Can Test Generation and Program Repair Inform Automated Assessment of Programming Projects?

1st Given Name Surname
dept. name of organization (of Aff.)
email address or ORCID

2nd Given Name Surname
dept. name of organization (of Aff.)
email address or ORCID

3rd Given Name Surname
dept. name of organization (of Aff.)
email address or ORCID

***Abstract*—Computer Science educators assessing student programming assignments are typically responsible for two challenging tasks: grading and providing feedback. Producing grades that are fair and feedback that is useful to students is a goal common to most educators. In this context, automated test generation and program repair offer promising solutions for detecting bugs and suggesting corrections in students' code which could be leveraged to inform grading and feedback generation. Previous research on the applicability of these techniques to simple programming tasks (e.g., single-method algorithms) has shown promising results, but their effectiveness for more complex programming tasks remains unexplored. To fill this gap, this paper investigates the feasibility of applying existing test generation and program repair tools for assessing complex programming assignment projects. In a case study using a real-world Java programming assignment project with 296 incorrect student submissions, we found that generated tests were insufficient in detecting bugs in over 50% of cases, while full repairs could only be automatically generated for only 2.1% of submissions. Our findings indicate significant limitations in current tools for detecting bugs and repairing student submissions, highlighting the need for more advanced techniques to support automated assessment of complex assignment projects.**

***Index Terms*—Automated Assessment, Test Generation, Program Repair**

I. Introduction

Learning computer programming is challenging, and effective assessment in higher education is key for helping students track progress and learn from mistakes. In modern Computer Science education, assessing programming assignments usually involves grading and providing feedback, both critical for students to understand their progress, identify errors, and enhance their skills [1]. Traditionally, educators manually or semi-automatically evaluate each student submission to determine correctness and provide personalized feedback [2]. However, this is a time-intensive task, particularly with large classes with hundreds of students and tight academic schedules. To alleviate manual efforts, educators seek automation for various assessment tasks, e.g., automated grading [3], feedback generation [4], and synthesis of code explanations [5].

Test-based grading is one of the primary approaches for evaluating student submissions in a (semi-)automated way [1]. Typically, as part of the assignment creation process or prior to the submission deadline, educators manually craft a test suite that embodies the assignment requirements to assess the correctness of student submissions. These tests aim to capture key functionalities that students are expected to implement in their submissions. The outcome of executing this test suite against each student submission can then be mapped to grades given to students (e.g., top grades if all tests pass) and test failure messages can be used as feedback.

While popular, the above test-based grading approach has shortcomings, not least because manually creating a high-quality test suite can be very costly for educators. Tests with low granularity (i.e., those that only assess broad functionalities) may fail to detect faults in student submissions, potentially leading to incorrect judgments [3]. On the other hand, writing tests with high granularity (i.e., covering all possible test scenarios and corner cases) requires a lot of effort and time not always available to educators.

To address the challenges of manual test creation, automated test generation techniques have been proposed as a means to reduce manual effort and increase test quality [6]. The expectation is that test generation can enhance test granularity, leading in turn to more accurate fault detection and fairer, more precise evaluation of student submissions. Additionally, useful feedback could be derived from the generated tests by reporting the failing tests and error messages to students. Consequently, automated test generation offers a valuable proxy for both grading and feedback in programming assignments.

But even when high-quality (incl. automatically generated) tests are available, the feedback provided by failing tests is often insufficient for students, particularly novice programmers, who struggle to map failing test executions back to specific errors in their code [4]. One approach to mitigate this involves leveraging automated program repair (APR) techniques. APR has seen rapid development in recent years, showing applicability in industrial settings to repair real-world faults [7] and also in educational contexts, offering more meaningful feedback to students for introductory programming assignments [8]. APR takes a buggy program with tests, including failing ones, as input and attempts to change the original program so that all the tests pass. In education, program repair can provide students with more actionable feedback by recommending potential corrections (i.e., repairs) for mistakes in their code. This feedback offers students the knowledge about how to proceed, helping to address their mistakes and guiding them toward a complete solution [1].

Although automated test generation and program repair have potential applicability in grading and feedback generation tasks to inform automated assessment, their effectiveness has been evaluated primarily on *introductory* programming assignments. These assignments typically require the implementation of a single-method routine using basic control flow and data structures (e.g., *"sort the elements in an input array"*). Common datasets used in these evaluations, such as those from massive open online courses (MOOCs), the IITK dataset [9], and a subset of the FalconCode dataset [10], fit this profile [6, 8, 11]. To the best of our knowledge, the application of test generation and program repair for automated assessment in the context of more complex programming assignments has not been investigated to date.

In this work, we seek to establish the feasibility of using existing test generation and program repair techniques to underpin automated assessment for more sophisticated *programming assignment projects* [10, 12]. These projects represent a step up in complexity from introductory assignments, often requiring the application of more advanced programming concepts (e.g., object-oriented programming and design patterns), and the implementation of multiple code files with interconnected functionalities. Consequently, they present further challenges for both grading and feedback generation. Can existing automated test generation and program repair techniques keep up and support educators in detecting faults and repairing incorrect submissions with this increased level of complexity?

We conduct and report on a case study using a dataset from a *real-world* programming assignment project from a Java Programming university course. The dataset contains 296 *incorrect* student submissions and includes a reference solution and test suites manually created by the course educator. We select the EvoSuite [13] state-of-the-art Java unit test generation tool as the representative of test generation tools. EvoSuite is designed to create tests that maximize code coverage and contain assertions capturing the behavior of the software under test. We compare their ability to detect bugs in student submissions against educator-written tests. As for program repair, we evaluate the extent to which existing tools can successfully repair incorrect student submissions. We chose ARJA-e [14], an advanced tool that employs a diverse set of repair operations and can handle various faults, making it well-suited to address diverse errors that may be present in student submissions. Furthermore, given the recent advent of large language models (LLMs) and their capability in code-related tasks, we also include the open-source Qwen2.5 LLM [1] in our evaluation. We run Qwen2.5 locally and deliberately avoid more advanced but closed-source LLMs like GPT-4[2] to prevent data privacy issues; closed-source models carry a risk of prompt leakage, which could expose sensitive content, including assignment details or student submissions [15, 16]. We use the LLM for both test generation and program repair tasks and compare its performance to resp. EvoSuite and ARJA-e, exploring the potential of LLMs in informing automated assessment for programming projects.

Our evaluation results show that, compared to the educator tests, the tests generated by both EvoSuite and Qwen2.5 are less effective in detecting bugs in over 50% of the 296 incorrect submissions. The generated tests perform equivalent to or outperform the educator tests in terms of bug detection only in 2% of the cases. For program repair, ARJA-e and Qwen2.5 demonstrated similar results, with ARJA-e fully and partially repaired 2.3% and 7.3% of the incorrect submissions, while Qwen2.5 resulted in 0.6% and 8.5%, respectively. The results indicate that while current test generation and program repair tools can inform assessments for introductory programming assignments, they are not yet adequate for more complex programming assignment projects.

To better understand the limitations of existing tools, we further explore the reasons behind these results. The findings highlight several factors, mostly related to the nature of assignment projects, such as code across multiple files and the presence of nuanced functionalities. Based on the insights, we produce practical guidelines for educators regarding using test generation and program repair tools to support assessment in assignment projects.

The main contributions of this paper are as follows:

(1) We conduct a case study on 296 incorrect student-submitted programming projects to investigate the feasibility of automated assessment. The results show that existing test generation and program repair tools are insufficient for effectively handling assignment projects.

