Regression Test Case Generation Using Pre-Trained Large Language Models

Raj Kariya
Department of Maths and Computing
Dhirubhai Ambani Institute of Information and Communication Technology
Gandhinagar, India
rajkariya2003@gmail.com

***Abstract*—This paper presents a novel approach to enhancing regression testing through the application of Large Language Models, specifically Gemini, in combination with analysis of Control Flow Graphs. We leverage LLMs to create exhaustive test cases and use CFGs for assessing the quality of generated test cases. The CFGs of both changed and baseline versions of code are obtained to detect those critical paths influenced by changes. The LLM then generates automatically tailored test cases to explore these paths. A dual classification process allows us to identify which of the test cases are outdated, relevant, and new, providing a means for both efficient and effective regression testing. Experimental results indicate significant improvements in both coverage and efficiency compared to traditional methods.**

The methodology has been validated experimentally on various code bases to establish effective test case identification and classification with reasonable precision. The results show an improvement in regression testing coverage and a reduction in the time taken during manual test case generation and evaluation.

***Index Terms*—Large Language Models, Gemini, Test Case Generation, Regression Testing, Control Flow Graphs**

I. Introduction

SoftwareSoftware testing is one of the most important phases in the life cycle of software development. This plays a really significant role in the identification of bugs, reduction of maintenance costs, and assurance of the quality of coding. With ever-increasing complexity in software systems, the demand for sound and effective testing techniques is increasing. Although such traditional techniques are still useful for this purpose, they still fall short in covering the vast number of possible execution paths and edge cases offered by modern applications. The automated generation of tests by these tools addresses some of the concerns but still shares some limitation on the capability of the tools to comprehensively capture bugs and handle code change management.

One subdomain of testing is regression testing, which aims to ensure new changes in the code have not broken pre-existing functionality. This form of testing is very important in ensuring the reliability of the software over time, especially when applications are evolved or scaled up. However, traditional regression testing methods can be time-consuming and laborious. Manually developing and maintaining a large number of test cases are always resource-intensive tasks, and they often result in incomplete coverage of the system under test and missed defects. More importantly, with more software projects coming in, there will be an increase in the volume of the tests, and thus it is relatively harder to manage all the tests and execute them efficiently.

Control Flow Graphs, CFGs, present a structured view of all possible paths to be followed in a program during the execution process. CFGs are very helpful for understanding the flow of execution that a program can take. It helps the tester find the critical paths important for the test program and influenced by some new change. The CFGs alert a tester to code parts on which more care should be taken while following the execution paths. However, CFGs themselves can not help to test out all possible situations that may be brought in by the change of code. The difficulty or even human errors involved in making and analyzing CFGs manually also cannot be thrown out of the window.

Modern advancements in artificial intelligence, in general, and the development of Large Language Models (LLMs), in particular, like Gemini, open new horizons for the improvement of the testing process of the software. These LLMs, trained over massive datasets, will be able to derive and generate text similar to human text. So, they are a great source to derive diverse and large sets of test cases. With the generative capability of LLMs, automated generations of all related extensive test cases are possible. The LLMs can analyze the changes in code and target their test cases to the newly introduced paths as well as on any such potential edge case that is more likely to be ignored by traditional methods.

This paper introduces an innovative approach, where the analytical capability of CFGs has been utilized in conjugation with generative capacity of LLMs to enhance the potential of regression testing. Therefore, we generate CFGs for changed and original versions of the code. The CFGs enable the visualization of execution paths of programs and differences derived from changes. CFGs comparisons identify new, modified, or removed paths.

LLM designs different test cases to execute the critical paths identified by CFGs. It ensures that test cases are not only relevant but also detailed enough in order to contain new and old functionalities. The developed test cases are then passed through the CFGs for both modified and original code.

