= n; i++){
        printf("%d\n", i);
    }
    return 0;
}
```

**Listing 7.2:** A semantically incorrect student's implementation since the variable `j` in the `main` function is not initialized.

```
void loop(int j, int l){
  while (l >= j){
    printf("%d\n", j);
    ++j;
  }
}
int main(){
  int j, l;
  scanf("%d", &l);
  loop(j, l);
  return 0;
}
```

Figure 7.1: Two implementations for the IPA of printing all the natural numbers from 1 to a given number $n$. The program in Listing 7.2 is semantically incorrect since the variable `j`, which is the variable being used to iterate over all the natural numbers until the number `l`, is not being initialized, i.e., the program has a bug of *missing expression*. The mapping between these programs' sets of variables is {`n` : `l`; `i` : `j`}.

semantically incorrect since the student forgot to initialize the variable $j$, a frequent bug among novice programmers called *missing expression/assignment* [176]. However, in this case, state-of-the-art program repair tools [2, 9] cannot fix the buggy program, since the control flow graphs do not match either due to using different loops (for-loop vs. while-loop) or due to the use of an auxiliary function. Thus, these program repair tools cannot leverage on the correct implementation in Listing 7.1 to repair the faulty program in Listing 7.2.

To overcome these limitations, in this chapter, we propose a novel graph program representation based on the structural information of the *abstract syntax trees (*ASTs*)* of imperative programs to learn how to map the set of variables between two programs using *graph neural networks (*GNNs*)*. Additionally, we present use-cases of program repair where these variable mappings can be applied to repair common bugs in incorrect students' programs that previous tools are not always capable of handling. For example, consider again the two programs presented in Figure 7.1. Note that having a mapping between both programs' variables (e.g., {`n` : `l`; `i` : `j`}) lets us reason about, on the level of expressions, which program fixes one can perform on the faulty program in Listing 7.2. In this case, when comparing variable `i` with variable `j` one would find the *missing assignment* i.e., `j = 1`.

Another useful application for mapping variables between different programs is fault localization. There is a body of research on fault localization [205–208], that requires the usage of assertions in order to verify programs. Variable mappings can be helpful in sharing these assertions among different programs. Additionally, several program repair techniques (e.g., SEARCHREPAIR [6], CLARA [2]) enumerate all possible mappings between two programs' variables during the search for possible fixes, using a correct program [2] or code snippets from a database [6]. Thus, variable mappings can drastically reduce the search space, by pruning all the other solutions that use a different mapping.

In programming courses, unlike in production code, typically, there is a reference implementation for each programming exercise. This comes with the challenge of comparing different names and structures between the reference implementation and a student's program. To deal with this challenging task,

we propose to map variables between programs using GNNs. Therefore, we explore three tasks to illustrate the advantages of using variable mappings to repair some frequent bugs without considering the incorrect/correct programs' control flow graphs. Hence, we propose to use our variable mappings to fix bugs of: *wrong comparison operator*, *variable misuse*, and *missing expression*. These bugs are recurrent among novice programmers [176] and have been studied by prior work in the field of automated program repair [193–195, 197].

Experiments on 4166 pairs of incorrect/correct programs show that our GNN model correctly maps 83% of the evaluation dataset. Furthermore, we also show that previous approaches can only repair about 72% of the dataset, mainly due to control flow mismatches. On the other hand, our approach, solely based on variable mappings, can fix 88.5%.

The main contributions of this work are:

- A novel graph program representation that is agnostic to the names of the variables and for each variable in the program contains a representative variable node that is connected to all the variable's occurrences;
- We propose to use GNNs for mapping variables between programs based on our program representation, ignoring the variables' identifiers;
- Our GNN model and the dataset used for this work's training and evaluation, are open-source and publicly available on GitHub: https://github.com/pmorvalho/ecai23-GNNs-for-mapping-variables-between-programs.

The structure of the remainder of this chapter is as follows. First, Section 7.2 presents our graph program representations. Next, Section 7.3 describes the GNNs used in this work. Section 7.4 introduces typical program repair tasks, as well as our program repair approach using variable mappings. Section 7.5 presents the experimental evaluation where we show the effectiveness of using GNNs to produce correct variable mappings between programs. Additionally, we compare our program repair approach based on the variable mappings generated by the GNN with state-of-the-art program repair tools. Finally, Section 7.6 describes related work, and the chapter concludes in Section 7.7.

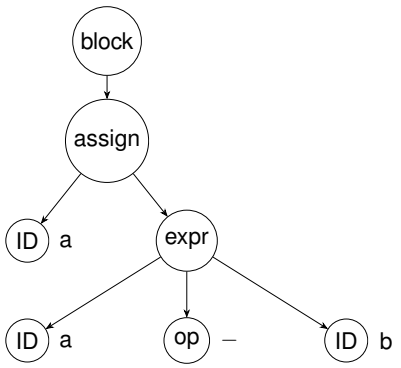
## 7.2 Program Representations

We represent programs as directed graphs so the information can propagate in both directions in the GNN. These graphs are based on the programs' *abstract syntax trees* (ASTs). An AST is described by a set of nodes that correspond to non-terminal symbols in the programming language's grammar and a set of tokens that correspond to terminal symbols [46]. An AST depicts a program's grammatical structure [60]. Figure 7.2a shows the AST for the small code snippet presented in Listing 7.3.

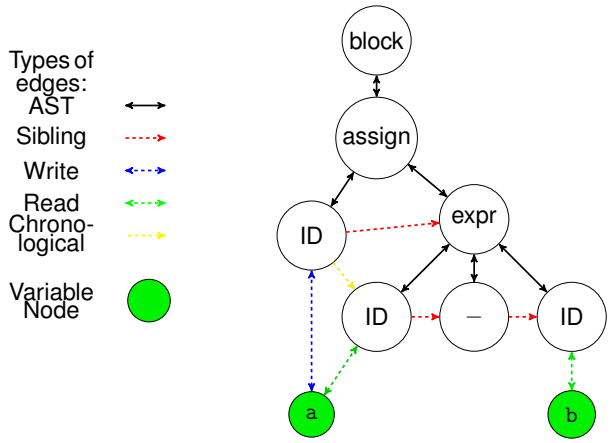
Regarding our graph program representation, firstly, we create a unique node in the AST for each distinct variable in the program and connect all the variable occurrences in the program to the same unique node. Figure 7.2b shows our graph representation for the small code snippet presented in Listing 7.3. Observe that our representation uses a single node for each variable in the program, the green

**Listing 7.3:** Small example of a C code block with an expression.

```
1 { // a and b are ints
2     a = a - b;
3 }
```




(a) Part of the AST representation of Listing 7.3.




(b) Our program representation for the snippet presented in Listing 7.3.

Figure 7.2: AST and our graph representation for the small code snippet presented in Listing 7.3.

nodes `a` and `b`. Moreover, we consider five types of edges in our representation: child, sibling, read, write, and chronological edges. *Child edges* correspond to the typical edges in the AST representation that connect each parent node to its children. Child edges are bidirectional in our representation. In Figure 7.2b, the black edges correspond to child edges. *Sibling edges* connect each child to its sibling successor. These edges denote the order of the arguments for a given node and have been used in other program representations [193]. Sibling edges allow the program representation to differentiate between different arguments when the order of the arguments is important (e.g. binary operation such as $\leq$). For example, consider the node that corresponds to the operation $\sigma(A_1, A_2, \ldots, A_m)$. The parent node $\sigma$ is connected to each one of its children by a child edge e.g. $\sigma \leftrightarrow A_1, \sigma \leftrightarrow A_2, \ldots, \sigma \leftrightarrow A_m$. Additionally, each child its connected to its successor by a sibling edge e.g. $A_1 \rightarrow A_2, A_2 \rightarrow A_3, \ldots, A_{m-1} \rightarrow A_m$. In Figure 7.2b, the red dashed edges correspond to sibling edges.

Regarding the *write and read edges*, these edges connect the `ID` nodes with the unique nodes corresponding to some variable. Write edges are connections between an `ID` node and its variable node. This edge indicates that the variable is being written. Read edges are also connections between an `ID` node and its variable node, although these edges indicate that the variable is being read. In Figure 7.2b, the blue dashed edge corresponds to a write edge while the green dashed edges correspond to read edges. Lastly, *chronological edges* establish an order between all the `ID` nodes connected to some variable. These edges denote the order of the `ID` nodes for a given variable node. For example, in Figure 7.2b, the yellow dashed edge corresponds to a chronological edge between the `ID` nodes of the variable `a`. Besides the siblings and the chronological edges, all the other edges are bidirectional in our representation.

*The novelty of our graph representation* is that we create a unique variable node for each variable in the program and connect each variable's occurrence to its unique node. This lets us map two variables in two programs, even if their number of occurrences is different in each program. Furthermore, the

variable's identifier is suppressed after we connect all the variable's occurrences to its unique node. This way, all the variables' identifiers are anonymized. Prior work on representing programs as graphs [193–195] use different nodes for each variable occurrence and take into consideration the variable identifier in the program representation. Furthermore, to the best of our knowledge, combining all five types of edges (sibling, write, read, chronological, and AST) is also novel. Section 7.5 presents an ablation study on the set of edges to analyze the impact of each type of edge.

## 7.3 Graph Neural Networks (GNNs)

Graph Neural Networks (GNNs) are a subclass of neural networks designed to operate on graph-structured data [209], which may be citation networks [210], mathematical logic [211] or representations of computer code [193]. Here, we use graph representations of a pair of ASTs, representing two programs for which we want to match variables, as the input. The main operative mechanism is to perform *message passing* between the nodes, so that information about the global problem can be passed between the local constituents. The content of these messages and the final representation of the nodes is parameterized by neural network operations (matrix multiplications composed with a non-linear function). For the variable matching task, we do the following to train the parameters of the network. After several message passing rounds through the edges defined by the program representations above, we obtain numerical vectors corresponding to each variable node in the two programs. We compute scalar products between each possible combination of variable nodes in the two programs, followed by a softmax function. Since the program samples are obtained by program mutation, the correct mapping of variables is known. Hence, we can compute a cross-entropy loss and minimize it so that the network output corresponds to the labeled variable matching. Note that the network has no information on the name of any object, which means that the task must be solved purely based on the structure of the graph representation. Therefore, our method is invariant to the consistent renaming of variables.

**Architecture Details.** The specific GNN architecture used in this work is the relational graph convolutional neural network (RGCN), which can handle multiple edges or relation types within one graph [212]. The numerical representation of nodes in the graph is updated in the message passing step according to the following equation:

$$\mathbf{x}_i' = \mathbf{\Theta}_{\text{root}} \cdot \mathbf{x}_i + \sum_{r \in \mathcal{R}} \sum_{j \in \mathcal{N}_r(i)} \frac{1}{|\mathcal{N}_r(i)|} \mathbf{\Theta}_r \cdot \mathbf{x}_j,$$

where $\mathbf{\Theta}$ are the trainable parameters, $\mathcal{R}$ stands for the different edge types that occur in the graph, and $\mathcal{N}_r$ the neighbouring nodes of the current node $i$ that are connected with the edge type $r$ [213]. After each step, we apply Layer Normalization [214] followed by a Rectified Linear Unit (ReLU) non-linear function.

We use two separate sets of parameters for the message passing phase for the program with the bug and the correct program. Five message passing steps are used in this work. After the message

passing phase, we obtain numerical vectors representing every node in both graphs. We then calculate dot products $\vec{a} \cdot \vec{b}$ between the vectors representing variable nodes in the buggy program graph $a \in A$ and the variable nodes from the correct graph $b \in B$, where $A$ and $B$ are the sets of variable node vectors. A score matrix $\mathcal{S}$ with dimensions $|A| \times |B|$ is obtained, to which we apply the softmax function on each row to obtain the matrix $\mathcal{P}$. The values in each row of $\mathcal{P}$ can now be interpreted as representing the probability that variable $a_i$ maps to each of the variables $b_i$.

## 7.4 Use-Cases: Program Repair

In this section, we propose a few use-cases on how to use variable mappings for program repair. More specifically, to repair bugs of: *wrong comparison operator*, *variable misuse*, and *missing expression*. These bugs are common among novice programmers [176] and have been studied by prior work in the field of automated program repair [193–195, 197]. The current state-of-the-art on semantic program repair tools focused on repairing IPAs, such as CLARA [2] and VERIFIX [9], are only able to fix these bugs if the correct expression in the correct program is located in a similar program structure as the incorrect expression in the incorrect implementation. For example, consider again the two programs presented in Figure 7.1. If the loop condition was incorrect in the faulty program, CLARA and VERIFIX could not fix it, since the control flow graphs do not match. Thus, these tools would fail due to *structural mismatch*.

The following sections present three program repair tasks that take advantage of variable mappings to repair an incorrect program using a correct implementation for the same IPA without considering the programs' structures. Our main goal is to show the usefulness of variable mappings. We claim that variable mappings are informative enough to repair these three realistic types of bugs. Given a buggy program, we search for and try to repair all three types of bugs. Whenever we find a possible fix, we check if the program is correct using the IPA's test suite.

**Bug #1: Wrong Comparison Operator (WCO).** Our first use-case are faulty programs with the bug of wrong comparison operator (WCO). This is a recurrent bug in students' submissions to IPAs since novice programmers frequently use the wrong operator, e.g., `i <= n` instead of `i < n`.

We propose tackling this problem solely based on the variable mapping between the faulty and correct programs, ignoring the programs' structure. First, we rename all the variables in the incorrect program based on the variable mapping by changing all the variables' identifiers in the incorrect program with the corresponding variables' identifiers in the correct implementation. Second, we count the number of times each comparison operation appears with a specific pair of variables/expressions in each program. Then, for each comparison operation in the correct program, we compute the mirrored expression, i.e., swapping the operator by its mirrored operator, and swapping the left-side and right-side of the operation. This way, if the incorrect program has the same correct mirrored expression, we can match it with an expression in the correct program. For example, in the programs shown in Figure 7.1, both loop conditions would match even if they are mirrored expressions, i.e., `i <= n` and `n >= i`.

Table 7.1: Validation mappings fully correct after 20 training epochs.

| | Buggy Programs | | | |
|---|---|---|---|---|
| | WCO Bug | VM Bug | ME Bug | All Bugs |
| Accuracy | 93.7% | 95.8% | 93.4% | 96.49% |

Afterwards, we iterate over all the pairs of variables/expressions that appear in comparison operations of the correct program (plus the mirrored expressions) and compare if the same pair of variables/-expressions appear the same number of times in the incorrect program, using the same comparison operator. If this is not the case, we try to fix the program using the correct implementation's operator in each operation of the incorrect program with the same pair of variables/expressions. Once the program is fixed, we rename all the variables based on the reverse variable mapping.

**Bug #2: Variable Misuse (VM).** Our second program repair task are buggy programs with variables being misused, i.e., the student uses the wrong variable in some program location. The wrong variable is of the same type as the correct variable that should be used. Hence, this bug does not produce any compilation errors. This type of bug is common among students and experienced programmers [199, 200]. The task of detecting this specific bug has received much attention from the Machine Learning (ML) research community [192, 193, 195].

Once again, we propose to tackle this problem based on the variable mapping between the faulty program and the correct one, ignoring the programs' structure. We start by renaming all the variables in the incorrect program based on the variable mapping. Then we count the number of times each variable appears in both programs. If a variable, `x`, appears more times in the incorrect program than in the correct implementation, and if another variable `y` appears more times in the correct program, then we try to replace each occurrence of `x` in the incorrect program with `y`. Once the program is fixed, we rename all the variables based on the reverse variable mapping.

**Bug #3: Missing Expression (ME).** The last use-case we will focus on is to repair the bug of *missing expressions/assignments*. This bug is also recurrent in students' implementations of IPAs [176]. Frequently, students forget to initialize some variable or to increment a variable of some loop, resulting in a bug of missing expression. However, unlike the previously mentioned bugs, this one has not received much attention from the ML community since it is more complex to repair this program fault. To search for a possible fix, we start by renaming all the variables in the incorrect program based on the variable mapping. Next, we count the number of times each expression appears in both programs. Expressions that appear more frequently in the correct implementation are considered possible repairs. Then, we try to inject these expressions, one at a time, into the incorrect implementation's code blocks and check the program's correctness. Once the program is fixed, we rename all variables based on the reverse variable mapping. This task is solely based on the variable mapping between the faulty and the correct programs.

Table 7.2: The number of correct variable mappings generated by our GNN on the evaluation dataset and the average overlap coefficients between the real mappings and our GNN's variable mappings.

| | Buggy Programs | | | |
|---|---|---|---|---|
| **Evaluation Metric** | WCO Bug | VM Bug | ME Bug | All Bugs |
| # Correct Mappings | 87.38% | 81.87% | 79.95% | 82.77% |
| Avg Overlap Coefficient | 96.99% | 94.28% | 94.51% | 95.05% |
| # Programs | 1078 | 1936 | 1152 | 4166 |

## 7.5 Experiments

**Experimental Setup.** We trained the Graph Neural Networks on an Intel(R) Xeon(R) Gold 6140 CPU @ 2.30GHz server with 72 CPUs and 692GB RAM. Networks were trained using NVIDIA GEFORCE GTX 1080 graphics cards with 12GB of memory. All the experiments related to our program repair tasks were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 64GB and a timeout of 60 seconds.

### 7.5.1 IPAs Dataset

To evaluate our work, we used C-PACK-IPAS [186], a benchmark of student programs developed during an introductory programming course in the C programming language for ten different IPAS (lab02), described in Chapter 4.

First, we selected a set of correct submissions, i.e., programs that compiled without any error and satisfied a set of input-output test cases for each IPA. We gathered 238 correct students' submissions from the first year and 78 submissions from the second year. We used the students' submissions from the first year for training and for validating our GNN and the submissions from the second year for evaluating our work.

Since we need to know the real variable mappings between programs (ground truth) to evaluate our representation, we generated a dataset of pairs of correct/incorrect programs to train and evaluate our work with specific bugs. This is a common procedure to evaluate machine learning models in the field of program repair [172, 192–195]. To generate this dataset, we used MULTIPAS [174] (see Chapter 6), a program modifier capable of mutating C programs syntactically, generating semantically equivalent programs, i.e., changing the program's structure but keeping its semantics. There are several program mutations available in MULTIPAS: mirroring comparison expressions, swapping the if's then-block with the else-block and negating the test condition, increment/decrement operators mirroring, variable declarations reordering, translating for-loops into equivalent while-loops, and all possible combinations of these program mutations. Hence, MULTIPAS has thirty-one different configurations for mutating a program. All these program mutations generate semantically equivalent programs. Afterwards, we also used MULTIPAS, to introduce bugs into the programs, such as *wrong comparison operator* (WCO), *variable misuse* (VM), *missing expression* (ME). Hence, we gathered a dataset of pairs of programs and the mappings between their sets of variables (see Appendix C). Each pair corresponds to a real correct student's implementation, and the second program is the student's program after being mutated

Table 7.3: Percentage of variable mappings fully correct on the validation set for different sets of edges used. Each type of edge is represented by an index using the mapping: {0: AST; 1: sibling; 2: write; 3: read; 4: chronological}.

| **Edges Used** | All | (1,2,3,4) | (0,2,3,4) | (0,1,3,4) | (0,1,2,4) | (0,1,2,3) | (0,1) |
|---|---|---|---|---|---|---|---|
| **Accuracy** | **96.49%** | 52.53% | 73.76% | 95.45% | 94.87% | 96.06% | 94.74% |

and with some bug introduced. Thus, this IPA dataset is generated, although based on real programs. The dataset is divided into three different sets: training set, validation set, and evaluation set. The programs generated from *first year* submissions are divided into a training and validation set based on which students' submissions they derive from. 80% of the students supply the training data, while 20% supply validation data. The evaluation set, which is not used during the machine learning optimization, is chronologically separate: it consists only of *second year* submissions, to simulate the real-world scenario of new, incoming students. The training set is composed of 3372, 5170, and 2908 pairs of programs from the first academic year for the WCO, VM, and ME bugs, respectively. The validation set, which was used during development to check the generalization of the prediction to unseen data, comprises 1457, 1457, and 1023 pairs of programs from the first year. Note that we subsample from the full spectrum of possible mutations, to keep the training data size small enough to train the network with reasonable time constraints. From each of the 31 combinations of mutations, we use one randomly created sample for each student per exercise. We found that this already introduced enough variation in the training dataset to generalize to unseen data. Finally, the evaluation set is composed of 4166 pairs of programs from the second year (see $3^{rd}$ row, Table 7.2). This dataset will be publicly available for reproducibility reasons.

### 7.5.2 Training

At training time, since the incorrect program is generated, the mapping between the variables of both programs is known. The network is trained by minimizing the cross entropy loss between the labels (which are categorical integer values indicating the correct mapping) and the values in each corresponding row of the matrix $\mathcal{P}$. As an optimizer, we used the Adam algorithm with its default settings in PyTorch [215]. The batch size was 1. As there are many different programs generated by the mutation procedures, we took one sample from each mutation for each student. Each network was trained for 20 full passes (epochs) over this dataset while shuffling the order of the training data before each pass. For validation purposes, data corresponding to 20$\%$ of the students from the first year of the dataset was kept separate and not trained on.

Table 7.1 shows the percentage of validation data mappings that were exactly correct (accuracy) after 20 epochs of training, using four different GNN models. Each GNN model was trained on programs with the bugs of wrong comparison operator (WCO), variable misuse (VM), missing expression (ME) or all of them (All). Furthermore, each GNN model has its own validation set with programs with a specific type of bug. The GNN model trained on All Bugs was validated using a mix of problems from each bug type. In the following sections, we focus only on this last GNN model (All Bugs).

Table 7.4: The number of programs repaired by each different repair technique: VERIFIX, CLARA, and our repair approach based on our GNN's variable mappings. The first row shows the results of repairing the programs using variable mappings generated based on uniform distributions (baseline).

| | Buggy Programs | | | | Not Succeeded | |
|---|---|---|---|---|---|---|
| **Repair Method** | WCO Bug | VM Bug | ME Bug | All Bugs | **% Failed** | **% Timeouts (60s)** |
| **Baseline** | 618 (57.33%) | 1187 (61.31%) | 287 (24.91%) | 2092 (50.22%) | 0 (0.0%) | **2074 (49.78%)** |
| **VERIFIX** | 555 (51.48%) | 1292 (66.74%) | 741 (64.32%) | 2588 (62.12%) | **1471 (35.31%)** | 107 (2.57%) |
| **CLARA** | 722 (66.98%) | 1517 (78.36%) | 764 (66.32%) | 3003 (72.08%) | 1153 (27.68%) | 10 (0.24%) |
| **GNN** | **992 (92.02%)** | **1714 (88.53%)** | **981 (85.16%)** | **3687 (88.5%)** | 0 (0.0%) | 479 (11.5%) |

### 7.5.3 Evaluation

Our GNN model was trained on programs with bugs of wrong comparison operator (WCO), variable misuse (VM), and missing expression (ME). We used two evaluation metrics to evaluate the variable mappings produced by the GNN. First, we counted the number of totally correct mappings our GNN was able to generate. We consider a mapping totally correct if it correctly maps all the variables between two programs. Secondly, we computed the overlap coefficient between the original variable mappings and the mappings generated by our GNN. The overlap coefficient is a similarity metric given by the intersection between the two mappings divided by the length of the variable mapping (see Appendix C).

The first row in Table 7.2 shows the number of totally correct variable mappings computed by our GNN model. One can see that the GNN maps correctly around 83% of the evaluation dataset. We have also looked into the number of variables in the mappings we were not getting entirely correct. The results showed that programs with more variables (e.g., six or seven variables) are the most difficult for our GNN to map their variables correctly (see Appendix C). For this reason, we have also computed the overlap coefficient between the GNN's variables mappings and the original mappings (ground truth). The second row in Table 7.2 shows the average of the overlap coefficients between the original variable mappings and the mappings generated by our GNN model. The overlap coefficient [216] measures the intersection (overlap) between two mappings. If the coefficient is $100\%$, both sets are equal. One set cannot be a subset of the other since both sets have the same number of variables in our case. The opposite is $0\%$ overlap, meaning there is no intersection between the two mappings. The GNN achieved at least 94% of overlap coefficients, i.e., even if the mappings are not always fully correct, almost 94% of the variables are correctly mapped by the GNN.

**Ablation Study.** To study the effect of each type of edge in our program representation, we have performed an ablation study on the set of edges. Prior works have done similar ablation studies [193]. Table 7.3 presents the accuracy of our GNN (i.e., number of correct mappings) on the evaluation dataset after 20 epochs. We can see that the accuracy of our GNN drops from 96% to 53% if we remove the AST edges (index 0), which was expected since these edges provide syntactic information about the program. Removing the sibling edges (index 1) also causes a great impact on the GNN's performance, dropping to 74%. The other edges are also important, and if we remove them, there is a negative impact on the GNN's performance. Lastly, since the AST and sibling edges caused the greatest impact, we evaluated using only these edges on our GNN and got an accuracy of 94.7%. However, the model using all the proposed edges has the highest accuracy of 96.49%.

### 7.5.4 Program Repair

This section presents the results of using variable mappings on the three use-cases described in Section 7.4, i.e., the tasks of repairing bugs of: *wrong comparison operator* (WCO), *variable misuse* (VM) and *missing expression* (ME). For this evaluation, we have also used the two current publicly available program repair tools for fixing introductory programming assignments (IPAs): CLARA [2] and VERIFIX [9]. Furthermore, we have tried to fix each pair of incorrect/correct programs in the evaluation dataset by passing each one of these pairs of programs to every repair method: VERIFIX, CLARA, and our repair approach based on the GNN's variable mappings.

If our repair procedure cannot fix the incorrect program using the most likely variable mapping according to the GNN model, then it generates the next most likely mapping based on the variables' distributions computed by the GNN. Therefore, the repair method iterates over all variable mappings based on the GNN's predictions. Lastly, we have also run the repair approach using as baseline variable mappings generated based on uniform distributions. This case simulates most repair techniques that compute all possible mappings between both programs' variables (e.g., SEARCHREPAIR [6]).

Table 7.4 presents the number of programs repaired by each different repair method. The first row presents the results for the baseline, which was only able to fix around 50% of the evaluation dataset. In the second row, the interested reader can see that VERIFIX can only repair about 62% of all programs. CLARA, presented in the third row, outperforms VERIFIX, being able to repair around 72% of the whole dataset. The last row presents the GNN model. This model is the best one repairing 88.5%.

The number of executions that resulted in a timeout (60 seconds) is relatively small for VERIFIX and CLARA. Regarding our repair procedure, it either fixes the incorrect program or iterates over all variable mappings until it finds one that fixes the program. Thus, the baseline and the GNN present no failed executions and considerably high rates of executions that end up in timeouts, almost 50% for the baseline and 11.5% in the case of the GNN model. Additionally, Table 7.4 also presents the failure rate of each technique, i.e., all the computations that ended within 60 seconds and did not succeed in fixing the given incorrect program. VERIFIX has the highest failure rate, around 35% of the entire evaluation set. CLARA also presents a significant failure rate, about 28%. As explained previously, this is the main drawback of these tools. Hence, these results support our claim that it is possible to repair these three realistic bugs solely based on the variable mappings' information without matching the structure of the incorrect/correct programs.

Furthermore, considering all executions, the average number of variable mappings used within 60 seconds is 1.24 variable mappings for the GNN model and 5.6 variable mappings when considering the baseline. The minimum number of mappings generated by both approaches is 1, i.e., both techniques were able to fix at least one incorrect program using the first generated variable mapping. The maximum number of variable mappings generated was 32 (resp. 48) for the GNN (resp. baseline). The maximum number of variable mappings used is high because the repair procedure iterates over all the variable mappings until the program is fixed or the time runs out. Moreover, even if we would only consider using the first variable mapping generated by the GNN model to repair the incorrect programs, we would be able to fix 3377 programs in 60 seconds, corresponding to 81% of the evaluation dataset.

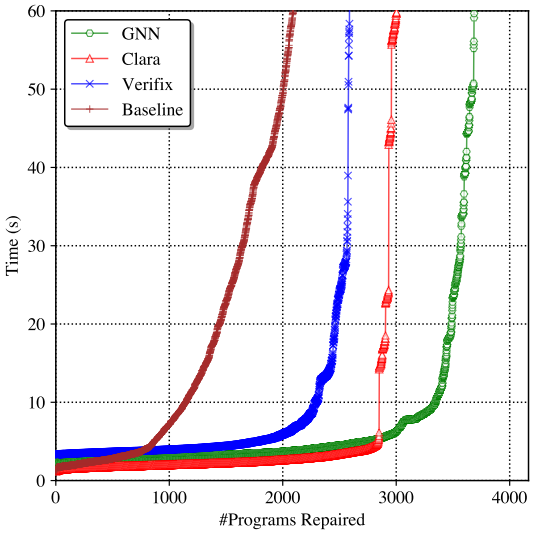



Figure 7.3: Cactus plot - The time spent by each method repairing each program of the evaluation dataset, using a timeout of 60 seconds.

Regarding the time performance of each technique, Figure 7.3 shows a cactus plot that presents the CPU time spent, in seconds, on repairing each program ($y$-axis) against the number of repaired programs ($x$-axis) using different repairing techniques. One can clearly see a gap between the different repair methods' time performances. For example, in 10 seconds, the baseline can only repair around 1150 programs, VERIFIX repairs around 2300, CLARA repairs around 2850 programs while using the GNN's variable mappings, we can repair around 3350 programs, i.e., around 17% more. We are considering the time the GNN takes to generate the variable mappings and the time spent on the repair procedure. However, the time spent by the GNN to generate one variable mapping is almost insignificant. The average time the GNN takes to produce a variable mapping is 0.025 seconds. The minimum (resp. maximum) time spent by the GNN, considering all the executions is 0.015s (resp. 0.183s).

## 7.6 Related Work

*Mapping variables* can also be helpful for the task of *code adaption*, where the repair framework tries to adapt all the variable names in a pasted snippet of code, copied from another program or a Stack Overflow post to the surrounding preexisting code [196]. ADAPTIVEPASTE [196] focused on a similar task to *variable misuse* (VM) repair, it uses a sequence-to-sequence with multi-decoder transformer training to learn programming language semantics to adapt variables in the pasted snippet of code. Recently, several systems were proposed to tackle the VM bug with ML models [193, 198, 203]. These tools classify the variable locations as faulty or correct and then replace the faulty ones through an enumerative prediction of each buggy location [193]. However, none of these methods takes program semantics into account, especially the long-range dependencies of variable usages [196].

## 7.7 Conclusions

This chapter tackles the highly challenging problem of mapping variables between programs. We propose the usage of graph neural networks (GNNs) to map the set of variables between two programs using our novel graph representation that is based on both programs' abstract syntax trees. In a dataset of 4166 pairs of incorrect/correct programs, experiments show that our GNN correctly maps 83% of the evaluation dataset. Furthermore, we leverage the variable mappings to perform automatic program repair. While the current state-of-the-art on program repair can only repair about 72% of the evaluation dataset due to structural mismatch errors, our approach, based on variable mappings, is able to fix 88.5%.

In Chapter 10, we integrate our variable mappings into a Large Language Model (LLM)-driven program repair tool to evaluate the impact of these mappings in assisting the repair process of LLMs.

# 8

# CFAULTS: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases

*"If debugging is the process of removing software bugs, then programming must be the process of putting them in."*

– Edsger Dijkstra.

## Contents

Debugging is one of the most time-consuming and expensive tasks in software development. Several formula-based fault localization (FBFL) methods have been proposed (see Section 3.2.2), but they fail to guarantee a set of diagnoses across all failing tests or may produce redundant diagnoses that are not subset-minimal, particularly for programs with multiple faults.

This chapter introduces a novel fault localization approach for C programs with multiple faults. CFAULTS leverages Model-Based Diagnosis (MBD) with multiple observations and aggregates all failing test cases into a unified MaxSAT formula. Consequently, our method guarantees consistency across observations and simplifies the fault localization procedure. Experimental results on two benchmark sets of C programs, TCAS and C-PACK-IPAS, show that CFAULTS is faster than other FBFL approaches like BUGASSIST and SNIPER. Moreover, CFAULTS only generates subset-minimal diagnoses of faulty statements, whereas the other approaches tend to enumerate redundant diagnoses.

This chapter has been published as a conference paper at the 26th International Symposium on Formal Methods, FM 2024 [16].

## 8.1 Introduction

Given a faulty program and a test suite with failing test cases, current *formula-based fault localization (FBFL)* methods encode the localization problem into several optimization problems to identify a minimal set of faulty statements (diagnoses) within a program. Typically, these methods find a minimal diagnosis considering each failing test case individually rather than simultaneously with all failing test cases. Moreover, these FBFL methods enumerate all *Minimal Correction Subsets (MCSes)* [53] to cover all diagnoses.

For instance, BUGASSIST [62, 100], a prominent FBFL tool, implements a ranking mechanism for bug locations. For each failing test, BUGASSIST enumerates all diagnoses of a Maximum Satisfiability (MaxSAT) formula corresponding to bug locations. Subsequently, BUGASSIST ranks diagnoses based on their frequency of appearance in each failing test. Other FBFL tools, like SNIPER [101], also enumerate all diagnoses for each failing test. However, the set of SNIPER's diagnoses is obtained by taking the Cartesian product of the diagnoses gathered using each failing test. As a result, while FBFL methods can determine minimal diagnoses per failing test, BUGASSIST cannot guarantee a minimal diagnosis considering all failing tests, and SNIPER may enumerate a significant number of redundant diagnoses that are not minimal [54]. These limitations may pose challenges for programs with multiple faulty statements, as shown in Example 13.

**Example 13** (Motivation)**.** Consider the program presented in Listing 8.1, which aims to determine the maximum among three given numbers. However, based on the test suite shown in Table 8.1, the program is faulty, as its output differs from the expected. The set of minimally faulty lines in this program is {5, 8, 11}, as all three `if`-conditions are incorrect according to the test suite. Fixing any subset of these lines would be insufficient to repair the program. One possible fix is to replace all these conditions with the suggested fixes in lines {6, 9, 12}.

**Listing 8.1:** Faulty program example. Faulty lines: {5,8,11}.