(2) We investigate the reasons behind the insufficiency of these tools and discuss practical implications for their use. These insights aim to enhance educators' understanding of test generation and program repair techniques in more complex programming assignments and to provide guidelines for effectively utilizing them in assignment projects.

This paper is structured as follows. Section II presents the background of this work, including test generation and program repair, different categories of programming assignments, and LLMs. Section III describes the selection of tools and the methodologies used for test generation and program repair. Section IV presents the experimental results. Section V discusses the implications and findings we derived from our study, Section VI presents related work, and Section VII concludes the paper and outlines future work directions.

II. Background

A. Test Generation

Manually writing test cases can be labor-intensive, which has prompted the development of various automated test generation techniques with a focus on efficiency in producing tests that achieve high code coverage and are effective at detecting faults. In the Java domain, Randoop [17] is well-known as a random unit test generation tool, and EvoSuite [13] is arguably the state-of-the-art search-based unit test generation tool. Both tools generate executable tests in JUnit [3] format, and both are capable of generating tests that achieve high coverage [18], but empirical evidence suggests EvoSuite is superior in terms

[1] Qwen2.5-Coder-7B-Instruct: https://huggingface.co/collections/Qwen

[2] https://openai.com/index/gpt-4/

[3] https://junit.org/

of fault effectiveness [19]. In a nutshell, given a Java class under test, EvoSuite starts by creating a *population* of test suites, each containing a random number of randomly created tests, *evolves* them using customary genetic operators and code coverage as optimization goal (*fitness function*) until the coverage criterion is met (e.g., full branch coverage) or the given budget (usually time) is exhausted. The result is the test suite with the highest code coverage, with each individual test enhanced with assertions for fault detection.

Writing unit tests to evaluate the functionality of student submissions is common practice among educators. However, this approach often leads to inaccurate grading and difficulties for students in mapping the failing tests to specific errors in their code [4]. Compared to manually checking the functionality solely by the program's outcome, unit testing offers more granularity of evaluation by assessing classes, methods, and statements within the program [1]. With many automated assessment tools leveraging testing for grading and providing feedback, test generation can assist educators in producing more granular and reliable assessments [20].

B. Program Repair

1) General Purpose Program Repair: General purpose program repair (hereafter, gPR) techniques aim to automatically fix bugs in real-world software. A variety of gPR approaches have been proposed over the years, including search-based, semantic-based, and deep learning-based techniques. As one of the most prominent gPR approaches, search-based repair, also known as *generate-and-validate*, takes a program and a test suite with at least one failing test as input (i.e., a test revealing that a given program does not implement a certain expected behavior). The approach first *generates* repairs by exploring a search space consisting of *repair ingredients* (i.e., code that can be used to generate repairs) and then *validates* the repairs against the given test suite. A valid repair should pass *all* the tests in the suite.

Most search-based repair approaches are *redundancy-based*, meaning they assume repair ingredients that could fulfill the tests already exist within the program under repair [21]. Existing prime examples of this type of approach include GenProg [22], ARJA [23], and PAR [24], which all leverage heuristic algorithms, such as genetic programming [25].

2) Educational Program Repair: gPR techniques typically assume the programs under repair are *mostly* correct, which limits their suitability for programming assignments for two primary reasons. First, compared to gPR defect benchmarks (e.g., Defects4J [26]) student submissions for programming assignments are relatively small, providing *limited repair ingredients*[4]. Second, student submissions often have high test failure rates, with multiple errors requiring *complex repairs*, which poses significant challenges for gPR [9]. While larger programming assignment *projects* may offer more repair ingredients, their complexity, especially with faults across multiple files, introduces further repair challenges.

To address these limitations, program repair approaches specially tailored to the programming assignment repair problem have been proposed; we refer to them as educational program repair (hereafter, ePR). Unlike gPR, ePR techniques tend to avoid solely relying on incorrect programs and test suites; instead, they assume the availability of (possibly multiple) reference solutions (e.g., educator-developed solutions or fully correct student solutions with respect to given tests) and leverage them to repair incorrect student submissions [8, 11, 28]. Existing ePR approaches primarily focus on *introductory* assignments, which are relatively simple in terms of both the programs themselves and their accompanying tests [11, 9].

C. Programming Assignments

Programming assignments vary in difficulty and can be categorized into three groups [10, 12, 29][5]: (1) *Introductory*: requiring students to use basic coding skills to complete a very specific task in a single file (e.g., find the maximum value in an array), (2) *Intermediate*: requiring students to use further coding skills, such as data structures and algorithms implemented across multiple methods (e.g., depth-first search using recursion), and (3) *Advanced*: requiring students to implement larger, more complex *projects* across multiple files (e.g., building an interactive Tic-tac-toe game).

Most existing ePR techniques focus on repairing introductory programming assignments, usually involving developing a single function. These programs are evaluated using simple input-output tests to assess correctness (e.g., "in: [3, 4], out: 4"). Such assignments are found in the IITK dataset [9], which is commonly used to evaluate ePR techniques, such as Clara [8], Refactory [11], and Verifix[28].

Compared to introductory ones, intermediate assignments involve problem-solving, algorithmic thinking, and optimization techniques, allowing students to handle larger and more complex code files [29]. Despite the increased complexity, these assignments often maintain a single-file structure with input-output testing, like the introductory ones. Defects4DS [29], a dataset comprising data structures and algorithms assignments, belongs to the intermediate level.

Advanced assignments often require developing a software project with multiple classes and methods, and tackling a set of problems, each exercising some functionality within the program. Taking the Tic-tac-toe game as an example, it may include tasks such as implementing the game mechanism, managing player interactions, and determining the winner.

Unlike simpler assignments, these projects use dedicated unit tests for assessing the program's correctness [10], commonly developed by comprehensive testing frameworks (e.g., JUnit for Java[6], unittest for Python[7]). Unit testing treats the

[4] There is informal evidence from users of the Astor [27] program repair library suggesting gPR tools may not directly apply to repairing programming assignments: https://github.com/SpoonLabs/astor/issues/155.

[5] These groups are alternatively termed as "skills", "labs", and "projects" by de Freitas et al. [10].

[6] https://junit.org/

[7] https://docs.python.org/3/library/unittest.html

program as a black box, interacting with and evaluating various program methods, asserting their expected behaviors.

D. Large Language Models

Large Language Models (LLMs) refer to a category of advanced, large-scale models pre-trained on extensive datasets, enabling them to generate human-like responses by predicting the next token given some text prefix. This training approach leverages the vast quantities of text available on the web, equipping LLMs with strong capabilities in natural language tasks such as conversation and reasoning [30]. Code language models (CLMs) or large language models trained on code (LLMCs), such as Codex from OpenAI[8], are a specialized subset of LLMs designed specifically for code-related tasks, such as code summarization, generation, and program repair [31].

Despite their promising performance across various tasks, LLMs are prone to two major problems: non-determinism and hallucinations [32, 33]. Non-determinism refers to the inconsistency in output when identical prompts yield different responses across multiple requests. Inconsistent responses, like generated code, can undermine reliability during the software development process [32]. Meanwhile, hallucinations describe LLMs' tendency to produce outputs that deviate from user intent, contain internal inconsistencies, or misalign with factual knowledge. This makes the LLMs' deployment potentially risky in many domains [33], including education where accuracy in grading and feedback is paramount.