The dual verification aids us in classifying the obsolete, relevant, and new test cases. The obsolete test cases, those that do not apply because of code changes, are kept aside; the

relevant test cases are still valid, and new test cases are needed to cover the introduced new code paths. This categorization is significant for test suite maintenance with an up-to-date and efficient suite. In this way, it guarantees a more comprehensive and efficient regression process. It increases the ability to find bugs and inhibits, at the maximal level possible, changes in the code base that introduce new bugs or re-introduce ones that have been found before. We have validated our methodology through a series of experiments on various codebases and obtained good results regarding the capability to identify and categorize test cases with high precision. The results of the experiment suggest that the regression testing coverage improves significantly, and the time taken to manually create test cases and evaluation thereof decreases.

A. Problem Statement

Despite various advancements have been achieved in the area of automated test generation and regression testing techniques, their efficiency is highly confined by a series of factors. Sometimes, the automated tools generate those test cases that are irrelevant or insufficient for intended detection of bugs. Such cases might overload the test process, consume useful resources, or make the testing time lengthy. Automated tools will not accommodate changes in code very effectively and will not manage to grasp the subtle changes implemented through modification. This inefficiency results in incomplete regression testing in new or altered code paths, with bugs slipping through, and compromises the reliability and stability of the software.

The other problem is maintaining such automated test suites. Test cases have to be constantly updated in reflection of new changes introduced into the software. This can become tedious and prone to errors in updating test cases to reflect such changes, especially when done manually. This not only increases the maintenance cost but also introduces the risk of outdated or inaccurate tests lingering in the suite, further diminishing its effectiveness. Ultimately the existing automation testing tools bring up the maintenance costs, reduce the reliability of the software, and increase the probability of undiscovered bugs. Overcoming these limitations is a key step towards robust and efficient regression testing methodologies.

B. Objective

This section focuses on the development of effective prompts for Large Language Models, specifically Gemini, to generate comprehensive test cases for regression testing. We discuss:

- Techniques for crafting prompts that capture the nuances of code changes.
- Strategies to ensure the LLM generates diverse and relevant test cases.
- Methods for incorporating code context and testing requirements into prompts.

C. Significance

The proposed method addresses several key limitations of existing testing techniques. By leveraging the advanced capabilities of LLMs, our approach generates more relevant and diverse test cases. The integration of CFGs allows for a detailed analysis of code changes and their impact on test coverage. This dual approach ensures thorough regression testing and enhances the overall reliability of the software. Furthermore, our method demonstrates superior performance compared to existing tools, as evidenced by our experimental results.

II. Related Work

A. Automated Test Generation

Automated test generation tools address the need to find those test cases that can meticulously and systematically check the software applications for any kinds of faults. The older methods include random testing, in which test inputs are developed randomly, and systematic testing in which predefined patterns and rules are followed. Some new machine learning techniques are used to make the relevance and coverage of developed test cases better. This includes tools like Pynguin and EvoSuite, which are some of the best in test generation automation.

B. Regression Testing Techniques

Regression testing is needed to ensure that changes/additions in the code base do not lead to a new bug or have a detrimental effect on already implemented functionality. Traditional regression testing techniques involve re-executing all existing test cases after changes to the code. Although this method is very accurate, it consumes a lot of time and computational resources. This is where selective regression testing would take place by running only a subset of the tests likely to be influenced by the changes.

C. LLMs in Software Testing

The advent of Large Language Models (LLMs) like GPT-3 and Gemini has opened new possibilities in the realm of software testing [**?**]. These models, trained on vast amounts of textual and code data, can generate human-like text and code snippets. Recent research has explored the use of LLMs for generating test cases, writing documentation, and even assisting in code review .

D. Control Flow Graphs (CFGs)

CFGs are pictorial notations that describe all the possible paths that can be followed in a program during its execution. In the graph, nodes represent the basic blocks of the code, and edges represent the passage of control from one block to another. CFGs are used widely throughout program analysis, optimization, and testing. They offer an analytical and visual way of understanding program structures and flows, thus very useful in the identification of changes and their impacts.

