HITS: High-coverage LLM-based Unit Test Generation

Kunj Bhuva
202201275
DA-IICT
Gandhinagar, India
202201275@daiict.ac.in

Maahi Shah
202201419
DA-IICT
Gandhinagar, India
202201419@daiict.ac.in

I. ABSTRACT

Large Language Models (LLMs) have demonstrated promising results in generating unit tests for Java and Python projects. However, their effectiveness drops when targeting complex focal methods—those containing multiple conditional statements and loops—which require diverse inputs to achieve comprehensive line and branch coverage. Existing LLM-based test generation approaches typically supply the entire method to the model without offering structural input guidance, making it difficult for the LLM to infer suitable test data for thorough coverage. To address this limitation, we propose a slicing-based technique that decomposes complex focal methods into smaller logical segments and prompts the LLM to generate tests for each slice independently. This targeted decomposition narrows the scope of analysis and helps the model focus on specific code paths, improving the diversity and relevance of the generated tests. We constructed a dataset comprising challenging focal methods from real-world projects and compared our method with state-of-the-art techniques, including ChatUniTest. Our experimental results show that our approach achieves negligible to slighly better line and branch coverage, demonstrating its advantage in enhancing automated test generation with LLMs.

II. INTRODUCTION

A. Problem Statement

Achieving high line and branch coverage remains a major challenge for automatic unit test generation methods, particularly when testing complex methods—those with a high degree of cyclomatic complexity (typically above 10). These methods often contain intricate control flows, including multiple conditions and nested loops, making it difficult for existing tools to generate sufficiently diverse and comprehensive test inputs.

While traditional search-based test generation (SBST) techniques such as Randoop and EvoSuite have been effective to some extent, their performance drops when it comes to complex methods. Even modern Large Language Model (LLM)-based tools like ChatUniTest show a significant decline in coverage when applied to such challenging cases. For example, evaluations using ChatUniTest reveal a slight reduction in both line and branch coverage scores when shifting from general methods to complex ones.

This underperformance largely stems from current LLM-based methods treating the entire method as a monolithic unit, making it harder for the model to infer all possible paths and edge cases. Without granular guidance or analysis of internal structures, these methods often miss significant portions of the code under test.

B. Proposed Solution

To overcome these limitations, we introduce HITS (High-coverage Test Synthesis)—a novel test generation approach designed to improve coverage specifically for complex methods by leveraging code slicing and LLM-driven generation.

The core idea behind HITS is to decompose a complex method into logical code slices, where each slice represents a coherent step in the method's problem-solving process. This divide-and-conquer strategy narrows the focus of the LLM during test generation, allowing it to handle a smaller scope at a time. As a result, the model can more easily reason

about the required inputs and behaviors needed to cover the lines and branches within each slice.

HITS works as follows:

- Dependency Analysis: We perform static analysis to understand the dependencies of the method under test.
- Method Decomposition: Using a chain-of-thought/few-shots prompting approach, we guide the LLM to decompose the method into smaller, testable slices.
- Slice-wise Test Generation: The LLM generates test cases for each slice individually using in-context learning with designed templates and examples.
- Error Correction: Before executing the generated test code, we instruct the large language model to identify and correct any potential errors it detects, reducing the likelihood of runtime failures.
- Test Aggregation: Finally, the tests for each slice are combined into a comprehensive suite for the entire method.

In our evaluation across 6 open-source Python projects, HITS outperforms state-of-the-art tools, including ChatUniTest, achieving 5–10% higher line and branch coverage on complex methods.

Key Contributions

- Introduced slice-level test generation using LLMs to tackle low coverage in complex methods.
- Developed HITS, a practical tool tailored to complex python methods, based on slice-by-slice unit test generation.
- Validated effectiveness through comprehensive experiments showing significant improvements over both traditional and LLM-based baselines.

III. APPROACH

The HITS system (High-coverage LLM-based Unit Test Generation via Method Slicing) employs a two-step process to generate high-quality, high-coverage unit tests for Python methods. The system leverages the power of large language models (LLMs) to systematically decompose methods into smaller, more manageable components and then generate unit tests for each of these components. This approach significantly enhances the accuracy and efficiency of test generation, addressing the challenges posed by the complexity and scale of modern software codebases.