III. Methodology

To explore the potential of automated test generation and program repair in the automated assessment of programming assignment projects, we conduct an empirical case study using a real-world project following the software engineering case study guideline [34]. Specifically, we aim to answer the following research questions:

RQ1 To what extent can existing *test generation techniques* inform automated assessment by detecting more buggy behaviors in programming assignment projects?

RQ2 To what extent can existing *program repair techniques* inform automated assessment by repairing more buggy behaviors in programming assignment projects?

A. Dataset

Our dataset is based on a programming assignment project from a Java Programming class for computer science undergraduate students; it includes a reference solution and a test suite both written by the educator as well as 299 student submissions, only three of which are fully correct.

The assignment tasked students to implement a coffee maker system, requiring features such as printing to console, conditionals, loops, arrays, inheritance, and exception handling. Students are provided with skeleton code containing fixed class names and fixed method names and signatures. While they are free to implement additional private methods, they are asked not to change the signatures of public ones.

The reference solution consists of 13 classes and 32 methods, 11 of which are considered *focal methods*, i.e., methods that students must implement in their submissions.

The educator-written test suite consists of 7 test classes with a total of 60 unit tests. Although not necessarily designed to achieve 100% code coverage, we assume this test suite adequately validates the expected behaviors of the program since it is carefully designed as the minimal requirement for achieving a perfect grade (e.g., 100 out of 100). However, the reference test suite may not sufficiently cover all test scenarios and corner cases within the student submissions; therein lies the opportunity of complementing this educator-written test suite with automatically generated tests.

B. Test Generation Tools

To investigate whether automated unit test generation can inform a more fine-grained assessment of the functionality implemented in programming assignments, we aim to select a representative traditional test generation tool for Java and a suitable LLM. As the representative for traditional test generation tools for Java, between Randoop and EvoSuite we choose EvoSuite due to its higher effectiveness in detecting real faults [35, 19], which can potentially contribute to more precise grading of programming assignment projects, and the more manageable nature of the unit tests it produces (minimized for coverage and usually shorter and more readable).

To identify an LLM suitable for test generation for programming assignment projects, we consider both the complexity of instructions involved, as described in Section III-C, and the ethical concerns of model selection. Closed-source LLMs, like GPT-4, pose risks related to potential misuse, including the possibility of training on user prompts and inadvertently leaking assignment content or solutions, where such risks could lead to student plagiarism [16, 15]. To mitigate these risks, we focus on selecting an open-source model with a relatively small size (i.e. number of parameters) that is feasible to run a consumer-grade personal computer. Based on benchmarks of the code generation capabilities under complex instructions, CodeQwen1.5-Chat-7B emerged as a strong candidate among models with approximately 7B parameters [36]. Considering the rapid advancements in LLMs, we choose an updated version of CodeQwen, Qwen2.5-Coder-7B-Instruct, which we operate locally using the Ollama framework [9].

C. Test Generation Methodology

The goal of using test generation is to create test suites that effectively detect bugs in student submissions, thereby providing detailed test results for automated assessment [6]. To achieve this, we run the two aforementioned test generation approaches (EvoSuite and LLM) using the educator's reference solution as input. The set of tests obtained as output (i.e., capturing the expected behavior) are then run against the students' submissions.

To evaluate whether test generation techniques can improve automated assessment for programming assignment projects,

[8] https://openai.com/index/openai-codex/

[9] https://ollama.com/

we examine if the tests generated by these techniques can match or even exceed the educator-developed test suite (hereafter, educator tests) in identifying bugs within student submissions. Specifically, we categorize each of the incorrect student submissions into five cases: (1) *Non-detection*: the generated tests fail to detect any of the buggy methods in a given incorrect student submission that the educator-written tests do identify; (2) *Insufficient*: the generated tests detect a non-empty subset of the buggy methods that the educator tests identify; (3) *Complementary*: generated and educator tests detect unique buggy methods missed by the other; (4) *Equivalent*: generated and educator tests detect the same buggy methods; (5) *Outperform*: the generated tests detect a superset of the buggy methods the educator tests identify.

When evaluating the bug detection ability of the generated tests, we focus on the 11 focal methods directly related to the tasks being assessed and contribute to students' grades. We isolate these methods from others, including skeleton code or any custom methods students create. Given the project's interdependency nature, isolating the focal methods minimizes cross-method interactions, easing the accurate detection of buggy methods.

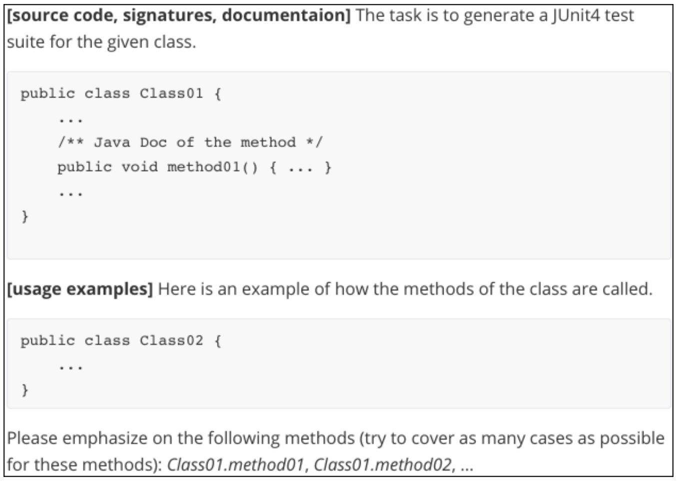
For test generation using EvoSuite, we applied the default settings to generate tests for all the methods within the assignment project. Due to the inherent randomness in EvoSuite's evolutionary algorithms [13], we ran five repetitions and averaged the results to better reflect its performance.

For Qwen2.5, we follow the unit test generation strategy proposed by Schäfer et al. [37], where the prompts contain (1) method signatures, (2) method documentation, (3) method usage examples, and (4) method source code. We present each class individually to the LLM, containing methods with their signatures, and request that it generate a test suite. Each method that comes with the skeleton code includes educator-written JavaDoc to clarify its intended functions. As additional context, we provide the source code of another class that frequently interacts with the class under test (e.g., via object instantiation and method calls) as a usage example. For classes with focal methods, we explicitly ask the LLM to focus on achieving high coverage on them. Figure 1 illustrates the complete prompt template. We use the default settings of Qwen2.5. If LLM-generated tests fail to compile due to missing library imports, we manually add them; otherwise, those tests are commented out (discarded) [37]. We also ensure the reference solution passes all the LLM-generated tests.

For our test generation using EvoSuite and LLM, as well as program repair with LLM (§III-E), we conducted experiments on a MacBook Pro with Apple M3 Pro Chip, 36 GB memory, and macOS Sequoia 15.0.1.

D. Program Repair Tools

Existing ePR tools, such as Clara [8], Refactory [11], and Verifix[28], do not apply to programming assignment projects, as discussed in Section II-B2, and thus, we exclude them from our study. Instead, we exhaustively consider the gPR tools in the community-driven online program repair catalog at https://program-repair.org/. However, gPR tools need to be adapted to fit the context of programming assignments. Inspired by ePR tools, we aim to equip a gPR tool with the ability to leverage a reference solution. As discussed in Section II-B, redundancy-based repair techniques are a primary category of search-based repair tools, which can repair an incorrect program by rearranging and transforming its existing code [21]. In this context, components from a reference solution can be utilized as promising repair ingredients if they are presented in the incorrect program (more details will come in Section III-E)). Accordingly, we focused on redundancy-based repair tools during the tool selection.



