High-Coverage LLM-based Unit Test Code Generation through Control Flow Graph Guidance

Xinran Luo
School of Software Engineering
South China University of Technology
Guangzhou, China
ellaxrluo@163.com

Cheng Liu
Institute of Computing Technology
Chinese Academy of Sciences
Beijing, China
liucheng@ict.ac.cn

Mengchen Zhao
School of Software Engineering
South China University of Technology
Guangzhou, China
zzmc@scut.edu.cn

Zhibin Wu
Shanghai Aircraft Design & Research Inst.
Commercial Aircraft Corporation of China
Shanghai, China
wuzhibin@comac.cc

Tao Lu
Shanghai Aviation Industrial Company
Commercial Aircraft Corporation of China
Shanghai, China
lutao4@comac.cc

Junying Chen*
School of Software Engineering
South China University of Technology
Guangzhou, China
jychense@scut.edu.cn

Abstract*—Unit testing is a critical yet labor intensive phase in software development. Although large language models (LLMs) have shown promise in unit test code generation, existing LLM-based methods often suffer from limited code understanding, high error rate, and low test coverage. We propose an approach that leverages control flow graphs (CFGs) to improve test quality and coverage. Our method comprises three phases. In the data pre-processing phase, we extract focal methods and dependencies from Java projects and construct CFGs from the abstract syntax tree, which allows LLMs to better focus on the core logic. In the code generation phase, we use LLM to decompose each focal method into CFG-based code slices and generate a coherent unit test code for each slice. In the post-processing optimization phase, we apply a two-stage repair strategy for failing tests to ensure greater test executability, combining template-based repair with LLM-based repair. To further boost coverage, we analyze uncovered control paths via the CFG and guide the LLM to optimize the unit test code test cases accordingly. Experimental results show that our approach significantly outperforms baselines in terms of test execution pass rate and code coverage, with ablation studies highlighting strong inter-module synergy. *All experimental data and source code are available at https://github.com/EllaDrCarrot/LUGC.

***Index Terms*—Unit Test, Code Generation, Control Flow Graph, Large Language Model, Software Testing Application**

I. Introduction

Unit testing plays a critical role in software development, allowing the early detection and correction of code-level errors [1]. It ensures code quality and system stability, forming the foundation for subsequent integration and system testing. However, in large-scale projects, writing unit test cases manually is time consuming and error prone due to complex logic and frequent code changes, resulting in high maintenance costs and inconsistent test styles.

To improve testing efficiency, developers often rely on automated unit test code generation tools, such as search-based software testing (SBST) methods [2], which aim to maximize code coverage as their main objective. For example, Randoop leverages feedback from the execution of randomly generated test cases to guide the search process [3], while EvoSuite employs genetic algorithms to evolve high-coverage test cases [4]. Although these traditional approaches are effective in improving coverage, the generated test code often lacks readability and interpretability, limiting its practical usefulness [5].

With the rise of large language models (LLMs), new opportunities have emerged for automated unit test code generation. Pre-trained on massive code corpora, LLMs show strong capabilities in code understanding and generation. Recent studies [5]–[8] demonstrate that LLMs can generate human-like test code, reducing development time and labor costs. However, the LLM-based unit test code generation still faces notable challenges, including difficulty in understanding complex input contexts, frequent generation of non-executable tests, limited coverage for methods with intricate logic, and intrinsic issues such as hallucination [9] and repetitive suppression [10], [11]. To address these limitations, we propose an LLM-based unit test code generation approach that leverages control flow graphs (CFGs) to enhance test quality and coverage. The contributions are as follows.

- **Enhanced understanding of code logic.** We propose a CFG analysis during data pre-processing. By parsing the abstract syntax tree (AST) and constructing a CFG for each focal method, we extract and serialize contextual and control-dependent information. This structured input helps LLMs to better focus on the core logic, avoiding distractions from irrelevant or verbose context.
- **Higher coverage for complex methods through code decomposition and optimization of the coverage rate.** During code generation, we decompose complex methods into CFG-aligned slices. Each slice is treated as an independent test target, allowing for the generation of

*Corresponding author.

fine-grained unit test code that preserves logic. For low-coverage cases, we further analyze uncovered execution paths and control dependencies via CFG and then guide the LLM to optimize the code.
- **Improved test executability through a two-stage repair strategy.** We propose a two-stage repair strategy in the post-processing optimization phase. Non-executable test cases are first handled using error-specific templates; if fail, the LLM-based repair process is initiated. This hierarchical repair reduces the LLM hallucination problem and the repetitive suppression problem.

II. Related Works

A. LLM-based unit test code generation

Recent research has explored the potential of LLMs in code generation [12], [13], including unit test code generation [14]. LLMs leverage the knowledge learned from large code corpora to analyze program logic and generate test cases that cover various scenarios, significantly reducing manual effort. They can also adapt to code updates and produce tests in standardized formats, improving test maintainability and readability. Researchers have proposed various LLM-based unit test code generation frameworks. ChatTester [7] improves efficiency through prompt optimization. ChatUniTest [5] integrates ChatGPT into Java testing workflows. Codamosa [15] combines SBST [2] tools and LLMs for complementary strengths. SymPrompt [16] uses execution paths as prompts for better accuracy. HITS [17] applies code slicing to improve the test coverage. TestART [18] focuses on repairing faulty LLM-generated tests using templates.