A. Step 1: Method Decomposition through Slicing

The first stage of the HITS system involves the decomposition of the target Python code into smaller logical units known as slices. Methods in real-world applications often contain complex logic with numerous branching conditions, making it difficult to generate effective unit tests for the entire method at once. To mitigate this, the method is broken down into distinct, isolated slices that represent smaller, coherent units of code, each with a specific focus or responsibility. This decomposition is achieved through a combination of static analysis and prompt engineering techniques. The static analysis step identifies all relevant components of the method, such as invoked fields, methods, and dependencies, and extracts detailed information from Python docs, method signatures, and other associated documentation.

```
//  Slice 1: Initialize and Prepare Rendering Data
    final String lpad = createPadding(leftPad);
    final String dpad = createPadding(descPad);

//  Slice 2: Create a list of option strings with proper formatting
    int max = 0;
    final List<StringBuffer> prefixList = new ArrayList<>();
    final List<Option> optList = options.helpOptions();
    if (getOptionComparator() != null) {
        Collections.sort(optList, getOptionComparator());
    }
    for (final Option option : optList) {
        // Create option string with proper formatting
        StringBuffer optBuf = new StringBuffer();
        // Append option details to optBuf
        prefixList.add(optBuf);
        max = Math.max(optBuf.length(), max);
    }

// Slice 3: Render each option with proper padding and description
    int x = 0;
    for (final Iterator<Option> it = optList.iterator(); it.hasNext(); ) {
        final Option option = it.next();
        StringBuilder optBuf = new StringBuilder(prefixList.get(x++).toString());
        if (optBuf.length() < max) {
            optBuf.append(createPadding(max - optBuf.length()));
        }
        optBuf.append(dpad);
        // Append option description
        if (option.getDescription() != null) {
            optBuf.append(option.getDescription());
        }
        // Render wrapped text
        renderWrappedText(sb, width, max + descPad, optBuf.toString());
        if (it.hasNext()) {
            sb.append(getNewLine());
        }
    }
```

Fig. 1. Slicing

Once the method has been analyzed, this information is used to generate human-readable prompts for the LLM. The LLM is prompted to summarize the method's purpose, extract key components and interactions, and break down the method into a series of logically connected tasks. These tasks or slices are represented in a structured format, typically in JSON, containing both textual summaries and the corresponding code snippets. The slices are designed to be small enough to allow targeted test

generation while retaining enough context to capture the full functional behavior of the original method. By decomposing the method in this way, the system reduces the complexity of generating tests for large methods, making the process more manageable and effective.

B. Step 2: Unit Test Generation for Each Slice

Once the method has been decomposed into smaller slices, the next step is to generate unit tests for each slice. This is achieved through a process of targeted test generation, where the LLM is prompted to create unit tests based on the previously identified method slices. Each slice, having its own set of logical conditions and dependencies, is tested independently to ensure comprehensive coverage. The LLM is guided to first identify the variables and method calls involved in each slice, followed by a determination of the necessary input conditions and potential branching paths.

The LLM is then tasked with constructing a test case for each slice. These test cases aim to cover all lines and branches within the slice by considering the various execution paths that could be taken during runtime. The generated test cases are designed to be self-contained, ensuring they are executable without needing the broader application context. They focus on testing the specific slice in isolation, allowing for clear and targeted validation of the method's logic. Once generated, these test cases are executed to verify their correctness and functionality. The system ensures that only the tests that compile and pass execution are considered valid, helping to ensure high-quality test coverage for the method under test.

Through these two steps—method decomposition and targeted test generation—HITS is able to produce comprehensive unit test suites with high coverage, all while addressing the complexities introduced by large and intricate methods. This approach utilizes the strengths of LLMs for reasoning about code structure and test generation, offering a significant improvement over traditional automated testing tools that rely on search-based or rule-based strategies.

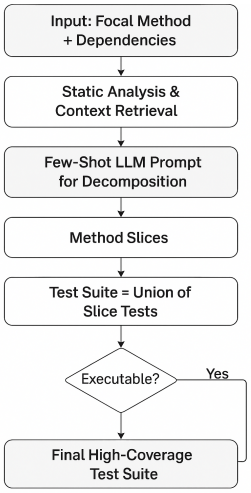

Fig. 2. Work Flow

IV. EVALUATION

In this section, we present the detailed evaluation methodology for testing the performance and effectiveness of the proposed system. The evaluation process consists of multiple phases, including dataset construction, experimental setup, performance metrics, and research questions guiding the evaluation.

A. Experiment Setup

The experiment setup involves the creation of a dedicated dataset, followed by the configuration of tests and baselines to evaluate the system under different conditions. We focus on evaluating the system using both synthetic and real-world projects to measure its generalization and effectiveness in different environments.

1) Dataset Construction: The dataset used in this evaluation is designed by collecting software projects from the Internet. In total, 6 projects are crawled from GitHub, each containing methods that vary in complexity. These projects are chosen to provide a diverse and representative sample for testing the system.

The key criteria used for selecting the projects include:

1) **Relevance to ChatUniTest**: The projects or their domains should be relevant to the fields evaluated by ChatUniTest. This tool is widely used for automated test generation, so it's inclusion ensures that the projects are aligned with modern software engineering practices.
2) **Complexity**: Projects must contain between 1 to 30 complex methods. This selection criterion ensures that the dataset is neither too

simple nor too large, providing a balanced challenge for the system.