[source code, signatures, documentaion] The task is to generate a JUnit4 test suite for the given class.

```
public class Class01 {
    ...
    /** Java Doc of the method */
    public void method01() { ... }
    ...
}
```

[usage examples] Here is an example of how the methods of the class are called.

```
public class Class02 {
    ...
}
```

Please emphasize on the following methods (try to cover as many cases as possible for these methods): *Class01.method01*, *Class01.method02*, ...

Fig. 1: Test generation prompt for LLM

We applied the following inclusion criteria: (1) applicable to Java programs, (2) publicly available, (3) not restricted to specific fault types (e.g., concurrency), (4) not restricted to specific repair scopes/operators (e.g., conditional statements), (5) not exclusive for benchmarks (e.g., Defects4J), (6) use a redundancy-based approach, (7) executable without compilation errors (or errors solvable with reasonable efforts). By adhering to these criteria, we ensure that the candidate tools were broadly applicable for repairing programming assignment projects. The shortlisted tools are ARJA [23], ARJA-e [14], jGenProg [27], Cardumen [38], and kGenProg [39][10].

To validate our choices, we conducted a preliminary evaluation of the shortlisted tools using IntroClassJava [40], a benchmark consisting of 297 student-written Java programs from an introductory programming course [41]. Previous studies indicate that ARJA outperforms both jGenProg and Cardumen in terms of the number of repaired programs [42]. Building on top of this, we evaluated ARJA-e and kGenProg on IntroClassJava, finding that ARJA-e surpassed the others in terms of the number of repaired programs. In addition, ARJA-e contains a rich collection of repair templates, such as adjusting method parameters, which is particularly useful for projects with frequent method calls. Given the time-consuming

[10] Detailed inclusion process available at https://github.com/ruizhengu/ICST-2025-Assignment-Projects.

nature of running traditional program repair tools against all submissions from a programming assignment (experimental setup and runtime are detailed in Section III-E), we chose ARJA-e as the representative tool.

Given the capabilities of LLMs across various code-related tasks, including program repair [31], we continue using Qwen2.5 as the LLM-based tool for the evaluation.

E. Program Repair Methodology

The objective of using program repair in our context is to modify incorrect student submissions so they pass all tests. We aim to mitigate some of the limitations of gPR and ePR discussed in Section II-B, while accommodating the unique characteristics of programming assignment projects described in Section II-B2.

We introduce valid repair ingredients from a reference solution into incorrect student submissions (cf. §III-D) and employ a method-level incremental repair strategy. This approach allows the repair of one buggy method at a time, minimizing the number of failed tests and the need for complex, multi-file repairs. The process takes an incorrect student submissions and a reference solution as input and attempts to repair each buggy method so that it passes its associated tests.

1) Reference Solutions: Previous work highlights a trade-off in program repair between expanding the search space with likely valid repair ingredients and the ability to produce accurate repairs [43]. Aiming for balance, we selectively incorporate only key *components* (i.e., correct implementations of specific buggy methods) from a reference solution rather than the entire solution. This approach helps limit redundant additions to the search space, optimizing repair efficiency.

2) Incremental Repair: Student submissions can be substantially incorrect, particularly in programming assignment projects where incorrect assumptions and subsequent programming mistakes may affect multiple classes and methods. In such cases, program repair techniques may struggle to generate complex fixes involving multiple code chunks and failing tests [9]. To alleviate this, we employ a method-level incremental repair strategy to repair one buggy method at a time. It helps simplify the repair process by breaking it down into smaller, manageable tasks. By confining repair generation to individual buggy methods, we reduce the number of failed tests and the search space size for potential repairs [43].

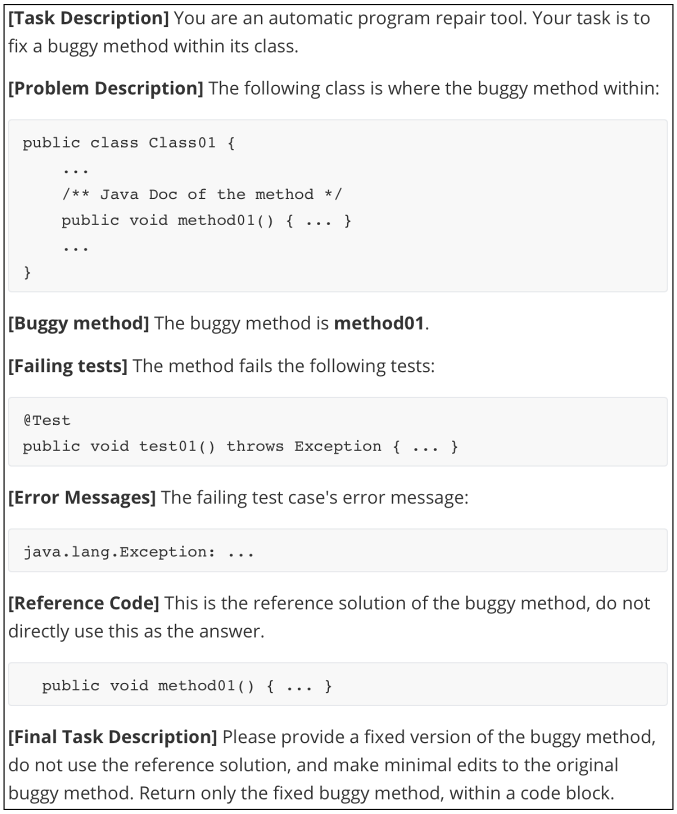
In the incremental repair process, we first identify a set of buggy methods from an incorrect submission based on the provided tests and the focal methods. We then isolate one buggy method as the *method under repair* and create an *intermediate submission* (hereafter referred to as *intermediate*) as the target for repair generation. To create an intermediate, we *replace* all buggy methods except the method under repair with their correct implementations from a reference solution. We also *add* the correct implementation of the method under repair to the intermediate. This approach limits the repair scope within the method under repair and introduces repair ingredients specifically targeting that method. By identifying all the buggy focal methods from 296 incorrect submissions, we derive a total of 2126 intermediates.

3) Experiment Setup and Procedure: We first run ARJA-e standalone (without reference solution or incremental repair) on 296 incorrect student submissions, followed by execution with the educator solution as reference and apply incremental repair against 2126 intermediates. To manage this resource-intensive task, we use a high-performance computing machine equipped with 2 x 32-core Intel Xeon Platinum 8358 CPUs and 256 GB of RAM, assigning about 70 intermediates to each of the 30 nodes. With a 10-minute time budget per ARJA-e execution, the total task duration is approximately 30 hours.

As programming assignments grow in size and complexity, the educator solution is not necessarily *the only* correct implementation; multiple valid solutions may exist. Furthermore, different correct student solutions can offer unique patterns that serve as valuable repair ingredients [23]. As developing reference solutions is time-consuming for educators, we also explore using correct student solutions as references.

To investigate how variations in reference solutions affect program repair effectiveness, we additionally consider three correct student solutions (CS1, CS2, CS3) as references, repeating the experiment three times. We assess the repair effectiveness of the traditional repair tool across three metrics: the number of fully repaired submissions (where all buggy methods are fixed), partially repaired submissions (at least one buggy method is repaired), and the total number of buggy methods repaired, using the educator solutions and three correct student solutions as references. To ensure valid outcomes, we excluded repairs generated by ARJA-e containing `System.exit(0);` to prevent early program termination.