B. CFG in Software Engineering

In software engineering practice, testers can leverage CFGs to identify all possible control flow paths within a program [19]. CFG is a directed graph that models the execution flow of a program. The nodes represent statements or code blocks, while the edges denote possible control transitions between them. CFGs capture various control structures, including conditional branches (`if-else`, `switch`), loops (`for`, `while`, `do-while`), jumps (`break`, `continue`), and exception handling (`try-catch-finally`). By abstracting these relationships into a graphical form, CFGs clarify the program logic and support systematic test case design. By analyzing the CFG, software testers can design a set of test cases that cover both normal execution branches and exception-handling scenarios.

III. The Proposed Method

A. Overview

The proposed method consists of three main phases: data pre-processing, code generation, and post-processing optimization. The overall architecture and work flow are illustrated in Figure 1. The control flow graph constructed during the data pre-processing phase acts as a structural guide throughout, supporting code decomposition and coverage optimization, as illustrated in Figure 2.

During data pre-processing, the system parses the Java project to identify focal methods and their dependencies, constructing CFGs for each method via AST analysis and serializing the data into JSON files. In the code generation phase, the LLM uses the CFG to decompose each method into logical slices, each slice representing a complete execution branch. The LLM then generates the test code for each slice, forming an initial set of unit test codes. Subsequently, these test codes are verified for executability, and non-executable tests are repaired using a hierarchical two-stage repair strategy, starting with error templates and followed by LLM-based corrections if needed. Executable tests are evaluated for coverage. If coverage is insufficient, the system analyzes the uncovered paths and feeds these data back to the LLM for further optimization. This iterative process continues until the coverage goals are met or the maximum iterations are reached. The system ultimately outputs a refined unit test code with high coverage after several rounds of repair and optimization.

B. Data Pre-processing

1) Code Parsing and Denoising: In the data pre-processing phase, we implement an automated Java project parsing tool that extracts target methods and their dependencies from Java project. Through static analysis, the tool constructs relevant contextual information [20], including method signatures, parameter constraints, and related class or method references. To enhance the quality of LLM input and reduce token overhead, the tool performs targeted code denoising. Remove comments, blank lines, and redundant formatting from the focal method, which minimizes noise without altering the core logic. Although comments can aid human understanding, they often consume substantial tokens and may mislead the model if they are inconsistent with actual behavior, potentially causing hallucinations [9] or semantic bias [21].

In addition, for auxiliary code elements, such as class and method invocations, constructors, and getter/setter functions, the system retains only essential information, such as method signatures, parameter lists, and invocation relationships while omitting internal implementation details. This selective abstraction preserves the inter-method and inter-class dependency structure and guides the LLM focus on the core logic of the focal methods, improving both the accuracy and efficiency of test case generation.

2) Control Flow Graph Construction: Existing research has shown that complex methods characterized by deep nested conditionals, loops, and intricate branching logic are particularly difficult for LLMs to analyze effectively, often resulting in low coverage [22]. To address this challenge, we incorporate control flow graph (CFG) representations into the LLM's input to improve its understanding of complex control structures in methods with high cyclomatic complexity. We begin by parsing the abstract syntax tree (AST) of each method using the Javalang parser. The AST serves as a hierarchical representation of the source code from which we extract control-related constructs such as `if-else`, `do-while`, and `try-catch-finally` statements. During traversal, we

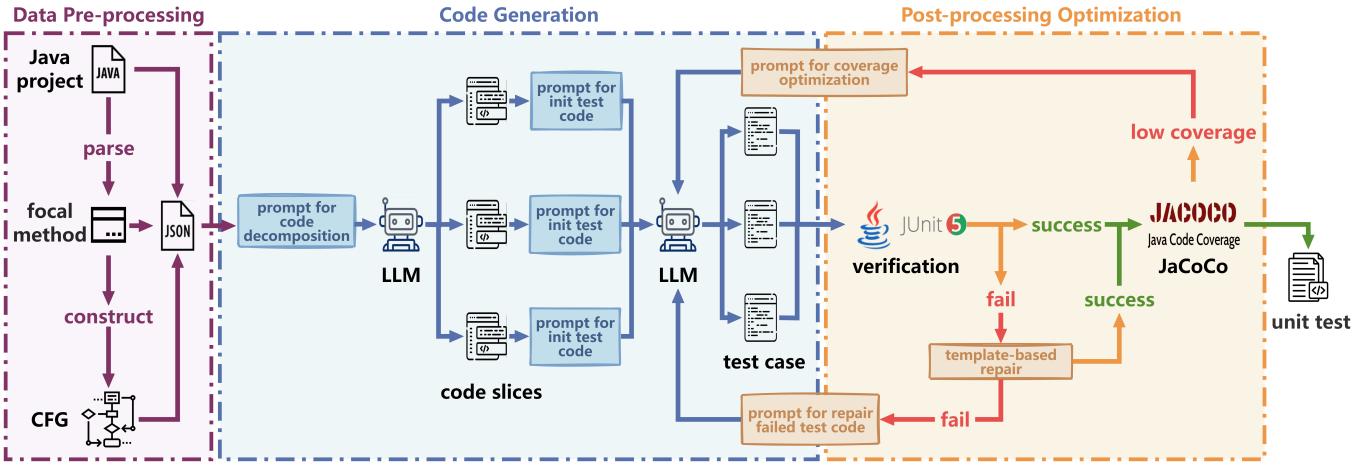

Fig. 1. The overall architecture and work flow of the proposed approach.