3) **Relevance and Popularity**: Preference is given to projects with high relevance to the domain and a significant number of GitHub stars. This helps ensure that the projects are widely used and well-maintained.

To prevent large projects from introducing bias due to high token costs, any projects with excessive size (in terms of the number of methods or overall complexity) are discarded. This ensures that the dataset remains manageable and that the evaluation results are not skewed by computational constraints.

2) Generation Configuration: We use OpenAI's `gpt-turbo-3.5-012` model for code generation and testing. This model is chosen because it offers a balance of efficiency and effectiveness in generating code snippets, test cases, and other necessary outputs. The following configurations are applied:

- **Prompt Design**: The model is provided with prompts that specify the type of output required (e.g., generating unit tests for complex methods).
- **Test Generation**: For each selected project, the model is tasked with generating test cases for methods based on the provided project data. The tests are evaluated for completeness, correctness, and the ability to cover edge cases.
- **Baseline Generation**: We compare the model's outputs with traditional test generation tools to measure the relative effectiveness.

B. Evaluation Metrics

The performance of the LLM-generated tests is evaluated using several key metrics:

- **Test Coverage**: The percentage of methods in the project for which tests are generated. Higher coverage indicates a more comprehensive test suite.
- **Test Accuracy**: The correctness of the tests in detecting bugs or errors within the code. This is assessed by running the tests on the project code and verifying that they correctly identify faults.
- **Edge-Case Detection**: The ability of the generated tests to identify edge cases or corner scenarios that might not be easily covered by standard test cases.

V. OPTIMIZING PROMPTS

Prompt Self-refinement: The technique of self-refinement assists the LLM in better comprehending our intentions. Self-refinement involves asking the LLM to optimize the prompt on its own. In our initial experiments, we noticed that the LLM often failed to meet output format requirements (such as generating JSON format), or it produced outputs with empty slices, irrelevant analysis, or no results at all. To address this, we applied the self-refinement technique by instructing the LLM to "Please refine the prompt with clearer structure and more fluent expressions." This led to an improved prompt, allowing the LLM to generate outputs that adhered to the desired format and followed the specified instructions. Since our instructions and requirements were complex and not immediately intuitive, self-refinement facilitated alignment between the LLM and the human user. The optimized prompt then guided the LLM to generate the expected output based on our detailed instructions.

VI. LIMITATIONS AND DEVIATIONS FROM ORIGINAL IMPLEMENTATION

While our approach is inspired by the original research paper, several components could not be implemented due to practical constraints. Firstly, we used Python instead of Java, whereas tools like EvoSuite are tailored specifically for Java-based test generation. Consequently, a direct comparison with EvoSuite was not technically feasible, and we excluded it from our evaluation. Secondly, we did not incorporate SymPrompt, which is more aligned with mutation testing objectives rather than coverage maximization. As our focus was on achieving high coverage for complex functions, integrating mutation-based techniques was deemed outside the current scope. Thirdly, the feedback loop for non-executable test cases was not automated. Since we utilized VSCode for execution, any non-executable test case had to be manually identified and revised, unlike the original paper's implementation where such test cases were automatically fed back for rectification. As a result, we encountered a higher number of failing test cases, which negatively impacted the pass rate of our method. This manual intervention slightly reduced the efficiency and accuracy of the test refinement process, highlighting

the importance of automating this loop in future work.

VII. CONCLUSION

In conclusion, the current limitations of LLM-based unit test generation tools, particularly in achieving high coverage scores for complex methods, highlight the need for a more refined approach. To address this challenge, we propose HITS, an innovative solution that decomposes the method-to-test into manageable slices, applying a 'divide-and-conquer' strategy to generate unit tests slice by slice. Through comprehensive evaluation, we have demonstrated that HITS outperforms existing state-of-the-art LLM-based test generation tools in terms of coverage and effectiveness. This approach significantly enhances the ability to generate high-coverage unit tests for complex Python methods, showcasing the potential of leveraging LLMs in combination with strategic decomposition techniques for improved software testing practices.

VIII. IMPORTANT LINKS

Click to access the Notebook on Google Drive
Click to access the PPT
Click to access GitHub repo

IX. REFERENCES

- Zhang, J., Chen, Y., Wang, S., Ou, Y. (2023). HITS: High-Coverage LLM-based Unit Test Generation via Method Slicing. In Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). ACM.
- HITS (Java) GitHub Repository – eecshope/HITS
- Python `unittest` Framework Documentation
- OpenAI API Documentation – LLMs for Code Generation and Repair
- Youtube: Using Coverage.py with Python coverage