VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Luong Anh Tuan

LEVERAGING LARGE LANGUAGE MODELS FOR AUTOMATED TEST DATA GENERATION

BACHELOR'S THESIS
Major: Computer Science

HANOI - 2025

VIETNAM NATIONAL UNIVERSITY, HANOI
UNIVERSITY OF ENGINEERING AND TECHNOLOGY

Luong Anh Tuan

LEVERAGING LARGE LANGUAGE MODELS FOR AUTOMATED TEST DATA GENERATION

BACHELOR'S THESIS
Major: Computer Science

Supervisor: PhD. Nguyen Duc Anh

HANOI - 2025

Statement of Integrity

I hereby declare that the graduation thesis entitled "Leveraging Large Language Models for Automated Test Data Generation" presented in this report is entirely my own work. The content does not involve any form of plagiarism or the use of others' results without proper citation. The proposed method in this thesis is the result of my own research, conducted under the supervision of PhD. Nguyen Duc Anh. I take full responsibility for any violations of the regulations of the University of Engineering and Technology, Vietnam National University, Hanoi.

Ha Noi, December 8th 2025

Supervisors

Student

Acknowledgements

First and foremost, I would like to express my sincere gratitude to the Faculty of Information Technology – University of Engineering and Technology, Vietnam National University, Hanoi, for providing me with the opportunity to study, practice, and accumulate the essential knowledge that has built a solid foundation for me today.

Next, I would like to extend my deepest appreciation to PhD. Nguyen Duc Anh for his dedicated teaching, guidance, and support, as well as for giving me the opportunity to study and conduct research at the Software Quality Assurance Laboratory over the past year and during the completion of this thesis. His patient mentorship and wholehearted assistance have been a great source of motivation, helping me to improve myself and achieve the results I have today. I always feel extremely fortunate and proud to have been one of his students.

Finally, I would like to express my heartfelt thanks to all my classmates in K67CS for their constant companionship, support, and for creating such a joyful and connected learning environment throughout our years at the university.

Abstract

Abstract: Software testing is an important phase in the software quality assurance process. With an increasingly large scale of modern software systems, automated testing is essential to increase efficiency and optimize costs in the testing process. Automated testing has 2 approaches: generating test data to detect errors and generating unit test data to maximize code coverage. To maximize code coverage, there are 2 approaches: Concolic Testing and LLM-based test data generation. However, these approaches have specific drawbacks. Concolic Testing is difficult to handle complex constraints, such as constraints related to the C++ Standard Template Library (STL). LLM-based methods face problems of large prompt sizes, outdated information, and lack of security. Therefore, the thesis proposes a method that fine-tunes LLM and integrates it with Concolic Testing to generate test data with higher code coverage. The input is the source code of the focal method and the output is the test cases to maximize code coverage. The main idea of the method is to leverage the fine-tuned LLM to generate test data to cover the execution paths that Concolic Testing does not cover. To prove the efficiency of the method, it was experimented on a dataset of 76 focal methods that were not in the training set, using a fine-tuned model of CodeT5+ 770M. In the cases with different coverage, 22 of 24 focal methods achieved higher coverage with the proposed method. The results demonstrate the practical usage of the proposed method.

Keywords: *Concolic Testing, Automated Test Data Generation, Large Language Models, Path Exploration*

List of Figures

List of Tables

Contents

Chapter 1

Introduction

Software testing is a critical phase of the quality assurance process. Software testing is divided into levels such as unit testing, integration testing, system testing, and acceptance testing [21, 9, 19]. In the context of software projects becoming larger and more complex, automated software testing is essential to enhance efficiency and optimize costs during the testing process, especially for a common language like C++. There are many published studies on automated testing for C++ source codes in the community with two main paths: generating test data for errors detection and generating test data to maximize code coverage [14, 8, 13, 15]. With the aim of generating high-coverage data, there are two typical methods: concolic testing and LLM-based test data generation [21, 6, 20]. However, both methods still suffer from specific limitations. Concolic testing has difficulty in solving sophisticated constraints while the LLM method struggles with big prompt sizes, outdated knowledge and a lack of privacy due to using models from third parties.