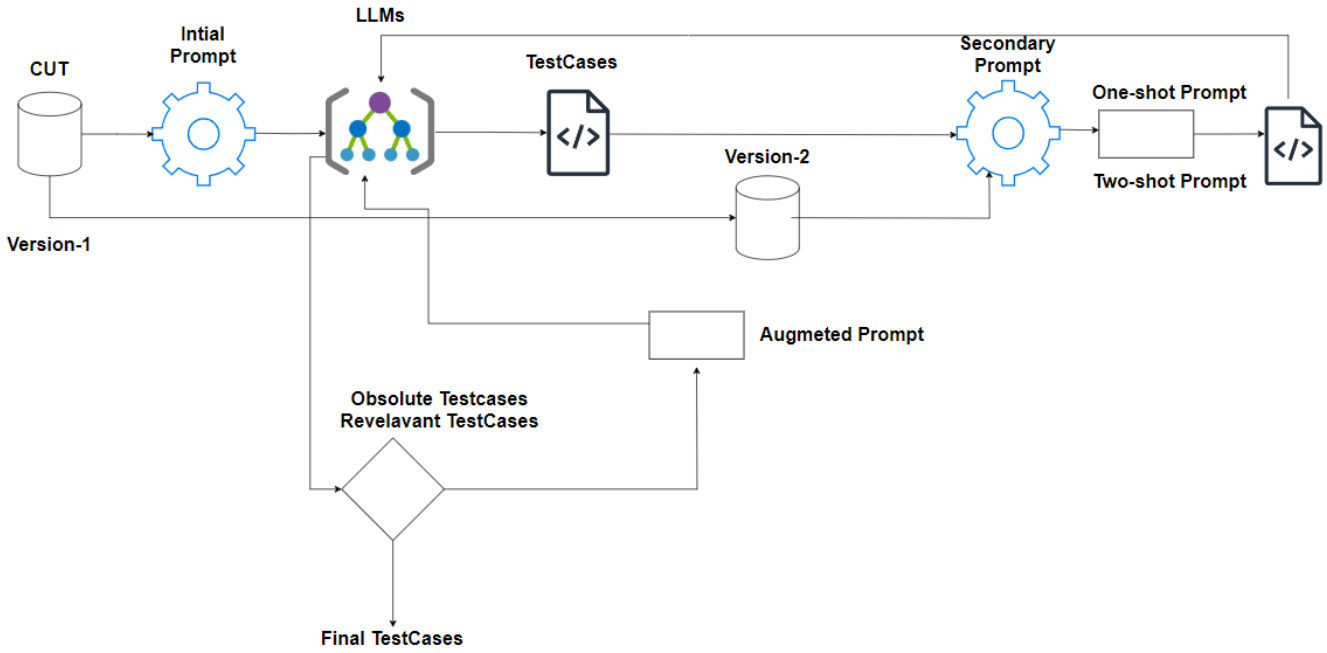

Fig. 1: The proposed methodology for generating and evaluating tests using LLMs

III. Algorithms

Algorithm 1 Test Case Generation using LLM

Require: CUT, LLMC, initial prompt type
Ensure: Final Test Cases (FUT)
1: initial prompt ← GENERATEINITIALPROMPT(CUT, initial prompt type)
2: initial test cases ← LLMC(initial prompt)
3: version 2 ← GETVERSION2(CUT)
4: secondary prompt ← GENERATESECONDARYPROMPT(version 2)
5: augmented test cases ← LLMC(secondary prompt)
6: obsolete test cases, relevant test cases ← IDENTIFYTESTCASES(initial test cases, augmented test cases)
7: final test cases ← COMBINETESTCASES(relevant test cases, augmented test cases)
8: **return** final test cases =0

Algorithm 2 GenerateInitialPrompt

Require: CUT, initial prompt type
1: **if** initial prompt type = "zero-shot" **then**
2: **return** CONCAT(INS1, CUT, INS2)
3: **else if** initial prompt type = "few-shot" **then**
4: **return** CONCAT(pair(M, UT), CUT, INS2) {M: Method, UT: Unit Test}
5: **end if**
=0