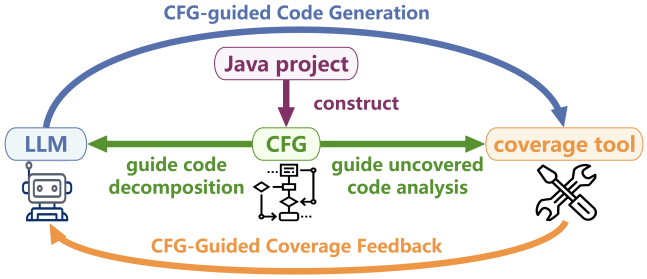

Fig. 2. CFG guidance in the workflow.

construct a directed CFG, where each node corresponds to a control-relevant statement, and each edge reflects a control dependency. As illustrated in Algorithm 1, the graph is built recursively. Control statements are pushed onto a stack to manage nested scopes, edges are added based on the syntactic structure, and various control constructs are handled according to their semantics.

The resulting CFG provides a structured semantic representation that guides the LLM in analyzing the method's execution paths and critical branches.This enables the generation of test cases with high structural adequacy, specifically by ensuring the systematic coverage of identified control paths, which effectively addresses the issue of low coverage in complex methods. All pre-processed data, including the CFG and contextual information, is stored in JSON format. This structured approach ensures data integrity, adheres to LLM input standards, and facilitates efficient data retrieval for subsequent code generation and post-processing optimization phases.

Algorithm 1 CFG Construction

Input: Java AST of method, line mapping info
Output: CFG nodes and edges
Init graph G, control stack S, metadata containers
procedure VISIT(node)
  Record node, push to S
  if node is `IfStatement` **then** ▷ // Handle conditional branches
    Add edge from S_{top} to current
    VISIT(then branch)
    while else branch is `IfStatement` **do**
      VISIT(else-if branch)
    end while
    if else branch exists **then** VISIT(else branch)
    end if
  else if node is loop **then** ▷ // Handle loop structures
    VISIT(loop body)
  else if node is `SwitchStatement` **then** ▷ // Handle switch-case blocks
    for all case & stmt in switch **do** VISIT(stmt)
    end for
  else if node is `TryStatement` **then** ▷ // Handle exception handling
    VISIT(try), VISIT(catch blocks), VISIT(finally)
  else if node is jump stmt **then** ▷ // Handle control flow jumps
    return
  else ▷ // Process standard statements
    if stmt-like **then** Connect to S_{top}
    end if
    for all child in node **do** VISIT(child)
    end for
  end if
  Pop S
end procedure
VISIT(AST root node)
Extract edges from G, return CFG

C. Code Generation

1) Prompt: During the LLM-based code generation phase, all prompts are constructed using the chain-of-thought (CoT) [23] format. This format is applied to four main tasks: code decomposition, unit test code generation for slices, repair for failed unit test code, and optimization for low coverage unit test code.

- **Code Decomposition Task:** The LLM begins by sum-

marizing the focal method under test and describing its required test environment. Then, leveraging the method's control flow graph (CFG), it analyzes the control flow structure to ensure semantic coherence in the generated code slices. Based on the semantic structure, the focal method is decomposed into sub-tasks. The corresponding code for each sub-task is then extracted and formatted into slices, output in a predefined JSON format.

- **Unit Test Code Generation Task:** The LLM restates the relevant input and method calls associated with the code slice and enumerates all potential usage scenarios and input combinations. Then it infers the necessary environment conditions for executing the code slice. Finally, it generates a unit test class that ensures full line and branch coverage.
- **Repair Failed Test Code Task:** The LLM uses JVM error messages to locate faulty statements, analyzes the root causes of the failures, proposes appropriate fixes, and generates a revised unit test code accordingly.
- **Code Coverage Optimization Task:** The LLM identifies the uncovered code lines, their associated control flow stages, and directly related control statements. It then analyzes the reasons for missed coverage, suggests additional test cases required to achieve full coverage, and generates the corresponding updated unit test code.

To improve the flexibility and efficiency of prompt construction, we adopt the Jinja2 template engine for prompt generation. Jinja2 supports dynamic variable interpolation, conditional logic, and code snippet rendering [24]. This allows for automated insertion of method names, control flow information, and contextual data, as well as the inclusion of relevant examples, thereby promoting prompt reuse and enabling scaling across diverse generation tasks.