Given that longer instructions tend to generate more buggy code by LLMs [32], we adopt a similar approach to augmenting LLM-based repair with reference solution and incremental repair used for ARJA-e. To avoid overwhelming the LLM with extensive prompts containing entire student programs across multiple code files, we only provide relevant context. For each intermediate, we provide the LLM with the class containing the buggy method and ask it to repair the method. We combine the prompting strategies from Silva and Monperrus [44] and Zhao et al. [29], structuring the prompt with the following components: (1) task description, introducing the program repair task, (2) problem description, including the class with the buggy method, its dependencies with other class methods or variables, and JavaDocs capturing the intended functionality for each method written by the educators, (3) failing tests, a list of test cases that the buggy method fails to pass, (4) failing tests' error message, detailed runtime error information to help identify faults, (5) reference code, the correct implementation of the buggy method from the reference solution, and (6) final task description, instructions to avoid directly using the reference code and return the corrected method. With both the reference solution and tests sourced from educators, this structured prompt ensures that LLM has sufficient contextual information while discouraging direct copying of the reference solution. The complete prompt



[Task Description] You are an automatic program repair tool. Your task is to fix a buggy method within its class.

[Problem Description] The following class is where the buggy method within:

```
public class Class01 {
    ...
    /** Java Doc of the method */
    public void method01() { ... }
    ...
}
```

[Buggy method] The buggy method is **method01**.

[Failing tests] The method fails the following tests:

```
@Test
public void test01() throws Exception { ... }
```

[Error Messages] The failing test case's error message:

```
java.lang.Exception: ...
```

[Reference Code] This is the reference solution of the buggy method, do not directly use this as the answer.

```
  public void method01() { ... }
```

[Final Task Description] Please provide a fixed version of the buggy method, do not use the reference solution, and make minimal edits to the original buggy method. Return only the fixed buggy method, within a code block.

Fig. 2: Program repair prompt for LLM

template is illustrated in Figure 2.

There are often multiple ways to implement the required functionalities, and similarly, various approaches can be taken to repair incorrect submissions. To encourage the LLM to generate creative and varied repairs that can better address the diversity in student submissions, we allow for non-deterministic outputs by setting Qwen2.5's temperature to 0.8. The temperature between 0 and 2 controls the output's randomness, where higher values make the results more varied, while lower values lead to more focused outputs [32].

To evaluate the effectiveness and consistency of Qwen2.5 for repair, we compute the Pass@k metrics, commonly used to evaluate LLM code generation [45, 12] For each intermediate, we generate $n = 5$ repair attempts. For a given k ($k \leq n$), we assess all combinations of k repairs from the n attempts, counting those with at least one successful repair (i.e., one that passes all tests). For instance, in Pass@1, if the first repair attempt is successful, it contributes to the total Pass@1 result.

We compute Pass@1, Pass@3, and Pass@5 metrics for LLM-generated repairs, covering: (1) fully repaired submissions: the number of incorrect submissions where all buggy methods are fixed in at least one combination of 5 repair attempts; (2) partially repaired submissions: the number of submissions with at least one buggy method is fixed in at least one combination of 5 attempts attempts; (3) buggy methods repaired: the total number of individual buggy methods repaired in at least one combination of 5 attempts

TABLE I: Coverage from educator and generated tests

Coverage Type	*Edu*	*EvoSuite*	*LLM*
Instruction	73%	94%	70%
Branch	79%	91%	63%

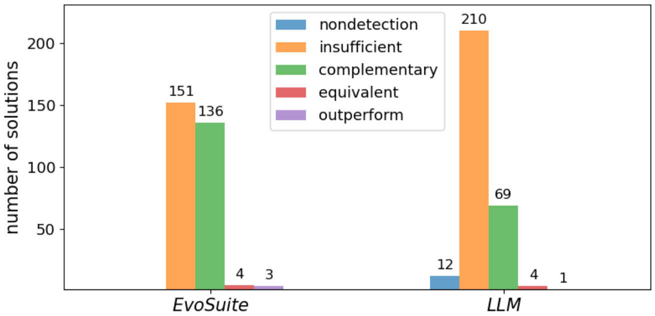


Fig. 3: Test effectiveness compared to educator tests

To further validate LLM-generated repairs, we examine whether the LLM genuinely repaired the incorrect submissions rather than copying the reference solution included in the prompt. To account for this, we manually review all LMM-generated repairs to identify cases where the repairs are either *identical* or *similar* to the reference solution.

IV. Results

In this section, we present the results of our study and answer the research questions formulated in Section III.

A. RQ1: Test Generation

Table I shows the instruction and branch coverage results. On average, EvoSuite ($EvoSuite$) generated 99 tests per run, compared to 60 educator tests (Edu) and 44 LLM-generated tests (LLM). $EvoSuite$ achieved the highest instruction and branch coverage, followed by Edu, while LLM exhibited the lowest coverage in both areas.

Figure 3 shows the bug detection results for $EvoSuite$ and LLM against Edu. Notably, LLM failed to detect bugs in 12 out of 296 incorrect submissions. Overall, both $EvoSuite$ and LLM showed insufficient bug detection compared to Edu, with few cases ($\sim$2%) where they were equivalent or outperformed Edu. Although many cases fell under the complementary category, Edu still demonstrated superior performance regarding the number of buggy methods detected. Specifically, $EvoSuite$ detected 180 buggy methods across all incorrect submissions, compared to 543 for Edu, which is a threefold increase. Similarly, LLM detected 80 buggy methods, while Edu detected 389, nearly 5 times higher.

These findings indicate that the generated tests are generally insufficient in detecting bugs in student submissions compared to educator tests. Even in complementary cases, educator tests consistently detect significantly more bugs, demonstrating the limited effectiveness of test generation for evaluating programming assignment projects.

TABLE II: Traditional program repair results with different reference solutions

Ref	S_{FR} (out of 296)	S_{PR} (out of 296)	M_{R} (out of 2126)
ES	7 (2.4%)	19 (6.4%)	32 (1.5%)
CS1	8 (2.7%)	19 (6.4%)	31 (1.5%)
CS2	6 (2%)	18 (6.1%)	28 (1.3%)
CS3	6 (2%)	18 (6.1%)	25 (1.2%)

TABLE III: LLM-generated repair results

Metrics	Pass@1	Pass@3	Pass@5
S_{FR} (out of 296)	2 (0.6%)	2 (0.6%)	2 (0.6%)
S_{PR} (out of 296)	25 (8.4%)	25 (8.4%)	26 (8.8%)
M_{R} (out of 2126)	73 (3.4%)	82 (3.9%)	86 (4.0%)

Answer to RQ1: The generated tests are insufficient in detecting bugs in over 50% of cases compared to the educator tests, and only ∼2% of cases show them as equivalent or superior. In complementary cases, the educator tests detect about 4 times more buggy methods than the generated tests.

B. RQ2: Program Repair

In executing ARJA-e standalone, only 2 out of 296 incorrect student submissions were repaired. However, with the educator solution as a reference and applying incremental repair, ARJA-e fully repaired 7 incorrect submissions. Using three correct student solutions as references shows consistent performance.