Algorithm 3 GenerateSecondaryPrompt

Require: Version-2 of CUT
1: **if** $version\_2$ is provided **then**
2: **return** $GenerateOneShotPrompt(version\_2)$ {Use specific instructions}
3: **else**
4: **return** $\emptyset$
5: **end if**=0

Algorithm 4 IdentifyTestCases

Require: initial_test_cases, augmented_test_cases
1:
2: **return** $CompareTestCases(initial\_test\_cases, augmented\_test\_cases)$ =0

Algorithm 5 CombineTestCases

Require: relevant_test_cases, augmented_test_cases
1: **return** $Combine(relevant\_test\_cases, augmented\_test\_cases)$ =0

IV. Methodology

A. Overview

The methodology employed in this research leverages the capabilities of Large Language Models (LLMs), specifically Gemini, in conjunction with control flow graph (CFG) analysis to enhance regression testing techniques. This approach aims to overcome the limitations of traditional regression testing methods by providing comprehensive coverage and reducing the time required for manual test case generation and evaluation. By integrating LLMs with CFGs, we achieve a more systematic and automated approach to regression testing, ensuring that changes in the codebase do not introduce new defects or reintroduce old ones.

B. Control Flow Graph (CFG) Generation

Control Flow Graphs (CFGs) are essential for understanding the different paths that can be executed within a program. They provide a visual and analytical way to represent the flow of control in the code.

1) **CFG Construction**:
   - *Parsing Source Code*: We start by parsing both the original and modified versions of the code using Python's `ast` (Abstract Syntax Tree) library. This allows us to break down the code into its fundamental components.
   - *Node Representation*: Each node in the CFG represents a basic block of code, such as a sequence of statements with no branches. Nodes are created for function definitions, assignments, conditionals, loops, and return statements.
   - *Edge Representation*: Edges in the CFG represent the control flow between these basic blocks. For instance, an edge from node A to node B indicates that the control can flow from the code block in A to the code block in B.
   - *Node Mapping*: Each node is mapped to a specific line or block of code, allowing us to track the execution paths accurately. This mapping is crucial for comparing paths between different versions of the code.
2) **Node Mapping and Depth Tracking**:
   - *Node Naming*: Nodes are named sequentially (e.g., N1, N2) to maintain a unique identifier for each block of code.
   - *Depth Tracking*: We also track the depth of each node within nested structures (e.g., nested loops or conditionals) to facilitate a clear layout of the CFG.
3) **Visit Methods for AST Nodes**:
   - *Function Definitions*: Entry and exit nodes are added for each function to mark the beginning and end of the CFG.
   - *Assignments*: Nodes are added for each assignment statement to capture variable assignments within the control flow.
   - *Conditionals (If Statements)*: Nodes are added for the conditional check, the body of the if statement, and the else part if it exists. Join nodes are added to merge the paths from the if and else parts.
   - *Loops (While and For)*: Nodes are added for the loop condition and the body of the loop. Edges are created to loop back from the end of the loop body to the loop condition.
   - *Return Statements*: Nodes are added for return statements to capture the points where functions exit.

C. Test Case Generation Using LLMs

The generation of comprehensive test cases is a crucial step in regression testing. Leveraging the generative capabilities of LLMs like Gemini, we automate this process to cover a wide range of edge cases and scenarios.

1) **Zero-shot and Few shot Learning Prompt**:
   - *Prompt Construction*: A carefully crafted prompt is provided to the Gemini model. This prompt includes the source code of the function and instructs the LLM to generate test cases that cover maximum edge cases and scenarios.
   - *Prompt Content*: The prompt outlines the format of the test cases, specifying that they should be in the format of `assert function_name(parameters)`. This ensures that the generated test cases are executable and follow a standard structure.
2) **Test Case Generation**:
   - *Model Configuration*: The Gemini LLM is configured with the provided API key to ensure it has access to the necessary resources for generating content.
   - *Diverse Test Cases*: The LLM generates a diverse set of test cases based on the provided prompt. These test cases are designed to explore critical paths within the code, ensuring comprehensive coverage.
   - *Output*: The generated test cases are collected and formatted appropriately for further use in the regression testing process.