2) Code Decomposition: In our approach, we propose a control-flow-based code decomposition strategy. This strategy utilizes the CFG of a method to segment it into logically complete code slices. Each code slice typically starts with a control statement and includes one or two control structures, thereby reducing the complexity of code that LLMs need to process during unit test code generation. Building on this, we introduce a conditional combination coverage strategy. Specifically, for each control statement within a slice, we extract all Boolean sub-conditions and require the LLM to generate inputs that satisfy both the true and false branches of each condition. This selective condition combination approach improves test coverage while avoiding the exponential cost of full path coverage. In real-world testing scenarios, complex methods often contain deep nested control structures, including conditionals, loops, switches, and jumps. A naive strategy that exhaustively enumerates all possible execution paths would require covering every possible combination of these control flows. The total number of paths can be approximated by:

$$P_{\text{origin}} = \left(\prod_{i=1}^{n_{\text{cond}}} 2^{m_i}\right) \cdot \left(\prod_{j=1}^{n_{\text{loop}}} (k_j + 1)\right) \cdot \left(\prod_{l=1}^{n_{\text{switch}}} s_l\right) \cdot 2^{n_{\text{jump}}} \quad (1)$$

where n_{cond} denotes the number of conditional statements (e.g., `if`, `if-else`), each with m_i Boolean sub-conditions; n_{loop} is the number of loops with maximum unrolling bound k_j; n_{switch} is the number of switch statements with s_l branches; and n_{jump} represents early exits such as `return`, `break`, or `throw`. The calculation grows exponentially, making exhaustive path exploration impractical for real-world methods.

In contrast, our control-flow-based decomposition isolates each control structure into an independent slice. Within each slice, the method only requires covering the true and false outcomes of each Boolean sub-condition, ignoring global control context. As a result, the total number of required condition combinations is linear in the number of sub-conditions across all slices:

$$P_{\text{decomposed}} = \sum_{j=1}^{k} 2 \cdot m_j \quad (2)$$

where k is the number of code slices and m_j is the number of sub-conditions in the j-th slice. Such a localized strategy avoids exponential blow-up while still achieving high condition coverage. Compared with the exponential complexity of the full path coverage as shown in Equation 1, our approach significantly reduces the condition combination space.

After the decomposition process, each generated code slice is annotated with a brief functional description to improve interpretability. These annotations provide the LLM with contextual information about the role and purpose of each slice, thereby improving the accuracy of code generation. The annotated slices, along with the necessary external information required for their execution, are serialized into a structured JSON format to facilitate standardized downstream processing. This structured representation is provided to the LLM for targeted unit test code generation.

D. Post-processing Optimization

1) Error Repair: The unit test code generated by the LLM will be put into the verification module to check the correctness of execution. The verification process includes syntax verification, compile verification, and runtime verification. Only test cases that pass all three processes are considered valid. When verification fails, the repair module attempts to fix unit test code using two strategies: **(1) template-based repair** for common errors; **(2) LLM-based repair** for complex or unhandled cases.

We design four error repair templates, as shown in Figure 3, to handle typical error categories observed in generated unit tests. **(1) Symbol Repair Template** Fixes syntactic issues such as unmatched parentheses, quotation marks, or missing semicolons by analyzing nesting structures and lexical contexts. **(2) Import Repair Template** Automatically adds missing import statements by extracting class names from compilation errors and resolving them against a dependency index. **(3) Exception Repair Template** Wraps error-prone code with `try-catch` blocks to catch and suppress known exceptions, allowing the rest of the test to execute uninterrupted. **(4) Assertion Repair Template** Modifies failed assertions based on runtime logs. If

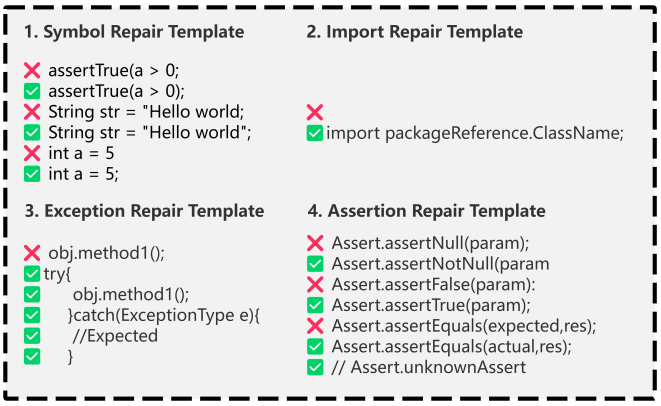

Fig. 3. Four error repair templates.

Algorithm 2 Uncovered Line Control Dependency Analysis

Input: CFG G, JaCoCo report, stmt positions `stmt_pos`
Output: Control dependencies for uncovered lines

Extract `U` ← uncovered lines, `B` ← branch lines from report
`P` ← $\{$`stmt_pos`$[id] \mid id \in G.nodes\}$
`missing_lines` ← `MergeList`(B, U, P)
Init sets: `roots` ← $\emptyset$, `targets` ← $\emptyset$
for all $n \in G.nodes$ **do**
  if `stmt_pos`$[n] \in B$ **then** `roots` ← `roots` $\cup\{n\}$
  end if
  if `stmt_pos`$[n] \in U$ **then** `targets` ← `targets` $\cup\{n\}$
  end if
end for
`deps` ← $\emptyset$
for all $n \in$ `targets` **do**
  Init visited $V \leftarrow \emptyset$, queue $Q \leftarrow [n]$
  while $Q \neq \emptyset$ **do**
    $c \leftarrow$ dequeue Q
    if $c \in$ `roots` **then**
      Add (`stmt_pos`$[n]$, `stmt_pos`$[c]$) to `deps`; **break**
    end if
    for all $p \in G.$`predecessors`(c) **do**
      if $p \notin V$ **then** Add p to V, enqueue p
      end if
    end for
  end while
end for
Sort `deps` by uncovered line number
return `deps`

the assertion type is ambiguous, the statement is commented out.