Table II summarizes the results, where S_{FR}, S_{PR}, and M_{R} denotes the number of *fully repaired* submissions (i.e., all buggy focal methods in the incorrect submission are repaired), *partially repaired* submissions (i.e., at least one buggy focal method in the incorrect submission is repaired), and the number of repaired methods among 2126 buggy focal methods, respectively. All four reference solutions yielded comparable results, where they contribute approximately 2.3% S_{FR} and around 6.2% S_{PR} out of 296 incorrect submissions, and about 1.4% M_{R} for 2126 buggy focal methods.

Table III presents the results of LLM-generated repairs, excluding 76 repair results identical to the reference solution (e.g., minor changes in variable names) as discussed at the end of Section III-E. Qwen2.5 demonstrated similar results to ARJA-e in terms of S_{PR}, where it achieved around 8.5% and ARJA-e at about 6.2% However, Qwen2.5 achieved a lower S_{FR}, reaching only 0.6%, compared to ARJA-e's 2.3%. Notably, Qwen2.5 significantly outperforms ARJA in M_{R}, with approximately 5.7% compared to ARJA-e's 1.4%. The results suggest that while LLM shows better results in repairing individual buggy methods, it may struggle with fully repairing the submission.

For both traditional and LLM-based program repair tools, the overall number of successful repairs remains low, indicating that current program repair techniques are inadequate for assessing programming assignment projects.

```java
public void test09()  {
    Recipe recipe0 = new Recipe("", 0);
    Recipe recipe1 = new Recipe("", 0);
    assertEquals(recipe0, recipe1);}
```

(a) EvoSuite-generated test

```java
public void testRecipeMultipleIngredients() {
    Recipe recipe0 = new Recipe("", 0);
    recipe0.addIngredient(new Coffee());
    Recipe recipe1 = new Recipe("", 0);
    recipe1.addIngredient(new Coffee());
    assertEquals(recipe0, recipe1);}
```

(b) Educator test

Fig. 4: A case where EvoSuite is insufficient compared to the educator test

Answer to RQ2: Traditional and LLM-based repair approaches fully repaired about 2.3% and 0.6% of the 296 incorrect student submissions and partially repaired around 7.3% and 8.5%, respectively. For 2126 buggy methods, the traditional approach repairs approximately 1.4%, whereas the LLM achieves a higher repair rate of 5.7%.

C. Threats to Validity

The process of selecting, building, and executing the program repair tools was mostly manual, which introduces a potential threat to internal validity due to human errors. To mitigate this threat, we follow a systematic approach to avoid biases during the selection process (§III-D). However, future work should explore a broader range of program repair tools to improve the generalizability of our findings, especially as LLM-based repair tools are rapidly evolving.

ARJA-e has a technical limitation in that it cannot identify faults in constructors. This limitation poses an internal validity threat as student submissions may have faults in constructors that ARJA-e cannot resolve. Although this limitation may impact the results of repair generation in our experiments, it falls out of the scope of our research. To maintain fairness in comparing ARJA-e with LLM, we did not instruct the LLM to specifically consider faults in constructors.

V. Findings and Discussions

In this section, we discuss the findings and implications derived from the experimental results to provide guidelines for improving existing test generation and program repair techniques for assessing programming assignment projects.

A. Test Generation

We first investigate why EvoSuite achieves higher code coverage than educator tests yet remains less effective in detecting student bugs. EvoSuite's limited effectiveness likely stems from lacking contextual knowledge about the assignments that the educator possesses when crafting tests. This knowledge allows educators to anticipate common student mistakes and incorporate them into the tests.

Figure 4 shows an example of such a case. Both EvoSuite and educator tests target the `Recipe.equal` method, which compares two `Recipe` objects based on default properties (e.g., name and size) and contained `Ingredient` objects. In Figure 4a, the EvoSuite test did not add any ingredients to the recipes, whereas highlighted in Figure 4b, the educator test did account for this by adding ingredients to the receipts when determining their equality. This example highlights the unique nature of programming assignment projects, where dependencies exist within programs, compared to introductory and intermediate assignments. With assignment-specific insights, educators can design tests that cover these interdependencies, thus detecting student error more effectively.

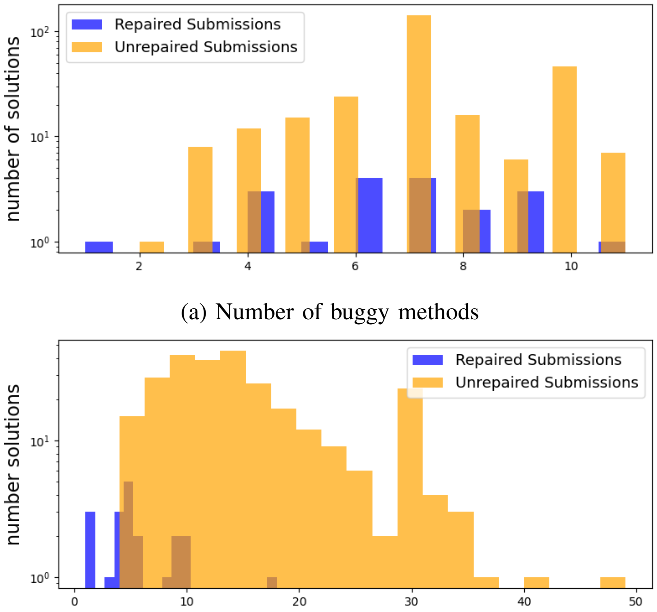
These findings suggest that current test generation techniques are not yet able to replace educator-written tests. However, research shows potential in using these tools to *augment* existing tests, rather than generating new ones from scratch [46]. To explore this in an educational setting, we augmented the educator tests with Qwen2.5.

By using the LLM to augment the existing educator tests, we expanded the test suite from 60 to 100 cases. The augmented tests slightly increased the coverage: instruction coverage rose from 73% to 74%, and branch coverage from 79% to 80%. More importantly, the LLM-augmented tests demonstrated a notable improvement in bug detection. Compared to the tests generated by EvoSuite or LLM (Figure 3), the augmented tests performed equivalently to the educator tests on 131 out of 296 incorrect student submissions and outperformed them on 29 submissions. These findings suggest that augmenting educator tests can enhance bug detection, supporting finer-grained assessments. However, this remains a preliminary exploration, underscoring the need for more systematic methodologies and rigorous evaluation to fully realize its potential.

B. Program Repair

To investigate the potential reasons behind the low repair rate, we first analyze the number of buggy methods and failed tests among the repaired and unrepaired student submissions from ARJA-e. Figure 5a shows no significant difference in the number of buggy methods between these two categories, with submissions from both repaired and unrepaired groups having between 1 and 11 buggy methods. However, as shown in Figure 5b, with 60 tests in total, the repaired submissions had significantly fewer failed tests, ranging from 1 to 18 (6 on average). The unrepaired group has failing tests ranging from 4 to 49 (15 on average), which is approximately x2.5 higher than the repaired ones. These findings align with prior research [9], suggesting that a high test failure rate stems from the significant incorrectness in student submissions, which is a primary reason for no repair generation. This highlights the need for new techniques that can effectively handle the complexity of repairing programming assignment projects.

To further draw insights from the unrepaired buggy methods, we manually investigated them. Due to the high number of such buggy methods, we randomly selected ten buggy methods having less than 8.25 failed tests, which is the third quartile of the number of failed tests from the repaired submissions.