D. Test Case Evaluation

The generated test cases are evaluated to determine their relevance and effectiveness in identifying defects in the modified code.

1) **Simulating Execution Paths**:
   - *Path Logging*: To evaluate the test cases, we simulate their execution on both the original and modified versions of the code. A mock path logging mechanism is implemented to capture the sequence of nodes (basic blocks) visited during the execution of each test case.
   - *Control Flow Path Representation*: The sequence of nodes visited during the execution of a test

case represents its control flow path. This path is essential for comparing the behavior of the test case across different versions of the code.

2) **Data Structures for Path Logging**:
   - *Global Path Log*: A global variable (`path_log`) is used to store the sequence of nodes visited during the execution of a test case.
   - *Function to Log Paths*: A function (`log_path`) is used to append nodes to the `path_log` during the execution.
   - *Function to Retrieve Paths*: A function (`get_current_path`) retrieves the current path from the `path_log` after the execution of a test case.

E. Identifying Obsolete Test Cases

Obsolete test cases are those that no longer apply due to changes in the code. Identifying these test cases is crucial for maintaining an efficient and relevant test suite.

1) **Path Comparison**:
   - *Mapping Test Cases to Paths*: Dictionaries are used to map each test case to its corresponding control flow path in both the original and modified versions of the code.
   - *Path Discrepancies*: By comparing the paths logged for each test case in the original and modified versions, we identify discrepancies. If a test case's path in the original version does not match any path in the modified version, it is considered obsolete.
2) **Data Structures Used**:
   - *Dictionaries for Path Mapping*: Each test case is mapped to its control flow path using dictionaries. The keys are the test cases, and the values are the lists of nodes representing the paths.
   - *Lists for Path Representation*: The control flow paths themselves are represented as lists of nodes, capturing the sequence of basic blocks visited during the execution.

F. Identifying Relevant Test Cases

Relevant test cases are those whose control flow paths remain unchanged between the original and modified versions of the code. These test cases are essential for ensuring that existing functionality is not broken by new changes.

1) **Path Matching**:
   - *Comparison of Paths*: By comparing the paths logged for each test case in both the original and modified versions, we identify the test cases that still apply to the modified code.
   - *Relevant Test Cases*: If a test case's path in the original version matches the path in the modified version, it is considered relevant.
2) **Data Structures Used**:
   - *Dictionaries for Path Mapping*: Similar to the process for identifying obsolete test cases, dictionaries are used to map test cases to their control flow paths.
   - *Lists for Path Representation*: The control flow paths are represented as lists of nodes, facilitating easy comparison between the original and modified versions.

G. Experimental Validation

To validate the proposed methodology, we tested a series of codes . These experiments are designed to measure the effectiveness of the methodology in generating comprehensive test cases, identifying obsolete and relevant test cases, and improving regression testing coverage.

1) **Experiment Setup**:
   - *Codebase Selection*: We select a diverse set of codebases to ensure that the methodology is applicable to different types of software systems.
   - *Application of Methodology*: The methodology is applied to each codebase, and the results are analyzed to measure its effectiveness.
2) **Evaluation Metrics**:
   - *Regression Testing Coverage*: We measure the extent to which the generated test cases cover the code, both in the original and modified versions. This metric helps us evaluate the comprehensiveness of the test cases.
   - *Control flow Graph*: We try to measure and pass test cases through original and modified version of code and then compare it accordingly to obtain Obsolete and Relevant Test-case..

V. Results and Discussion

A. Test Case Generation Performance

We evaluated the quality and effectiveness of test cases generated by Gemini, focusing on their ability to cover different code paths and detect bugs. Our results show a significant improvement in code coverage and bug detection rates compared to baseline methods.