```
1  int main(){
2    // finds maximum of 3 numbers
3    int f,s,t;
4    scanf("%d%d%d",&f,&s,&t);
5    if (f < s && f >= t)
6      // fix: f >= s
7      printf("%d",f);
8    if (f > s && s <= t)
9      // fix: f < s and s >= t
10     printf("%d",s);
11   if (f > t && s > t)
12     // fix: f < t and s < t
13     printf("%d",t);
14
15   return 0;
16 }
```

| | Input | | | Output |
|---|---|---|---|---|
| $t_0$ | 1 | 2 | 3 | 3 |
| $t_1$ | 6 | 2 | 1 | 6 |
| $t_2$ | -1 | 3 | 1 | 3 |

Table 8.1: Test-suite.

| | BugAssist | SNIPER |
|---|---|---|
| **#Diagnoses** $t_0$ | 8 | 8 |
| **#Diagnoses** $t_1$ | 21 | 21 |
| **#Diagnoses** $t_2$ | 9 | 9 |
| **#Total Unique Diagnoses** | 32 | 1297 |
| **Final Diagnosis** | {4,13} | {5,8,11} |

Table 8.2: Number of diagnoses (faulty statements) generated by BugAssist [100] and SNIPER [101] per test.

In a typical FBFL approach, the minimal set of statements identified as faulty might include, for example, lines 4 and 5. Removing the `scanf` statement and an `if`-statement would allow an FBFL tool to assign any value to the input variables in order to always produce the expected output. However, considering an approach that prioritizes identifying faulty statements within the program's logic before evaluating issues in the input/output statements (such as `scanf` and `printf`), one might identify lines {5, 8, 11} as the faulty statements. When applying BugAssist's and SNIPER's approach on the program in Listing 8.1 with the described optimization criterion and utilizing the inputs/outputs detailed in Table 8.1 as specification, distinct sets of faults are identified for each failing test. Table 8.2 presents the diagnosis (set of faulty lines) produced by each tool, along with the number of diagnoses enumerated for each failing test case and the total number of unique diagnoses after aggregating the diagnoses from all tests, using each tool's respective method.

In the case of BugAssist, diagnoses are prioritized based on their occurrence frequency. Consequently, BugAssist yields 32 unique diagnoses and selects {4, 13} since this diagnosis is identified in every failing test. In contrast, SNIPER computes the Cartesian product of all diagnoses, resulting in 1297 unique diagnoses. Note that BugAssist's diagnoses may not adequately identify all faulty program statements. Conversely, SNIPER's diagnosis {5, 8, 11} is minimal, even though it enumerates an additional 1296 diagnoses. Hence, existing FBFL methods do not ensure a minimal diagnosis across all failing tests (e.g., BugAssist) or may produce an overwhelming number of redundant sets of diagnoses (e.g., SNIPER), especially for programs with multiple faults.

This chapter tackles this challenge by formulating the FL problem as a single optimization problem in Section 8.2. Our MaxSAT-based *Model-Based Diagnosis (MBD)* with multiple observations approach allows us to generate only minimal diagnoses to identify all faulty program components within a C program. Furthermore, we have implemented the MBD problem with multiple test cases in CFaults, a fault localization tool for ANSI-C programs, presented in Section 8.3. CFaults begins by unrolling and instrumentalizing C programs at the code-level, ensuring independence from the bounded model checker. Next, CFaults utilizes CBMC [57], a well-known bounded model checker for C, to generate a trace formula of the program. Finally, CFaults encodes the problem into MaxSAT to identify the minimal set

of diagnoses corresponding to the buggy statements.

Experimental results presented in Section 8.4 on two benchmarks of C programs, TCAS [178] (industrial), and C-PACK-IPAS [171] (programming exercises), show that CFAULTS effectively detects minimal sets of diagnoses. In contrast, SNIPER and BUGASSIST either generate an overwhelming number of redundant diagnoses or fail to produce a minimal set required to fix each program.

To summarize, the contributions of this work are: (1) we tackle the fault localization problem in C programs using a Model-Based Diagnosis (MBD) approach considering multiple failing test cases, and formulating it as a unified optimization problem; (2) we implement this MBD approach in a publicly available tool called CFAULTS [217] [1] that unrolls and instrumentalizes C programs at the code level, making it independent of the bounded model checker used; (3) CFAULTS allows refinement of localized faults to pinpoint the bug's location more precisely; (4) we evaluate CFAULTS on two sets of C programs (TCAS and C-PACK-IPAS), showing that CFAULTS is fast and only produces subset-minimal diagnoses, unlike other state-of-the-art formula-based fault localization tools.

## 8.2 Model-Based Diagnosis with Multiple Test Cases

This chapter encodes the fault localization problem as a Model-Based Diagnosis (MBD) with multiple observations using a single optimization problem. We simultaneously integrate all failing test cases (observations) in a single MaxSAT formula. This approach allows us to generate only minimal diagnoses capable of identifying all faulty components within the system, in our case, a C program. The interested reader is referred to Section 3.2.2 for a description of the MBD theory with a single observation.

Given $m$ observations, $\mathcal{O} = \{o_1, \ldots, o_m\}$, a distinct replica of the system, denoted as $\mathcal{P}_i$, is required for each observation $o_i$. The hard clauses, $\phi_h$, in our MaxSAT formulation correspond to each observation's encoding ($o_i$) and $m$ system replicas, one for each observation, $\mathcal{P}_i$. Hence, $\phi_h = \bigwedge_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i)$. Additionally, we aim to maximize the set of healthy components. Therefore, the soft clauses are formulated as: $\phi_s = \bigwedge_{c \in \mathcal{C}} h(c)$. Thus, given the MaxSAT solution of $(\phi_h, \phi_s)$, the set of unhealthy components ($h(c) = 0$), corresponds to a subset-minimal aggregated diagnosis, which also represents the smallest minimal diagnosis. This diagnosis is a subset-minimal of components that, when declared unhealthy (deactivated), make the system consistent with all observations, as follows:

$$\bigwedge\nolimits_{o_i \in \mathcal{O}} (\mathcal{P}_i \wedge o_i) \wedge \bigwedge\nolimits_{c \in \mathcal{C} \setminus \Delta} h(c) \wedge \bigwedge\nolimits_{c \in \Delta} \neg h(c) \nvDash \bot \tag{8.1}$$

We assume that the system remains unchanged given different observations, where the components are replicated for each observation, but the healthy variables are shared. This is necessary because we analyze all observations jointly, which can affect the component's behaviour. In our work, the observations consist of a test suite containing failing test cases.

The HSD [54] algorithm was proposed to localize single faults in circuits given multiple observations. The HSD algorithm is based on hitting set dualization (HSD). For each observation $o_i$, this algorithm

[1] `https://github.com/pmorvalho/CFaults`

computes minimal unsatisfiable subsets (MUSes) of the MaxSAT formula encoded by (3.5). Next, the HSD algorithm computes a minimum hitting set $\mathcal{H}$ on the MUSes, and checks if $\mathcal{H}$ makes the system consistent with each observation individually. Hence, to compute all subset-minimal aggregated diagnoses of a faulty system $\mathcal{P}$, the algorithm performs at most $m$ oracle calls for each minimum hitting set computed, where $m$ is the number of observations. Each oracle call uses a different system replica (3.5).

Our approach encodes the problem into a single MaxSAT formula, while HSD [54] divides the problem into $m$ MaxSAT formulas, one for each observation. Additionally, for each minimal hitting set computed in HSD, $m$ oracle calls are needed to check if a diagnosis is consistent with all observations. However, in our case, we just need to perform a single MaxSAT call that returns a cardinality minimal diagnosis, which is, by definition, consistent with all observations since all observations are encoded into the formula. Furthermore, the HSD algorithm was solely evaluated using single faults in circuits given multiple observations, and it was not implemented to work with programs. A potential drawback is that our MaxSAT formula grows with the number of observations. This could result in a large formula and affect the performance of the MaxSAT solver. However, this scenario was not observed in our experimental results (see Section 8.4).

## 8.3 CFAULTS: Model-Based Diagnosis with Multiple Observations for C

CFAULTS is a new model-based diagnosis (MBD) tool for fault localization in C programs with multiple test cases. Unlike previous works, CFAULTS uses the approach proposed in Section 8.2, and C programs are relaxed at the code level, enabling users to leverage other bounded model checkers effectively. Figure 8.1 provides an overview of CFAULTS consisting of six main steps: program unrolling, program instrumentalization, bounded model checking (CBMC), encoding to MaxSAT, an Oracle (MaxSAT solver), and a refinement step. Hence, CFAULTS formulates the MBD problem with multiple test cases as the 3-tuple $\langle \mathcal{P}, \mathcal{C}, \mathcal{O} \rangle$, where the observations $\mathcal{O}$ consist of failing test cases (inputs and assertions), the components $\mathcal{C}$ represent the set of program statements, and the system description $\mathcal{P}$ is a trace formula of the unrolled and instrumentalized program. The program is instrumented at the code level with relaxation variables corresponding to our *healthy variables*.

### 8.3.1 Program unrolling

CFAULTS starts the unrolling process by expanding the faulty program using the set of failed tests from the test suite. In this context, an unrolled program signifies the original program expanded $m$ times ($m$ program scopes), where $m$ denotes the number of failed test cases. An unrolled program encodes the execution of all failing tests within the program, along with their corresponding inputs and specifications (assertions).

The unrolling process encompasses three primary steps. Initially, CFAULTS generates fresh variables and functions for each of the $m$ program scopes, ensuring each scope possesses unique variables and

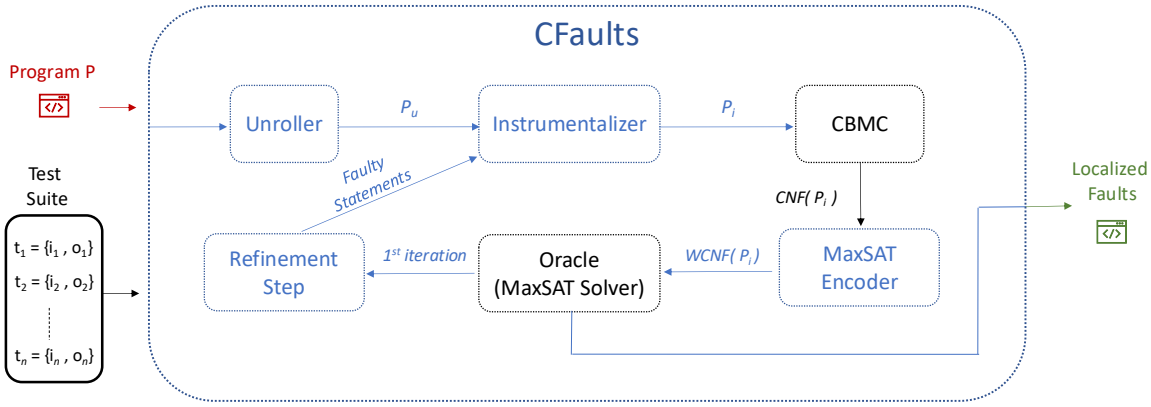



Figure 8.1: Overview of CFAULTS.

functions. Subsequently, CFAULTS establishes variables representing the inputs and outputs for each program scope corresponding to the failing tests. Input operations, such as `scanf`, undergo translation into read accesses to arrays corresponding to the inputs, while output operations, such as `printf`, are replaced by write operations into arrays representing the program's output. Every exit point of the program (e.g., a `return` statement in the `main` function) is replaced with a `goto` statement directing the program flow to the next failing test's scope. Lastly, at the end of the unrolled program, CFAULTS embeds an assertion capturing all the specifications of the failing tests. Consequently, the unrolled program encapsulates the execution of all failing tests within a single program.

Listing 8.2 exhibits a program segment generated through the unrolling process applied to Listing 8.1. CFAULTS establishes global variables to represent the inputs and outputs of each failing test (lines 1–3, Listing 8.2). For the sake of simplicity, the depicted listing illustrates solely the initial scope corresponding to test 0 from the test suite outlined in Table 8.1. Distinct variables are introduced for each failing test. Furthermore, the `scanf` function call is substituted with input array operations (lines 8–10), while the `printf` calls are replaced with CFAULTS' print functions, akin to `sprintf` functions, which direct output to a buffer. Lastly, the unrolled program concludes with an assertion representing the disjunction of the negation of all failing test assertions. For instance, suppose there are $m$ failing tests, where $A_i$ denotes the assertion of test $t_i$. In this scenario, CFAULTS injects the following assertion into the program: $\neg A_1 \vee \cdots \vee \neg A_m$.

### 8.3.2 Program Intrumentalization

After integrating all possible executions and assertions from failing tests during the unrolling step, CFAULTS proceeds to instrumentalize the unrolled C program by introducing relaxation variables for each program component (statement/instruction). Each relaxation variable activates (or deactivates) the program component being relaxed when assigned to true (or false) respectively. CFAULTS ensures that there are no conflicts between the names of the relaxation variables and the names of the program's original variables. For this step, CFAULTS needs to receive a maximum number of iterations that the program should be unwound.

The relaxation process introduces relaxation variables that deactivate or activate program compo-

**Listing 8.2:** The program from Listing 8.1 after being subjected to CFAULTS' unrolling process, using the test suite presented in Table 8.1. For simplicity, only the initial scope corresponding to test $t_0$ is displayed. The scopes `scope_1` and `scope_2` associated with failing tests $t_1$ and $t_2$ are omitted.

```
1  float _input_f0[3] = {1, 2, 3};
2  char _out_0[2] = "3";
3  int _ioff_f0 = 0, _ooff_0 = 0;
4  // ... inputs and outputs for the other tests
5  int main(){
6    scope_0:{
7      int f_0, s_0, t_0;
8      f_0 = _input_f0[_ioff_f0++];
9      s_0 = _input_f0[_ioff_f0++];
10     t_0 = _input_f0[_ioff_f0++];
11     if ((f_0 < s_0) && (f_0 >= t_0))
12         _ooff_0 = printInt(_out_0, _ooff_0, f_0);
13     if ((f_0 > s_0) && (s_0 <= t_0))
14         _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     if ((f_0 > t_0) && (s_0 > t_0))
16         _ooff_0 = printInt(_out_0, _ooff_0, t_0);
17     goto scope_1;
18   }
19   // ... scope_1 and scope_2
20   final_step:
21   assert(strcmp(_out_0, "3") != 0 || /* other assertions */ );
22 }
```

nents. This process involves four distinct relaxation rules for: (1) conditions of `if`-statements, (2) expression lists (e.g., an expression list executed at the beginning of a for-loop), (3) loop conditions, and (4) other program statements.

**Example 14.** Listings 8.3 shows a code snippet that sums all the numbers between `1` and `n`. Listings 8.4 depicts the same program statements after undergoing relaxation by CFAULTS. For the sake of simplicity, all relaxation variables' and offsets' names were simplified.

In more detail, the rule for relaxing a general program statement is to envelop the statement with an `if`-statement, whose condition is a relaxation variable. For example, consider lines 5 and 6 in the program on Listings 8.3. These lines are relaxed by CFAULTS using relaxation variables `_rv1` and `_rv2` respectively, appearing as lines 11 and 12 on Listings 8.4.

Furthermore, when relaxing `if`-statements, the statements inside the `then` and `else` blocks adhere to the previously explained relaxation rule. However, the conditions of `if`-statements are relaxed using a ternary operator, as shown in line 14 of Listings 8.4. Note that if the relaxation variable is assigned true, then the original `if` condition is executed. Otherwise, a different relaxation variable (e.g., `_ev4` in Listings 8.4) determines whether the program execution enters the `then`-block or the `else`-block (if one exists). These relaxation variables (`else`*'s relaxation variables*) are local to each failing test scope and enable different tests to determine whether to enter the `then` or `else`-block.

When handling expression lists, CFAULTS adopts a comparable strategy to that of generic program statements, enclosing each expression within a ternary operator instead of an `if`-statement. If the program component is deactivated, the expression is replaced by `1`. For example, the initialization of variable `i` in line 11 of Listings 8.3 is relaxed into the ternary operation in line 17 of Listings 8.4.

Lastly, all relaxation variables inside a loop are Boolean vectors to relax statements within a loop. Each entry of these vectors relaxes the loop's statements for a given iteration. The maximum number

**Listing 8.3:** Program statements.

```
1  int i;
2  int n;
3  int s;
4
5  s = 0;
6  n = _input_f0[_ioff_f0++];
7
8  if (n == 0)
9      return 0;
10
11 for (i=1; i < n; i++){
12     s = s + i;
13 }
```

**Listing 8.4:** Program statements relaxed.

```
1  //main scope
2  bool _rv1, _rv2, _rv3, _rv5;
3  bool _rv6[UNWIND],..., _rv8[UNWIND];
4  int _los; // loop1 offset
5
6  //test scope
7  bool _ev4;
8  int i,n,s;
9  _los=1;
10
11 if (_rv1) s = 0;
12 if (_rv2) n = _input_f0[_ioff_f0++];
13
14 if ( _rv3 ? (n == 0) : _ev4)
15     return 0;
16
17 for (_rv5 ? (i = 1) : 1;
18       !_rv6[_los] || (i<n);
19       _rv8[_los] ? i++ : 1, _los++){
20     if (_rv7[_los]) s = s + i;
21 }
```

of iterations of the loops is defined by the CFAULTS user. CFAULTS follows a similar approach for inner loops, creating arrays of arrays. Thus, for simple program statements within a loop, CFAULTS encapsulates them with `if`-statements, with the relaxation variables indexed to the iteration number. Line 20 of Listings 8.4 illustrates a relaxed statement inside a loop. The loop's condition is relaxed by implication of the relaxation variable, as demonstrated in line 18 of Listings 8.4. Furthermore, each loop has its own offsets to index relaxation variables. These offsets are initialized just before the loop and incremented at the end of each iteration (e.g., line 19 in Listing 8.4).

When handling auxiliary functions, CFAULTS declares the relaxation variables needed in the main scope of the program and passes these variables as parameters. Hence, CFAULTS ensures that the same variables are used throughout the auxiliary functions' calls.

Listing 8.5 depicts the program resulting from the instrumentalization process of Listing 8.2 performed by CFAULTS. The same program components (statements/instructions) across different failing test scopes are assigned the same relaxation variable declared in the main scope. Consequently, if a relaxation variable is set to 0, the corresponding program component is deactivated across all test executions. Additionally, the relaxation variables are left uninitialized, allowing CFAULTS to determine the minimal number of faulty components requiring deactivation. Note that relaxation variables are not declared as global variables but as local variables within the `main` scope. This is to prevent the C compiler from automatically initializing all these variables to 0.

### 8.3.3 CBMC

After unrolling and instrumentalizing the C program, CFAULTS invokes CBMC, a bounded model checker for C [57]. CBMC initially transforms the unrolled and relaxed program into *Static Single Assignment (SSA)* form, an intermediate representation ensuring that variables are assigned values only once and are defined before use [218]. SSA achieves this by converting existing variables into multiple versions, each uniquely representing an assignment. Next, CBMC translates the SSA rep-

**Listing 8.5:** Instrumentalized program.

```
1  //global vars
2  int main(){
3    bool _rv1, _rv2, ..., _rv12;
4    scope_0:{
5      bool _ev5, _ev8, _ev11;
6      int f_0, s_0, t_0;
7      if (_rv1) f_0 = _input_f0[_ioff_f0++];
8      if (_rv2) s_0 = _input_f0[_ioff_f0++];
9      if (_rv3) t_0 = _input_f0[_ioff_f0++];
10     if (_rv4 ? ((f_0 < s_0) && (f_0 >= t_0)) : _ev5 ){
11         if (_rv6) _ooff_0 = printInt(_out_0, _ooff_0, f_0);
12     }
13     if (_rv7 ? ((f_0 > s_0) && (s_0 <= t_0)) : _ev8 ){
14         if (_rv9) _ooff_0 = printInt(_out_0, _ooff_0, s_0);
15     }
16     if (_rv10? ((f_0 > t_0) && (s_0 > t_0)) : _ev11 ){
17         if (_rv12) _ooff_0 = printInt(_out_0, _ooff_0, t_0);
18     }
19     goto scope_1;
20   }
21   // scope_1 and scope_2
22   final_step:
23   assert(strcmp(_out_0, "3") != 0 || /* ... other assertions */ );
24 }
```

resentation into a CNF formula, which represents the trace formula of the program. During the CNF formula generation, CBMC negates the program's assertion ($\neg(\neg A_1 \vee \cdots \vee \neg A_m)$) to compute a counter-example. Moreover, the CNF formula, $\phi$, encodes each failing test's input ($I_i$), assertion ($A_i$), and all execution paths of the unrolled and relaxed incorrect program encoded by the trace formula ($P$), i.e., $\phi = (I_1 \wedge \ldots \wedge I_m) \wedge P \wedge (A_1 \wedge \cdots \wedge A_m)$. Thus, if $\phi$ is $SAT$, an assignment exists that activates or deactivates each relaxation variable and makes all failing test assertions true. Hence, each satisfiable assignment is a diagnosis of the C program, considering all failing tests.

### 8.3.4 MaxSAT Encoder

Let $\phi$ denote the CNF formula generated by CBMC in the previous step. Next, CFAULTS generates a weighted partial MaxSAT formula $(\mathcal{H}, \mathcal{S})$ to maximize the satisfaction of relaxation variables in the program, aiming to minimize the necessary code alterations. The set of hard clauses is defined by CBMC's CNF formula (i.e., $\mathcal{H} = \phi$), while the soft clauses consist of unit clauses representing relaxation variables used to instrument the C program, expressed as $\mathcal{S} = \bigwedge_{c \in \mathcal{C}} (rv_c)$. Additionally, we assign a hierarchical weight to each relaxation variable based on the height of its sub-AST (Abstract Syntax Tree). For instance, in the case of an `if`-statement without an `else`-block, the relaxation variable for its condition will be assigned a weight equal to the sum of the weights of the relaxation variables within the `then`-block. Furthermore, to prioritize the identification of faulty statements within the program's logic over evaluating issues in the input/output, these statements (such as `scanf` and `printf`) are assigned a significantly higher cost compared to other program statements. Moreover, due to the use of hierarchical weights in the relaxation variables, CFAULTS enumerates all MaxSAT solutions to identify all subset-minimal diagnoses with the smallest weight, since there can be more than one MaxSAT solution (with the same cost) that differ in the number of relaxed program statements.

### 8.3.5 Oracle

CFAULTS invokes a MaxSAT solver to determine the program's diagnoses with minimum weight. Note that all these diagnoses are also subset-minimal, aligning with the principles of Model-Based Diagnosis (MBD) theory. By consolidating all failing tests into a unified, unrolled, and instrumentalized program, the MaxSAT solution identifies the minimum subset of statements requiring removal to fulfil the assertions of all failing tests.

### 8.3.6 Refinement

The standard Model-Based Diagnosis (MBD) theory focuses on faulty components (program statements) whose removal can rectify the system (program's assertions). However, addressing program faults in software may necessitate introducing, relocating, or replacing statements. Hence, CFAULTS incorporates a refinement step that introduces nondeterminism into the program, enabling the Oracle to simulate actions such as introducing, reallocating or replacing existing program statements. During the first iteration of CFAULTS, the refinement step is invoked to introduce non-determinism, with the aim of minimizing the number of faulty statements. This step can improve fault localization by conducting a more detailed analysis of previously identified faulty statements. For example, in the scenario outlined in Example 13, refining line 5 into