If template-based repair fails, we resort to LLM-based repair. The failed test code and full error logs are provided to the LLM, which is prompted to regenerate a corrected version. The LLM-based repair process essentially re-enters the previous code generation phase, reapplying the same code generation and verification pipeline with updated inputs, ensuring consistency and minimizing redundant logic across components. The repair cycle is repeated until the test passes all verification steps or a maximum retry limit is reached.

2) Coverage Rate Optimization: Once the unit test code passes the execution verification, the system proceeds to the coverage evaluation stage. We employ the open-source Java coverage tool JaCoCo to assess test quality [25]. The system extracts uncovered lines and branches from the report and maps these lines to the corresponding statements in the CFG of the focal method. By cross-referencing line numbers with CFG node positions, we identify the control structures that govern the execution of uncovered lines. To determine these control dependencies, we perform a reverse breadth-first traversal (BFS) on the CFG, starting from uncovered blocks and tracing back to their control points, thereby reconstructing the control path leading to the uncovered region. The detailed steps are illustrated in the Algorithm 2

The original test code, along with CFG-guided dependency information on uncovered regions, is provided to the LLM, which analyzes potential causes of coverage gaps and generates an improved test code. This process essentially revisits the previous code generation phase, where the LLM reapplies the code generation and verification pipeline with updated inputs. This iterative process progressively refines the test code, increasing coverage while maintaining correctness and semantic integrity. The updated test is re-evaluated, continuing until the desired coverage threshold is met or a maximum number of optimization attempts is reached.

TABLE I
DATASET INFORMATION.

Project	Version	Domain	Incl. in DS
CLI	1.10.0	CLI parser	No
CSV	1.10.0	CSV I/O	Yes
COD	1.17.2	Encoding utilities	No
COL	4.5.0	Collection utilities	Yes
Ruler	1.2.1	Rule engine	Yes

IV. Experimental Results and Discussions

A. Datasets, Setup, Baselines, and Metrics

1) Datasets: To ensure the reliability of the experimental results, we select five open-source projects on GitHub: *Commons-CLI (CLI)*, *Commons-CSV (CSV)*, *Commons-Codec (COD)*, *Commons-Collections (COL)*, and *Ruler*. The CLI (1.10.0) and COD (1.17.2) were released after the DeepSeek-R1 knowledge cutoff date (2024.07) [26], while the other datasets were released prior to it. Table I presents the version information, application domain, and whether the project is included in the Deepseek-R1 (DS) training dataset. To further explore the research potential of the proposed approach in complex testing scenarios, we extracted methods with cyclomatic complexity$>$10 from the aforementioned projects to construct a focused experimental dataset. This enables an in-depth evaluation of whether the proposed approach can enhance test quality for highly complex methods.

2) Experimental Setup: Considering both performance and cost, we selected the DeepSeek-R1 for our experiments by calling the model API. For the configuration of the test environment, the Java8 version JDK and JVM are selected as the running basis, JUnit5 is chosen as the unit test framework, and JaCoCo is used as the code coverage test tool. The

optimization targets 100% coverage with a 10-iteration limit to balance coverage gains and API costs.

3) Baselines: To assess the effectiveness of our proposed method, we compare it with several baselines in automated unit test code generation, including the classic SBST tool Evosuite [4] and LLM-based approaches: ChatUniTest [5], ChatTester [7], and HITS [17].

- **Evosuite** is a classic automated unit test code generation tool. It employs evolutionary algorithms to create test cases and leverages fitness functions to guide the search process, aiming to maximize code coverage metrics.
- **ChatUniTest** is an LLM-based unit test code generation tool. It extracts adaptive contextual information for the target method and prompts the model to generate the corresponding unit tests. If the generated code fails to execute, the framework enables iterative repair by feeding runtime error messages back into the model.
- **ChatTester** is another LLM-based framework similar to ChatUniTest, but it constructs the method context incrementally based on the needs of the generation process, aiming to provide more targeted and efficient prompts.
- **HITS** is an advanced framework that decomposes the focal method into multiple slices and then generates unit test codes for each slice individually.

4) Evaluation Metrics: To evaluate the experimental results, we adopt three critical metrics: execution pass rate (PR), line coverage rate (LC) and branch coverage rate (BC), from the dimensions of execution effectiveness and code coverage integrity.

- **Execution pass rate (PR)** measures the proportion of generated unit tests that pass syntax checks, compilation, and runtime execution.

$$PR = \frac{\text{Passed Tests}}{\text{Total Generated Tests}} \tag{3}$$

- **Line coverage rate (LC)** measures the proportion of executable lines[1] in the focal method that are actually executed (i.e., covered) during the execution of unit tests.

$$LC = \frac{\text{Executed Lines}}{\text{Total Executable Lines}} \tag{4}$$

- **Branch coverage rate (BC)** measures the proportion of conditional branches in the focal method that are executed (i.e., covered) during the execution of unit tests.

$$BC = \frac{\text{Executed Branches}}{\text{Total Branches}} \tag{5}$$

B. Experimental Results and Observation

1) Comparison of Execution Pass Rate: Table II shows the execution pass rates across five Java projects. Our method achieved an average pass rate of 62.30%, with 72.19% on the CLI project. While it is lower than HITS (70.86%), it outperformed ChatUniTest (51.98%) and ChatTester (24.97%).