(a) Number of buggy methods

(b) Number of failed tests

Fig. 5: Comparison of repaired and unrepaired submissions

One major cause of no repair generation is *limited program context*, as program repair techniques may fail to generate repairs due to insufficient analysis of the program context (e.g., the surrounding code), a common challenge faced by program repair tools [14, 47]. Although ARJA-e mitigates it to some extent by considering broader program context beyond just variables within the buggy methods [14], we observed that it still struggles when repairing a buggy method involves expressions related to field variables updated elsewhere in the program. This also helps explain why LLM repaired more buggy methods and achieved more partially repaired submissions than ARJA-e, yet produced fewer fully repaired submissions. Despite isolating the buggy methods by creating intermediate programs, ARJA-e still received an entire program as input. In contrast, while the prompts for LLM included relevant information, they only covered one or a few classes, offering less program context than ARJA-e. This limitation of LLM-based approaches is particularly relevant to the nature of programming assignment projects, which typically involve multiple code files and significantly longer contextual information compared to simpler introductory assignments.

To ensure reliable results from LLM, only a portion of the program and related information could be used as input. This hinders the LLM's ability to generate fully repaired submissions across the entire program, despite its effectiveness in repairing single methods. Although JavaDocs of the methods were provided in the prompts to clarify functionalities, other contexts, such as the assignment's brief or additional program structure, could serve as useful supplementary information, where future work should explore these possibilities.

C. Implications for educators

Based on the experiments and findings derived from this study, we offer several implications for educators regarding the use of test generation and program repair tools to support assessment in programming assignment projects.

It is necessary for educators to develop tests that cover the intended scenarios of the assignments, as the current test generation techniques, such as EvoSuite, are not advanced enough to fully account for the complex dependencies and scenarios present in assignment projects. These tools often miss important cases that educators can better anticipate due to their deep understanding of the assignments' goals.

While techniques like LLMs or test generation show the potential to augment existing tests to enhance bug detection, there is limited research into the systematic evaluation of this approach. However, our findings suggest that LLM-augmented tests can improve bug detection to an extent, and educator might explore this as a viable way to enhance their tests.

Educators can leverage correct student solutions to drive program repair tools for repairing incorrect student submissions. This approach reduces the need for manually developing reference solutions without significantly impacting the effectiveness of program repair tools.

While program repair tools can address bugs within method bodies at the statement level, they often struggle with repairs requiring changes to class members, such as method signatures, annotations, and constructors. To mitigate this issue, educators typically provide skeleton code with predefined class members when releasing assignments. However, there is a trade-off between the applicability of program repair techniques and students' learning experiences, as providing too much supporting information may negatively impact students' learning. Therefore, it is recommended that educators carefully design assignments to balance adequate support for program repair techniques with encouraging students to develop problem-solving skills independently.

VI. Related Work

Paiva et al. [1] conducted a systematic review on automated assessment in programming education, analyzing 778 primary studies on various topics, including testing techniques and feedback types for automated assessment. The study identified *output comparison* and *unit testing* as the primary methods to assess student programs' functionality, highlighting that testing techniques are essential for automated assessment. It also discussed program repair as a promising method for generating feedback that suggests corrections for students' code.

Tang et al. [6] proposed FEAT (Feedback and Evaluation via Auto-generated Tests), a toolchain that generates high-quality tests for automated assessment systems. By inputting the problem specification, the tool combines exhaustive and random testing approaches to create tests that detect every erroneous solution identified by a larger test set. The authors evaluated the generated tests against expect-created tests for eight programming problems from online courses. Their findings indicate that the generated tests achieved higher coverage, identifying 1.3 - 64.6% more incorrect solutions than expert tests. While the source of the evaluated program was not specified, searching for their names indicates that the problems are likely at the introductory level[11].

Yi et al. [9] investigated the feasibility of using existing program repair tools for introductory programming assignments. They evaluated four program repair tools on the IITK dataset and successfully repaired 208 out of 661 C programs. The study identified two key reasons for the relatively low repair rate: high test failure rate, where 60% of the programs failed more than half of the tests, and complex repair, in which most successful repairs only involved one-line changes.

Zhao et al. [29] introduced PaR, an LLM-powered framework for repairing programming assignments that involve complex data structures and algorithms. They also created Defects4DS, a dataset of 682 incorrect solutions for such assignments. While the solutions are still single-filed programs, they pose unique repair challenges due to larger codebases and complex syntax compared to introductory ones. PaR utilizes prompts that include relevant peer solutions, program descriptions, input/output formats, and the buggy code. Results show that PaR outperforms LLM baselines (e.g., GPT-3.5) and traditional program repair tools (e.g., Verifix [28]) on both the Defects4DS and IITK [9] datasets, indicating its effectiveness in repairing both introductory and intermediate assignments.

VII. Conclusions and Future Work

In this paper, we evaluate the feasibility of applying existing test generation and program repair approaches for automated assessment of programming assignment projects. We employed EvoSuite and Qwen2.5-Coder, an LLM for test generation, and ARJA-e and the LLM for program repair. Our evaluation reveals noticeable limitations for both bug detection and the repair of incorrect student solutions in complex programming projects.

We discuss common challenges in generating tests and repairing programming assignment projects, suggesting potential improvements. Our findings highlight the need for advanced test generation and program repair techniques, or alternative methods, to better inform automated assessments that reflect the nature of programming projects. We identified several future research directions, first, augmenting educator tests to improve bug detection, leading to more accurate assessments and fine-grained grading. Second, exploring supplementary information to guide LLMs to effectively repair student solutions that spread multiple files and interdependencies. This could involve extracting minimal, useful information from assignment requirements or runtime data for more comprehensive repairs. Third, integrating test generation and program repair, where tests are foundational for bug detection, enables repair techniques to enhance overall performance.

Addressing these directions could advance both test generation and program repair, making automated assessment systems applicable to more complex programming assignments beyond the introductory level.

[11] https://linzifan.github.io/python_courses/PoC-Project4

References

[1] J. C. Paiva, J. P. Leal, and A. Figueira, "Automated assessment in computer science education: A state-of-the-art review," *ACM Trans. Comput. Educ.*, vol. 22, no. 3, jun 2022.

[2] B. Clegg, M.-C. Villa-Uriol, P. McMinn, and G. Fraser, "Gradeer: An open-source modular hybrid grader," 2021. [Online]. Available: https://arxiv.org/abs/2102.09400

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

[4] R. Singh, S. Gulwani, and A. Solar-Lezama, "Automated feedback generation for introductory programming assignments," *SIGPLAN Not.*, vol. 48, no. 6, p. 15–26, 2013.

[5] S. Sarsa, P. Denny, A. Hellas, and J. Leinonen, "Automatic generation of programming exercises and code explanations using large language models," in *ACM Conf. on Intl. Computing Education Research - Volume 1 (ICER)*. ACM, 2022, p. 27–43.

[6] T. Tang, R. Smith, S. Rixner, and J. Warren, "Data-driven test case generation for automated programming assessment," in *Conf. on Innovation and Technology in Computer Science Education (ITiCSE)*. ACM, 2016, p. 260–265.

[7] X. B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *2016 IEEE 23rd Intl. Conf. on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 1, 2016, pp. 213–224.

[8] S. Gulwani, I. Radiček, and F. Zuleger, "Automated clustering and program repair for introductory programming assignments," in *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 2018, p. 465–480.

[9] 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 *11th Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2017, p. 740–751.

[10] A. de Freitas, J. Coffman, M. de Freitas, J. Wilson, and T. Weingart, "Falconcode: A multiyear dataset of python code samples from an introductory computer science course," in *ACM Technical Symposium on Computer Science Education (SIGCSE)*. ACM, 2023, p. 938–944.