```
if ((_rv1 ? (f < s) : nondet_bool()) && (_rv2 ? (f >= t) : nondet_bool()))
```

enables CFAULTS to determine that only the left part of the binary operation (`f < s`) is faulty, while the right part remains unaffected. This fine-grained approach allows for more precise detection of program faults. When the refinement step is triggered, CFAULTS instrumentalizes the program again, introducing nondeterminism exclusively to the statements previously identified as faulty during the initial Oracle call. Through this process, CFAULTS aims to reduce the set of faulty program components by executing them or assigning them to nondeterministic functions. All remaining program components are executed, meaning their relaxation variables are activated during this step.

## 8.4 Experimental Results

All of the experiments were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz running Linux Debian 10.2, using a memory limit of 32 GB and a timeout of 3600s, for each program. CFAULTS has been evaluated using two distinct benchmarks of C programs: TCAS [178] and C-PACK-IPAS [186]. TCAS stands out as a well-known program benchmark extensively utilized in the fault localization literature [62, 101]. This benchmark comprises a C program from Siemens and 41 versions with intentionally introduced faults, with known positions and types of these faults. Conversely, C-PACK-IPAS is a set of student programs collected during an introductory programming course. For this evaluation, we used the first lab class of C-PACK-IPAS, which consists of ten programming assignments, comprising 486 faulty programs and 799 correct implementations. C-PACK-IPAS has proven

| Benchmark: **TCAS** | | | |
|---|---|---|---|
| | **Valid Diagnosis** | **Memouts** | **Timeouts** |
| **BugAssist** | 41 (100.0%) | 0 (0.0%) | 0 (0.0%) |
| **SNIPER** | 7 (17.07%) | 34 (82.93%) | 0 (0.0%) |
| **CFaults** | 41 (100.0%) | 0 (0.0%) | 0 (0.0%) |
| **CFaults-Refined** | 41 (100.0%) | 0 (0.0%) | 0 (0.0%) |

| Benchmark: **C-Pack-IPAs** | | | |
|---|---|---|---|
| | **Valid Diagnosis** | **Memouts** | **Timeouts** |
| **BugAssist** | 454 (93.42%) | 0 (0.0%) | 32 (6.58%) |
| **SNIPER** | 446 (91.77%) | 4 (0.82%) | 36 (7.41%) |
| **CFaults** | 483 (99.38%) | 1 (0.21%) | 2 (0.41%) |
| **CFaults-Refined** | 482 (99.18%) | 1 (0.21%) | 3 (0.62%) |

Table 8.3: BUGASSIST, SNIPER and CFAULTS fault localization results.

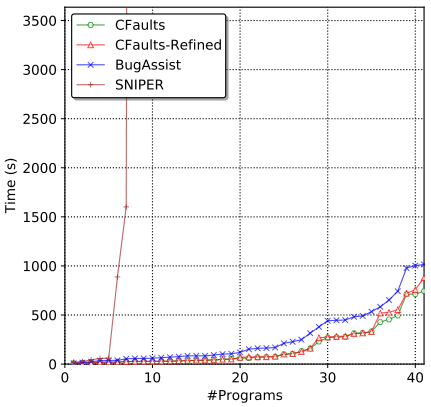
successful in evaluating various works across program analysis [173], program transformation [174], and clustering [42].

CFAULTS uses `pycparser` [219] for unrolling and instrumentalizing C programs. Additionally, CBMC version 5.11 is used to encode C programs into CNF formulas. Furthermore, since the source code of BUGASSIST and SNIPER is either unavailable or no longer maintained (resulting in compilation and linking issues), prototypes of their algorithms were implemented. It is worth noting that the original version of SNIPER could only analyze programs that utilized a subset of ANSI-C, lacked support for loops and recursion, and could only partially handle global variables, arrays, and pointers. In this work, both SNIPER and BUGASSIST handle ANSI-C programs, as their algorithms are built on top of CFAULTS's unroller and instrumentalizer modules. For the MaxSAT oracle, RC2Stratified [220] from the `PySAT` toolkit [221] (v. 0.1.7.dev19) was used.

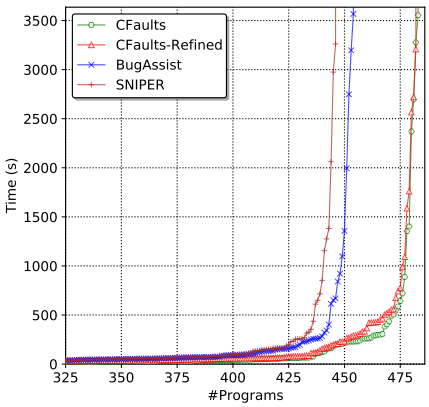
Furthermore, all three FBFL algorithms evaluated (CFAULTS, BUGASSIST, and SNIPER) consistently generate diagnoses that are consistent with (8.1), indicating that all proposed diagnoses undergo validation by CBMC once the algorithm provides a diagnosis. However, this validation primarily serves to verify diagnoses generated by BUGASSIST, as it has the capability to produce diagnoses that may not align with all failing test cases. In contrast, CFAULTS' MaxSAT solution, by definition, aligns with all observations, and SNIPER's aggregation method (Cartesian product) produces only valid diagnoses, although they may not always be subset-minimal. When considering BUGASSIST, we iterate through all computed diagnoses based on BUGASSIST's voting score, until we identify one diagnosis that is consistent with all observations, i.e., conforms to (8.1).

Table 8.3 provides an overview of the results obtained using SNIPER, BUGASSIST, and CFAULTS on the two benchmarks of C programs. The TCAS program comprises approximately 180 lines of code and has a maximum of 131 failing tests for each program. This leads SNIPER to reach the memory limit of 32GB for almost 83% of the programs when aggregating the sets of MCSes computed for each failing test. Additionally, a higher rate of timeouts is observed for SNIPER and BUGASSIST than for CFAULTS. Figures 8.2a and 8.2b depict cactus plots that present the CPU time spent on fault localization in each program (y-axis) versus the number of programs with all faults successfully localized (x-axis) using BUGASSIST, SNIPER, and CFAULTS (with and without refinement) on TCAS and C-PACK-IPAS, respectively. Notably, CFAULTS generally exhibits faster performance compared to BUGASSIST and SNIPER across both benchmarks. In Figure 8.2a, SNIPER's performance is due to its memout rate on TCAS.

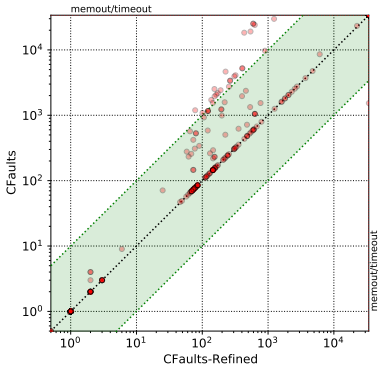
In TCAS, CFAULTS, whether invoking the refinement step or not, identifies faults in the entire dataset. However, in C-PACK-IPAS, CFAULTS localizes faults in one additional program when the refinement step is not called. Even if the refinement step reaches the time limit, CFAULTS still possesses a subset-



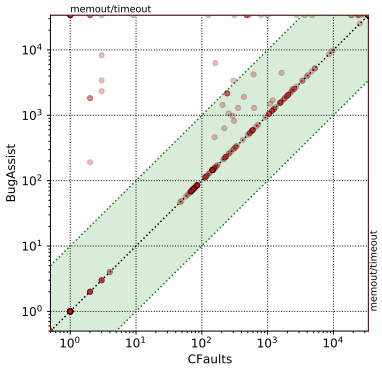

(a) Time Performance on TCAS.



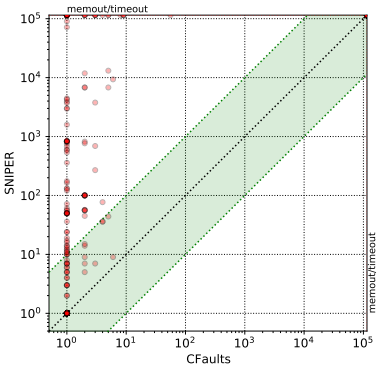

(b) Time Performance on C-PACK-IPAS.




(c) Costs of refined diagnoses on C-PACK-IPAS.




(d) Costs of diagnoses on C-PACK-IPAS.




(e) #Diagnoses generated on C-PACK-IPAS.

Figure 8.2: Comparison between BUGASSIST's, SNIPER's and CFAULTS' diagnoses.

minimal diagnosis from the preceding step that has not undergone refinement. The refinement step slightly slows down CFAULTS, as shown in Figures 8.2a and 8.2b. Nonetheless, Figure 8.2c illustrates a scatter plot comparing the optimum costs (MaxSAT solution's cost) achieved by CFAULTS with and without calling the refinement step on C-PACK-IPAS. Each point on this plot represents a faulty program, where the x-value (resp. y-value) represents the optimum cost of CFAULTS' with refinement (resp. without refinement) diagnosis. If a point lies above the diagonal, it indicates that a non-refined diagnosis has a higher cost than a refined diagnosis for the same program. Therefore, while the refinement step may marginally slow down CFAULTS, it enables CFAULTS to identify smaller diagnoses at a reduced cost in approximately 16% of C-PACK-IPAS's programs. Moreover, this observation was not noted in the TCAS dataset, as each program contains a maximum of two faults, and the refinement step did not yield improved outcomes in this particular dataset.

Additionally, Figure 8.2d illustrates a scatter plot comparing the diagnoses' costs achieved by CFAULTS (x-axis) against BUGASSIST (y-axis) on C-PACK-IPAS. BUGASSIST fails to provide an optimal diagnosis in almost 6% of cases. In the TCAS benchmark, although BUGASSIST manages to localize faults in all programs, it yields a non-optimal diagnosis in 10% of the programs. Furthermore, Figure 8.2e depicts a scatter plot comparing the number of diagnoses generated by CFAULTS (x-axis) against SNIPER (y-axis). While CFAULTS needs to enumerate all MaxSAT solutions due to the weighted MaxSAT formula, it is evident that SNIPER generates significantly more diagnoses than CFAULTS. This discrepancy sug-

gests that SNIPER overlooks the possibility of redundant diagnoses being computed. The number of such redundant diagnoses is much larger than the number of diagnoses generated by CFAULTS. Figure 8.2e illustrates that in some instances, SNIPER may enumerate up to 100K diagnoses, whereas CFAULTS generates less than 10.

As a validation step for our implementation, we analyzed all three fault localization methods on the collection of 799 correct programs in C-PACK-IPAS. This was done to ensure that all methods yielded zero faults for all correct implementations of each programming exercise. Moreover, we conducted a comparison between CFAULTS and the HSD algorithm [54] (see Section 8.2) on the ISCAS85 dataset [222], which is a widely studied collection of single-fault circuits. It is worth noting that HSD's implementation currently only supports fault localization in circuits. We encountered no performance issues during this comparison, and both approaches successfully localized all faults within each circuit.

## 8.5 Conclusion

This chapter introduces a novel formula-based fault localization technique for C programs capable of addressing any number of faults. Leveraging Model-Based Diagnosis (MBD) with multiple observations, CFAULTS consolidates all failing test cases into a unified MaxSAT formula, ensuring consistency in the fault localization process. Experimental evaluations on TCAS and C-PACK-IPAS, show that CFAULTS is faster than other FBFL approaches like BUGASSIST and SNIPER. Furthermore, CFAULTS only generates minimal diagnoses of faulty statements, while other methods tend to produce redundant diagnoses. In fact, leveraging on the usage of MaxSAT, CFAULTS only generates minimal diagnoses that minimize the total weight of the unhealthy components.

Next, in Chapter 9 CFAULTS is evaluated in a classroom setting. Furthermore, in Chapter 10, CFAULTS is integrated into a Large Language Model (LLM)-driven program repair tool to assess the impact of using formula-based fault localization (FBFL) to guide the repair process of LLMS.

# 9

# GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education

*"Everybody should learn to program a computer, because it teaches you how to think."*

– Steve Jobs.

## Contents

Due to the substantial number of enrollments in programming courses, a key challenge is delivering personalized feedback to students. The nature of this feedback varies significantly, contingent on the subject and the chosen evaluation method. However, tailoring current Automated Assessment Tools (AATs) to integrate other program analysis tools is not straightforward. Moreover, AATs usually support only specific programming languages, providing feedback exclusively through dedicated websites based on test suites.

This chapter introduces GITSEED, a language-agnostic automated assessment tool designed for Programming Education and Software Engineering (SE) and backed by GITLAB. The students interact with GITSEED through GITLAB. Using GITSEED, students in Computer Science (CS) and SE can master the fundamentals of git while receiving personalized feedback on their programming assignments and projects. Furthermore, faculty members can easily tailor GITSEED's pipeline by integrating various code evaluation tools (e.g., memory leak detection, fault localization, program repair, etc.) to offer personalized feedback that aligns with the needs of each CS/SE course. Our experiments assess GITSEED's efficacy via comprehensive user evaluation, examining the impact of feedback mechanisms and features on student learning outcomes. Findings reveal positive correlations between GITSEED usage and student engagement.

This chapter has been published as a conference paper at the 1st ACM Virtual Global Computing Education Conference, SIGCSE Virtual 2024 [223].

## 9.1 Introduction

Providing feedback to CS students on their programming assignments and software projects demands considerable time and effort from the faculty. Hence, there is a rising demand for systems, such as Automated Assessment Tools (AATs), that can deliver automated, comprehensive, and personalized feedback to students. When compared to non-automated evaluators, such as teaching assistants, AATs exhibit the ability to evaluate several assignments or code submissions efficiently and quickly. Hence, AATs facilitate the learning process since students get their feedback much faster. Moreover, AATs offer objectivity and consistency, adhering to some evaluation metric (e.g., a test suite).

The interest and development of AATs dates back to the 1960s [142, 224]. Over the past two decades, there has been a surge in the growth and adoption of AATs [144–149, 153]. However, despite the remarkable growth in the development and usage of AATs, certain drawbacks have become increasingly apparent. Primarily, a majority of AATs [144, 149] merely display the outcomes of a set of input/output tests used for the student's evaluation and lack other kinds of feedback. Secondly, ATTs tend to be specific to one programming language or a limited set of languages. AATs that are language-agnostic are scarce [145, 147]. Thirdly, AATs typically offer feedback solely through dedicated websites, necessitating students to familiarize themselves with new GUI interfaces. Finally, it is either challenging or impractical to adapt most AATs to integrate other program analysis tools, which might be essential to provide more personalized feedback in some CS/SE courses.

This chapter introduces GITSEED, a new tool that overcomes the aforementioned limitations of previ-

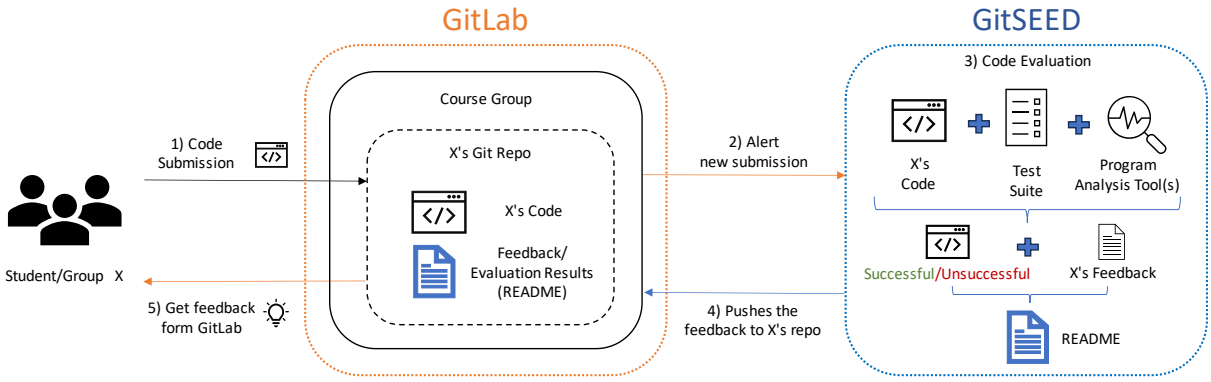



Figure 9.1: The overview of GITSEED.

ous AATs. GITSEED is a novel **Git**-backed AAT for **S**oftware **E**ngineering and Programming **Ed**ucation. Figure 9.1 presents the overview of GITSEED. As Figure 9.1 shows, the students interact with GITSEED through GITLAB. This way, CS/SE students can learn the fundamentals of git while receiving personalized feedback on their programming assignments and projects. Afterwards, using GITLAB's runners, GITSEED is notified whenever there is a new submission from group X. GITSEED evaluates this new submission against a test suite and using program analysis tools defined by the faculty. Finally, the resulting evaluation report is pushed into X's git repository (repo) so that the students have access to personalized feedback right away.

Furthermore, GITSEED is language agnostic, i.e., it can be used for any CS/SE course no matter the programming language(s) used. Moreover, most CS/SE students are familiar or will be familiar throughout their courses, with several git web interfaces (e.g., GITLAB, GITHUB, GITEA). Therefore, GITSEED eliminates the necessity for students to acquaint themselves with an unfamiliar GUI interface, which happens frequently in several universities where different CS/SE courses use different GUI interfaces for automated assessment of programming tasks [142, 144–146, 149].

GITSEED has two different categories of assessments: labs and projects. Either one is optional, and it is possible to have an unlimited number of projects depending on the chosen configuration. Faculty can choose which assessment model aligns best with their courses. Moreover, faculty members can easily tailor the pipeline of GITSEED, enhancing the quality of feedback provided to the students aligned with the needs of each CS/SE course. For example, code evaluation tools can be integrated into GITSEED, such as memory leak detection [225], fault localization [226, 227], program repair [2, 9, 173], plagiarism detection [183], code coverage, among others.

The chapter is organized as follows. Section 9.2 presents the implementation and possible configurations of GITSEED in more detail. GITSEED has already proven successful in two separate courses, a first-year programming course and a CS graduate course. Section 9.3 discusses our experiments and evaluates the effectiveness of GITSEED in enhancing programming education. Through the analysis of feedback from students enrolled in a first-year undergraduate course, we explore the role of GITSEED's features, including dashboards and feedback mechanisms, in facilitating learning and improving student performance. Finally, Section 9.4 briefly compares GITSEED with GITHUB CLASSROOM, and this chapter concludes in Section 9.5.

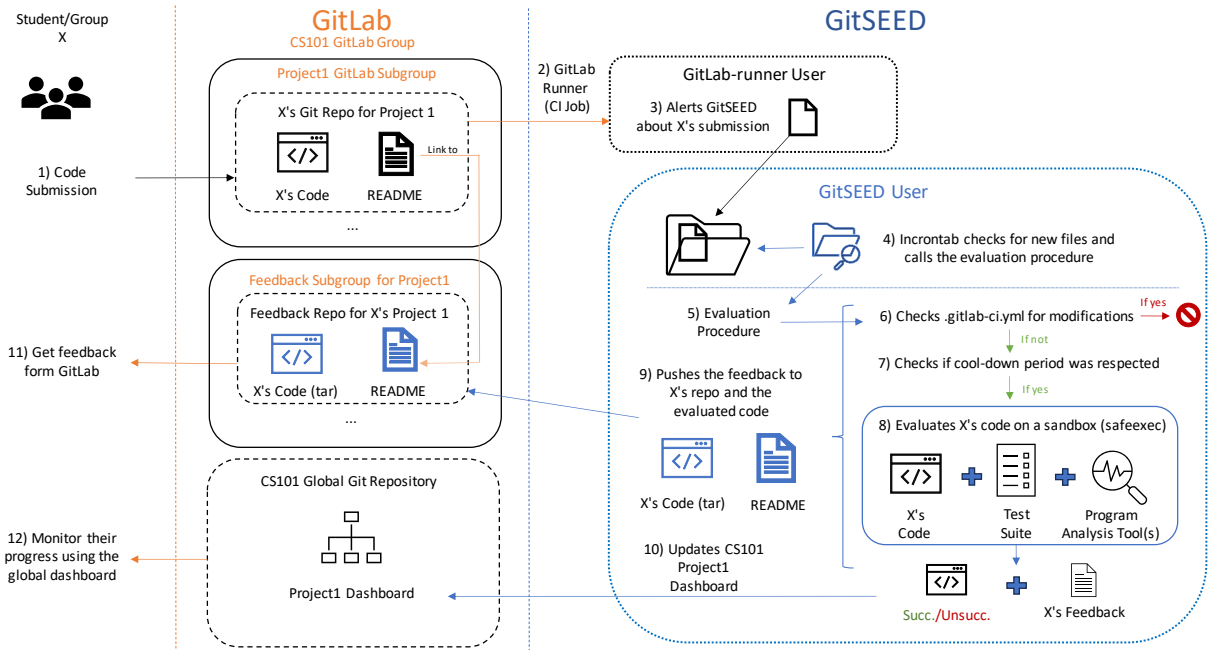



Figure 9.2: Workflow of GITSEED for processing a new project submission from group X in CS101.

To summarize, this chapter makes the following contributions:

- We present GITSEED, an open-source language-agnostic automated assessment tool designed for Software Engineering (SE) and Programming Education and backed by GITLAB;
- GITSEED is integrated into GITLAB's continuous integration (CI) workflow, which adopts educational assessment within a professional version control platform rather than a dedicated website, like so many other AATs;
- Students interact with GITSEED through GITLAB, learning this way the fundamentals of git while receiving personalized feedback on their assignments;
- Faculty members can easily adapt GITSEED by integrating other code analysis tools to offer personalized feedback that aligns with the needs of each CS/SE course.
- GITSEED is publicly available on GitLab [228] and on ZENODO [229].

## 9.2 GITSEED

This section presents the internals of GITSEED and configuration options. Section 9.2.1 describes the GITLAB features required by GITSEED. Next, Section 9.2.2 focuses on the back-end of GITSEED and its workflow, and Section 9.2.3 details the measures taken in order to ensure that all stages of the GITSEED pipeline are safe. Finally, Section 9.2.4 explains our implementation of GITSEED.

### 9.2.1 GITLAB

Figure 9.2 illustrates the complete workflow of GITSEED, where GITLAB plays the role of an intermediary between the students and GITSEED. Notice that students only interact with GITLAB, and then GITLAB triggers the processing of submissions in GITSEED. Furthermore, GITSEED was designed to work with other git web interfaces. Modern, widely-used programming editors/environments (e.g., VS CODE) already feature user-friendly interfaces for managing git repos. Hence, the submission process and getting feedback can be easily done within the student's programming environment, eliminating the need to exit their coding workspace. Alternatively, students can also use GITLAB's web interface.

#### 9.2.1.1 GitLab Group, Subgroups and Course Repo

GITSEED expects the following structure of groups in GITLAB: a main GITLAB group with all the students (e.g., CS101), a subgroup for each distinct evaluation element (e.g., labs, project, etc.), an additional subgroup for feedback, and a git repo for the entire course (CS101, in Figure 9.2) that contains all the course's information and dashboards for each evaluation element.

GITSEED has different repos for labs and projects. The rationale behind this choice is that while labs remain open throughout the semester, projects have distinct deadlines and may involve different groups of students. Furthermore, the feedback can also be pushed directly to the same git repos used by students for code submissions. However, having two different repos, one for code submissions and one for getting feedback, is the best approach for first-year students. This approach mitigates the chances of merging conflicts or git conflicts between students and GITSEED. Hence, GITSEED uses different repos to simplify the students' repos synchronization. This way, students manage their code development repos, and GITSEED only submits to the feedback repo.

In GITLAB, each project member is assigned a role that determines which actions they can take in the git repo [1]. In GITSEED, students assume the role of "Developers" for their own repos (e.g., projects, labs), granting them read and write access. However, students assume the role of "Reporters" for their feedback git repos and in the course global project, granting them viewing but not editing privileges. Note that the group of students can only see their own feedback repo and not those of other groups. Finally, faculty members hold the roles of "Maintainers" or "Owners" for all repos, depending on the chosen configuration.

#### 9.2.1.2 Continuous Integration (CI)

GITSEED takes advantage of the CI pipeline[2] available on GITLAB. The CI pipeline, essential for testing and deploying software projects, operates through a `.gitlab-ci.yml` script outlining all testing and deployment actions. It utilizes "runners", agents executing these actions, such as tests, as defined in the script.

[1] `https://docs.gitlab.com/ee/user/permissions.html`
[2] `https://docs.gitlab.com/ee/ci/`

**GitLab Runner** GITSEED requires a GITLAB runner to be installed and self-hosted in the machine where GITSEED is running (e.g., a Linux virtual machine (VM)). In this machine, the runner runs the job described in the `.gitlab-ci.yml`. The runner's job is to add information to GITSEED's queue that student X made a new submission to project Y. The faculty needs to adapt the `.gitlab-ci.yml` based on the programming language(s) being evaluated in the course. Distinct students can use different programming languages for the same programming task.

### 9.2.2 GITSEED's Back-end

Next, we detail the stages of the GITSEED workflow from Figure 9.2.

#### 9.2.2.1 GITLAB Manager

The GITLAB manager interacts with GITLAB and creates/modifies/clones all the necessary subgroups and repos and assigns the students and faculty to their own repos. GITSEED works for single-student groups and for groups with several students. Using the GITLAB manager, the faculty can easily manage students' access to their git repos. The students' accesses can be removed, for example, after a project deadline or if the students modified something in the git repo that were not supposed to (e.g., `.gitlab-ci.yml`).

#### 9.2.2.2 Assessments

GITSEED has two categories of assessments: labs and projects. Either one is optional, and it is possible to have an unlimited number of projects depending on the chosen configuration. Each type of assessment has its own evaluation script.

**Labs** GITSEED allows faculty members to publish each different lab's exercises in the student's repos. Each lab corresponds to a practical class. Additionally, with GITSEED, students who do not finish all the lab exercises during the class can still conclude and automatically check their programs afterwards.

**Projects** GITSEED allows the publication of the projects' descriptions and related data in the students' repos. Moreover, after a project's deadline, faculty members can remove the students' access to write into their repos and then reevaluate all the projects one last time. This last reevaluation might be necessary in case there was any submission that was not assessed due to the *cool-down period* (see description of cool-down periods next).

#### 9.2.2.3 Commits Database

GITSEED maintains a database containing all students' commits timestamps for every evaluation element. The goal is to have a cool-down period for each different evaluation element, ensuring that only the submissions respecting their previous cool-down period are evaluated. This measure is implemented to prevent overloading GITSEED's machine. Note that some CS/SE courses have thousands of

students, who tend to submit multiple times, especially near project deadlines. Moreover, a cool-down period forces students to think more thoroughly about their program before making new submissions, as no new feedback will be provided for commits made during this time. The default cool-down period is set at 1 minute for lab exercises and 10 minutes for project assignments. Nevertheless, these periods can be easily modified (see Section 9.2.2.8). Furthermore, the feedback report (README) lets the students know when their current cool-down period is over.

#### 9.2.2.4 Dashboard

GITSEED keeps a dashboard/leaderboard for each distinct evaluation element. These dashboards are automatically posted by GITSEED in the course's central git repo so all students can monitor their progress with regard to their colleagues. The dashboards keep track of each student/group's number of successful/unsuccessful tests, their number of submissions, and the number of days since the beginning of the project/lab assignment.

#### 9.2.2.5 GITLAB-Runner

The GITLAB runner described in Section 9.2.1.2 needs to be installed in the same machine where GITSEED is running and needs to have write access to the folder that keeps track of new submissions.

#### 9.2.2.6 Incrontab

The machine where GITSEED is installed has an incrontab daemon that is triggered by the GITLAB runner. The GITLAB runner adds the information that a new commit was performed on a given repo, and that triggers the evaluation procedure.

#### 9.2.2.7 Evaluation

During the evaluation process, GITSEED first checks if the students modified the `.gitlab-ci.yml` script. If this is the case, students lose their rights to push/modify the git repo and are asked to reach out to faculty members. Otherwise, GITSEED checks if the cool-down period was respected. If not, the student's new submission is not evaluated. Otherwise, if the cool-down period was respected, then GITSEED proceeds to the next evaluation step.

**`safeexec`** To run the students' code safely, GITSEED uses `safeexec` [3] which is a lightweight sandbox for executing user programs. Alongside `safeexec`, other program analysis tools can be run on the students' code. After completing the evaluation, GITSEED submits the evaluation report (README) and a tar file containing the evaluated code (e.g., a programming assignment's implementation) to the respective feedback repo. Finally, GITSEED updates the respective dashboard in the course's central git repo with the student's performance.

[3] `https://github.com/ochko/safeexec`

#### 9.2.2.8 Configurations

GITSEED has several predefined configurations that can be easily modified in the configuration file:

- Cool-down Period (default: 1 min): Amount of time students need to wait between their own submissions;
- Output Visible (default: false): GITSEED shows (or does not) the output of all tests to the students;
- Only First Wrong Output Visible (default: true): GITSEED only shows students their first incorrect output. This option is only used if the previous option is set to true;
- CPU Time Limit (default: 5 sec): GITSEED runs the students' code with this CPU time limit for each test case;
- Memory Limit (default: 8 GB): GITSEED runs the students' code with this memory limit for each test case.

**Easily Tailored** GITSEED's current methods for evaluation are fully language agnostic. The evaluation scripts can be easily tailored to evaluate different programming languages. Furthermore, faculty members can quickly adapt the GITSEED pipeline by integrating or replacing various code evaluation tools (e.g., memory leak detection, fault localization, program repair, plagiarism checks, solution checkers) to offer personalized feedback that aligns with the needs of each CS/SE course. Lastly, GITSEED was designed with modularity in mind. On that account, one can easily remove, add, or modify any component of GITSEED without compromising it.

### 9.2.3 Safety Measures

Several measures must be ensured for GITSEED to operate safely. Firstly, the GITLAB runner user needs to be granted write access to GITSEED's folder for new submissions. However, this user should not have access to any other folders, as the GITLAB runner executes code from the `.gitlab-ci.yml` script, which may be tampered with by students. By limiting access to only that folder, the GITLAB runner cannot alter or read anything else from GITSEED. Furthermore, GITSEED runs the students' code using `safeexec`, which simulates a sandbox controlling read/write accesses. Note that GITSEED is not dependent on `safeexec`. Due to GITSEED modularity, `safeexec` can be quickly replaced with some other sandbox application. Lastly, given the crucial role of `.gitlab-ci.yml` in GITSEED's functionality, this script is added to the `gitignore` file and students' READMEs explicitly instruct them not to edit this `yml` script. Nevertheless, GITSEED checks for any tampering with this script before evaluating the student's code. If detected, GITSEED restricts the student's access to the repo until the faculty checks the situation.

### 9.2.4 Implementation

For this chapter, we used a GITLAB instance self-hosted at Instituto Superior Técnico. The GITSEED system was deployed on a dedicated virtual machine running Linux (Debian 4.19) on a AMD Opteron(TM) Processor 6276 with 16GB of RAM. Additionally, the virtual machine hosted the gitlab-runner package, version 15.9.1. GITSEED is implemented using `bash` and python3 (version 3.9.16). GITSEED uses `bash` to execute and evaluate the students' code. On the other hand, GITSEED relies on python3 and `curl` to communicate with GITLAB, through its API (v3.15.0). Furthermore, GITSEED utilizes python3 and `sqlite3` for the maintenance of the course's dashboards and the database containing the commit history.

## 9.3 Impact Discussion

This section discusses our experiments with GITSEED, between Spring 2023 and Spring 2024, in two distinct academic courses at Instituto Superior Técnico, a first-year undergraduate and a graduate course. GITSEED offers both formative and summative assessments. Formative assignments, such as lab classes, remain accessible throughout the semester, allowing students to revise until correct. Summative assignments, such as projects, also permit unlimited attempts but come with strict deadlines. Students were briefed that formative assignments serve as learning aids, encouraging exploration without fear of repercussions for errors. The aim is for students to utilize GITSEED until mastery is achieved. Conversely, summative assignments serve as assessments of acquired knowledge and skill, showcasing proficiency in the subject. Since these summative assignments require more computation time and memory, higher cool-down periods were established between each group's submissions, to prevent overloading GITSEED's machine.

### 9.3.1 Courses Setup

#### 9.3.1.1 Undergraduate Course (Spring 2023)

GITSEED was initially used in Spring 2023 in a first-year undergraduate course, Introduction to Algorithms and Data Structures, where students learn how to program in C, with a total of 528 enrolled students. GITSEED was used to assess this course's labs (formative assignments) and projects (summative assignments).

**Assignments** For formative assignments, GITSEED created individual git repos for the lab classes. Configuration settings included a one-minute cool-down period, a five-second CPU time limit, and an 8GB memory limit for each programming assignment across eight labs. Throughout these eight labs, students made a total of 10338 code contributions to their repos. Regarding the summative assignments, this course had two different programming projects, each configured for single-student groups. Configuration settings included a five-second CPU time limit per test case, a 16GB memory limit, and a

10-minute cool-down period for project evaluations. Students made a total of 15061 code contributions to their repos, 7916 to the first project and 7145 to the second one.

**Program Analysis Tools** For projects evaluation, GITSEED was tailored to: (1) identify forbidden libraries in student projects, and (2) run VALGRIND [225] to detect memory leaks in their code. Providing feedback on memory leaks proved beneficial, particularly for first-year students unfamiliar with these tools.

**Opportunities for improvement** Throughout the course, we noticed that some students forgot to pull the feedback from GITLAB to their local repos before pushing new modifications, resulting in git-merge issues. Consequently, GITLAB's CI would ignore these commits. This happened because students use GITLAB web interface to get their feedback while coding in their local git repos. To address this issue, GITSEED now publishes feedback in separate repositories, with students' READMEs containing links for easy synchronization, especially for first-year students.

#### 9.3.1.2 Graduate Course (Fall 2023)

In Fall 2023, GITSEED was also used in a graduate course on Automated Reasoning with 38 students. The project consisted of solving an NP-Hard optimization problem through a Boolean logic solver using Python. There were 21 groups, each consisting of a maximum of two students. Configuration settings included a one-minute CPU time limit, a 16GB memory limit, and a 20-minute cool-down period for project evaluations. Throughout the project, students made a total of 269 contributions to their repos. Once again, we customized GITSEED, in this case, to incorporate both private and public test cases for evaluating students' code. Moreover, we also inserted additional software into the GITSEED pipeline that gave students feedback about the satisfiability and optimality of their projects' solutions.

#### 9.3.1.3 Undergraduate Course (Spring 2024)

In Spring 2024, GITSEED once again played a pivotal role by supporting the first-year undergraduate course, Introduction to Algorithms and Data Structures, which had a total of 509 students. GITSEED served as the platform for assessing labs and projects in the course, employing configurations similar to those outlined in Section 9.3.1.1. However, notable adjustments were made for this course iteration.

**Feedback** Feedback was provided in separate repositories based on the insights gained from the previous year.

**Program Analysis Tools** A significant enhancement was the integration of four additional program analysis tools into GITSEED: CFAULTS, CPPCHECK, CLANG-TIDY, and `Lizard`. `Lizard` [230] is a cyclomatic complexity analyzer for various programming languages, aiding in evaluating code length and complexity. The fault localization tool pinpointed faulty statements within the programs using a test suite. Additionally, CPPCHECK [231] and CLANG-TIDY [232] are static analyzers used to detect uninitialized variables and various errors, such as division by zero. Finally, CFAULTS [16] is a formula-based fault

localization tool, described in Chapter 8, that pinpoints bug locations within the programs. The insights generated by these tools were compiled into feedback reports and appended alongside test-suite evaluation outcomes in the students' feedback repositories. The results from both the fault localization tool and the static analyzers were presented to students as "Hints", strategically guiding them towards potential problematic statements within their programs. This approach aimed to provide students with targeted assistance in identifying and rectifying programming errors. Moreover, GITSEED was configured to display only the first incorrect output to students, fostering a focused learning environment.

### 9.3.2 User Study

In Spring 2024, we conducted a comprehensive user study with students to gather valuable feedback on their experience with GITSEED, particularly focusing on its dashboards and the various types of feedback provided by analysis tools, namely `valgrind`, `lizard`, and "hints" (generated by fault localization and static analyzers). Throughout the course, we noticed that incorporating motivational elements, such as the dashboards available within GITSEED, effectively encouraged student engagement and facilitated their progress. Approximately 20% of the students who were enrolled in the course for the entire semester took part in the questionnaire. They were asked anonymously to evaluate the usefulness of the different feedback mechanisms and features of GITSEED they encountered during the semester (see [223]). The findings revealed that students perceived the following aspects as beneficial:

**GITSEED**: 91.8% of students found GITSEED to be a valuable resource. Its role in providing a centralized platform for assignment submission, feedback reception, and revision evidently streamlined the learning process and enhanced overall comprehension. **Dashboards**: 82.2% of students acknowledged the significance of dashboards in tracking their progress and monitoring their performance relative to course objectives. **Hints**: Despite being less prevalent than other feedback mechanisms, 68.5% of students recognized the utility of hints generated by fault localization and static analyzers. These hints acted as invaluable pointers, directing students towards potential errors in their code and fostering a deeper understanding of programming concepts through self-correction. `Valgrind`: 90.4% of students found the feedback from `valgrind` to be beneficial. This tool's ability to detect memory management issues and provide detailed diagnostics undoubtedly aided students in debugging their programs and writing more robust code. `Lizard`: 75.3% of students appreciated the insights offered by `lizard`, particularly its analysis of code complexity and length. By highlighting areas of code that might require simplification or restructuring, `lizard` contributed to the optimization of students' coding practices and the cultivation of clearer, more efficient programming habits. In addition to evaluating the specific components of GITSEED, students were given the opportunity to provide general feedback through short-answer responses. These open-ended questions allowed students to express their thoughts, suggestions, and concerns regarding their overall experience with GITSEED. Overall, the user study underscored the positive impact of GITSEED's features and feedback mechanisms on students' learning experiences, reaffirming its value as a comprehensive educational tool for programming courses.

## 9.4 GITSEED VS GITHUB CLASSROOM

As previously mentioned in Section 3.4, GITHUB CLASSROOM [154] is an AAT tool available on GITHUB that allows faculty to create and manage digital classrooms and assignments. GITHUB CLASSROOM uses the same mechanism of a CI runner (GitHub Actions) to process student code and report on quality aspects. GITHUB CLASSROOM shares several benefits with GITSEED, is language agnostic, and enables students to learn the fundamentals of git while receiving feedback on their assignments. However, GITHUB CLASSROOM operates on a third-party platform. There may be regulatory or institutional policies that restrict the use of cloud-based services for certain types of data. On the other hand, GITLAB is open-source and can be self-hosted by educational institutes. Additionally, GITHUB CLASSROOM may not seamlessly integrate with existing learning management systems (LMS) used by educational institutions. This lack of integration can result in administrative challenges, such as maintaining separate platforms for course materials, grades, and communication. While GITSEED can be easily integrated with this kind of systems. Finally, while GITHUB CLASSROOM is free to use, some advanced features or integrations may require paid GITHUB plans. For example, the number of minutes available for GitHub Actions (CI) is limited per month. Instructors may need to consider the cost of providing GITHUB accounts or repositories for students, especially in cases where institutional resources are limited. In contrast, educational institutions can use the premium version of GITLAB for free, and both GITLAB and GITSEED are open-source projects. Finally, it is worth noting that there is no equivalent to GITHUB CLASSROOM on GITLAB, highlighting an opportunity for GITSEED to fill this gap.

## 9.5 Conclusion

This chapter presents GITSEED, an open-source, language-agnostic automated assessment tool (AAT) seamlessly integrated with GITLAB. Students benefit from personalized feedback on programming assignments and projects, mastering Git fundamentals simultaneously. Notably, GITSEED eliminates the need for students to navigate new GUI interfaces. Integrated into GITLAB's continuous integration (CI) workflow, GITSEED brings educational assessment into a professional version control platform rather than a dedicated web-based platform. Furthermore, faculty can easily customize GITSEED's pipeline with various code evaluation tools. Our experiments showcased GITSEED's success in both undergraduate and graduate courses, affirming its efficacy in programming education. It enhances student engagement and learning outcomes. Positive student feedback highlights GITSEED contribution to active learning and a supportive educational environment.

# 10

# Counterexample Guided Program Repair Using Zero-Shot Learning and MaxSAT-based Fault Localization

*"By harnessing the precision of formula-based fault localization, we can transform LLM-driven program repair from guesswork into a targeted, intelligent process, where each fix is guided by logic and clarity, unlocking the true potential of AI in debugging."*

– Chat-GPT 4.0.

## Contents

Symbolic semantic program repair approaches, which rely on Formal Methods (FM), check a program's execution against a test suite or reference solution, are effective but limited. These tools excel at identifying buggy parts but can only fix programs if the correct implementation and the faulty one share the same control flow graph. Conversely, Large Language Models (LLMs) are used for program repair but often make extensive rewrites instead of minimal adjustments. This tends to lead to more invasive fixes, making it harder for students to learn from their mistakes. In summary, LLMs excel at completing strings, while FM-based fault localization excels at identifying buggy parts of a program.

In this chapter, we propose a novel approach that combines the strengths of both FM-based fault localization and LLMs, via zero-shot learning, to enhance automated program repair (APR) for introductory programming assignments (IPAs). Our method uses MaxSAT-based fault localization to identify buggy parts of a program, then presents the LLM with a program sketch devoid of these buggy statements. This hybrid approach follows a Counterexample Guided Inductive Synthesis (CEGIS) loop to iteratively refine the program (see Definition 32). We ask the LLM to synthesize the missing parts, which are then checked against a test suite. If the suggested program is incorrect, a counterexample from the test suite is fed back to the LLM for revised synthesis. Our experiments on 1,431 incorrect student programs show that our counterexample guided approach, using MaxSAT-based bug-free program sketches (see Definition 23), significantly improves the repair capabilities of all six evaluated LLMs. This method allows LLMs to repair more programs and produce smaller fixes, outperforming other configurations and state-of-the-art symbolic program repair tools.

This chapter has been published as a conference paper at the 39th Annual AAAI Conference on Artificial Intelligence, AAAI 2025 [233].

## 10.1 Introduction

Traditional semantic APR techniques based on *Formal Methods* (FM), while providing high-quality fixes, are often slow and may struggle when the correct implementation diverges significantly from the erroneous one [135]. These APR approaches do not guarantee minimal repairs, as they align an incorrect submission with a correct implementation for the same IPA. If the alignment is not possible, these tools return a structural mismatch error, leaving the program unrepaired. In the past decade, there has been a surge in Machine Learning (ML) techniques for APR [3, 12–15, 18, 19]. ML-based approaches require multiple correct implementations to generate high-quality repairs, and need considerable time and resources to train on correct programs. While trained ML-based approaches generate repairs more quickly, they often produce imprecise and non-minimal fixes [10].

More recently, *Large Language Models* (LLMs) trained on code (LLMCs) have shown great potential in generating program fixes [20–23, 38–41]. LLM-based APR can be performed using zero-shot learning [234], few-shot learning [20] or fine-tuned models [23]. Fine-tuned models are the most commonly used, where the model is trained for a specific task. Conversely, zero-shot learning refers to the ability of a model to correctly perform a task without having seen any examples of that task during training. Few-shot learning refers to the LLMs's ability to perform tasks correctly with only a small number of examples

provided. Furthermore, the ability to generalize using zero or few-shot learning enables LLMs to handle a wide range of tasks without the need for costly retraining or fine-tuning. Nonetheless, few-shot learning can lead to larger fixes than necessary, as it is based on a limited number of examples. LLMs do not guarantee minimal repairs and typically rewrite most of the student's implementation to fix it, rather than making minimal adjustments, making their fixes less efficient and harder for students to learn from.

In this chapter, we propose a novel approach that combines the strengths of both FM and LLMs to enhance APR of IPAs via zero-shot learning. Our method involves using MaxSAT-based fault localization to identify the set of minimal buggy parts of a program and then presenting an off-the-self LLM with a program sketch devoid of these buggy statements. This hybrid approach follows a Counterexample Guided Inductive Synthesis (CEGIS) loop [70] to iteratively refine the program. We provide the LLM with a bug-free program sketch and ask it to synthesize the missing parts. After each iteration, the synthesized program is checked against a test suite. If the program is incorrect, a counterexample from the test suite is fed back to the LLM, prompting a revised synthesis.

Our experiments with 1431 incorrect student programs reveal that our counterexample guided approach, utilizing MaxSAT-based bug-free program sketches, significantly boosts the repair capabilities of all six evaluated LLMs. This method enables LLMs to repair more programs and produce superior fixes with smaller patches, outperforming both other configurations and state-of-the-art symbolic program repair tools [2, 9].

In summary, this chapter makes the following contributions:

- We tackle the Automated Program Repair (APR) problem using an LLM-Driven Counterexample Guided Inductive Synthesis (CEGIS) approach;
- We employ MaxSAT-based Fault Localization to guide and minimize LLMs' patches to incorrect programs by feeding them bug-free program sketches;
- Experiments show that our approach enables all six evaluated LLMs to fix more programs and produce smaller patches than other configurations and symbolic tools;
- Our code will be made publicly available on GitHub and Zenodo.

## 10.2 Motivation

Consider the program presented in Listing 10.1, which aims to determine the maximum among three given numbers. However, based on the test suite shown in Table 10.1, the program is buggy as its output differs from the expected results. The set of minimal faulty lines in this program includes lines 5 and 10, as these two `if` conditions are incorrect according to the test suite. A good way to provide personalized feedback to students on their IPAs is to highlight these two buggy lines. However, it is essential to check these faults by fixing the program and evaluating it against the test suite.

Using traditional Automated Program Repair (APR) tools for IPAs based on Formal Methods, such as CLARA [2] or VERIFIX [9], the program in Listing 10.1 cannot be fixed within 90 seconds. CLARA takes too long to compute a 'minimal' repair by considering several correct implementations for the same IPA,

**Listing 10.1:** Semantically incorrect program. Faulty lines: {5,10}.