[1]This refers to the code lines that has undergone code denoising and can be executed.

This superiority over ChatUniTest and ChatTester stems from our two-stage repair strategy. Although ChatUniTest and ChatTester also perform automatic error repair, they rely solely on the LLM to revise the failing test cases. In contrast, our method adopts a two-stage repair process: it first attempts template-based repairs tailored to specific error messages, and only invokes the LLM if this fails. This structured approach helps avoid redundant or hallucinated fixes often produced by LLMs when lacking true execution context. Moreover, when LLM-based repair is triggered, our method dynamically adjusts the generation parameters (e.g., top-k, top-p, temperature, and frequency penalty) to increase diversity and reduce repetition, thereby improving repair robustness and execution fidelity.

Although HITS achieves a higher execution pass rate, it allows the LLM to autonomously decompose code without explicit guidance. This results in extremely fine-grained slices, often at the granularity of individual lines of code, producing dozens or even hundreds of slices per method. Generating unit tests for such small slices is relatively easier, which likely contributes to the higher pass rate observed in HITS. However, this fine granularity lacks clear semantic coherence, leading to an excessive number of slices and consequently significantly increasing computational costs due to multiple rounds of model invocation. In contrast, our approach explicitly decomposes code based on CFG, producing semantically meaningful units that better capture the underlying logical structure of the code, enabling more accurate localization and analysis of uncovered code during the coverage optimization phase. Although our execution pass rate is lower, the resulting test code achieves a higher overall code coverage. Furthermore, our method is designed to be more targeted by only re-triggering the LLM for identified uncovered control paths, rather than re-generating the entire test suite, which aims to optimize the efficiency of model interactions.

2) Comparison of Code Coverage Rate: To evaluate the effectiveness of our method in improving the code coverage for complex methods, we measured line coverage (LC) and branch coverage (BC) in five open-source projects with complex methods (cyclomatic complexity $>$10). As shown in Table III, our method achieves the highest average coverage 68.33% for LC and 62.50% for BC, outperforming all LLM-based baselines and EvoSuite in most scenarios. Notably, on the CLI project, our method achieves 86.50% LC and 81.62% BC, showing a substantial lead. Furthermore, across all methods, LC consistently exceeds BC, which aligns with expectations, as complex control flows often make full branch coverage more challenging than line execution.

The performance gap between our method and the baselines stems from differences in architecture. ChatUniTest and ChatTester generate tests from LLMs without further refinement, while HITS incorporates an optimization loop but lacks detailed analysis of uncovered code. In contrast, our method explicitly analyzes uncovered lines using CFG, guiding the LLM to iteratively refine the test code. This targeted optimization reduces blind generation and improves test coverage.

TABLE II
COMPARISON OF EXECUTION PASS RATE (PR) (%).

Project	HITS	ChatUniTest	ChatTester	Ours
CLI	100.00	85.71	41.18	72.19
CSV	82.04	34.78	9.09	62.37
COD	82.04	57.67	23.48	68.49
COL	55.88	59.74	47.06	70.39
Ruler	34.33	21.98	4.04	38.05
Avg.	70.86	51.98	24.97	62.30

TABLE III
COMPARISON OF LC (%) AND BC (%)

Project	HITS		ChatUniTest		ChatTester		Evosuite		Ours	
	LC	BC	LC	BC	LC	BC	LC	BC	LC	BC
CLI	77.83	73.00	51.17	45.83	50.33	47.17	53.50	52.50	86.50	81.62
CSV	57.67	52.33	19.67	15.00	16.67	15.00	36.20	36.50	78.57	70.71
COD	67.84	58.63	26.26	20.47	30.05	25.00	79.30	79.32	75.50	74.20
COL	43.71	33.64	46.21	37.50	38.93	31.26	31.70	29.50	66.33	59.88
Ruler	24.13	17.63	5.56	4.13	2.00	0.94	14.30	10.36	34.77	26.11
Avg.	54.24	47.05	29.77	24.59	27.59	23.87	43.00	41.64	68.33	62.50

However, the experimental results also demonstrate that LLM-based methods are not always superior to traditional tools. In the COD project, EvoSuite surpasses all LLM-based methods. This suggests that heuristic-driven evolutionary algorithms may be more effective for certain code structures. Unit test code generation methods should be chosen based on the characteristics of the project.

3) Observation from Experiments on PR, LC and BC: The experimental results of the execution pass rate and code coverage rate indicate that while a low pass rate may limit code coverage, a high pass rate does not guarantee high coverage. For instance, ChatUniTest and ChatTester suffer from a low pass rate and low coverage, demonstrating that execution errors impede coverage. In contrast, HITS achieves a high execution pass rate but lower coverage than our method, indicating that successful execution alone is insufficient. Effective unit test code generation requires both executable code and targeted coverage optimization. This highlights the necessity of the repair module and the optimization module in the post-processing optimization phase.