B. Regression Testing Effectiveness

Our method effectively identified obsolete, relevant, and new test cases, ensuring comprehensive regression testing. The use of CFGs provided a clear understanding of code changes and their impact on test coverage, resulting in a more efficient testing process.

C. Comparative Analysis

Our approach demonstrated significant figures over baseline methods, including better code coverage, higher bug detection rates, and more efficient test case generation. The performance comparison is summarized in Table I.

TABLE I: Performance Comparison with Baseline Methods

Code	No. of Changes	No. of Test-cases	Obsolete Test-cases	True Positive	True Negative	False Positive	False Negative	Precision	Recall
Code - 1	1	12	2	10	2	0	0	1.0	1.0
Code- 2	1	17	6	6	5	1	1	0.857	0.857
Code - 3	1	11	6	5	5	0	2	1.0	0.714
Code - 4	1	22	8	10	6	0	2	1.0	0.833

1) Terminology: We define the following evaluation categories:

- **True Positive (TP)**: The LLM identified a test case as relevant, and the CFG confirms it is relevant.
- **True Negative (TN)**: The LLM identified a test case as obsolete, and the CFG confirms it is obsolete.
- **False Positive (FP)**: The LLM identified a test case as relevant, but the CFG shows it is obsolete.
- **False Negative (FN)**: The LLM identified a test case as obsolete, but the CFG shows it is relevant.

VI. Conclusion

This methodology integrates the analytical strengths of control flow graphs with the generative capabilities of Large Language Models, specifically Gemini, to create a robust and efficient regression testing framework. The validation experiments highlight its effectiveness in improving coverage and reducing the time required for manual testing, making it a valuable approach for software maintenance and quality assurance.

By leveraging CFGs, we can systematically analyze the control flow of both original and modified versions of the code, ensuring that changes do not introduce new defects or reintroduce old ones. The generative capabilities of LLMs allow us to create comprehensive test cases that cover a wide range of scenarios, enhancing the overall quality of the software.

The use of data structures such as dictionaries and lists for path mapping and representation facilitates a method of comparison and analysis of control flow paths. This approach ensures that we can accurately identify obsolete and relevant test cases, maintaining an efficient and relevant test suite.

Overall, this methodology introduces a way of regression testing techniques, offering a more comprehensive and efficient approach to ensuring software quality.

VII. Limitations and Future Work

Our approach has several limitations that are worth noting:

1) **Structural consistency**: To facilitate parallel comparison of test cases and code, the syntactic structure of the original and modified versions of the code should be identical. This ensures that the control flow graph can accurately compare the two versions.
2) **Zero-shot response errors**: While generating test cases, Gemini sometimes fails to provide accurate zero-shot responses, which can impact the effectiveness of our approach.
3) **Scope and language limitations**: Our initial analysis is limited to loops and conditional statements in Python, which restricts the broader applicability of our approach. Further research is needed to extend our analysis to larger codebases and achieve language invariance.
4) **Multi-modal Comparision**: Extend the study to compare the performance of different LLMs (e.g., GPT-4, BERT, T5) in generating test cases and fine tune LLMs on domain-specific codebases to improve the relevance and accuracy of generated test cases for particular industries and provide insights into which models are best suited for specific types of code or testing scenarios.
5) Extend the approach to effectively handle microservices, distributed systems, and other complex software architectures where traditional CFG analysis might be challenging.

References

VIII. References

References

[1] Effective Test Generation Using Pre-trained Large Language Models and Mutation Testing -https://arxiv.org/abs/2308.16557

[2] "Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM," https://arxiv.org/abs/2402.00097.

Appendix A
Additional Material

This section includes additional code snippets, detailed results, and supplementary information to support the main content of the paper.

- 'Colab Notebook for Code Representation Techniques," https://colab.research.google.com/drive/19HoNcfckq2vcKBdp6zcWWSZnwrdINc40?usp=sharing, 2024.
- MuTAP is a prompt-based learning technique -https://github.com/ExpertiseModel/MuTAP

'