```
1  // finds maximum of 3 numbers
2  int main(){
3     int f,s,t;
4     scanf("%d%d%d",&f,&s,&t);
5     if (f < s && f >= t)
6     //fix: f >= s
7        printf("%d",f);
8     else if (s > f && s >= t)
9        printf("%d",s);
10    else if (t < f && t < s)
11    //fix: t > f and t > s
12       printf("%d",t);
13
14    return 0;
15 }
```

**Listing 10.2:** Reference implementation.

```
1  int main() {
2     int m1,m2,m3,m;
3     scanf("%d%d%d",&m1,&m2,&m3);
4     m = m1 > m2 ? m1 : m2;
5     m = m3 > m ? m3 : m;
6     printf("%d\n", m);
7
8     return 0;
9  }
```

**Listing 10.3:** Program sketch with holes.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     @ HOLE 1 @
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     @ HOLE 2 @
9        printf("%d",t);
10
11    return 0;
12 }
```

**Listing 10.4:** GRANITE's fix using the program sketch.

```
1  int main(){
2     int f,s,t;
3     scanf("%d%d%d",&f,&s,&t);
4     if (f >= s && f >= t)
5        printf("%d",f);
6     else if (s > f && s >= t)
7        printf("%d",s);
8     else
9        printf("%d",t);
10
11    return 0;
12 }
```

| | Input | | | Output |
|---|---|---|---|---|
| $t_0$ | 1 | 2 | 3 | 3 |
| $t_1$ | 6 | 2 | 1 | 6 |
| $t_2$ | -1 | 3 | 1 | 3 |

Table 10.1: Test-suite.

while VERIFIX returns a compilation error. Conversely, using state-of-the-art LLMs trained for coding tasks (LLMCs), GRANITE [235] or CODEGEMMA [236], would involve providing the description of the programming assignment and some examples of input-output tests. Even with these features, neither LLM could fix the buggy program in Listing 10.1 within 90 seconds when repeatedly testing and refining their fixes. If the lecturer's reference implementation shown in Listing 10.2 is suggested as a reference in the prompt, both LLMs simply copy the correct program, ignoring instructions not to do so.

Hence, symbolic approaches demand an excessive amount of time to produce an answer, and LLMs, while fast, often produce incorrect fixes. A promising strategy to provide feedback to students on IPAs is to combine the strengths of both approaches. MaxSAT-based Fault localization [54, 100] can rigorously identify buggy statements, which can then be highlighted in the LLM prompt to focus on the specific parts of the program that need fixing. Listing 10.3 shows an example of a program sketch, which is a partially incomplete program where each buggy statement from the original incorrect program in Listing 10.1 is replaced with a `@ HOLE @`. Instructing the LLMs to complete this incomplete program allows both GRANITE and CODEGEMMA to fix the program in a single interaction, returning the program in Listing 10.4.

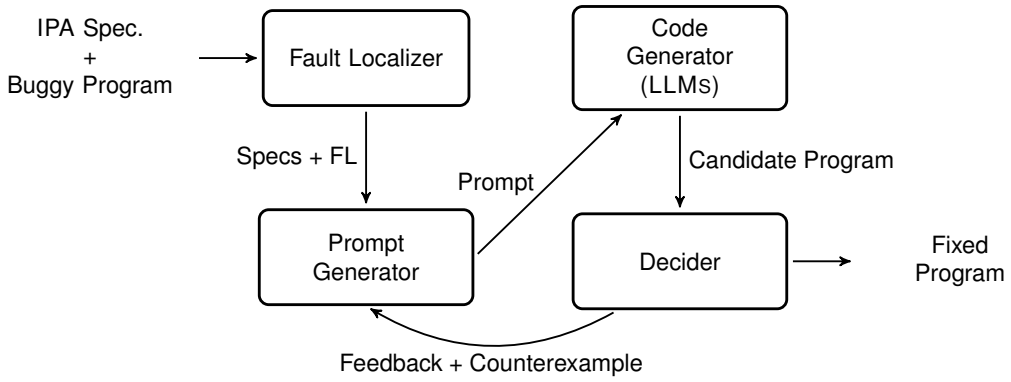



Figure 10.1: Counterexample Guided Automated Repair.

## 10.3 Counterexample Guided Automated Repair

Our approach combines the strengths of both Formal Methods (FM) and LLMs to enhance Automated Program Repair (APR). Firstly, we employ MaxSAT-based fault localization techniques to rigorously identify the minimal set of buggy parts of a program [16, 54]. Afterwards, we leverage LLMs to quickly synthesize the missing parts in the program sketch. Finally, we use a counterexample from the test suite to guide LLMs in generating patches that make the synthesized program compliant with the entire test suite, thus completing the repair. The rationale of our approach follows a Counterexample Guided Inductive Synthesis (CEGIS) [130] loop to iteratively refine the program. Figure 10.1 provides an overview of our APR approach. The input is a buggy program and the specifications for an introductory programming assignment (IPA), including a test suite, a description of the IPA, and the lecturer's reference implementation. We start by using MaxSAT-based fault localization techniques to identify the program's minimal set of faulty statements. Next, the prompt generator builds a prompt based on the specifications of the IPA and a bug-free program sketch reflecting the localized faults, then feeds this information to the LLM. The LLM generates a program based on the provided prompt. After each iteration, the Decider module evaluates the synthesised program against a test suite. If the program is incorrect, a counterexample chosen from the test suite is sent back to the prompt generator, which then provides this counterexample to the LLM to prompt a revised synthesis.

**Prompts.** The prompts fed to LLMs can contain various types of information related to the IPA. The typical information available in every programming course includes the description of the IPA, the test suite to check the students' submissions corresponding to the IPA's specifications, and the lecturer's reference implementation.

The syntax used in our prompts is similar to that in other works on LLM-driven program repair [21]. We have evaluated several types of prompts. Basic prompts are the simplest prompts that can be fed to an LLM without additional computation, including all the programming assignment's basic information. An example of such a prompt is the following:

````
Fix all semantic bugs in the buggy program below. Modify the code as little as possible.
Do not provide any explanation.

### Problem Description ###
Write a program that determines and  prints the largest of three integers  given by the user.

### Test Suite
#input:
6 2 1
#output:
6
// The other input-output tests

# Reference Implementation (Do not copy this program) <c> #
```c
int main(){
  // Reference Implementation
}
```

### Buggy Program <c> ###
```c
int main(){
  // Buggy program from Listing 1
}
```

### Fixed Program <c> ###
```c
````

In order to incorporate information about the faults localized in the program using MaxSAT-based fault localization, we utilized two different types of prompts: (1) `FIXME` annotations and (2) program sketches. `FIXME` annotated prompts are prompts where each buggy line identified by the fault localization tool is marked with a `/* FIXME */` comment. These prompts are quite similar to the basic prompt described previously, with the primary differences being the annotations in the buggy program and the first command given to the LLMs, which is modified as follows:

```
Fix all buggy lines with '/* FIXME */' comments in the buggy program below.
```

In the second type of prompt, to address program repair as a string completion problem, we evaluated the use of prompts where the buggy program is replaced by an incomplete program (program sketch), with each line identified as buggy by our fault localization module replaced by a *hole*. The command given to the LLMs is now to complete the incomplete program. Consequently, the sections
```

'Buggy Program' and 'Fixed Program' are replaced by 'Incomplete Program' and 'Complete Program', respectively, as follows:

```
Complete all the '@ HOLES N @' in the incomplete program below.
// ...
### Incomplete Program <c> ###
// ...
### Complete Program <c> ###
```c
```

**Feedback.** If the candidate program generated by the LLM is not compliant with the test suite, this feedback is provided to the LLM in a new message through iterative querying. This new prompt indicates that the LLM's previous suggestion to fix the buggy program was incorrect and provides a counterexample (i.e., an IO test) where the suggested fixed program produces an incorrect output. Hence, we provide the LLM with a feedback prompt similar to:

```
### Feedback ###
Your previous suggestion was incorrect! Try again. Code only. Provide no explanation.

### Counterexample  ###
#input:
6 2 1
#output:
6

### Fixed Program <c> ###
```c
```

## 10.4 Experimental Results

The goal of our evaluation is to answer the following research questions:

**RQ1.** How effective are state-of-the-art (SOTA) LLMs in repairing introductory programming assignments (IPAs) compared to different SOTA semantic repair approaches?

**RQ2.** How do different prompt configurations impact the performance of LLMs?

**RQ3.** How does FM-based fault localization impact LLM-driven APR?

**RQ4.** How helpful is it to provide a reference implementation for the same IPA to the LLMs?

**RQ5.** How helpful is it to provide LLMs with a variable mapping between the buggy program and a reference implementation of the same IPA?

**RQ6.** What is the performance impact of using a Counterexample Guided approach in LLM-driven APR?

**Experimental Setup.** All LLMs were run using NVIDIA RTX A4000 graphics cards with 16GB of memory on an Intel(R) Xeon(R) Silver 4130 CPU @ 2.10GHz with 48 CPUs and 128GB RAM. All the experiments related to the program repair tasks were conducted on an Intel(R) Xeon(R) Silver computer with 4210R CPUs @ 2.40GHz, using a memory limit of 10GB and a timeout of 90 seconds.

**Evaluation Benchmark.** To evaluate our work, we used C-PACK-IPAS [171], which is a set of student programs developed during an introductory programming course in the C programming language. These programs were collected over three distinct lab classes for 25 different programming assignments throughout three academic years. Each lab class focuses on a different topic of the C programming language. The first class deals with integers and input-output operations. Secondly, the second class focuses on loops and chars. Lastly, in the third lab class, students learn to use vectors and strings. Since this work focuses only on semantic program repair, only submissions that compile without any errors were selected. The set of submissions was split into two sets: correct submissions and incorrect submissions. The students' submissions that satisfied a set of input-output test cases for each IPA were considered correct. The submissions that failed at least one input-output test were considered incorrect. C-PACK-IPAS contains 1431 semantically incorrect programs submitted for 25 different IPAS.

### 10.4.1 Large Language Models (LLMs)

In our experiments, we used only open-access LLMs available on Hugging Face [237] with approximately 7 billion parameters for three primary reasons. Firstly, closed-access models like Chat-GPT are cost-prohibitive and raise concerns over student data privacy. Secondly, models with a very large number of parameters (e.g., 70B) need significant computational resources, such as GPUs with higher RAM capacities, and take longer to generate responses, which is unsuitable for a classroom setting. Thirdly, we used these off-the-shelf LLMs to evaluate the publicly available versions without fine-tuning them. This approach ensures that the LLMs used in this chapter are available to anyone without investing time and resources into fine-tuning these models. Thus, we evaluated six different LLMs for this study through iterative querying. Three of these models are LLMCs, i.e., LLMs fine-tuned for coding tasks: IBM's GRANITE [235], Google's CODEGEMMA [236] and Meta's CODELLAMA [238]. The other three models are general-purpose LLMs not specifically tailored for coding tasks: Google's GEMMA [239], Meta's LLAMA3 (latest version of the LLAMA family [240]) and Microsoft's PHI3 [241].

We selected specific variants of each model to optimize their performance for our program repair tasks. For Meta's LLAMA3, we utilized the 8B-parameter instruction-tuned variant. This model is designed to follow instructions more accurately, making it suitable for a range of tasks, including program repair. For CODELLAMA, we used the 7B-parameter instruct-tuned version, which is specifically designed for general code synthesis and understanding, making it highly effective for coding tasks. We employed GRANITE model with 8B-parameters, fine-tuned to respond to coding-related instructions. For PHI3, we opted for the mini version, which has 3.8B-parameters and a context length of 128K. This smaller model is efficient yet capable of handling extensive context, making it practical for educational settings. For GEMMA, we used the 7B-parameter instruction-tuned version, optimized to follow detailed instructions.

Lastly, for CODEGEMMA, we selected the 7B-parameter instruction-tuned variant, designed specifically for code chat and instruction, enhancing its capability in handling programming-related queries and tasks. To fit all LLMS into 16GB GPUs, we used model quantization of 4bit. Moreover, all LLMS were run using Hugging Face's Pipeline architecture. By using these different LLMS, we aimed to balance computational efficiency with the ability to effectively generate and refine code, facilitating a practical APR approach in an educational environment.

### 10.4.2 Fault Localization

We used CFAULTS [16] which is a formula-based fault localization (FL) tool, that pinpoints bug locations within the programs. It aggregates all failing test cases into a unified MaxSAT formula (see Chapter 8). This FL tool can be easily replaced by other FL tools (e.g., spectrum-based).

### 10.4.3 Evaluation

To assess the effectiveness of the program fixes generated by the LLMS under different prompt configurations, we used two key metrics: the number of programs successfully repaired and the quality of the repairs. For assessing the patches quality, we use the *Tree Edit Distance* (TED) [134, 242] to compute the distance between the student's buggy program and the fixed program returned by the LLMS. TED computes the structural differences between two Abstract Abstract Syntax Trees (AASTS) by calculating the minimum number of edit operations (i.e., insertions, deletions, and substitutions) needed to transform one AST into another. Based on this metric for measuring program distances, we computed the *distance score*, defined by Equation 10.1. This score aims to identify and penalize LLMS that replace the buggy program with the reference implementation rather than fixing it. The distance score is zero when the TED of the original buggy program ($T_o$) to the program suggested by the LLM ($T_f$) is the same as the TED of the reference implementation ($T_r$) to $T_o$. Otherwise, it penalizes larger fixes than necessary to align the program with the correct implementation.

$$ds(T_f, T_o, T_r) = max\Big(0, 1 - \frac{\mathrm{TED}(T_f, T_o)}{\mathrm{TED}(T_r, T_o)}\Big) \tag{10.1}$$

**Baseline.** We used two state-of-the-art symbolic semantic program repair tools for IPAS as baselines: VERIFIX [9] and CLARA [2]. VERIFIX employs MaxSMT to align a buggy program with a reference solution provided by the lecturer, while CLARA clusters multiple correct implementations and selects the one that produces the smallest fix when aligned with the buggy program. Both tools require an exact match between the control flow graphs (e.g., branches, loops) and a bijective relationship between the variables; otherwise, they return a structural mismatch error. VERIFIX was provided with each buggy program, the reference implementation, and a test suite. CLARA was given all correct programs from different academic years to generate clusters for each IPA. With a time limit of 90 seconds, VERIFIX can only repair 6.3% of the benchmark due to structural and unsupported errors, while CLARA repairs 34.6%.

| | Prompt Configurations | | | | |
|---|---|---|---|---|---|
| **LLMs** | **CE** | **De** | **De-TS** | **De-TS-CE** | **FIXME** |
| **CodeGemma** | 451 (31.5%) | 564 (39.4%) | 597 (41.7%) | 606 (42.3%) | 413 (28.9%) |
| **CodeLlama** | 447 (31.2%) | 472 (33.0%) | 492 (34.4%) | 500 (34.9%) | 403 (28.2%) |
| **Gemma** | 379 (26.5%) | 462 (32.3%) | 496 (34.7%) | 492 (34.4%) | 328 (22.9%) |
| **Granite** | 529 (37.0%) | 584 (40.8%) | 626 (43.7%) | 624 (43.6%) | 459 (32.1%) |
| **Llama3** | 496 (34.7%) | 533 (37.2%) | 564 (39.4%) | 590 (41.2%) | 400 (28.0%) |
| **Phi3** | 367 (25.6%) | 462 (32.3%) | 494 (34.5%) | 489 (34.2%) | 313 (21.9%) |
| **Portfolio (All LLMs)** | 732 (51.2%) | 821 (57.4%) | 842 (58.8%) | 846 (59.1%) | 655 (45.8%) |
| **LLMs** | **FIXME_De-CE** | **FIXME_De-TS** | **FIXME_De-TS-CE** | **Sk** | **Sk_De-CE** |
| **CodeGemma** | 563 (39.3%) | 592 (41.4%) | 601 (42.0%) | 484 (33.8%) | 615 (43.0%) |
| **CodeLlama** | 485 (33.9%) | 481 (33.6%) | 463 (32.4%) | 467 (32.6%) | 547 (38.2%) |
| **Gemma** | 443 (31.0%) | 446 (31.2%) | 444 (31.0%) | 367 (25.6%) | 524 (36.6%) |
| **Granite** | 574 (40.1%) | 566 (39.6%) | 583 (40.7%) | 550 (38.4%) | 653 (45.6%) |
| **Llama3** | 531 (37.1%) | 535 (37.4%) | 557 (38.9%) | 434 (30.3%) | 565 (39.5%) |
| **Phi3** | 448 (31.3%) | 460 (32.1%) | 474 (33.1%) | 367 (25.6%) | 506 (35.4%) |
| **Portfolio (All LLMs)** | 806 (56.3%) | 796 (55.6%) | 820 (57.3%) | 717 (50.1%) | 890 (62.2%) |
| **LLMs** | **Sk_De-TS** | **Sk_De-TS-CE** | **TS** | **Portfolio (All Configurations)** | |
| **CodeGemma** | 682 (47.7%) | **688 (48.1%)** | 511 (35.7%) | 850 (59.4%) | |
| **CodeLlama** | **573 (40.0%)** | 561 (39.2%) | 466 (32.6%) | 748 (52.3%) | |
| **Gemma** | 532 (37.2%) | **534 (37.3%)** | 404 (28.2%) | 780 (54.5%) | |
| **Granite** | **691 (48.3%)** | 681 (47.6%) | 577 (40.3%) | 887 (62.0%) | |
| **Llama3** | 578 (40.4%) | **591 (41.3%)** | 505 (35.3%) | 929 (64.9%) | |
| **Phi3** | **547 (38.2%)** | 535 (37.4%) | 400 (28.0%) | 759 (53.0%) | |
| **Portfolio (All LLMs)** | 900 (62.9%) | **907 (63.4%)** | 767 (53.6%) | 1050 (73.4%) | |
| **Verifix** [9] | 90 (6.3%) | | | | |
| **Clara** [2] | 495 (34.6%) | | | | |

Table 10.2: The number of programs fixed by each LLM under various configurations. Row *Portfolio (All* LLMs*)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **De** - IPA *Description*, **FIXME** - *FIXME Annotations*, **SK** - *Sketches*, **TS** - *Test Suite*.