C. Ablation Study

To systematically evaluate the individual impact of the Error Repair and Coverage Optimization modules in our approach, we conducted an ablation study by independently disabling each component while keeping the rest of the pipeline unchanged. Table IV summarizes the results, illustrating the contribution of each module to the overall performance.

1) Effect of Error Repair Module: LLM-generated unit test codes often suffer from syntactic issues, unresolved imports, invalid assertions, or unhandled exceptions, making many of them fail to compile or run. We first evaluated the impact of the Error Repair Module by comparing the execution pass rate before and after its application. As shown in Table IV, the module significantly improves the pass rate in all five projects. For instance, in the CSV project, the rate increases from 4.12% to 62.37%, confirming that the error repair module is highly

TABLE IV
COMPARISON OF PR (%) BEFORE/AFTER REPAIR, AND COMPARISON OF LC (%) AND BC (%) BEFORE/AFTER OPTIMIZATION

Project	PR		LC		BC	
	Before	After	Before	After	Before	After
CLI	28.88	72.19	83.25	86.50	78.87	81.62
CSV	4.12	62.37	74.28	78.57	69.42	70.71
COD	18.72	68.49	68.40	75.50	71.30	74.20
COL	14.47	70.39	64.88	66.33	58.66	59.88
Ruler	7.32	38.05	43.66	45.11	33.55	35.11

effective. Despite the improvements, perfect execution (100% execution pass rate) is not achieved. This is partly due to the limitation that our repair templates handle only common error patterns and partly because LLMs cannot simulate complex execution contexts during unit test code generation or repair. Some environment-dependent failures, such as missing resources or stateful setups, remain unresolved.

2) Effect of Coverage Optimization Module: After improving execution reliability, we evaluated the contribution of the Coverage Optimization Module. This module aims to maximize coverage by analyzing uncovered paths using CFGs and then feeding this information back to guide iterative test refinement via LLMs. As shown in Table IV, after applying the optimization module, both line coverage and branch coverage improve consistently across all projects. In particular, the COD project exhibits a large gain in line coverage, from 68.40% to 75.50%, highlighting the module's strength in generating more targeted test cases.

However, the degree of improvement varies. Projects with already high initial coverage like CLI benefit less, showing a saturation effect due to diminishing returns. In contrast, low-coverage projects like Ruler still exhibit modest gains, indicating that the optimizer is effective even under sparse test scenarios.

3) Synergistic Effects of Module Combination: A detailed analysis combining the ablation study and previous comparative experiments reveals that each module in our system not only contributes individually but also interacts synergistically throughout the unit test code generation pipeline.The CFG acts as a structural guide throughout the pipeline, providing vital structural information that guides both the code generation and coverage optimization modules. The inter-module interaction results in a compounded coverage gain that significantly outperforms what each module can achieve independently. The control flow of information and functional complementarity among modules highlights the architectural cohesiveness of our approach. It demonstrates that performance improvements are not merely additive but synergistic, reflecting the advantage of a tightly integrated design for automated unit test code generation.

V. CONCLUSION

This work proposes an approach that takes advantage of control flow graphs to improve test quality and coverage. By deeply integrating the control flow relationships, a complete

framework is constructed combining data pre-processing, code generation, and post-processing optimization. This approach exhibits strong inter-module synergy, which significantly outperforms the baselines in terms of the execution pass rate and the code coverage rate, fully verifying the effectiveness of the proposed approach. Building upon this foundation, future research will focus on exploring the framework's adaptability across a broader range of LLM architectures and further evaluating fault-detection capabilities through mutation analysis.

ACKNOWLEDGMENT

This work was partially supported by the Shenzhen Science and Technology Program (Grant No. KJZD20240903104200001), the Guangdong Basic and Applied Basic Research Foundation (Grant No. 2024A1515030128), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDB0660103), and the State Key Laboratory of Processors, Institute of Computing Technology, Chinese Academy of Sciences (Grant No. CLQ202403).

REFERENCES

[1] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Trans. Softw. Eng.*, vol. 50, no. 4, p. 911–936, Apr. 2024. [Online]. Available: https://doi.org/10.1109/TSE.2024.3368208

[2] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, Dec. 2012. [Online]. Available: https://doi.org/10.1145/2379776.2379787

[3] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, "Feedback-directed random test generation," in *Proceedings of the 29th International Conference on Software Engineering*, ser. ICSE '07. USA: IEEE Computer Society, 2007, p. 75–84. [Online]. Available: https://doi.org/10.1109/ICSE.2007.37

[4] G. Fraser and A. Arcuri, "Evolutionary generation of whole test suites," in *2011 11th International Conference on Quality Software*, 2011, pp. 31–40.

[5] Y. Chen, Z. Hu, C. Zhi, J. Han, S. Deng, and J. Yin, "Chatunitest: A framework for llm-based test generation," 2024. [Online]. Available: https://arxiv.org/abs/2305.04764

[6] S. Bhatia, T. Gandhi, D. Kumar, and P. Jalote, "Unit test generation using generative ai : A comparative performance analysis of autogeneration tools," in *2024 IEEE/ACM International Workshop on Large Language Models for Code (LLM4Code)*, 2024, pp. 54–61.