[11] Y. Hu, U. Z. Ahmed, S. Mechtaev, B. Leong, and A. Roychoudhury, "Re-factoring based program repair applied to programming assignments," in *2019 34th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*. IEEE Computer Society, nov 2019, pp. 388–398.

[12] C. Koutcheme, N. Dainese, S. Sarsa, J. Leinonen, A. Hellas, and P. Denny, "Benchmarking educational program repair," 2024.

[13] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *19th ACM SIGSOFT Symposium and the 13th European Conf. on Foundations of Software Engineering (ESEC/FSE)*. ACM, 2011, p. 416–419.

[14] Y. Yuan and W. Banzhaf, "Toward better evolutionary program repair: An integrated approach," *ACM Trans. Softw. Eng. Methodol.*, vol. 29, no. 1, jan 2020.

[15] S. Kim, S. Yun, H. Lee, M. Gubri, S. Yoon, and S. J. Oh, "Propile: Probing privacy leakage in large language models," in *Advances in Neural Information Processing Systems*, vol. 36. Curran Associates, Inc., 2023, pp. 20 750–20 762.

[16] X. Wu, R. Duan, and J. Ni, "Unveiling security, privacy, and ethical concerns of chatgpt," *Journal of Information and Intelligence*, vol. 2, no. 2, pp. 102–115, 2024.

[17] C. Pacheco and M. D. Ernst, "Randoop: feedback-directed random testing for java," in *Companion to the 22nd ACM SIGPLAN Conf. on Object-Oriented Programming Systems and Applications (OOPSLA)*. ACM, 2007, p. 815–816.

[18] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri, "Random or evolutionary search for object-oriented test suite generation?" *STVR*, vol. 28, no. 4, p. e1660, 2018.

[19] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges," in *30th IEEE/ACM Intl. Conf. on Automated Software Engineering (ASE)*, 2015, pp. 201–211.

[20] S. Nayak, R. Agarwal, and S. K. Khatri, "Automated assessment tools for grading of programming assignments: A review," in *Intl. Conf. on Computer Communication and Informatics (ICCCI)*, 2022, pp. 1–4.

[21] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? an empirical inquiry into the redundancy assumptions of program repair approaches," in *Companion Proc. of the 36th Intl. Conf. on Software Engineering (ICSE)*. ACM, 2014, p. 492–495.

[22] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *31st Intl. Conf. on Software Engineering (ICSE)*. IEEE, 2009, pp. 364–374.

[23] Y. Yuan and W. Banzhaf, "Arja: Automated repair of java programs via multi-objective genetic programming," *IEEE Transactions on Software Engineering*, vol. 46, no. 10, pp. 1040–1067, oct 2020.

[24] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Intl. Conf. on Software Engineering (ICSE)*. IEEE, 2013, pp. 802–811.

[25] J. R. Koza, *Genetic programming: on the programming of computers by means of natural selection*. Cambridge, MA, USA: MIT Press, 1992.

[26] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: a database of existing faults to enable controlled testing studies for java programs," in *Intl. Symposium on Software Testing*

and Analysis (ISSTA). ACM, 2014, p. 437–440.

[27] M. Martinez and M. Monperrus, "Astor: a program repair library for java (demo)," in *Intl. Symposium on Software Testing and Analysis (ISSTA)*. ACM, 2016, p. 441–444.

[28] 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.*, vol. 31, no. 4, jul 2022.

[29] Q. Zhao, F. Liu, L. Zhang, Y. Liu, Z. Yan, Z. Chen, Y. Zhou, J. Jiang, and G. Li, "Peer-aided repairer: Empowering large language models to repair advanced student assignments," 2024.

[30] W. X. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, Y. Hou, Y. Min, B. Zhang, J. Zhang, Z. Dong, Y. Du, C. Yang, Y. Chen, Z. Chen, J. Jiang, R. Ren, Y. Li, X. Tang, Z. Liu, P. Liu, J.-Y. Nie, and J.-R. Wen, "A survey of large language models," 2024.

[31] A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," 2023.

[32] S. Ouyang, J. M. Zhang, M. Harman, and M. Wang, "An Empirical Study of the Non-determinism of ChatGPT in Code Generation," *ACM Trans. Softw. Eng. Methodol.*, Sep. 2024.

[33] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, L. Zhang, Z. Li, and Y. Ma, "Exploring and evaluating hallucinations in llm-powered code generation," 2024.

[34] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, p. 131–164, 2009.

[35] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds, "An industrial evaluation of unit test generation: Finding real faults in a financial application," in *IEEE/ACM 39th Intl. Conf. on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2017, pp. 263–272.

[36] T. Y. Zhuo, M. C. Vu, J. Chim, H. Hu, W. Yu, R. Widyasari, I. N. B. Yusuf, H. Zhan, J. He, I. Paul, S. Brunner, C. Gong, T. Hoang, A. R. Zebaze, X. Hong, W.-D. Li, J. Kaddour, M. Xu, Z. Zhang, P. Yadav, N. Jain, A. Gu, Z. Cheng, J. Liu, Q. Liu, Z. Wang, D. Lo, B. Hui, N. Muennighoff, D. Fried, X. Du, H. de Vries, and L. V. Werra, "BigCodeBench: Benchmarking code generation with diverse function calls and complex instructions," 2024.

[37] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," 2023.

[38] M. Martinez and M. Martin, "Ultra-large repair search space with automatically mined templates: The cardumen mode of astor," in *Intl. Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2017.

[39] Y. Higo, S. Matsumoto, R. Arima, A. Tanikado, K. Naitou, J. Matsumoto, Y. Tomida, and S. Kusumoto, "kgenprog: A high-performance, high-extensibility and high-portability apr system," in *Asia-Pacific Software Engineering Conference (APSEC)*, 2018, pp. 697–698.

[40] T. Durieux and M. Monperrus, "IntroClassJava: A Benchmark of 297 Small and Buggy Java Programs," Universite Lille 1, Research Report hal-01272126, 2016.

[41] 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*, vol. 41, no. 12, pp. 1236–1256, 2015.

[42] T. Durieux, F. Madeiral, M. Martinez, and R. Abreu, "Empirical review of java program repair tools: a large-scale experiment on 2,141 bugs and 23,551 repair attempts," in *ACM Joint Meeting on European Software Engineering Conf. and Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2019, p. 302–313.

[43] F. Long and M. Rinard, "An analysis of the search spaces for generate and validate patch generation systems," in *Intl. Conf. on Software Engineering (ICSE)*. ACM, 2016, p. 702–713.

[44] A. Silva and M. Monperrus, "Repairbench: Leaderboard of frontier models for program repair," 2024.

[45] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, R. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, W. Saunders, C. Hesse, A. N. Carr, J. Leike, J. Achiam, V. Misra, E. Morikawa, A. Radford, M. Knight, M. Brundage, M. Murati, K. Mayer, P. Welinder, B. McGrew, D. Amodei, S. McCandlish, I. Sutskever, and W. Zaremba, "Evaluating large language models trained on code," 2021.

[46] M. F. Roslan, J. M. Rojas, and P. McMinn, "An empirical comparison of evosuite and dspot for improving developer-written test suites with respect to mutation score," in *Symposium on Search-Based Software Engineering (SSBSE)*. Springer, 2022, pp. 19–34.

[47] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *40th Intl. Conf. on Software Engineering (ICSE)*. ACM, 2018, p. 1–11.