Table 10.2 presents the number of programs repaired by each LLM under various configurations. The row labeled `Portfolio` represents the best possible outcomes by selecting the optimal configuration for each program across all LLMs. Meanwhile, `Portfolio` column highlights the best results achieved by a particular LLM across all tested configurations. Entries highlighted in bold correspond to the highest success rates for each LLM. The configurations yielding the highest success rates for the six evaluated LLMs involve incorporating a reference implementation of the IPA into the prompt. However, rather than genuinely fixing the buggy program, the LLMs often replace it with the reference implementation. For instance, GRANITE repairs 876 programs using a configuration that includes bug-free program sketches (Sk), an IPA description, counterexamples, a test suite, and the reference implementation (Sk_De-TS-CE-RI). Notably, 442 of these repaired programs exhibit a TED value of zero between the reference implementation and the fixed program, indicating that GRANITE is replicating the reference implementation. To address this, we separately analyzed configurations that include and exclude access to a reference implementation. When no reference implementation is provided (top of Table 10.2), GRANITE still leads among the LLMs, fixing up to 59.1% of the programs across all configurations and 48.3% when using sketches (SK), the IPA description, and a test suite (SK_De-TS). CODEGEMMA also

| | Prompt configurations with access to Reference Implementations | | | | |
|---|---|---|---|---|---|
| **LLMs** | **CPA** | **CPIA** | **De-TS-CE-CPA** | **De-TS-CE-RI** | **FIXME_De-TS-CE-CPA** |
| **CodeGemma** | 505 (35.3%) | 447 (31.2%) | 578 (40.4%) | 576 (40.3%) | 637 (44.5%) |
| **CodeLlama** | 532 (37.2%) | 517 (36.1%) | 528 (36.9%) | 525 (36.7%) | 565 (39.5%) |
| **Gemma** | 633 (44.2%) | 364 (25.4%) | 595 (41.6%) | 607 (42.4%) | 563 (39.3%) |
| **Granite** | 758 (53.0%) | 516 (36.1%) | 773 (54.0%) | 828 (57.9%) | 794 (55.5%) |
| **Llama3** | 595 (41.6%) | 464 (32.4%) | 685 (47.9%) | 691 (48.3%) | 657 (45.9%) |
| **Phi3** | 465 (32.5%) | 367 (25.6%) | 552 (38.6%) | 444 (31.0%) | 545 (38.1%) |
| **Portfolio (All LLMs)** | 1021 (71.3%) | 726 (50.7%) | 1033 (72.2%) | 1046 (73.1%) | 1011 (70.6%) |
| **LLMs** | **FIXME_De-TS-CE-RI** | **FIXME_De-TS-CPA** | **FIXME_De-TS-RI** | **RI** | **Sk_De-TS-CE-CPA** |
| **CodeGemma** | 638 (44.6%) | 647 (45.2%) | 640 (44.7%) | 445 (31.1%) | 725 (50.7%) |
| **CodeLlama** | 609 (42.6%) | 604 (42.2%) | 611 (42.7%) | 455 (31.8%) | 633 (44.2%) |
| **Gemma** | 616 (43.0%) | 580 (40.5%) | 655 (45.8%) | 582 (40.7%) | 664 (46.4%) |
| **Granite** | 857 (59.9%) | 838 (58.6%) | 882 (61.6%) | 775 (54.2%) | 838 (58.6%) |
| **Llama3** | 681 (47.6%) | 662 (46.3%) | 661 (46.2%) | 555 (38.8%) | 725 (50.7%) |
| **Phi3** | 492 (34.4%) | 572 (40.0%) | 508 (35.5%) | 358 (25.0%) | 639 (44.7%) |
| **Portfolio (All LLMs)** | 1056 (73.8%) | 1036 (72.4%) | 1082 (75.6%) | 1039 (72.6%) | 1050 (73.4%) |
| **LLMs** | **Sk_De-TS-CE-RI** | **Sk_De-TS-CPA** | **Sk_De-TS-RI** | **Portfolio (All Configurations)** | |
| **CodeGemma** | 739 (51.6%) | **744 (52.0%)** | 729 (50.9%) | 950 (66.4%) | |
| **CodeLlama** | 675 (47.2%) | 673 (47.0%) | **677 (47.3%)** | 959 (67.0%) | |
| **Gemma** | **732 (51.2%)** | 681 (47.6%) | 720 (50.3%) | 1025 (71.6%) | |
| **Granite** | 876 (61.2%) | 881 (61.6%) | **921 (64.4%)** | 1169 (81.7%) | |
| **Llama3** | 730 (51.0%) | **783 (54.7%)** | 706 (49.3%) | 1073 (75.0%) | |
| **Phi3** | 647 (45.2%) | **661 (46.2%)** | 653 (45.6%) | 959 (67.0%) | |
| **Portfolio (All LLMs)** | 1077 (75.3%) | 1080 (75.5%) | **1089 (76.1%)** | 1218 (85.1%) | |

Table 10.3: The number of programs fixed by each LLM under various configurations *with access to a reference implementation* of each IPA. Row *Portfolio (All* LLMS*)*, shows the best results across all LLMS for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **CPA** - *Closest Program using* AASTS, **CPIA** - *Closest Program using Invariants +* AASTS, **De** - IPA *Description*, **FIXME** - *FIXME Annotations*, **RI** - *Reference Implementation*, **SK** - *Sketches*, **TS** - *Test Suite*. Entries highlighted in bold correspond to the highest success rates for each LLM.

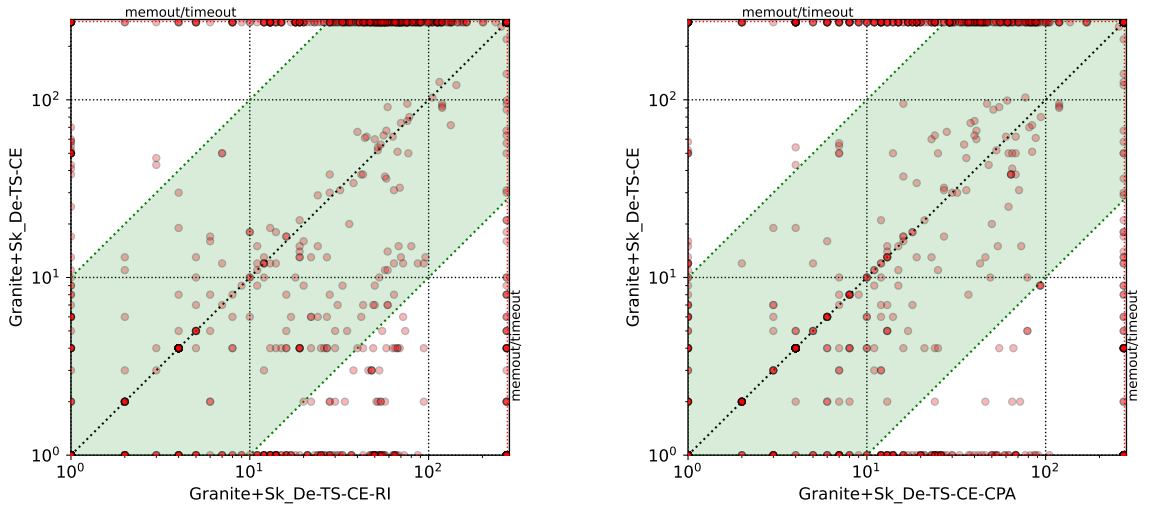
performs well, achieving up to 57.5% success in a portfolio approach and showing particular strength in configurations involving sketches (SK). For instance, CODEGEMMA can repair 48.1% of the evaluation benchmark using bug-free sketches, IPA description, test suite, and counterexample (SK_De-TS-CE). Configurations incorporating sketches (SK) and FIXME annotations generally yield better results. Including counterexamples (CE), IPA descriptions, and test suites (De-TS) further boosts the success rate across different LLMS. The portfolio approach, which combines the strengths of all LLMS and configurations without using reference implementation, achieves the highest overall success rate, fixing 70.8% of the programs. This demonstrates that leveraging multiple LLMS together can significantly enhance repair success.

#### 10.4.3.1 Correct Implementations

Furthermore, Table 10.3 provides the results of LLMS with a correct implementation for the same programming assignment. The correct implementation can be either the lecturer's implementation for the same IPA, the closest correct program based on the programs' Anonymous Abstract Syntax Trees (AASTS), or the closest correct program based on the programs' AASTS and their sets of invariants (CPIA) from a previously submitted student program, determined by INVAASTCLUSTER [42] (see Chapter 5). The intent was to allow the model to reuse correct code snippets to generate repairs. Results

| Metric: **sum(Distance Score)** | | | | | |
|---|---|---|---|---|---|
| | **Prompt Configurations** | | | | |
| **LLMs** | **FIXME_De-TS** | **FIXME_De-TS-CE** | **FIXME_De-TS-CE-CPA** | **FIXME_De-TS-CE-RI** | **FIXME_De-TS-CPA** |
| **CodeGemma** | 469.6 | 479.3 | 249.2 | 426.3 | 250.6 |
| **CodeLlama** | 413.5 | 403.9 | 240.4 | 409.2 | 258.3 |
| **Gemma** | 284.3 | 282.2 | 142.2 | 264.0 | 144.1 |
| **Granite** | 463.1 | 470.6 | 171.2 | 298.8 | 160.4 |
| **Llama3** | 353.4 | 353.6 | 175.9 | 392.7 | 176.0 |
| **Phi3** | 276.0 | 291.9 | 96.8 | 223.8 | 96.8 |
| **LLMs** | **FIXME_De-TS-RI** | **RI** | **Sk** | **Sk_De-CE** | **Sk_De-TS** |
| **CodeGemma** | 435.7 | 373.5 | 421.8 | 473.8 | 524.4 |
| **CodeLlama** | 417.3 | 332.3 | 421.6 | 464.8 | **477.9** |
| **Gemma** | 272.5 | 227.2 | 280.1 | 328.6 | 338.8 |
| **Granite** | 282.7 | 93.8 | 479.4 | 506.2 | **539.8** |
| **Llama3** | 390.6 | 409.8 | 354.6 | 383.1 | 379.8 |
| **Phi3** | 232.6 | 203.2 | 265.1 | 299.5 | **326.5** |
| **LLMs** | **Sk_De-TS-CE** | **Sk_De-TS-CE-CPA** | **Sk_De-TS-CE-RI** | **Sk_De-TS-CPA** | **Sk_De-TS-RI** |
| **CodeGemma** | **529.5** | 249.8 | 497.3 | 268.8 | 484.3 |
| **CodeLlama** | 464.5 | 251.3 | 459.0 | 270.8 | 452.1 |
| **Gemma** | **340.3** | 156.4 | 316.2 | 153.9 | 307.2 |
| **Granite** | 533.6 | 172.3 | 334.5 | 175.4 | 349.4 |
| **Llama3** | 384.5 | 172.7 | **423.0** | 196.4 | 407.3 |
| **Phi3** | 321.4 | 98.2 | 253.4 | 97.2 | 256.6 |

Table 10.4: The cumulative distance scores for each program successfully repaired by each LLM across various configurations. Entries highlighted in bold correspond to the highest score for each LLM.




(a) Reference Implementation.

(b) Closest Program using AASTs.

Figure 10.2: Comparison of tree edit distances (TED) for GRANITE's repairs when using (x-axis) versus not using (y-axis) correct implementations with configuration Sk_De-TS-CE.

show that including a reference implementation allows for better repair results. However, as mentioned earlier, the LLMs often simply copy the provided reference implementation.

Table 10.4 presents the sum of the distance scores (see Equation 10.1) for the top-performing LLMs from Tables 10.2 and 10.3 across different configurations. This summation aims to penalize LLMs that either copy the provided reference implementation or generate unnecessarily large repairs. For example, GRANITE using configuration Sk_De-TS-CE-RI can repair 876 programs but yields a total distance score of 334.5, whereas using the same configuration without a correct implementation repairs 681 programs resulting in a higher distance score of 533.6.

Figure 10.2a shows a scatter plot that compares the tree edit distance (TED) of the buggy program to the program fixed by GRANITE with and without a reference implementation, using configuration Sk_De-

| | Prompt configurations with access to Reference Implementations and Variable Mappings | | | | |
|---|---|---|---|---|---|
| **LLMs** | **CPA-VM** | **CPIA-VM** | **De-TS-CE-CPA-VM** | **De-TS-CE-RI-VM** | **FIXME_De-TS-CE-CPA-VM** |
| **CodeGemma** | 597 (41.7%) | 579 (40.5%) | 651 (45.5%) | 624 (43.6%) | 700 (48.9%) |
| **CodeLlama** | 660 (46.1%) | **699 (48.8%)** | 589 (41.2%) | 568 (39.7%) | 614 (42.9%) |
| **Gemma** | 738 (51.6%) | 700 (48.9%) | 675 (47.2%) | 680 (47.5%) | 656 (45.8%) |
| **Granite** | 827 (57.8%) | 721 (50.4%) | 821 (57.4%) | 869 (60.7%) | 846 (59.1%) |
| **Llama3** | 591 (41.3%) | 565 (39.5%) | 729 (50.9%) | 711 (49.7%) | 689 (48.1%) |
| **Phi3** | 552 (38.6%) | 516 (36.1%) | 598 (41.8%) | 555 (38.8%) | 601 (42.0%) |
| **Portfolio (All LLMs)** | 1036 (72.4%) | 991 (69.3%) | 1042 (72.8%) | 1070 (74.8%) | 1033 (72.2%) |

| **LLMs** | **FIXME_De-TS-CE-RI-VM** | **RI-VM** | **Sk_De-TS-CE-CPA-VM** | **Sk_De-TS-CE-RI-VM** | **Portfolio (All Configurations)** |
|---|---|---|---|---|---|
| **CodeGemma** | 705 (49.3%) | 526 (36.8%) | **782 (54.6%)** | 780 (54.5%) | 959 (67.0%) |
| **CodeLlama** | 618 (43.2%) | 543 (37.9%) | 681 (47.6%) | 677 (47.3%) | 984 (68.8%) |
| **Gemma** | 650 (45.4%) | 639 (44.7%) | 756 (52.8%) | **766 (53.5%)** | 1082 (75.6%) |
| **Granite** | 882 (61.6%) | 832 (58.1%) | 901 (63.0%) | **921 (64.4%)** | 1167 (81.6%) |
| **Llama3** | 669 (46.8%) | 559 (39.1%) | **792 (55.3%)** | 720 (50.3%) | 1060 (74.1%) |
| **Phi3** | 531 (37.1%) | 519 (36.3%) | **691 (48.3%)** | **691 (48.3%)** | 988 (69.0%) |
| **Portfolio (All LLMs)** | 1075 (75.1%) | 1082 (75.6%) | 1078 (75.3%) | **1093 (76.4%)** | 1210 (84.6%) |

Table 10.5: The number of programs fixed by each LLM under various configurations with access to variable mappings. Row *Portfolio (All* LLMs*)*, shows the best results across all LLMs for each configuration. Column *Portfolio (All Configurations)* shows the best results for each LLM across all configurations. Mapping abbreviations to configuration names: **CE** - *Counterexample*, **CPA** - *Closest Program using* AASTs, **CPIA** - *Closest Program using Invariants +* AASTs, **De** - IPA *Description*, **FIXME** - *FIXME Annotations*, **RI** - *Reference Implementation*, **SK** - *Sketches*, **TS** - *Test Suite*, **VM** - *Variable Mapping*.

TS-CE. Each point represents a faulty program, where the x-value (resp. y-value) represents the TED cost of GRANITE' with access to a reference implementation (resp. without it). Points below the diagonal indicate that fixing a program with access to a correct implementation incurs a higher TED cost than fixing it without access. This suggests that while access to a reference implementation enables GRANITE and other LLMs to repair more programs, it often results in larger changes to the student's program than when no correct implementation is given.

Similarly, Figure 10.2b shows a scatter plot that compares the TED cost between the buggy program and the program fixed by GRANITE using the closest correct implementation determined by AASTs (CPA) and without using this implementation. This plot shows that while having a correct implementation helps GRANITE repair more programs, it generally results in fewer modifications to the student's code when the CPA is used instead of the lecturer's reference implementation.

We did not run the same prompt configuration using the closest correct program considering both AASTs and invariants (CPIA) because, as shown in Table 10.3, this approach did not contribute positively to the repair process of the LLMs.

#### 10.4.3.2 Variable Mappings

Comparing two programs is a highly challenging task due to the undecidability of the program equivalence problem. Consequently, mapping variables between two programs is essential for various applications, such as program repair [173, 243]. In this context, we have evaluated the impact of incorporating variable mappings between the buggy program and a given correct implementation into the prompts used by LLMs. We compute these variable mappings using Graph Neural Networks (GNNs)[173], which map the set of variables between each correct program and the incorrect submission based on

| Metric: **sum(Distance Score)** | | | | | |
|---|---|---|---|---|---|
| | **Prompt Configurations** | | | | |
| **LLMs** | **CPA-VM** | **CPIA-VM** | **De-TS-CE-CPA-VM** | **De-TS-CE-RI-VM** | **FIXME_De-TS-CE-CPA-VM** |
| **CodeGemma** | 268.6 | 382.2 | 257.6 | 470.5 | 275.5 |
| **CodeLlama** | 217.3 | 351.6 | 253.9 | 426.8 | 231.0 |
| **Gemma** | 137.2 | 185.3 | 156.5 | 243.4 | 166.1 |
| **Granite** | 78.1 | 109.4 | 148.1 | 283.5 | 141.8 |
| **Llama3** | 249.9 | 316.3 | 199.2 | 426.0 | 189.1 |
| **Phi3** | 134.2 | 182.1 | 76.4 | 214.2 | 75.3 |
| **LLMs** | **FIXME_De-TS-CE-RI-VM** | **RI-VM** | **Sk_De-TS-CE-CPA-VM** | **Sk_De-TS-CE-RI-VM** | |
| **CodeGemma** | 475.3 | 410.1 | 286.9 | **509.5** | |
| **CodeLlama** | 424.4 | 372.7 | 250.0 | **455.2** | |
| **Gemma** | 263.3 | 195.7 | 171.5 | **306.4** | |
| **Granite** | 281.3 | 114.8 | 165.7 | **330.9** | |
| **Llama3** | 407.0 | 429.8 | 196.1 | **432.1** | |
| **Phi3** | 213.0 | **252.4** | 78.2 | 247.5 | |

Table 10.6: The cumulative distance scores for each program successfully repaired by each LLM across various configurations considering variable mappings. Entries highlighted in bold correspond to the highest score for each LLM.

both programs' ASTs (see Chapter 7). These GNNs [173] were trained on the first lab class of C-Pack-IPAs (see Chapter 4), with MultIPAs [174] augmenting C-Pack-IPAs by generating pairs of buggy/correct programs (see Chapter 6).

Table 10.5 shows the results for LLMs when provided with a correct implementation for the same IPA and a variable mapping between the buggy and correct implementation. The correct implementation can either be the lecturer's solution for the same IPA, the closest correct program based on the AASTs (CPA), or the closest correct program based on the AASTs and their invariants (CPIA). As indicated in Table 10.5, all LLMs, except for Granite, are able to repair more programs when they have access to variable mappings between the buggy and correct programs. For instance, CodeGemma, using the prompt configuration Sk_De-TS-CE-CPA without variable mappings, fixes 725 programs, while the same configuration plus variable mappings fixes 782 programs, representing an improvement of nearly 4%.

Table 10.6 presents the sum of the distance scores (see Equation 10.1) for the LLMs from Table 10.5 across different prompt configurations. Notably, only Llama3's score improves compared to Table 10.4, increasing from a distance score of 423 to 432.1 in Table 10.6. This suggests that while access to variable mappings aids LLMs in repairing more programs, it does not significantly enhance the LLMs' distance score in 80% of the evaluated models.

### 10.4.4 Discussion

To answer our research questions: For RQ1, all six LLMs using different prompt configurations repair more programs than traditional repair tools. For RQ2, prompt configurations with FL-based Sketches, IPA description and test suite yield the most successful repair outcomes. Moreover, for RQ3, it is clear that incorporating FL-based Sketches (or even FIXME annotations) allows the LLMs to repair more programs than only providing the buggy program. For RQ4, including a reference implementation allows for more repaired programs but with potentially less efficient fixes. For RQ5, incorporating a variable mapping alongside the reference implementation enables even more programs to be repaired, though it may similarly lead to less efficient fixes. Finally, for RQ6, employing a Counterexample guided approach significantly improves the accuracy of LLM-driven APR across various configurations.

## 10.5 Conclusion

Large Language Models (LLMs) excel at completing strings, while MaxSAT-based fault localization (FL) excels at identifying buggy parts of a program. We proposed a novel approach combining MaxSAT-based FL and LLMs via zero-shot learning to enhance Automated Program Repair (APR) for introductory programming assignments (IPAs). Experiments show that our bug-free program sketches, significantly improves the repair capabilities of all six evaluated LLMs, enabling them to repair more programs and produce smaller patches compared to other configurations and state-of-the-art symbolic program repair tools. Therefore, this interaction between Formal Methods and LLMs yields more accurate and efficient program fixes, enhancing feedback mechanisms in programming education.

# 11

# Conclusion

*“This is what happens when a project ends reasonably well: once you understand the main conclusion, it seems it was always obvious.”*

– Daniel Kahneman, Thinking Fast and Slow [244].

Automated Program Repair (APR) for introductory programming assignments (IPAs) is driven by the growing number of students enrolling in programming courses each year. Providing personalized feedback on these assignments requires significant time and effort from faculty, making automated feedback essential. One common approach is to suggest potential repairs to students' incorrect programs.

Given the large volume of student enrollments, courses can gather numerous correct implementations for the IPAs. When a student submits a faulty program, these correct submissions can be leveraged to automatically propose repairs, offering valuable guidance and improving the learning experience. However, in order to repair a faulty student's submission current state-of-the-art program repair frameworks require the existence of correct implementations for the introductory programming assignment (IPA) with the same control flow graph (CFG).

To address these limitations, we propose MENTOR, a clustering-based program repair tool designed to provide Auto**m**ated F**e**edback for I**n**troduc**t**ory Pr**o**gramming Exe**r**cises. MENTOR was designed to overcome the current state-of-the-art program repair tools' drawbacks listed in Chapter 3. MENTOR's four primary objectives are: (1) to advance the current state of program clustering and program repair; (2) to enhance MaxSAT-based fault localization techniques; (3) to enable the repair of buggy programs using correct implementations without the necessity for matching control-flow graphs; and (4) to deliver automated, personalized, and sound feedback to students.

MENTOR operates by accepting an incorrect submission for a specific IPA, a test suite, and a set of $N$ correct submissions for that same assignment. The tool is structured into several modules: (1) program clustering, (2) variable alignment, (3) fault localization, and (4) program fixing. All modules of MENTOR have been evaluated using C-Pack-IPAs [171], our pack of IPAs detailed in Chapter 4.

Chapter 5 introduced InvAASTCluster [42], MENTOR's novel clustering approach that leverages dynamically generated program invariants and anonymized abstract syntax trees (AASTs) to accurately cluster semantically equivalent programs. Experimental results show that InvAASTCluster significantly outperforms syntax-based clustering techniques and boosts the repair rate of Clara, a state-of-the-art clustering-based program repair tool, by approximately 13%. Moreover, InvAASTCluster also reduces the time required by Clara to repair incorrect student submissions, highlighting its efficiency and positive impact on APR performance.

Next, MENTOR's variable aligner module is presented in Chapter 7. This novel approach for variable mapping utilizes graph neural networks (GNNs) to leverage programs' abstract syntax trees (ASTs) in mapping variable sets between two programs [173]. Experiments show the effectiveness of these mappings across various program repair scenarios, successfully mapping 83% of 4,166 evaluated program pairs generated by our program transformation tool, MultIPAs [174], described in Chapter 6. Furthermore, incorporating variable mappings significantly enhances repair performance, allowing our approach to repair 88.5% of the evaluation benchmark, compared to 72% achieved by structure-based methods.

Regarding MENTOR's novel fault localization technique, Chapter 8 proposed CFaults [16], a cutting-edge fault localization technique for C programs with multiple faults that employs Model-Based Diagnosis (MBD) and a unified MaxSAT formula to ensure consistency across failing test cases. Experimental results on TCAS and C-Pack-IPAs demonstrate that CFaults outperforms state-of-the-art formula-

based fault localization (FBFL) tools such as BUGASSIST and SNIPER in terms of both speed and diagnosis quality, generating only subset-minimal diagnoses without redundant results.

Additionally, Chapter 9 introduced GITSEED [223], a language-agnostic automated assessment tool integrated with GITLAB to support Programming Education and Software Engineering (SE). GITSEED allows students to learn `git` fundamentals while receiving personalized feedback on their code submissions. Notably, CFAULTS was successfully integrated into GITSEED to pinpoint faults in students' programs, and the feedback from students regarding its usage has been positive. Our evaluation indicates that GITSEED enhances student engagement and learning outcomes, enabling instructors to customize the tool's pipeline for course-specific feedback.

Lastly, Chapter 10 presents MENTOR's program fixer module, a novel approach that combines the strengths of Formal Methods (FM)-based fault localization and Large Language Models (LLMS) using zero-shot learning to enhance APR for IPAS. This method leverages MaxSAT-based fault localization through CFAULTS to identify buggy code segments and generate a program sketch that excludes these faulty parts. Using a Counterexample Guided Inductive Synthesis (CEGIS) loop, MENTOR iteratively refines the program by prompting an LLM to synthesize the missing segments, which are validated against a test suite. Moreover, this chapter incorporates INVAASTCLUSTER's closest correct program and our variable mappings into the repair process to further guide the LLMS.

Experimental results on C-PACK-IPAS show that MENTOR's hybrid repair method, which combines FM-based fault localization and LLMS, significantly improves repair success rates and results in smaller, more precise fixes, outperforming other repair strategies and state-of-the-art symbolic tools. For example, VERIFIX can only repair 6.3% of C-PACK-IPAS, while CLARA repairs 34.6%. In contrast, MENTOR, depending on the prompt configuration and the LLM used, achieves repair rates ranging from 37.3% to 64.4% on C-PACK-IPAS.

As future directions, we propose integrating MENTOR into the evaluation pipeline of introductory programming courses to enable students to benefit from its capabilities, particularly in receiving hints derived from the program statements that MENTOR has repaired. Additionally, it would be worthwhile to explore the use of Large Language Models (LLMS) to generate plain-text summaries explaining why students' programs were incorrect. This enhancement would provide students with a more detailed understanding of their programming faults, fostering a deeper learning experience.

*“The saddest aspect of life right now is that science gathers knowledge faster than society gathers wisdom.”*

– Isaac Asimov.

# Bibliography


[1] D. R. Hofstadter. *Godel, Escher, Bach: An Eternal Golden Braid*. Basic Books, Inc., USA, 1979. ISBN 0465026850.

[2] S. Gulwani, I. Radicek, and F. Zuleger. Automated clustering and program repair for introductory programming assignments. In *PLDI 2018*, pages 465–480. ACM, 2018.

[3] M. Yasunaga and P. Liang. Graph-based, self-supervised program repair from diagnostic feedback. In *ICML 2020*, volume 119 of *Proceedings of Machine Learning Research*, pages 10799–10808. PMLR, 2020.

[4] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: program repair via semantic analysis. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *35th International Conference on Software Engineering, ICSE '13*, pages 772–781. IEEE Computer Society, 2013.

[5] S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In A. Bertolino, G. Canfora, and S. G. Elbaum, editors, *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015*, pages 448–458. IEEE Computer Society, 2015.

[6] Y. Ke, K. T. Stolee, C. L. Goues, and Y. Brun. Repairing programs with semantic code search (T). In M. B. Cohen, L. Grunske, and M. Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 295–306. IEEE Computer Society, 2015.

[7] S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: scalable multiline program patch synthesis via symbolic analysis. In L. K. Dillon, W. Visser, and L. A. Williams, editors, *ICSE 2016*, pages 691–701. ACM, 2016.

[8] 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 E. Bodden, W. Schäfer, A. van Deursen, and A. Zisman, editors, *ESEC/FSE 2017*, pages 740–751. ACM, 2017.

[9] U. Z. Ahmed, Z. Fan, J. Yi, O. I. Al-Bataineh, and A. Roychoudhury. Verifix: Verified repair of programming assignments. *ACM Trans. Softw. Eng. Methodol.*, jan 2022. ISSN 1049-331X. doi: 10.1145/3510418. URL `https://doi.org/10.1145/3510418`.

[10] K. Wang, R. Singh, and Z. Su. Search, align, and repair: data-driven feedback generation for introductory programming exercises. In *PLDI 2018*, pages 481–495. ACM, 2018.

[11] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury. Re-factoring based program repair applied to programming assignments. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 388–398. IEEE, 2019.

[12] 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 S. Uchitel, A. Orso, and M. P. Robillard, editors, *ICSE 2017*, pages 404–415. IEEE / ACM, 2017.

[13] R. Gupta, S. Pal, A. Kanade, and S. K. Shevade. Deepfix: Fixing common C language errors by deep learning. In S. P. Singh and S. Markovitch, editors, *AAAI 2017*, pages 1345–1351. AAAI Press, 2017.

[14] A. Mesbah, A. Rice, E. Johnston, N. Glorioso, and E. Aftandilian. Deepdelta: learning to repair compilation errors. In *ESEC/SIGSOFT FSE 2019*, pages 925–936. ACM, 2019.

[15] R. Gupta, A. Kanade, and S. K. Shevade. Deep reinforcement learning for syntactic error repair in student programs. In *The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019*, pages 930–937. AAAI Press, 2019.

[16] P. Orvalho, M. Janota, and V. Manquinho. CFaults: Model-Based Diagnosis for Fault Localization in C Programs with Multiple Test Cases. In *Formal Methods - 26th International Symposium, FM 2024, Milan, Italy, 2024, Proceedings*, volume 14933 of *Lecture Notes in Computer Science*, pages 463–481, Cham, 2024. Springer Nature Switzerland. ISBN 978-3-031-71162-6. doi: 10.1007/978-3-031-71162-6_24.

[17] M. H. Liffiton, B. E. Sheese, J. Savelka, and P. Denny. Codehelp: Using large language models with guardrails for scalable support in programming classes. In A. Mühling and I. Jormanainen, editors, *Proceedings of the 23rd Koli Calling International Conference on Computing Education Research, Koli Calling 2023, Koli, Finland, November 13-18, 2023*, pages 8:1–8:11. ACM, 2023. doi: 10.1145/3631802.3631830. URL https://doi.org/10.1145/3631802.3631830.

[18] S. Bhatia, P. Kohli, and R. Singh. Neuro-symbolic program corrector for introductory programming assignments. In *ICSE 2018*, pages 60–70. ACM, 2018.

[19] Y. Pu, K. Narasimhan, A. Solar-Lezama, and R. Barzilay. sk_p: a neural program corrector for moocs. In E. Visser, editor, *Companion Proceedings of the 2016 ACM SIGPLAN International Conference on Systems, Programming, Languages and Applications: Software for Humanity, SPLASH 2016*, pages 39–40. ACM, 2016.

[20] J. Zhang, J. P. Cambronero, S. Gulwani, V. Le, R. Piskac, G. Soares, and G. Verbruggen. Pydex: Repairing bugs in introductory python assignments using llms. *Proc. ACM Program. Lang.*, 8(OOPSLA1):1100–1124, 2024. doi: 10.1145/3649850. URL https://doi.org/10.1145/3649850.

[21] H. Joshi, J. P. C. Sánchez, S. Gulwani, V. Le, G. Verbruggen, and I. Radicek. Repair is nearly generation: Multilingual program repair with llms. In B. Williams, Y. Chen, and J. Neville, editors, *Thirty-Seventh AAAI Conference on Artificial Intelligence, AAAI 2023, Thirty-Fifth Conference on Innovative Applications of Artificial Intelligence, IAAI 2023, Thirteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2023, Washington, DC, USA, February 7-14, 2023*, pages 5131–5140. AAAI Press, 2023. doi: 10.1609/AAAI.V37I4.25642. URL https://doi.org/10.1609/aaai.v37i4.25642.