[7] Z. Yuan, M. Liu, S. Ding, K. Wang, Y. Chen, X. Peng, and Y. Lou, "Evaluating and improving chatgpt for unit test generation," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3660783

[8] N. Alshahwan, J. Chheda, A. Finogenova, B. Gokkaya, M. Harman, I. Harper, A. Marginean, S. Sengupta, and E. Wang, "Automated unit test improvement using large language models at meta," in *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*, ser. FSE 2024. New York, NY, USA: Association for Computing Machinery, 2024, p. 185–196. [Online]. Available: https://doi.org/10.1145/3663529.3663839

[9] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *ACM Trans. Inf. Syst.*, vol. 43, no. 2, Jan. 2025. [Online]. Available: https://doi.org/10.1145/3703155

[10] J. Xu, X. Liu, J. Yan, D. Cai, H. Li, and J. Li, "Learning to break the loop: analyzing and mitigating repetitions for neural text generation," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[11] M. Zhang, A. Sokolov, W. Cai, and S.-Q. Chen, "Multi-aspect repetition suppression and content moderation of large language models," *CoRR*, vol. abs/2304.10611, 2023. [Online]. Available: https://doi.org/10.48550/arXiv.2304.10611

[12] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. D. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[13] H. Liu, M. Shen, J. Zhu, N. Niu, G. Li, and L. Zhang, "Deep learning based program generation from requirements text: Are we there yet?" *IEEE Transactions on Software Engineering*, vol. 48, no. 4, pp. 1268–1289, 2022.

[14] M. Schäfer, S. Nadi, A. Eghbali, and F. Tip, "An empirical evaluation of using large language models for automated unit test generation," *IEEE Transactions on Software Engineering*, vol. 50, no. 1, pp. 85–105, 2024.

[15] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, "Codamosa: Escaping coverage plateaus in test generation with pre-trained large language models," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, 2023, pp. 919–931.

[16] G. Ryan, S. Jain, M. Shang, S. Wang, X. Ma, M. K. Ramanathan, and B. Ray, "Code-aware prompting: A study of coverage-guided test generation in regression setting using llm," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024. [Online]. Available: https://doi.org/10.1145/3643769

[17] Z. Wang, K. Liu, G. Li, and Z. Jin, "Hits: High-coverage llm-based unit test generation via method slicing," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1258–1268. [Online]. Available: https://doi.org/10.1145/3691620.3695501

[18] S. Gu, C. Fang, Q. Zhang, F. Tian, J. Zhou, and Z. Chen, "Testart: Improving llm-based unit test via co-evolution of automated generation and repair iteration," *CoRR*, vol. abs/2408.03095, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.2408.03095

[19] S. Kauffman, C. Moreno, and S. Fischmeister, "Annotating control-flow graphs for formalized test coverage criteria," in *2024 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2024, pp. 55–62.

[20] Q. Zhang, Y. Shang, C. Fang, S. Gu, J. Zhou, and Z. Chen, "Testbench: Evaluating class-level test case generation capability of large language models," 2024. [Online]. Available: https://arxiv.org/abs/2409.17561

[21] L. Yang, C. Yang, S. Gao, W. Wang, B. Wang, Q. Zhu, X. Chu, J. Zhou, G. Liang, Q. Wang, and J. Chen, "On the evaluation of large language models in unit test generation," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '24. New York, NY, USA: Association for Computing Machinery, 2024, p. 1607–1619. [Online]. Available: https://doi.org/10.1145/3691620.3695529

[22] S. B.A., O. S.O., A. A.O., and A. Olowoye, "Development of an enhanced automated software complexity measurement system," *Journal of Advances in Computational Intelligence Theory*, vol. 1, no. 3, Jan. 2020. [Online]. Available: https://doi.org/10.5281/zenodo.3597631

[23] J. Wei, X. Wang, D. Schuurmans, M. Bosma, B. Ichter, F. Xia, E. H. Chi, Q. V. Le, and D. Zhou, "Chain-of-thought prompting elicits reasoning in large language models," in *Proceedings of the 36th International Conference on Neural Information Processing Systems*, ser. NIPS '22. Red Hook, NY, USA: Curran Associates Inc., 2022.

[24] S. Bach, V. Sanh, Z. X. Yong, A. Webson, C. Raffel, N. V. Nayak, A. Sharma, T. Kim, M. S. Bari, T. Fevry, Z. Alyafeai, M. Dey, A. Santilli, Z. Sun, S. Ben-david, C. Xu, G. Chhablani, H. Wang, J. Fries, M. Al-shaibani, S. Sharma, U. Thakker, K. Almubarak, X. Tang, D. Radev, M. T.-j. Jiang, and A. Rush, "PromptSource: An integrated development environment and repository for natural language prompts," in *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, V. Basile, Z. Kozareva, and S. Stajner, Eds. Dublin, Ireland: Association for Computational Linguistics, May 2022, pp. 93–104. [Online]. Available: https://aclanthology.org/2022.acl-demo.9/

[25] A. Khatami and A. Zaidman, "State-of-the-practice in quality assurance in java-based open source software development," 2023. [Online]. Available: https://arxiv.org/abs/2306.09665

[26] DeepSeek-AI, "Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning," 2025. [Online]. Available: https://arxiv.org/abs/2501.12948