[22] 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. In M. Feng, T. Käser, P. P. Talukdar, R. Agrawal, Y. Narahari, and M. Pechenizkiy, editors, *Proceedings of the 16th International Conference on Educational Data Mining, EDM 2023, Bengaluru, India, July 11-14, 2023*. International Educational Data Mining Society, 2023. URL https://educationaldatamining.org/2023.EDM-short-papers.37/index.html.

[23] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, and A. Svyatkovskiy. Inferfix: End-to-end program repair with llms. In S. Chandra, K. Blincoe, and P. Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 1646–1656. ACM, 2023. doi: 10.1145/3611643.3613892. URL https://doi.org/10.1145/3611643.3613892.

[24] M. Monperrus. Automatic software repair: A bibliography. *ACM Comput. Surv.*, 51(1):17:1–17:24, 2018.

[25] L. Gazzola, D. Micucci, and L. Mariani. Automatic software repair: A survey. *IEEE Trans. Software Eng.*, 45(1):34–67, 2019.

[26] C. L. Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Commun. ACM*, 62 (12):56–65, 2019.

[27] R. Bavishi, H. Yoshida, and M. R. Prasad. Phoenix: automated data-driven synthesis of repairs for static analysis violations. In M. Dumas, D. Pfahl, S. Apel, and A. Russo, editors, *ESEC/SIGSOFT FSE 2019*, pages 613–624. ACM, 2019.

[28] H. Yoshida, R. Bavishi, K. Hotta, Y. Nemoto, M. R. Prasad, and S. Kikuchi. Phoenix: a tool for automated data-driven synthesis of repairs for static analysis violations. In G. Rothermel and D. Bae, editors, *ICSE 2020*, pages 53–56. ACM, 2020.

[29] D. Ramos, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues. Batfix: Repairing language model-based transpilation. *ACM Trans. Softw. Eng. Methodol.*, 33(6):161, 2024. doi: 10.1145/3658668. URL https://doi.org/10.1145/3658668.

[30] A. Ketkar, D. Ramos, L. Clapp, R. Barik, and M. K. Ramanathan. A lightweight polyglot code transformation language. *Proc. ACM Program. Lang.*, 8(PLDI):1288–1312, 2024. doi: 10.1145/3656429. URL https://doi.org/10.1145/3656429.

[31] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[32] C. L. Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *IEEE Trans. Software Eng.*, 38(1):54–72, 2012.

[33] W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In E. Denney, T. Bultan, and A. Zeller, editors, *ASE 2013*, pages 356–366. IEEE, 2013.

[34] D. M. Perry, D. Kim, R. Samanta, and X. Zhang. Semcluster: clustering of imperative programming assignments based on quantitative semantic features. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 860–873. ACM, 2019.

[35] R. Milner. An algebraic definition of simulation between programs. In D. C. Cooper, editor, *Proceedings of the 2nd International Joint Conference on Artificial Intelligence. London, UK, September 1-3, 1971*, pages 481–489. William Kaufmann, 1971.

[36] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[37] B. R. Churchill, O. Padon, R. Sharma, and A. Aiken. Semantic program alignment for equivalence checking. In K. S. McKinley and K. Fisher, editors, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 1027–1040. ACM, 2019.

[38] C. S. Xia, Y. Ding, and L. Zhang. The plastic surgery hypothesis in the era of large language models. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 522–534. IEEE, 2023. doi: 10.1109/ASE56229.2023.00047. URL https://doi.org/10.1109/ASE56229.2023.00047.

[39] Y. Wei, C. S. Xia, and L. Zhang. Copiloting the copilots: Fusing large language models with completion engines for automated program repair. In S. Chandra, K. Blincoe, and P. Tonella, editors, *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pages 172–184. ACM, 2023. doi: 10.1145/3611643.3616271. URL https://doi.org/10.1145/3611643.3616271.

[40] Z. Fan, X. Gao, M. Mirchev, A. Roychoudhury, and S. H. Tan. Automated repair of programs from large language models. In *45th IEEE/ACM International Conference on Software Engineering,*

*ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1469–1481. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00128. URL https://doi.org/10.1109/ICSE48619.2023.00128.

[41] C. S. Xia, Y. Wei, and L. Zhang. Automated program repair in the era of large pre-trained language models. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 1482–1494. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00129. URL https://doi.org/10.1109/ICSE48619.2023.00129.

[42] P. Orvalho, M. Janota, and V. Manquinho. InvAASTCluster: On Applying Invariant-Based Program Clustering to Introductory Programming Assignments. *CoRR*, abs/2206.14175, 2022. doi: 10.48550/ARXIV.2206.14175. URL https://doi.org/10.48550/arXiv.2206.14175.

[43] P. da Silva. SQUARES : A SQL Synthesizer Using Query Reverse Engineering. Master's thesis, Instituto Superior Técnico, Lisboa, Portugal, 2019.

[44] A. Biere, M. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[45] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. Encodings for enumeration-based program synthesis. In *Principles and Practice of Constraint Programming - 25th International Conference, CP 2019, Stamford, CT, USA, September 30 - October 4, 2019, Proceedings*, pages 583–599, 2019.

[46] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition*. Pearson international edition. Addison-Wesley, 2007.

[47] Y. Feng, R. Martins, O. Bastani, and I. Dillig. Program synthesis using conflict-driven learning. In *PLDI*, pages 420–435, 2018.

[48] F. E. Allen. Control flow analysis. In R. S. Northcote, editor, *Proceedings of a Symposium on Compiler Optimization, Urbana-Champaign, Illinois, USA, July 27-28, 1970*, pages 1–19. ACM, 1970.

[49] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158, 1971.

[50] R. Martins, S. Joshi, V. Manquinho, and I. Lynce. Incremental cardinality constraints for maxsat. In *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, pages 531–548, 2014.

[51] P. Orvalho, V. Manquinho, and R. Martins. UpMax: User Partitioning for MaxSAT. In M. Mahajan and F. Slivovsky, editors, *26th International Conference on Theory and Applications of Satisfiability Testing, SAT 2023, July 4-8, 2023, Alghero, Italy*, volume 271 of *LIPIcs*, pages 19:1–19:13. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2023. doi: 10.4230/LIPICS.SAT.2023.19. URL https://doi.org/10.4230/LIPIcs.SAT.2023.19.

[52] C. Zhang, R. Wagner, P. Orvalho, D. Garlan, V. Manquinho, R. Martins, and E. Kang. Alloy-max: Bringing maximum satisfaction to relational specifications. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2021*., pages –, 2021.

[53] M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *J. Autom. Reason.*, 40(1):1–33, 2008. doi: 10.1007/S10817-007-9084-Z. URL https://doi.org/10.1007/s10817-007-9084-z.

[54] A. Ignatiev, A. Morgado, G. Weissenbacher, and J. Marques-Silva. Model-based diagnosis with multiple observations. In S. Kraus, editor, *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI 2019, Macao, China, August 10-16, 2019*, pages 1108–1115. ijcai.org, 2019. doi: 10.24963/IJCAI.2019/155. URL https://doi.org/10.24963/ijcai.2019/155.

[55] M. A. Arbib, A. J. Kfoury, and R. N. Moll. *A Basis for Theoretical Computer Science*. Texts and Monographs in Computer Science. Springer, 1981. ISBN 978-0-387-90573-0. doi: 10.1007/978-1-4613-9455-6. URL https://doi.org/10.1007/978-1-4613-9455-6.

[56] Z. S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[57] E. M. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In K. Jensen and A. Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004. doi: 10.1007/978-3-540-24730-2_15. URL https://doi.org/10.1007/978-3-540-24730-2_15.

[58] E. M. Clarke, D. Kroening, and K. Yorav. Behavioral consistency of C and verilog programs using bounded model checking. In *Proceedings of the 40th Design Automation Conference, DAC 2003, Anaheim, CA, USA, June 2-6, 2003*, pages 368–371. ACM, 2003. doi: 10.1145/775832.775928. URL https://doi.org/10.1145/775832.775928.

[59] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao. The daikon system for dynamic detection of likely invariants. *Sci. Comput. Program.*, 69(1-3):35–45, 2007. URL https://doi.org/10.1016/j.scico.2007.01.015.

[60] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley series in computer science / World student series edition. Addison-Wesley, 1986.

[61] R. Reiter. A theory of diagnosis from first principles. *Artif. Intell.*, 32(1):57–95, 1987. doi: 10.1016/0004-3702(87)90062-2. URL https://doi.org/10.1016/0004-3702(87)90062-2.

[62] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 437–446. ACM, 2011.

[63] J. Marques-Silva, M. Janota, A. Ignatiev, and A. Morgado. Efficient model based diagnosis with maximum satisfiability. In Q. Yang and M. J. Wooldridge, editors, *Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015*, pages 1966–1972. AAAI Press, 2015. URL http://ijcai.org/Abstract/15/279.

[64] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst.*, 9(3):319–349, 1987. doi: 10.1145/24039.24041. URL https://doi.org/10.1145/24039.24041.

[65] Y. Feng, R. Martins, J. V. Geffen, I. Dillig, and S. Chaudhuri. Component-based synthesis of table consolidation and transformation tasks from examples. In *PLDI*, pages 422–436, 2017.

[66] A. S. Lezama. *Program synthesis by sketching*. PhD thesis, UC Berkeley, 2008.

[67] Z. Manna and R. J. Waldinger. Toward automatic program synthesis. *Commun. ACM*, 14(3): 151–165, 1971.

[68] A. Abate, C. David, P. Kesseli, D. Kroening, and E. Polgreen. Counterexample guided inductive synthesis modulo theories. In H. Chockler and G. Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 270–288. Springer, 2018. doi: 10.1007/978-3-319-96145-3_15. URL https://doi.org/10.1007/978-3-319-96145-3_15.

[69] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May 2010*, pages 215–224, 2010.

[70] A. Solar-Lezama, R. M. Rabbah, R. Bodík, and K. Ebcioglu. Programming by sketching for bit-streaming programs. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 281–294. ACM, 2005. doi: 10.1145/1065010.1065045. URL https://doi.org/10.1145/1065010.1065045.

[71] D. Ramos, J. Pereira, I. Lynce, V. Manquinho, and R. Martins. UNCHARTIT: an interactive framework for program recovery from charts. In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 175–186. IEEE, 2020. doi: 10.1145/3324884.3416613. URL https://doi.org/10.1145/3324884.3416613.

[72] A. Ni, D. Ramos, A. Z. H. Yang, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues. SOAR: A synthesis approach for data science API refactoring. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pages 112–124. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00023. URL https://doi.org/10.1109/ICSE43902.2021.00023.

[73] D. Ramos, H. Mitchell, I. Lynce, V. Manquinho, R. Martins, and C. Le Goues. MELT: mining effective lightweight transformations from pull requests. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 1516–1528. IEEE, 2023. doi: 10.1109/ASE56229.2023.00117. URL https://doi.org/10.1109/ASE56229.2023.00117.

[74] P. Orvalho, M. Terra-Neves, M. Ventura, R. Martins, and V. M. Manquinho. SQUARES : A SQL synthesizer using query reverse engineering. *Proc. VLDB Endow.*, 13(12):2853–2856, 2020. URL http://www.vldb.org/pvldb/vol13/p2853-orvalho.pdf.

[75] R. Brancas, M. Terra-Neves, M. Ventura, V. Manquinho, and R. Martins. Towards reliable SQL synthesis: Fuzzing-based evaluation and disambiguation. In D. Beyer and A. Cavalcanti, editors, *Fundamental Approaches to Software Engineering - 27th International Conference, FASE 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings*, volume 14573 of *Lecture Notes in Computer Science*, pages 232–254. Springer, 2024. doi: 10.1007/978-3-031-57259-3_11. URL https://doi.org/10.1007/978-3-031-57259-3_11.

[76] M. Ferreira, M. Terra-Neves, M. Ventura, I. Lynce, and R. Martins. FOREST: an interactive multi-tree synthesizer for regular expressions. In J. F. Groote and K. G. Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 27th International Conference, TACAS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings, Part I*, volume 12651 of *Lecture Notes in Computer Science*, pages 152–169. Springer, 2021. doi: 10.1007/978-3-030-72016-2_9. URL https://doi.org/10.1007/978-3-030-72016-2_9.

[77] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Software Eng.*, 2(3):215–222, 1976.

[78] L. A. Clarke and D. J. Richardson. Applications of symbolic evaluation. *J. Syst. Softw.*, 5(1):15–35, 1985.

[79] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In R. Draves and R. van Renesse, editors, *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*, pages 209–224. USENIX Association, 2008.

[80] S. Person, G. Yang, N. Rungta, and S. Khurshid. Directed incremental symbolic execution. In M. W. Hall and D. A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 504–515. ACM, 2011.

[81] C. S. Pasareanu and N. Rungta. Symbolic pathfinder: symbolic execution of java bytecode. In C. Pecheur, J. Andrews, and E. D. Nitto, editors, *ASE 2010, 25th IEEE/ACM International Conference on Automated Software Engineering, Antwerp, Belgium, September 20-24, 2010*, pages 179–180. ACM, 2010. doi: 10.1145/1858996.1859035. URL `https://doi.org/10.1145/1858996.1859035`.

[82] J. F. Santos, P. Maksimovic, T. Grohens, J. Dolby, and P. Gardner. Symbolic execution for javascript. In D. Sabel and P. Thiemann, editors, *Proceedings of the 20th International Symposium on Principles and Practice of Declarative Programming, PPDP 2018, Frankfurt am Main, Germany, September 03-05, 2018*, pages 11:1–11:14. ACM, 2018.

[83] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and java. In M. G. Bobaru, K. Havelund, G. J. Holzmann, and R. Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 41–55. Springer, 2011.

[84] Y. Kim, M. Kim, Y. J. Kim, and Y. Jang. Industrial application of concolic testing approach: A case study on libexif by using CREST-BV and KLEE. In M. Glinz, G. C. Murphy, and M. Pezzè, editors, *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*, pages 1143–1152. IEEE Computer Society, 2012.

[85] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), 15-19 September 2008, L'Aquila, Italy*, pages 443–446. IEEE Computer Society, 2008.

[86] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. In M. Wermelinger and H. C. Gall, editors, *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2005, Lisbon, Portugal, September 5-9, 2005*, 2005.

[87] K. Sen and G. Agha. CUTE and jcute: Concolic unit testing and explicit path model-checking tools. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006.

[88] P. Godefroid, N. Klarlund, and K. Sen. DART: directed automated random testing. In V. Sarkar and M. W. Hall, editors, *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation, Chicago, IL, USA, June 12-15, 2005*, pages 213–223. ACM, 2005.

[89] P. Godefroid. Random testing for security: blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd international workshop on Random testing: co-located with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007)*, pages 1–1, 2007.

[90] E. O. Soremekun, L. Kirschner, M. Böhme, and A. Zeller. Locating faults with program slicing: an empirical analysis. *Empir. Softw. Eng.*, 26(3):51, 2021. doi: 10.1007/S10664-020-09931-7. URL https://doi.org/10.1007/s10664-020-09931-7.

[91] B. Rothenberg and O. Grumberg. Must fault localization for program repair. In S. K. Lahiri and C. Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part II*, volume 12225 of *Lecture Notes in Computer Science*, pages 658–680. Springer, 2020. doi: 10.1007/978-3-030-53291-8_33. URL https://doi.org/10.1007/978-3-030-53291-8_33.

[92] A. Zeller. Yesterday, my program worked. today, it does not. why? In O. Nierstrasz and M. Lemoine, editors, *Software Engineering - ESEC/FSE'99, 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999, Proceedings*, volume 1687 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 1999.

[93] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund. Spectrum-based multiple fault localization. In *ASE 2009, 24th IEEE/ACM International Conference on Automated Software Engineering, Auckland, New Zealand, November 16-20, 2009*, pages 88–99. IEEE Computer Society, 2009. doi: 10.1109/ASE.2009.25. URL https://doi.org/10.1109/ASE.2009.25.

[94] W. E. Wong, V. Debroy, and B. Choi. A family of code coverage-based heuristics for effective fault localization. *J. Syst. Softw.*, 83(2):188–208, 2010. doi: 10.1016/J.JSS.2009.09.037. URL https://doi.org/10.1016/j.jss.2009.09.037.

[95] L. Naish, H. J. Lee, and K. Ramamohanarao. A model for spectra-based software diagnosis. *ACM Trans. Softw. Eng. Methodol.*, 20(3):11:1–11:32, 2011. doi: 10.1145/2000791.2000795. URL https://doi.org/10.1145/2000791.2000795.

[96] W. E. Wong, V. Debroy, R. Gao, and Y. Li. The dstar method for effective software fault localization. *IEEE Trans. Reliab.*, 63(1):290–308, 2014. doi: 10.1109/TR.2013.2285319. URL https://doi.org/10.1109/TR.2013.2285319.

[97] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Trans. Software Eng.*, 42(8):707–740, 2016. doi: 10.1109/TSE.2016.2521368. URL https://doi.org/10.1109/TSE.2016.2521368.

[98] R. Abreu, P. Zoeteweij, R. Golsteijn, and A. J. C. van Gemund. A practical evaluation of spectrum-based fault localization. *J. Syst. Softw.*, 82(11):1780–1792, 2009. doi: 10.1016/J.JSS.2009.06.035. URL https://doi.org/10.1016/j.jss.2009.06.035.

[99] K. Liu, A. Koyuncu, T. F. Bissyandé, D. Kim, J. Klein, and Y. Le Traon. You cannot fix what you cannot find! an investigation of fault localization bias in benchmarking automated program repair systems. In *2019 12th IEEE conference on software testing, validation and verification (ICST)*, pages 102–113. IEEE, 2019.

[100] M. Jose and R. Majumdar. Bug-assist: Assisting fault localization in ANSI-C programs. In G. Gopalakrishnan and S. Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 504–509. Springer, 2011. doi: 10.1007/978-3-642-22110-1_40. URL https://doi.org/10.1007/978-3-642-22110-1_40.

[101] S. Lamraoui and S. Nakajima. A formula-based approach for automatic fault localization of multi-fault programs. *J. Inf. Process.*, 24(1):88–98, 2016. doi: 10.2197/IPSJJIP.24.88. URL https://doi.org/10.2197/ipsjjip.24.88.

[102] J. K. Feser, S. Chaudhuri, and I. Dillig. Synthesizing data structure transformations from input-output examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 229–239, 2015.

[103] S. Lamraoui and S. Nakajima. A formula-based approach for automatic fault localization of imperative programs. In S. Merz and J. Pang, editors, *Formal Methods and Software Engineering - 16th International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*, volume 8829 of *Lecture Notes in Computer Science*, pages 251–266. Springer, 2014. doi: 10.1007/978-3-319-11737-9_17. URL https://doi.org/10.1007/978-3-319-11737-9_17.

[104] A. Griesmayer, S. Staber, and R. Bloem. Automated fault localization for C programs. In R. Bloem, M. Roveri, and F. Somenzi, editors, *Proceedings of the Workshop on Verification and Debugging, V&D@FLoC 2006, Seattle, WA, USA, August 21, 2006*, volume 174 of *Electronic Notes in Theoretical Computer Science*, pages 95–111. Elsevier, 2006. doi: 10.1016/J.ENTCS.2006.12.032. URL https://doi.org/10.1016/j.entcs.2006.12.032.

[105] F. Wotawa, M. Nica, and I. Moraru. Automated debugging based on a constraint model of the program and a test case. *J. Log. Algebraic Methods Program.*, 81(4):390–407, 2012. doi: 10.1016/J.JLAP.2012.03.002. URL https://doi.org/10.1016/j.jlap.2012.03.002.

[106] Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In J. Palsberg and M. Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 351–363. ACM, 2005. doi: 10.1145/1040305.1040334. URL https://doi.org/10.1145/1040305.1040334.

[107] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In P. Bjesse and A. Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 91–100. FMCAD Inc., 2011. URL http://dl.acm.org/citation.cfm?id=2157671.

[108] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. SATABS: sat-based predicate abstraction for ANSI-C. In N. Halbwachs and L. D. Zuck, editors, *Tools and Algorithms for the Con-*

*struction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Edinburgh, UK, April 4-8, 2005, Proceedings*, volume 3440 of *Lecture Notes in Computer Science*, pages 570–574. Springer, 2005. doi: 10.1007/978-3-540-31980-1_40. URL https://doi.org/10.1007/978-3-540-31980-1_40.

[109] E. M. Clarke, D. Kroening, N. Sharygina, and K. Yorav. Predicate abstraction of ANSI-C programs using SAT. *Formal Methods Syst. Des.*, 25(2-3):105–127, 2004. doi: 10.1023/B:FORM.0000040025.89719.F3. URL https://doi.org/10.1023/B:FORM.0000040025.89719.f3.

[110] E. M. Clarke, O. Grumberg, D. Kroening, D. A. Peled, and H. Veith. *Model checking, 2nd Edition*. MIT Press, 2018. ISBN 978-0-262-03883-6. URL https://mitpress.mit.edu/books/model-checking-second-edition.

[111] A. Metodi, R. Stern, M. Kalech, and M. Codish. A novel sat-based approach to model based diagnosis. *J. Artif. Intell. Res.*, 51:377–411, 2014. doi: 10.1613/JAIR.4503. URL https://doi.org/10.1613/jair.4503.

[112] S. Safarpour, H. Mangassarian, A. G. Veneris, M. H. Liffiton, and K. A. Sakallah. Improved design debugging using maximum satisfiability. In *Formal Methods in Computer-Aided Design, 7th International Conference, FMCAD 2007, Austin, Texas, USA, November 11-14, 2007, Proceedings*, pages 13–19. IEEE Computer Society, 2007. doi: 10.1109/FAMCAD.2007.26. URL https://doi.org/10.1109/FAMCAD.2007.26.

[113] A. Adam and J. H. Laurent. Laura, A system to debug student programs. *Artif. Intell.*, 15(1-2): 75–122, 1980.

[114] C. W. Johnson and C. Runciman. Semantic errors - diagnosis and repair. In J. R. White and F. E. Allen, editors, *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction, Boston, Massachusetts, USA, June 23-25, 1982*, pages 88–97. ACM, 1982.

[115] S. Parihar, Z. Dadachanji, P. K. Singh, R. Das, A. Karkare, and A. Bhattacharya. Automatic grading and feedback using program repair for introductory programming courses. In R. Davoli, M. Goldweber, G. Rößling, and I. Polycarpou, editors, *ITiCSE 2017*, pages 92–97. ACM, 2017.

[116] U. Z. Ahmed, P. Kumar, A. Karkare, P. Kar, and S. Gulwani. Compilation error repair: for the student programs, from the student programs. In *ICSE (SEET) 2018*, pages 78–87. ACM, 2018.

[117] U. Z. Ahmed, R. Sindhgatta, N. Srivastava, and A. Karkare. Targeted example generation for compilation errors. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 327–338. IEEE, 2019. doi: 10.1109/ASE.2019.00039. URL https://doi.org/10.1109/ASE.2019.00039.

[118] C. Lattner and V. S. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization (CGO*

*2004), 20-24 March 2004, San Jose, CA, USA*, pages 75–88. IEEE Computer Society, 2004. doi: 10.1109/CGO.2004.1281665. URL https://doi.org/10.1109/CGO.2004.1281665.

[119] Z. Li, F. Sun, H. Wang, Y. Ding, Y. Liu, and X. Chen. CLACER: A deep learning-based compilation error classification method for novice students' programs. In *IEEE 45th Annual Computers, Software, and Applications Conference, COMPSAC 2021, Madrid, Spain, July 12-16, 2021*, pages 74–83. IEEE, 2021. doi: 10.1109/COMPSAC51774.2021.00022. URL https://doi.org/10.1109/COMPSAC51774.2021.00022.

[120] Y. Lu, N. Meng, and W. Li. FAPR: fast and accurate program repair for introductory programming courses. *CoRR*, abs/2107.06550, 2021.

[121] X. Liu, S. Wang, P. Wang, and D. Wu. Automatic grading of programming assignments: an approach based on formal semantics. In S. Beecham and D. E. Damian, editors, *Proceedings of the 41st International Conference on Software Engineering: Software Engineering Education and Training, ICSE (SEET) 2019*, pages 126–137. IEEE / ACM, 2019.

[122] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. L. Goues. Sosrepair: Expressive semantic search for real-world program repair. *IEEE Trans. Software Eng.*, 47(10):2162–2181, 2019. doi: 10.1109/TSE.2019.2944914.

[123] K. T. Stolee and S. G. Elbaum. Toward semantic search via SMT solver. In W. Tracz, M. P. Robillard, and T. Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 25. ACM, 2012. doi: 10.1145/2393596.2393625.

[124] K. T. Stolee, S. G. Elbaum, and M. B. Dwyer. Code search with input/output queries: Generalizing, ranking, and assessment. *J. Syst. Softw.*, 116:35–48, 2016. doi: 10.1016/j.jss.2015.04.081.

[125] K. T. Stolee, S. G. Elbaum, and D. Dobos. Solving the search for source code. *ACM Trans. Softw. Eng. Methodol.*, 23(3):26:1–26:45, 2014. doi: 10.1145/2581377. URL https://doi.org/10.1145/2581377.

[126] S. P. Reiss. Semantics-based code search. In *31st International Conference on Software Engineering, ICSE 2009, May 16-24, 2009, Vancouver, Canada, Proceedings*, pages 243–253. IEEE, 2009. doi: 10.1109/ICSE.2009.5070525.

[127] F. Long and M. Rinard. Automatic patch generation by learning correct code. In R. Bodík and R. Majumdar, editors, *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2016, St. Petersburg, FL, USA, January 20 - 22, 2016*, pages 298–312. ACM, 2016. doi: 10.1145/2837614.2837617.

[128] J. Gao, B. Pang, and S. S. Lumetta. Automated feedback framework for introductory programming courses. In A. Clear, E. Cuadros-Vargas, J. Carter, and Y. Túpac, editors, *ITiCSE 2016*, pages 53–58. ACM, 2016.

[129] S. Wang, M. Wen, B. Lin, H. Wu, Y. Qin, D. Zou, X. Mao, and H. Jin. Automated patch correctness assessment: How far are we? In *35th IEEE/ACM International Conference on Automated Software Engineering, ASE 2020, Melbourne, Australia, September 21-25, 2020*, pages 968–980. IEEE, 2020.

[130] A. Solar-Lezama, L. Tancau, R. Bodík, S. A. Seshia, and V. A. Saraswat. Combinatorial sketching for finite programs. In J. P. Shen and M. Martonosi, editors, *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 404–415, 2006.

[131] J. Clune, V. Ramamurthy, R. Martins, and U. A. Acar. Program equivalence for assisted grading of functional programs. *Proc. ACM Program. Lang.*, 4(OOPSLA):171:1–171:29, 2020. doi: 10.1145/3428239. URL https://doi.org/10.1145/3428239.

[132] E. L. Glassman, J. Scott, R. Singh, P. J. Guo, and R. C. Miller. Overcode: Visualizing variation in student solutions to programming problems at scale. *ACM Trans. Comput. Hum. Interact.*, 22(2):7:1–7:35, 2015.

[133] Y. Fu, J. Osei-Owusu, A. Astorga, Z. N. Zhao, W. Zhang, and T. Xie. Pacon: a symbolic analysis approach for tactic-oriented clustering of programming submissions. In C. Curtsinger and T. N. Nguyen, editors, *Proceedings of the 2021 ACM SIGPLAN International Symposium on SPLASH-E, Chicago, IL, USA. October 20, 2021*, pages 32–42. ACM, 2021.

[134] K. Tai. The tree-to-tree correction problem. *J. ACM*, 26(3):422–433, 1979. doi: 10.1145/322139.322143. URL https://doi.org/10.1145/322139.322143.

[135] M. R. Contractor and C. R. Rivero. Improving program matching toăautomatically repair introductory programs. In S. Crossley and E. Popescu, editors, *Intelligent Tutoring Systems*, pages 323–335, Cham, 2022. Springer International Publishing.

[136] M. T. A. Chowdhury, M. R. Contractor, and C. R. Rivero. Flexible control flow graph alignment for delivering data-driven feedback to novice programming learners. *J. Syst. Softw.*, 210:111960, 2024. doi: 10.1016/J.JSS.2024.111960. URL https://doi.org/10.1016/j.jss.2024.111960.

[137] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. DECKARD: scalable and accurate tree-based detection of code clones. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26, 2007*, pages 96–105. IEEE Computer Society, 2007.

[138] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra. Aroma: code recommendation via structural code search. *Proc. ACM Program. Lang.*, 3(OOPSLA):152:1–152:28, 2019.

[139] G. Mathew and K. T. Stolee. Cross-language code search using static and dynamic analyses. In D. Spinellis, G. Gousios, M. Chechik, and M. D. Penta, editors, *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*, pages 205–217. ACM, 2021.

[140] J. A. Jones, M. J. Harrold, and J. T. Stasko. Visualization of test information to assist fault localization. In W. Tracz, M. Young, and J. Magee, editors, *Proceedings of the 24th International Conference on Software Engineering, ICSE 2002, 19-25 May 2002, Orlando, Florida, USA*, pages 467–477. ACM, 2002.

[141] S. Mechtaev, M. Nguyen, Y. Noller, L. Grunske, and A. Roychoudhury. Semantic program repair using a reference implementation. In M. Chaudron, I. Crnkovic, M. Chechik, and M. Harman, editors, *Proceedings of the 40th International Conference on Software Engineering, ICSE 2018*, pages 129–139. ACM, 2018.

[142] J. C. Paiva, J. P. Leal, and Á. Figueira. Automated assessment in computer science education: A state-of-the-art review. *ACM Trans. Comput. Educ.*, 22(3):34:1–34:40, 2022. doi: 10.1145/3513140. URL https://doi.org/10.1145/3513140.

[143] T. Staubitz, H. Klement, R. Teusner, J. Renz, and C. Meinel. Codeocean - A versatile platform for practical programming excercises in online environments. In *2016 IEEE Global Engineering Education Conference, EDUCON 2016, Abu Dhabi, United Arab Emirates, April 10-13, 2016*, pages 314–323, "", 2016. IEEE. doi: 10.1109/EDUCON.2016.7474573. URL https://doi.org/10.1109/EDUCON.2016.7474573.

[144] J. P. Leal and F. M. A. Silva. Mooshak: a web-based multi-site programming contest system. *Softw. Pract. Exp.*, 33(6):567–581, 2003. doi: 10.1002/spe.522. URL https://doi.org/10.1002/spe.522.

[145] S. H. Edwards and M. A. Pérez-Quiñones. Web-cat: automatically grading programming assignments. In J. Amillo, C. Laxer, E. M. Ruiz, and A. Young, editors, *Proceedings of the 13th Annual SIGCSE Conference on Innovation and Technology in Computer Science Education, ITiCSE 2008, Madrid, Spain, June 30 - July 2, 2008*, page 328, "", 2008. ACM. doi: 10.1145/1384271.1384371. URL https://doi.org/10.1145/1384271.1384371.

[146] autolab. https://autolabproject.com, 2023. Accessed: 2023-10-08.

[147] M. Peveler, J. Tyler, S. Breese, B. Cutler, and A. L. Milanova. Submitty: An open source, highly-configurable platform for grading of programming assignments (abstract only). In M. E. Caspersen, S. H. Edwards, T. Barnes, and D. D. Garcia, editors, *Proceedings of the 2017 ACM SIGCSE Technical Symposium on Computer Science Education, SIGCSE 2017, Seattle, WA, USA, March 8-11, 2017*, page 641, "", 2017. ACM. doi: 10.1145/3017680.3022384. URL https://doi.org/10.1145/3017680.3022384.

[148] C. Sharp, J. van Assema, B. Yu, K. Zidane, and D. J. Malan. An open-source, api-based framework for assessing the correctness of code in CS50. In M. N. Giannakos, G. Sindre, A. Luxton-Reilly, and M. Divitini, editors, *Proceedings of the 2020 ACM Conference on Innovation and Technology in Computer Science Education, ITiCSE 2020, Trondheim, Norway, June 15-19, 2020*, pages

487–492, "", 2020. ACM. doi: 10.1145/3341525.3387417. URL https://doi.org/10.1145/3341525.3387417.

[149] Codeboard.io. https://codeboard.io, 2023. Accessed: 2023-10-08.

[150] B. P. Cipriano, N. Fachada, and P. Alves. Drop project: An automatic assessment tool for programming assignments. *SoftwareX*, 18:101079, 2022. doi: 10.1016/j.softx.2022.101079. URL https://doi.org/10.1016/j.softx.2022.101079.

[151] C. Iddon, N. Giacaman, and V. Terragni. GRADESTYLE: github-integrated and automated assessment of java code style. In *45th IEEE/ACM International Conference on Software Engineering: Software Engineering Education and Training, SEET@ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pages 192–197, "", 2023. IEEE. doi: 10.1109/ICSE-SEET58685.2023.00024. URL https://doi.org/10.1109/ICSE-SEET58685.2023.00024.

[152] GradeScope. . https://www.gradescope.com, 2024. Accessed: 2024-05-01.

[153] B. Coleman and N. Sommer. Fostering a testing mindset through automated feedback on multiple submissions using git-keeper. In *IEEE International Conference on Software Testing, Verification and Validation, ICST 2024 - Workshops, Toronto, ON, Canada, May 27-31, 2024*, pages 349–353. IEEE, 2024. doi: 10.1109/ICSTW60967.2024.00066. URL https://doi.org/10.1109/ICSTW60967.2024.00066.

[154] G. Classroom. . https://classroom.github.com, 2024. Accessed: 2024-05-01.

[155] Leetcode. https://leetcode.com, 2023. Accessed: 2023-10-08.

[156] topcoder. https://www.topcoder.com, 2023. Accessed: 2023-10-08.

[157] Codeforces. https://codeforces.com, 2023. Accessed: 2023-10-08.

[158] replit. https://replit.com, 2023. Accessed: 2023-10-08.

[159] S. Gulwani, V. A. Korthikanti, and A. Tiwari. Synthesizing geometry constructions. In M. W. Hall and D. A. Padua, editors, *PLDI 2011*, pages 50–61. ACM, 2011.

[160] R. Singh, S. Gulwani, and S. K. Rajamani. Automatically generating algebra problems. In J. Hoffmann and B. Selman, editors, *Proceedings of the Twenty-Sixth AAAI Conference on Artificial Intelligence, July 22-26, 2012, Toronto, Ontario, Canada*. AAAI Press, 2012.

[161] U. Z. Ahmed, S. Gulwani, and A. Karkare. Automatically generating problems and solutions for natural deduction. In F. Rossi, editor, *IJCAI 2013, Proceedings of the 23rd International Joint Conference on Artificial Intelligence, Beijing, China, August 3-9, 2013*, pages 1968–1975. IJCAI/AAAI, 2013.

[162] N. Tillmann, J. de Halleux, T. Xie, and J. Bishop. Pex4fun: A web-based environment for educational gaming via automated test generation. In E. Denney, T. Bultan, and A. Zeller, editors, *2013*

*28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 730–733. IEEE, 2013.

[163] V. Kuncak, M. Mayer, R. Piskac, and P. Suter. Software synthesis procedures. *Commun. ACM*, 55 (2):103–111, 2012.

[164] A. Solar-Lezama. Program sketching. *Int. J. Softw. Tools Technol. Transf.*, 15(5-6):475–495, 2013.

[165] S. Gulwani, I. Radicek, and F. Zuleger. Feedback generation for performance problems in introductory programming assignments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, (FSE-22), Hong Kong, China, November 16 - 22, 2014*, pages 41–51. ACM, 2014.

[166] K. Zimmerman and C. R. Rupakheti. An automated framework for recommending program elements to novices (N). In M. B. Cohen, L. Grunske, and M. Whalen, editors, *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 283–288. IEEE Computer Society, 2015.

[167] S. Kaleeswaran, A. Santhiar, A. Kanade, and S. Gulwani. Semi-supervised verified feedback generation. In T. Zimmermann, J. Cleland-Huang, and Z. Su, editors, *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 739–750. ACM, 2016. doi: 10.1145/2950290.2950363. URL `https://doi.org/10.1145/2950290.2950363`.

[168] H. J. B. Rocha, E. de Barros Costa, and P. C. d. A. R. Tedesco. Helping to provide adaptive feedback to novice programmers: a framework to assist the teachers. In *2023 18th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6. IEEE, 2023.

[169] S. Lau and P. J. Guo. Codehound: Helping instructors track pedagogical code dependencies in course materials. In M. Henz and B. S. Lerner, editors, *Proceedings of the 2022 ACM SIGPLAN International Symposium on SPLASH-E, SPLASH-E 2022, Auckland, New Zealand, 5 December 2022*, pages 1–6. ACM, 2022. doi: 10.1145/3563767.3568126. URL `https://doi.org/10.1145/3563767.3568126`.

[170] H. Ye, J. Gu, M. Martinez, T. Durieux, and M. Monperrus. Automated classification of overfitting patches with statically extracted code features. *IEEE Trans. Software Eng.*, 48(8):2920–2938, 2022. doi: 10.1109/TSE.2021.3071750. URL `https://doi.org/10.1109/TSE.2021.3071750`.

[171] P. Orvalho, M. Janota, and V. Manquinho. C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments. In *2024 IEEE/ACM International Workshop on Automated Program Repair (APR)*, pages 14–21, ., 2024. ACM. doi: 10.1145/3643788.3648010. URL `https://doi.org/10.1145/3643788.3648010`.

[172] P. Orvalho, J. Piepenbrock, M. Janota, and V. Manquinho. Project Proposal: Learning Variable Mappings to Repair Programs. In *7th Conference on Artificial Intelligence and Theorem Proving, AITP*, 2022.

[173] P. Orvalho, J. Piepenbrock, M. Janota, and V. M. Manquinho. Graph Neural Networks for Mapping Variables Between Programs. In *ECAI 2023 - 26th European Conference on Artificial Intelligence*, volume 372 of *Frontiers in Artificial Intelligence and Applications*, pages 1811–1818, Poland, 2023. IOS Press. doi: 10.3233/FAIA230468. URL https://doi.org/10.3233/FAIA230468.

[174] P. Orvalho, M. Janota, and V. M. Manquinho. MultIPAs: Applying Program Transformations To Introductory Programming Assignments For Data Augmentation. In A. Roychoudhury, C. Cadar, and M. Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 1657–1661. ACM, 2022. doi: 10.1145/3540250.3558931. URL https://doi.org/10.1145/3540250.3558931.

[175] C. Le Goues, N. Holtschulte, E. K. Smith, Y. Brun, P. Devanbu, S. Forrest, and W. Weimer. The manybugs and introclass benchmarks for automated repair of c programs. *IEEE Transactions on Software Engineering*, 41(12):1236–1256, 2015.

[176] S. H. Tan, J. Yi, Yulis, S. Mechtaev, and A. Roychoudhury. Codeflaws: a programming competition benchmark for evaluating automated program repair tools. In S. Uchitel, A. Orso, and M. P. Robillard, editors, *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, pages 180–182. IEEE Computer Society, 2017.

[177] G. An, M. Kwon, K. Choi, J. Yi, and S. Yoo. BUGSC++: A highly usable real world defect benchmark for C/C++. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pages 2034–2037. IEEE, 2023. doi: 10.1109/ASE56229.2023.00208. URL https://doi.org/10.1109/ASE56229.2023.00208.

[178] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empir. Softw. Eng.*, 10(4):405–435, 2005. doi: 10.1007/S10664-005-3861-2.

[179] G. Polya and G. Pólya. *How to solve it: A new aspect of mathematical method*, volume 34. Princeton university press, 2014.

[180] P. Cashin, C. Martinez, W. Weimer, and S. Forrest. Understanding automatically-generated patches through symbolic invariant differences. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pages 411–414. IEEE, 2019. URL https://doi.org/10.1109/ASE.2019.00046.

[181] Z. Y. Ding, Y. Lyu, C. S. Timperley, and C. L. Goues. Leveraging program invariants to promote population diversity in search-based automatic program repair. In J. Petke, S. H. Tan, W. B. Langdon, and W. Weimer, editors, *Proceedings of the 6th International Workshop on Genetic Improvement, GI@ICSE 2019, Montreal, Quebec, Canada, May 28, 2019*, pages 2–9. ACM, 2019. URL https://doi.org/10.1109/GI.2019.00011.

[182] B. Yang and J. Yang. Exploring the differences between plausible and correct patches at fine-grained level. In *2020 IEEE 2nd International Workshop on Intelligent Bug Fixing (IBF)*, pages 1–8. IEEE, 2020.

[183] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In A. Y. Halevy, Z. G. Ives, and A. Doan, editors, *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, San Diego, California, USA, June 9-12, 2003*, pages 76–85. ACM, 2003.

[184] V. I. Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady academii nauk*, volume 10, pages 707–710. Soviet Union, 1966.

[185] H. Steinhaus. Sur la division des corps matériels en parties. *Bull. Acad. Polon. Sci*, 1(804):801, 1956.

[186] P. Orvalho, M. Janota, and V. M. Manquinho. C-Pack of IPAs: A C90 Program Benchmark of Introductory Programming Assignments, 2022. URL `https://doi.org/10.48550/arXiv.2206.08768`.

[187] D. A. Reynolds. Gaussian mixture models. *Encyclopedia of biometrics*, 741:659–663, 2009.

[188] H. Schütze, C. D. Manning, and P. Raghavan. *Introduction to information retrieval*, volume 39. Cambridge University Press Cambridge, 2008.

[189] L. Van der Maaten and G. Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(11), 2008.

[190] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.

[191] U. Alon, S. Brody, O. Levy, and E. Yahav. code2seq: Generating sequences from structured representations of code. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[192] D. Zügner, T. Kirschstein, M. Catasta, J. Leskovec, and S. Günnemann. Language-agnostic representation learning of source code from structure and context. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[193] M. Allamanis, M. Brockschmidt, and M. Khademi. Learning to represent programs with graphs. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*, 2018.

[194] M. Allamanis, H. Jackson-Flux, and M. Brockschmidt. Self-supervised bug detection and repair. In *NeurIPS*, 2021.

[195] M. Vasic, A. Kanade, P. Maniatis, D. Bieber, and R. Singh. Neural program repair by jointly learning to localize and repair. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[196] X. Liu, J. Jang, N. Sundaresan, M. Allamanis, and A. Svyatkovskiy. Adaptivepaste: Code adaptation through learning semantics-aware variable usage representations. *CoRR*, abs/2205.11023, 2022.

[197] M. Pradel and K. Sen. Deepbugs: a learning approach to name-based bug detection. *Proc. ACM Program. Lang.*, 2(OOPSLA):147:1–147:25, 2018.

[198] V. J. Hellendoorn, C. Sutton, R. Singh, P. Maniatis, and D. Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[199] R. Karampatsis and C. Sutton. How often do single-statement bugs occur?: The manysstubs4j dataset. In S. Kim, G. Gousios, S. Nadi, and J. Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 573–577. ACM, 2020.

[200] D. Tarlow, S. Moitra, A. Rice, Z. Chen, P. Manzagol, C. Sutton, and E. Aftandilian. Learning to fix build errors with graph2diff neural networks. In *ICSE '20: 42nd International Conference on Software Engineering, Workshops, Seoul, Republic of Korea, 27 June - 19 July, 2020*, pages 19–20. ACM, 2020.

[201] S. Yu, T. Wang, and J. Wang. Data augmentation by program transformation. *J. Syst. Softw.*, 190: 111304, 2022. doi: 10.1016/j.jss.2022.111304.

[202] X. Liu and H. Zhong. Mining stackoverflow for program repair. In R. Oliveto, M. D. Penta, and D. C. Shepherd, editors, *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*, pages 118–129. IEEE Computer Society, 2018. doi: 10.1109/SANER.2018.8330202. URL `https://doi.org/10.1109/SANER.2018.8330202`.

[203] Y. Wang, K. Wang, F. Gao, and L. Wang. Learning semantic program embeddings with graph interval neural network. *Proc. ACM Program. Lang.*, 4(OOPSLA):137:1–137:27, 2020. doi: 10.1145/3428205. URL `https://doi.org/10.1145/3428205`.

[204] K. Wang, R. Singh, and Z. Su. Dynamic neural program embeddings for program repair. In *6th International Conference on Learning Representations, ICLR 2018*. OpenReview.net, 2018.

[205] M. Jose and R. Majumdar. Cause clue clauses: error localization using maximum satisfiability. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 437–446. ACM, 2011.

[206] R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In P. Bjesse and A. Slobodová, editors, *International Conference on Formal Methods in Computer-Aided Design, FMCAD '11, Austin, TX, USA, October 30 - November 02, 2011*, pages 91–100. FMCAD Inc., 2011. URL http://dl.acm.org/citation.cfm?id=2157671.

[207] S. Lamraoui and S. Nakajima. A formula-based approach for automatic fault localization of imperative programs. In S. Merz and J. Pang, editors, *International Conference on Formal Engineering Methods, ICFEM 2014, Luxembourg, Luxembourg, November 3-5, 2014. Proceedings*. Springer, 2014. doi: 10.1007/978-3-319-11737-9_17.

[208] S. Lamraoui and S. Nakajima. A formula-based approach for automatic fault localization of multi-fault programs. *JIP*, 2016. doi: 10.2197/ipsjjip.24.88.

[209] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. In *5th International Conference on Learning Representations, ICLR 2017*, 2017.

[210] D. Cummings and M. Nassar. Structured citation trend prediction using graph neural networks. In *ICASSP 2020 - 2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 3897–3901, 2020. doi: 10.1109/ICASSP40776.2020.9054769.

[211] Z. A. Goertzel, J. Jakubuv, C. Kaliszyk, M. Olsák, J. Piepenbrock, and J. Urban. The isabelle ENIGMA. In *13th International Conference on Interactive Theorem Proving, ITP 2022*, volume 237 of *LIPIcs*, pages 16:1–16:21, 2022. doi: 10.4230/LIPIcs.ITP.2022.16.

[212] M. S. Schlichtkrull, T. N. Kipf, P. Bloem, R. van den Berg, I. Titov, and M. Welling. Modeling relational data with graph convolutional networks. In *The Semantic Web - 15th International Conference, ESWC 2018*, volume 10843 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2018. doi: 10.1007/978-3-319-93417-4_38. URL https://doi.org/10.1007/978-3-319-93417-4_38.

[213] PyTorchGeometric. Documentation. https://pytorch-geometric.readthedocs.io/en/latest/modules/nn.html#torch_geometric.nn.conv.RGCNConv, 2022. Accessed 2022-08-12.

[214] L. J. Ba, J. R. Kiros, and G. E. Hinton. Layer normalization. *CoRR*, http://arxiv.org/abs/1607.06450, 2016. URL http://arxiv.org/abs/1607.06450.

[215] D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015. URL http://arxiv.org/abs/1412.6980.

[216] M. Vijaymeena and K. Kavitha. A survey on similarity measures in text mining. *Machine Learning and Applications: An International Journal*, 3(2):19–28, 2016.

[217] P. Orvalho, M. Janota, and V. Manquinho. CFaults: Model-Based Diagnosis for Fault Localization in C with Multiple Test Cases, June 2024. URL https://github.com/pmorvalho/CFaults.

[218] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991. doi: 10.1145/115372.115320. URL https://doi.org/10.1145/115372.115320.

[219] pycparser. . https://github.com/eliben/pycparser, 2024. [Online; accessed 18-April-2024].

[220] A. Ignatiev, A. Morgado, and J. Marques-Silva. RC2: an efficient MaxSAT solver. *J. Satisf. Boolean Model. Comput.*, 11(1):53–64, 2019.

[221] A. Ignatiev, A. Morgado, and J. Marques-Silva. PySAT: A python toolkit for prototyping with SAT oracles. In O. Beyersdorff and C. M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 428–437. Springer, 2018. doi: 10.1007/978-3-319-94144-8_26. URL https://doi.org/10.1007/978-3-319-94144-8_26.

[222] M. C. Hansen, H. Yalcin, and J. P. Hayes. Unveiling the ISCAS-85 benchmarks: A case study in reverse engineering. *IEEE Des. Test Comput.*, 16(3):72–80, 1999. doi: 10.1109/54.785838. URL https://doi.org/10.1109/54.785838.

[223] P. Orvalho, M. Janota, and V. Manquinho. GitSEED: A Git-backed Automated Assessment Tool for Software Engineering and Programming Education. In *Proceedings of the 2024 ACM Virtual Global Computing Education Conference, SIGCSE Virtual 2024*, volume 1, 2024. ISBN 979-8-4007-0598-4/24/12. doi: 10.1145/3649165.3690106. URL https://doi.org/10.1145/3649165.3690106.

[224] J. Hollingsworth. Automatic graders for programming classes. *Commun. ACM*, 3(10):528–529, 1960. doi: 10.1145/367415.367422. URL https://doi.org/10.1145/367415.367422.

[225] N. Nethercote and J. Seward. Valgrind: A program supervision framework. *Electronic notes in theoretical computer science*, 89(2):44–66, 2003.

[226] Z. Li, S. Wu, Y. Liu, J. Shen, Y. Wu, Z. Zhang, and X. Chen. Vsusfl: Variable-suspiciousness-based fault localization for novice programs. *Journal of Systems and Software*, 205:111822, 2023.

[227] P. Orvalho, M. Janota, and V. M. Manquinho. Cfaults: Model-based diagnosis for fault localization in C programs with multiple test cases. *CoRR*, abs/2407.09337, 2024. doi: 10.48550/ARXIV.2407.09337. URL https://doi.org/10.48550/arXiv.2407.09337.

[228] P. Orvalho, M. Janota, and V. Manquinho. . https://gitlab.inesc-id.pt/u020557/GitSEED, 2024. Accessed: 2024-09-10.

[229] P. Orvalho, M. Janota, and V. Manquinho. GitSEED: A Git-backed Automated Assessment Tool for Software Engineering Education (Artifact), Sept. 2024. URL T.

[230] Lizard. Lizard: A simple code complexity analyser. https://github.com/terryyin/lizard, 2024. Accessed: 2024-05-01.

[231] cppcheck. . https://cppcheck.sourceforge.io, 2024. Accessed: 2024-05-01.

[232] clang tidy. . https://clang.llvm.org/extra/clang-tidy/, 2024. Accessed: 2024-05-01.

[233] P. Orvalho, M. Janota, and V. M. Manquinho. Counterexample guided program repair using zero-shot learning and maxsat-based fault localization. *Proceedings of the AAAI Conference on Artificial Intelligence, AAAI 2025*, 39(1):649–657, Apr. 2025. doi: 10.1609/aaai.v39i1.32046. URL https://ojs.aaai.org/index.php/AAAI/article/view/32046.

[234] C. S. Xia and L. Zhang. Less training, more repairing please: revisiting automated program repair via zero-shot learning. In A. Roychoudhury, C. Cadar, and M. Kim, editors, *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pages 959–971. ACM, 2022. doi: 10.1145/3540250.3549101. URL https://doi.org/10.1145/3540250.3549101.

[235] M. Mishra, M. Stallone, G. Zhang, Y. Shen, A. Prasad, A. M. Soria, M. Merler, P. Selvam, S. Surendran, S. Singh, M. Sethi, X. Dang, P. Li, K. Wu, S. Zawad, A. Coleman, M. White, M. Lewis, R. Pavuluri, Y. Koyfman, B. Lublinsky, M. de Bayser, I. Abdelaziz, K. Basu, M. Agarwal, Y. Zhou, C. Johnson, A. Goyal, H. Patel, S. Y. Shah, P. Zerfos, H. Ludwig, A. Munawar, M. Crouse, P. Kapanipathi, S. Salaria, B. Calio, S. Wen, S. Seelam, B. Belgodere, C. A. Fonseca, A. Singhee, N. Desai, D. D. Cox, R. Puri, and R. Panda. Granite code models: A family of open foundation models for code intelligence. *CoRR*, abs/2405.04324, 2024. doi: 10.48550/ARXIV.2405.04324. URL https://doi.org/10.48550/arXiv.2405.04324.

[236] H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley, K. Bansal, L. Vilnis, M. Wirth, P. Michel, P. Choy, P. Joshi, R. Kumar, S. Hashmi, S. Agrawal, Z. Gong, J. Fine, T. Warkentin, A. J. Hartman, B. Ni, K. Korevec, K. Schaefer, and S. Huffman. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409, 2024. doi: 10.48550/ARXIV.2406.11409. URL https://doi.org/10.48550/arXiv.2406.11409.

[237] HuggingFace. . https://huggingface.co, 2024. [Online; accessed 1-July-2024].

[238] B. Rozière, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, A. Kozhevnikov, I. Evtimov, J. Bitton, M. Bhatt, C. Canton-Ferrer, A. Grattafiori, W. Xiong, A. Défossez, J. Copet, F. Azhar, H. Touvron, L. Martin, N. Usunier, T. Scialom, and G. Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

[239] T. Mesnard, C. Hardin, R. Dadashi, S. Bhupatiraju, S. Pathak, L. Sifre, M. Rivière, M. S. Kale, J. Love, P. Tafti, L. Hussenot, A. Chowdhery, A. Roberts, A. Barua, A. Botev, A. Castro-Ros, A. Slone, A. Héliou, A. Tacchetti, A. Bulanova, A. Paterson, B. Tsai, B. Shahriari, C. L. Lan, C. A. Choquette-Choo, C. Crepy, D. Cer, D. Ippolito, D. Reid, E. Buchatskaya, E. Ni, E. Noland, G. Yan, G. Tucker, G. Muraru, G. Rozhdestvenskiy, H. Michalewski, I. Tenney, I. Grishchenko, J. Austin, J. Keeling, J. Labanowski, J. Lespiau, J. Stanway, J. Brennan, J. Chen, J. Ferret, J. Chiu, and et al. Gemma: Open models based on gemini research and technology. *CoRR*, abs/2403.08295, 2024. doi: 10.48550/ARXIV.2403.08295. URL https://doi.org/10.48550/arXiv.2403.08295.

[240] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, A. Rodriguez, A. Joulin, E. Grave, and G. Lample. Llama: Open and efficient foundation language models. *CoRR*, abs/2302.13971, 2023. doi: 10.48550/ARXIV.2302.13971. URL https://doi.org/10.48550/arXiv.2302.13971.

[241] M. I. Abdin, S. A. Jacobs, A. A. Awan, J. Aneja, A. Awadallah, H. Awadalla, N. Bach, A. Bahree, A. Bakhtiari, H. S. Behl, A. Benhaim, M. Bilenko, J. Bjorck, S. Bubeck, M. Cai, C. C. T. Mendes, W. Chen, V. Chaudhary, P. Chopra, A. D. Giorno, G. de Rosa, M. Dixon, R. Eldan, D. Iter, A. Garg, A. Goswami, S. Gunasekar, E. Haider, J. Hao, R. J. Hewett, J. Huynh, M. Javaheripi, X. Jin, P. Kauffmann, N. Karampatziakis, D. Kim, M. Khademi, L. Kurilenko, J. R. Lee, Y. T. Lee, Y. Li, C. Liang, W. Liu, E. Lin, Z. Lin, P. Madan, A. Mitra, H. Modi, A. Nguyen, B. Norick, B. Patra, D. Perez-Becker, T. Portet, R. Pryzant, H. Qin, M. Radmilac, C. Rosset, S. Roy, O. Ruwase, O. Saarikivi, A. Saied, A. Salim, M. Santacroce, S. Shah, N. Shang, H. Sharma, X. Song, M. Tanaka, X. Wang, R. Ward, G. Wang, P. Witte, M. Wyatt, C. Xu, J. Xu, S. Yadav, F. Yang, Z. Yang, D. Yu, C. Zhang, C. Zhang, J. Zhang, L. L. Zhang, Y. Zhang, Y. Zhang, Y. Zhang, and X. Zhou. Phi-3 technical report: A highly capable language model locally on your phone. *CoRR*, abs/2404.14219, 2024. doi: 10.48550/ARXIV.2404.14219. URL https://doi.org/10.48550/arXiv.2404.14219.

[242] K. Zhang and D. E. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM J. Comput.*, 18(6):1245–1262, 1989. doi: 10.1137/0218082. URL https://doi.org/10.1137/0218082.

[243] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In F. Tip and E. Bodden, editors, *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*, pages 298–309. ACM, 2018. doi: 10.1145/3213846.3213871. URL https://doi.org/10.1145/3213846.3213871.

[244] D. Kahneman. Thinking, fast and slow. *Farrar, Straus and Giroux*, 2011.

[245] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical association*, 66(336):846–850, 1971.

[246] L. Hubert and P. Arabie. Comparing partitions. *Journal of classification*, 2(1):193–218, 1985.

[247] A. Strehl and J. Ghosh. Cluster ensembles—a knowledge reuse framework for combining multiple partitions. *Journal of machine learning research*, 3(Dec):583–617, 2002.

[248] N. X. Vinh, J. Epps, and J. Bailey. Information theoretic measures for clusterings comparison: is a correction for chance necessary? In *Proceedings of the 26th annual international conference on machine learning*, pages 1073–1080, 2009.

[249] E. B. Fowlkes and C. L. Mallows. A method for comparing two hierarchical clusterings. *Journal of the American statistical association*, 78(383):553–569, 1983.

[250] A. Rosenberg and J. Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In *Proceedings of the 2007 joint conference on empirical methods in natural language processing and computational natural language learning (EMNLP-CoNLL)*, pages 410–420, 2007.

# A

# C-PACK-IPAS

## A.1 List of Introductory Programming Assignments (IPAS)

### A.1.1 Lab02 - Integers and IO operations.

In Lab02, the students learn how to program with integers, floats, IO operations (mainly `printf` and `scanf`), conditionals (if-statements), and simple loops (for and while-loops).

**IPA #1: Lab02 - Ex01.** Write a program that determines and prints the largest of three integers given by the user.

**IPA #2: Lab02 - Ex02.** Write a program that reads two integers ‘N, M‘ and prints the smallest of them in the first row and the largest in the second.

**IPA #3: Lab02 - Ex03.** Write a program that reads two positive integers ‘N, M‘ and prints "yes" if ‘M‘ is a divisor of ‘N‘, otherwise prints "no".

**IPA #4: Lab02 - Ex04.** Write a program that reads three integers and prints them in order on the same line. The smallest number must appear first.

**IPA #5: Lab02 - Ex05.** Write a program that reads a positive integer ‘N‘ and prints the numbers ‘1..N‘, one per line.

**IPA #6: Lab02 - Ex06.** Write a program that determines the largest and smallest number of ‘N‘ real numbers given by the user. Consider that ‘N‘ is a value requested from the user. The result must be printed with the command ‘printf("min: %f, max: %fñ", min, max)‘. *Hint: initialize the largest and smallest to the first read value.*

**IPA #7: Lab02 - Ex07.** Write a program that asks the user for a positive integer ‘N‘ and prints the number of divisors of ‘N‘. Remember that prime numbers have 2 divisors.

**IPA #8: Lab02 - Ex08.** Write a program that calculates and prints the average of ‘N‘ real numbers given by user. The program should first ask the user for an integer ‘N‘, representing the number of numbers to be entered. The real numbers must be represented by float type. The result must be printed with the command ‘printf("%.2f", avg);‘.

**IPA #9: Lab02 - Ex09.** Write a program that asks the user for a value ‘N‘ corresponding to a certain period of time in seconds. The program should output this period of time in the format ‘HH:MM:SS‘. *Hint: use the operator that calculates the remainder of division (‘%‘).*

**IPA #10: Lab02 - Ex10.** Write a program that asks the user for a positive value ‘N‘. The output should present the number of digits that make up ‘N‘ (on the first line), as well as the sum of the digits of ‘N‘ (on the second line). For example, the number 12345 has 5 digits and the sum of these digits is 15.

### A.1.2 Lab03 - Loops and Chars.

In this lab, the students learn how to program with loops, nested loops, auxilary functions and chars.

**IPA #11: Lab03 - Ex01.** Write a program that draws a square of numbers like the following using the function ‘void square(int N);‘. The value of ‘N‘, given by the user, must be greater than or equal to 2. The tab (character ‘’‘‘) must be used as the separator. The square shown is the example for ‘N = 5‘.

*1 2 3 4 5*

*2 3 4 5 6*

*3 4 5 6 7*

*4 5 6 7 8*

*5 6 7 8 9*

**IPA #12: Lab03 - Ex02.** Write a program that draws a pyramid of numbers using the ‘void pyramid(int N);‘ function. The value of ‘N‘, given by the user, must be greater than or equal to 2. The space (character ‘’ ’‘) must be used as the separator. The pyramid shown is the example for ‘N = 5‘.

*1*

*1 2 1*

*1 2 3 2 1*

*1 2 3 4 3 2 1*

*1 2 3 4 5 4 3 2 1*

**IPA #13: Lab03 - Ex03.** Write a program that draws a cross on diagonals using the ‘void cross(int N);‘ function. The asterisk (‘’*’‘ character) must be used to draw the cross; hyphen (‘’-’‘ character) must be used as the separator. The crosses shown are the examples for ‘N = 3‘ and ‘N=8‘.

```
        *-------*
        -*-----*-
        --*---*--
        ---*-*---
        ---*-*---
  *-*   --*---*--
  -*-   -*-----*-
  *-*   *-------*
```

**IPA #14: Lab03 - Ex04.** Write a program that reads a sequence of numbers separated by spaces and newlines, and print the same string, but the numbers in the output should not contain 0 at the beginning, eg ‘007‘ should print ‘7‘. The exception is the number 0, which should be printed as 0. The string in the input ends with ‘EOF‘. *Warning: Number values may be greater than the maximum value of type ‘int‘ or any primitive type in C. Hint: the ‘int getchar()‘ function can be used to read a character.*

**IPA #15: Lab03 - Ex05.** Write a program that reads a sequence of messages and prints them out, one per line. Each message is delimited by quotation marks (character ‘"‘). The message can contain an "escape sequence" - the character loses special meaning if it is preceded by the character ‘̀(backslash). For example, the input ‘"a̋f̈oõbar"̈‘ matches the message ‘a"foõbar"‘. So the backslash allows you to include quotes in the message just like the backslash itself.

**IPA #16: Lab03 - Ex06.** Write a program that reads a positive integer from the input (such as a sequence of characters up to 100 chars) and that decides whether the number read is divisible by 9. If the number is divisible by 9, the program should print the message ‘yes‘, and should print ‘no‘ otherwise. *Warning: Number values can be greater than the maximum value of type ‘int‘ or any primitive type in C. Hint: A number is divisible by 9 iff the sum of its digits is divisible by 9.* For example, the sum of the digits

of the number 729 is 18, so it is divisible by 9. The fact can be seen from the following equation: 7 x 100 + 2 x 10 + 9 = (7 x 99 + 7) + (2 x 9 + 2) + 9.

**IPA #17: Lab03 - Ex07.** Write a program that takes a sequence of numbers and operators (‘+‘, ‘-‘) representing an arithmetic expression and returns the result of that arithmetic expression. The string in the input ends with ‘ñ‘. You can assume that every two numbers are always separated by ‘space, operator, space‘, i.e., ‘’op’‘, for either of the 2 operators above. Example: Input ‘70 + 22 - 3‘ should return ‘89‘. *Hint: You should start by converting a sequence of digits (characters) to an integer.*

### A.1.3 Lab04 - Vectors and Strings.

In this lab, the students learn how to program with integers arrays and strings.

**IPA #18: Lab04 - Ex01.** Write a program that asks the user for a positive integer ‘n < VECMAX‘, where ‘VECMAX=100‘. Then read ‘n‘ positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for ‘n = 3‘ and values ‘1 3 4‘.

```
*
***
****
```

**IPA #19: Lab04 - Ex02.** Write a program that asks the user for a positive integer ‘n < VECMAX‘, where ‘VECMAX=100‘. Then read ‘n‘ positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for ‘n = 3‘ and values ‘1 3 4‘.

```
***
 **
 **
  *
```

**IPA #20: Lab04 - Ex03.** Write a program that asks the user for a positive integer ‘n < VECMAX‘, where ‘VECMAX=100‘. Then read ‘n‘ positive integers. At the end the program should write a graphical representation of the values read as follows. The graph shown is the example for ‘n = 3‘ and values ‘1 3 4‘.

```
  *
 **
 **
***
```

Consider that in the following IPAs, all strings have a maximum of ‘MAX = 80‘ characters (including the end-of-string character).

**IPA #21: Lab04 - Ex04.** Write a program that reads a word from the terminal and checks whether the word is a palindrome or not. A word is a palindrome if it is spelled the same way from left to right and vice versa (eg "AMA" is a palindrome). If the word is a palindrome, the program should print the value ‘yes‘, and ‘no‘ if not. *Hint: You can use ‘scanf("%s", s)‘ to read a word. Note that the string ‘s‘ does not ask for ‘&‘ in ‘scanf‘.*

**IPA #22: Lab04 - Ex05.** Write a program that reads characters from the keyboard, character by character, until it finds the character ‘ñ‘ or EOF and writes the line read to the terminal. Implement the ‘int leLinha(char s[])‘ function which reads the line into the string ‘s‘ and returns the number of characters read. *Hint: After solving this exercise, try using the ‘fgets‘ command.*

**IPA #23: Lab04 - Ex06.** Write a program that reads a line from the terminal (use the function from the previous exercise) and writes the same text to the terminal, but with the lowercase letters replaced by the respective uppercase letters. Implement the ‘void uppercase(char s[])‘ function. *Note: Remember that the string ‘s‘ is changed by the ‘uppercase‘ function.*

**IPA #24: Lab04 - Ex07.** Write a program that reads a line and a character and writes to the terminal the same line where all occurrences of the character were removed. Implement the ‘void eraseCharacter(char s[], char c)‘ function that erases the character ‘c‘ from the string ‘s‘.

**IPA #25: Lab04 - Ex08.** Write a program that reads two integers in decimal representation and prints the larger of those two numbers. You can assume that the two numbers have the same number of digits and a maximum of 100 characters. *Note: The numbers may be too large to be stored in a ‘long long‘ variable, for example ‘99888888888888888887‘ and ‘99888888888888888888‘.*

## A.2 Number of submissions

Table A.1 presents the number of semantically correct student submissions received for 25 different programming assignments over three lab classes for three different years. Next, Table A.2 presents the set of semantically incorrect submissions, while Table A.3 describes the set of syntactically incorrect programs.

Table A.1: The number of semantically correct student submissions received for 25 different programming assignments over three lab classes for three different years.

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 1** | **Lab02** | 25 | 25 | 25 | 23 | 25 | 23 | 22 | 23 | 24 | 23 | **238** |
| | **Lab03** | 20 | 18 | 16 | 7 | 16 | 17 | 20 | - | - | - | **114** |
| | **Lab04** | 22 | 22 | 19 | 22 | 18 | 19 | 21 | 13 | - | - | **153** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 2** | **Lab02** | 13 | 8 | 8 | 7 | 8 | 9 | 7 | 6 | 7 | 6 | **79** |
| | **Lab03** | 6 | 5 | 3 | 1 | 4 | 7 | 7 | - | - | - | **33** |
| | **Lab04** | 6 | 7 | 6 | 6 | 6 | 5 | 4 | 3 | - | - | **43** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 3** | **Lab02** | 52 | 50 | 54 | 44 | 50 | 51 | 44 | 47 | 43 | 47 | **482** |
| | **Lab03** | 40 | 39 | 37 | 10 | 15 | 37 | 26 | - | - | - | **204** |
| | **Lab04** | 38 | 34 | 30 | 49 | 36 | 32 | 27 | 23 | - | - | **269** |

Table A.2: The number of semantically incorrect student submissions received for 25 different programming assignments over three lab classes for three different years.

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 1** | **Lab02** | 36 | 10 | 7 | 12 | 3 | 5 | 7 | 9 | 21 | 3 | **113** |
| | **Lab03** | 32 | 35 | 20 | 69 | 16 | 17 | 9 | - | - | - | **198** |
| | **Lab04** | 5 | 11 | 5 | 6 | 10 | 5 | 14 | 10 | - | - | **66** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 2** | **Lab02** | 28 | 2 | 1 | 7 | 2 | 4 | 7 | 2 | 3 | 4 | **60** |
| | **Lab03** | 14 | 10 | 11 | 17 | 15 | 6 | 4 | - | - | - | **77** |
| | **Lab04** | 6 | 1 | 1 | 2 | 7 | 1 | 4 | 6 | - | - | **28** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 3** | **Lab02** | 51 | 43 | 31 | 33 | 4 | 28 | 22 | 41 | 36 | 24 | **313** |
| | **Lab03** | 58 | 76 | 44 | 121 | 63 | 31 | 31 | - | - | - | **424** |
| | **Lab04** | 5 | 17 | 5 | 41 | 19 | 8 | 21 | 36 | - | - | **152** |

Table A.3: The number of syntactically incorrect student submissions received for 25 different programming assignments over three lab classes for three different years.

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 1** | **Lab02** | 6 | 0 | 1 | 5 | 4 | 4 | 4 | 2 | 1 | 2 | **29** |
| | **Lab03** | 6 | 3 | 1 | 6 | 2 | 1 | 2 | - | - | - | **21** |
| | **Lab04** | 2 | 1 | 1 | 0 | 5 | 0 | 1 | 2 | - | - | **12** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 2** | **Lab02** | 6 | 3 | 0 | 5 | 1 | 5 | 0 | 0 | 0 | 0 | **20** |
| | **Lab03** | 1 | 0 | 0 | 1 | 1 | 1 | 1 | - | - | - | **5** |
| | **Lab04** | 0 | 0 | 0 | 1 | 1 | 1 | 4 | 0 | - | - | **7** |

| | | E1 | E2 | E3 | E4 | E5 | E6 | E7 | E8 | E9 | E10 | Total |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Year 3** | **Lab02** | 24 | 17 | 14 | 20 | 10 | 27 | 11 | 31 | 8 | 12 | **174** |
| | **Lab03** | 18 | 8 | 7 | 10 | 15 | 9 | 8 | - | - | - | **80** |
| | **Lab04** | 14 | 7 | 4 | 18 | 15 | 10 | 11 | 21 | - | - | **100** |

# B

# Program Clustering

## B.1 Use Case #1: Clustering IPAs

### B.1.1 Clustering Accuracy

Figure B.1 shows a matrix with the different values of the cluster accuracy using the MINIBATCH KMEANS algorithm on each program representation using ten different seeds. Each entry is highlighted accordingly to its value. The lowest value is highlighted in black, and the highest is highlighted in white. Intermediate values are highlighted in different shades of grey, depending on how far they are from the lowest value.

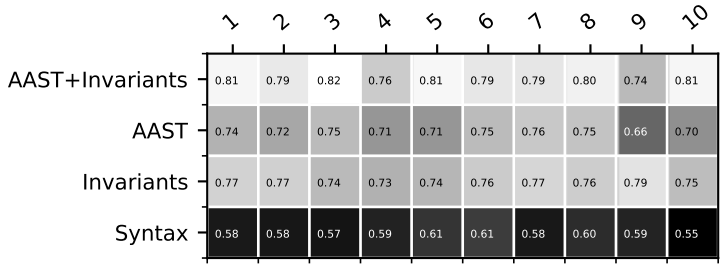



Figure B.1: The values for cluster accuracy using the MINIBATCH KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

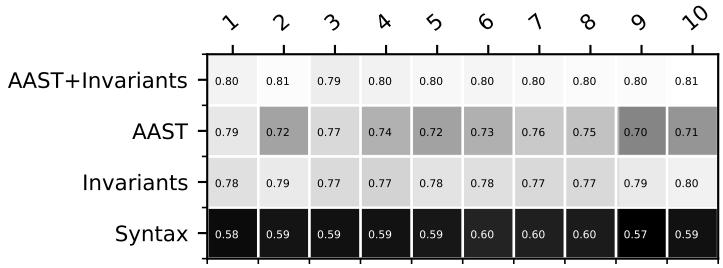



Figure B.2: The values for cluster accuracy using the BIRCH algorithm on each program representation after ten different runs, each run using a different seed.

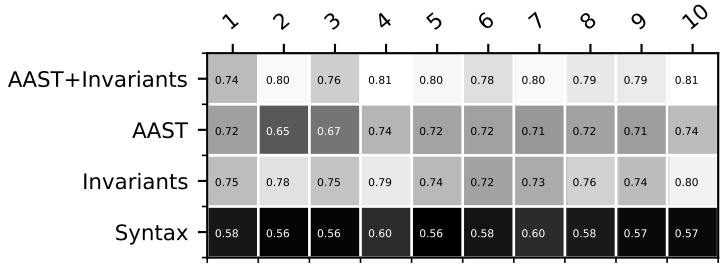



Figure B.3: The values for cluster accuracy using the GAUSSIAN MIXTURE algorithm on each program representation after ten different runs, each run using a different seed.

Secondly, Figure B.2 shows a matrix with the different values of the cluster accuracy using the BIRCH algorithm on each program representation using ten different seeds.

Lastly, Figure B.3 shows a matrix with the different values of the cluster accuracy using the GAUSSIAN MIXTURE algorithm on each program representation for ten different seeds.

### B.1.2 Other Evaluation Metrics

In this section, we present other clustering evaluation metrics for the KMEANS algorithm, such as: the *Rand index*, the *adjusted Rand index*, the *normalized mutual information*, the *adjusted mutual information*, the *FowlkesMallows index*, the *completeness score*, the *homogeneity score*, and the *V measure*.

**Rand Index.** The *Rand index* measures the similarity of the two assignments, ignoring permutations [245]. The Rand index is given by the following equation B.1:

$$RI = \frac{TP + TN}{TP + FP + FN + TN} \tag{B.1}$$

Figure B.4 presents the values for the Rand index using the KMEANS algorithm on each program representation after ten different runs.

**Adjusted Rand Index.** The *adjusted Rand index* is the corrected-for-chance version of the Rand index since the Rand index does not guarantee that random label assignments will get a value close to zero [246]. The adjusted Rand index is given by equation B.2, where $TP$ is the number of true positives,

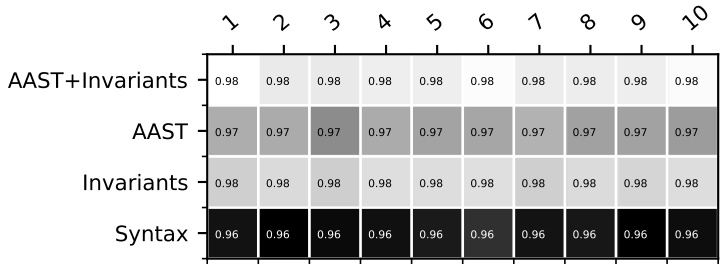



Figure B.4: The values for the Rand index using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

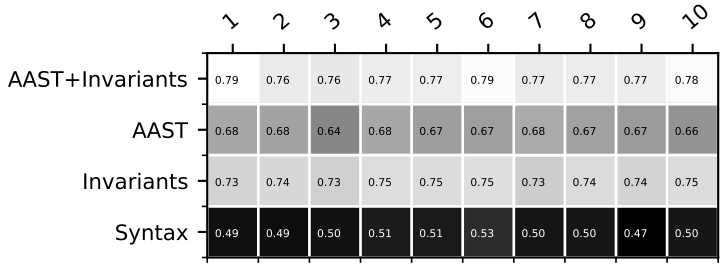



Figure B.5: The values for the adjusted Rand index using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

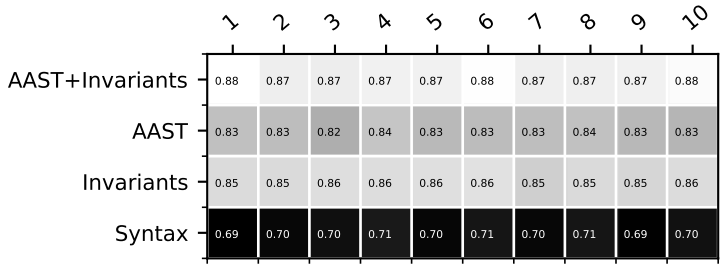



Figure B.6: The values for the normalized mutual information using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

$TN$ is the number of true negatives, $FP$ is the number of false positives, and $FN$ is the number of false negatives.

$$ARI = \frac{RI - E[RI]}{max(RI) - E[RI]} \tag{B.2}$$

Figure B.5 presents the values for the adjusted Rand index using the KMEANS algorithm on each program representation after ten different runs.

**Normalized Mutual Information.** The *normalized mutual information* of two random variables is a measure of the mutual dependence between the two variables [247]. Figure B.6 shows the values for the normalized mutual information using the KMEANS algorithm on each program representation after 10 different runs.

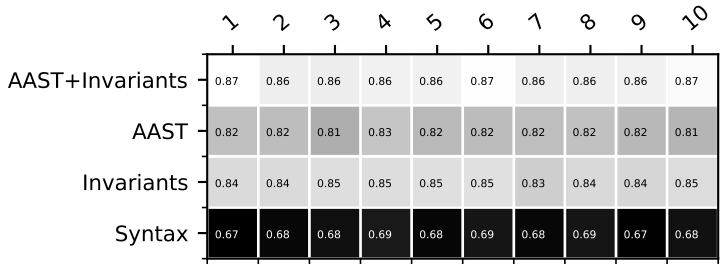



Figure B.7: The values for the adjusted mutual information using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

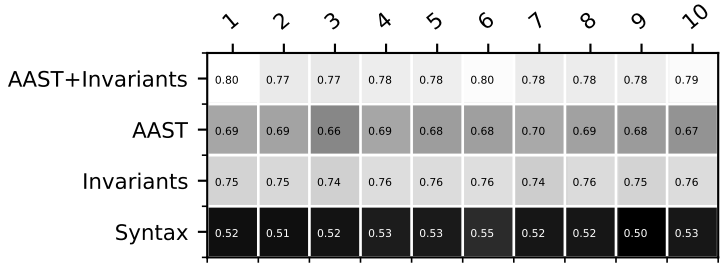



Figure B.8: The values for the FowlkesMallows index using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

**Adjusted Mutual Information.** The *adjusted mutual information* corrects the effect of the agreement solely due to chance between clusterings, similar to the way the adjusted rand index corrects the Rand index [248]. Figure B.7 presents the values for the adjusted mutual information using the KMEANS algorithm on each program representation after ten different runs.

**FowlkesMallows index.** The *Fowlkes-Mallows index* measures the similarity between two clusters. A high value for the FowlkesMallows index indicates a great similarity between the clusters and the benchmark classifications [249]. The Fowlkes-Mallows index can be computed using equation B.3, where $TP$ is the number of true positives, $FP$ is the number of false positives, and $FN$ is the number of false negatives. $TPR$ is the true positive rate, also called sensitivity or recall, and $PPV$ is the positive predictive rate, also known as precision.

$$FM = \sqrt{PPV \cdot TPR} = \sqrt{\frac{TP}{TP + FP} \cdot \frac{TP}{TP + FN}} \tag{B.3}$$

Figure B.8 shows the values for the FowlkesMallows index using the KMEANS algorithm on each program representation after ten different runs.

**Completeness score.** The *completeness score* measure if all members of a given class are assigned to the same cluster [250]. Figure B.9 shows the values for the completeness score using the KMEANS algorithm on each program representation after ten different runs.

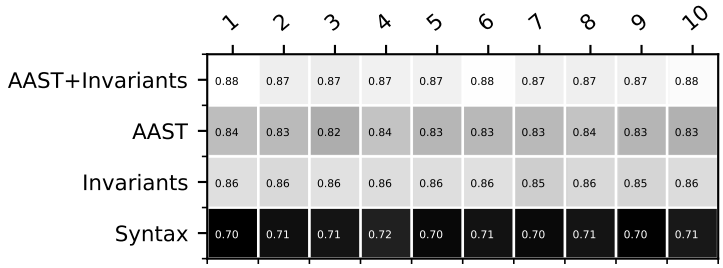



Figure B.9: The values for the completeness score using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

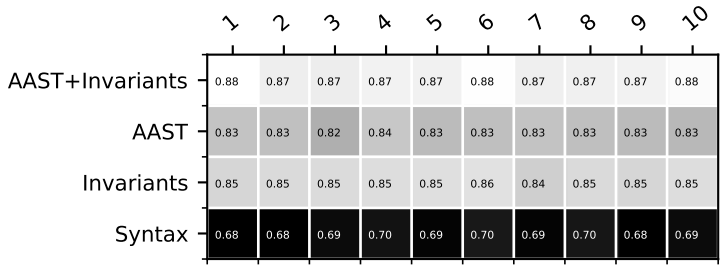



Figure B.10: The values for the homogeneity score using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

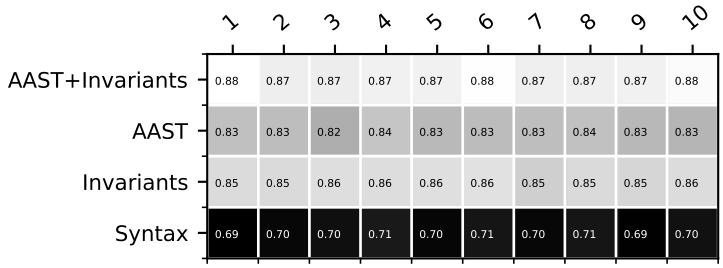



Figure B.11: The values for the V measure using the KMEANS algorithm on each program representation after ten different runs, each run using a different seed.

**Homogeneity score.** The *homogeneity score* checks if each cluster contains only members of a single class [250]. Figure B.10 presents the values for the homogeneity score using the KMEANS algorithm on each program representation after ten different runs.

**V measure.** The *V measure* is the harmonic mean between the homogeneity and completeness scores [250]. Figure B.11 shows the values for the V measure using the KMEANS algorithm on each program representation after ten different runs.

## B.2 Use Case #2: Repairing IPAs

Table B.1 presents the number of clusters each clustering method uses for each IPA. This table was used to generate the cactus plot in Figure 5.6.

Table B.1: The number of clusters generated using each clustering approach for each IPA.

| Exercise | #Correct Submissions | Clara Clusters | KMeans AAST | KMeans AAST+Invs | KMeans Invs | KMeans Syntax | Closest Program (KMeans) AAST+Invs |
|---|---|---|---|---|---|---|---|
| **lab02/ex01** | 92 | 18 | 9 | 9 | 9 | 9 | 1 |
| **lab02/ex02** | 84 | 6 | 8 | 8 | 8 | 8 | 1 |
| **lab02/ex03** | 83 | 4 | 8 | 8 | 7 | 8 | 1 |
| **lab02/ex04** | 76 | 20 | 7 | 7 | 7 | 7 | 1 |
| **lab02/ex05** | 80 | 10 | 7 | 7 | 7 | 7 | 1 |
| **lab02/ex06** | 68 | 25 | 6 | 6 | 6 | 6 | 1 |
| **lab02/ex07** | 67 | 17 | 6 | 6 | 6 | 6 | 1 |
| **lab02/ex08** | 49 | 21 | 4 | 4 | 4 | 4 | 1 |
| **lab02/ex09** | 74 | 12 | 7 | 7 | 7 | 7 | 1 |
| **lab02/ex10** | 65 | 17 | 6 | 6 | 6 | 6 | 1 |
| **lab03/ex01** | 70 | 51 | 6 | 6 | 6 | 6 | 1 |
| **lab03/ex02** | 55 | 49 | 5 | 5 | 5 | 5 | 1 |
| **lab03/ex03** | 45 | 27 | 4 | 4 | 4 | 4 | 1 |
| **lab03/ex04** | 28 | 8 | 2 | 2 | 2 | 2 | 1 |
| **lab03/ex06** | 46 | 8 | 4 | 4 | 4 | 4 | 1 |
| **lab04/ex01** | 59 | 32 | 5 | 5 | 5 | 5 | 1 |
| **lab04/ex02** | 47 | 32 | 4 | 4 | 4 | 4 | 1 |
| **lab04/ex03** | 41 | 33 | 4 | 4 | 4 | 4 | 1 |
| **lab04/ex08** | 8 | 6 | 1 | 1 | 1 | 1 | 1 |
| **lab05/ex01** | 4 | 3 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab3/ex2810** | 17 | 9 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab3/ex2811** | 7 | 3 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab3/ex2812** | 17 | 7 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab3/ex2813** | 4 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2824** | 15 | 5 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2825** | 10 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2827** | 6 | 6 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2831** | 7 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2832** | 17 | 7 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab4/ex2833** | 19 | 9 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2865** | 7 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2866** | 11 | 10 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2867** | 7 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2868** | 8 | 6 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2869** | 7 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2870** | 9 | 8 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab5/ex2871** | 15 | 10 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2932** | 3 | 3 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2933** | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2936** | 5 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2937** | 2 | 2 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2938** | 6 | 4 | 1 | 1 | 1 | 1 | 1 |
| **itsp/lab6/ex2939** | 2 | 1 | 1 | 1 | 1 | 1 | 1 |

# C

# Variable Mapping

## C.1 IPAs Dataset Generation

To evaluate our work, we have generated a dataset of pairs programs based on a benchmark of student programs developed during an introductory programming course in the C programming language for ten different introductory programming assignments (IPAs), over two distinct academic years. We selected only semantically correct submissions i.e., programs that compiled without any error and satisfied a set of input-output test cases for each IPA.

Afterwards, we generated a dataset of pairs of correct/incorrect programs to train and evaluate our work with specific bugs. The reason to generate programs is that we need to know the real variable mappings between two programs (ground truth) to evaluate our representation. As explained in Chapter 7, we used MULTIPAS to generate this dataset. This tool can mutate our programs syntactically, generating semantically equivalent programs. There are several program mutations available in MULTIPAS such as: mirroring comparison expressions, swapping the if's then-block with the else-block and negating the test condition, increment/decrement operators mirroring, variable declarations reordering, translating for-loops into equivalent while-loops, and all possible combinations of these program mutations. Hence, MULTIPAS has 31 different configurations for mutating a program. Each program mutation can be applied in more than one place for a given program. Hence, each program mutation can generate several different mutated programs. For example, using the program mutation that reorders variable

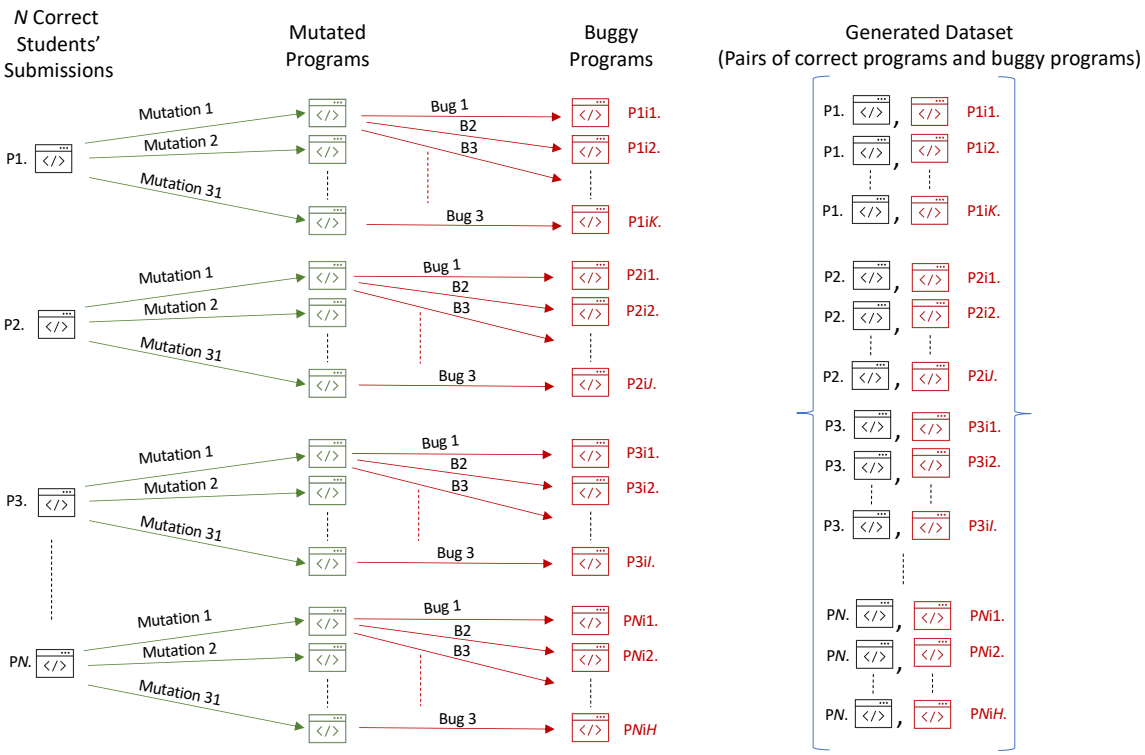



Figure C.1: IPAs Dataset Generation

declarations, each possible reordering generates a different mutated program.

Regarding the generation of buggy programs, we also used MULTIPAS, for introducing bugs into the programs, such as *wrong comparison operator* (WCO), *variable misuse* (VM) and *missing expression* (ME). Each bug can be applied in more than one place for a given program. Thus, one program can generate several different buggy programs using the same bug. For example, the bug of variable misuse can be applied in each variable occurrence in the program, each one generates a single buggy program.

Figure C.1 presents the generation of our dataset. Firstly, we applied all the available program mutations to each correct student's submission. Then, for each mutated program, we applied all three types of bugs: WCO, VM and ME. Finally, we gathered a dataset of pairs of programs and the mappings between their sets of variables. As Figure C.1 shows, each pair of programs, in our generated dataset, corresponds to a correct student's implementation and the student's program after being mutated and with some bug introduced.

## C.2 #Correct/Incorrect Mappings vs #Variables

Figure C.2 shows a histogram with the number of programs, $y$-axis, whose variables (number of variables in the $x$-axis) our GNN models can map totally correct (#Correct Mappings) in green and programs with at least one variable being mapped incorrectly (#Incorrect Mappings) in red.

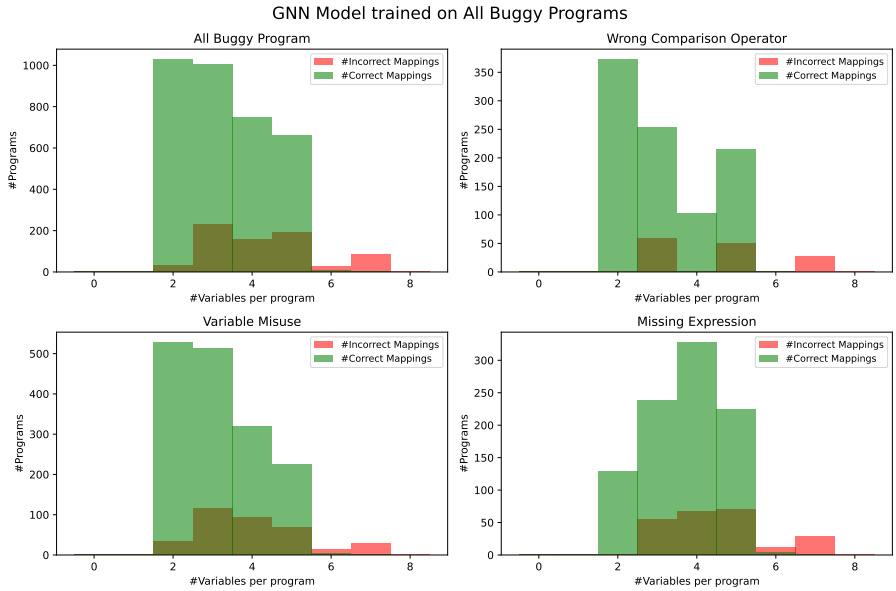



Figure C.2: Histograms showing the number of programs that our GNN, trained on all buggy programs, mapped all their variables correctly. The results are presented for programs with the bugs of wrong comparison operator (WCO), variable misuse (VM), missing expression (ME) or all of them (All).

## C.3 Overlap Coefficient

The overlap or SzymkiewiczSimpson coefficient measures the overlap between two sets (e.g., mappings). This metric can be calculated by dividing the size of the intersection of two sets by the size of the smaller set, as follows:

$$overlap(A, B) = \frac{|A \cap B|}{min(|A|, |B|)} \tag{C.1}$$

An overlap of $100\%$ means that both sets are equal or one of them is a subset of the other. The opposite, $0\%$ overlap, means there is no intersection between both sets.