Concolic testing [18, 12] is a method combining static and dynamic execution. There are 3 main phases in this method: initializing test data, executing test data and analyzing execution paths, creating new test data. In the beginning, it initializes random test data for execution. After execution, it analyzes execution paths in CFG to locate suitable unvisited execution paths. Finishing the analyzing process, it generates new test data by using a constraint solver to solve the constraints of selected execution paths. The method repeats the process of executing test data, analyzing execution paths, and generating new test data until maximum coverage is reached or resources are exhausted. The problem of this method is creating new test data. After choosing suitable execution paths, it extracts and sends constraints' information to constraint solver such as Z3 to create new test data for this execution path. Nevertheless, solvers such as Z3 just work well on primitive data types and accuracy significantly decreases with data using libraries such as C++ STL or complex data structures. Solvers

encountering these data types often run out of time, generate failed test data, or generate unsatisfactory test data. This makes the method face challenges in creating test data that fulfill the constraints and the chosen execution path.

The development of LLMs opens up many new opportunities in software testing [21, 23, 11]. Traditional testing methods such as concolic testing often meets difficulty when dealing with data types from libraries, such as C++ Standard Template Library (STL)., or complex data structures. Meanwhile, LLMs are able to understand source code and contexts since they are pre-trained on a large amount of source code and related context documents. This helps LLMs generate test data which can satisfies the constraints that traditional testing methods have difficulty processing. Despite such potential, the method of using LLMs still has limitations. First, LLMs may not be pre-trained on data about the latest versions of libraries. This makes generating test data for new versions less accurate, if that version has major changes. In addition, to generate test data suitable for the purpose and desired format, the prompt for the LLMs may be very large so that models can understand and generate data properly. Finally, when using LLMs like GPT or Deepseek, the data must be sent to their servers for processing. This is a taboo for businesses in securing their companies and customers' important information.

In order to solve these problems, the thesis proposes a method called AkaLLM. The main idea of the method is to combine fine-tuned LLMs and concolic testing to generate test data that can achieve high coverage. Fine-tuning LLMs and using them in a local environment helps solve the security problem above. The input of the method is the unit source code that needs test data. The output is the test data with maximum coverage when executed. The method consists of three main phases: concolic testing, minimizing the number of paths required to generate new test data, and using fine-tuned LLMs to generate new test data. In the concolic testing phase, the method performs test data generation as described above. At the end of the concolic testing phase, the method decides whether to execute the next phases by checking whether the maximum coverage has been achieved. If the maximum coverage has not been achieved, the method continues with the next phases. In the phase of minimizing the number of paths required to generate new test data, the method analyzes CFGs to collect the execution paths and constraints that have not been visited. Information of the execution paths and related context are combined to form a prompt in the third phase, which solves the problem of the prompt length. In the third phase, from the generated prompts, the models generate test data that suit the execution paths. These new test data are executed to calculate the final coverage of all test cases. This phase helps overcome the difficulty of dealing with complex constraints of concolic testing.

To demonstrate the effectiveness of the proposed method, the method is implemented on 174 unused functions in the training dataset. The method is compared by implementing the proposed method and Concolic testing on the AKAAUTO tool to compare the coverage achieved by the two methods. The results show that in 61 functions there are differences in coverage between the two methods, up to 56 functions the proposed method achieves higher coverage. The main contributions of the method can be summarized as follows:

- The thesis proposes the LLM fine-tune method to generate test data with high coverage, overcoming the limitations of Concolic testing.
- The thesis proposes the use of partial method instead of focal method to reduce the prompt size while still retaining full information about the execution path to be covered.

The remaining chapters of this thesis are organized as follows. Chapter 2 introduces the background knowledge required to understand the topic of automated test data generation for C++ projects. Chapter 3 presents an overview of the proposed method together with a detailed explanation of its phases. Chapter 4 describes the experiments that were carried out to evaluate the effectiveness of the approach. Chapter 5 discusses about the threats to validity. Chapter 6 introduces the related works to the thesis. Finally, Chapter 7 provides the conclusions, highlights the main results, and discusses the limitations as well as possible directions for future development.

Chapter 2

Background

2.1 Control Flow Graph

The control flow graph (CFG) is a directed graph used to represent the execution flow of a program. The automated testing techniques usually depend on analyzing the control flow graph to generate test data. The the control flow graph is defined as follows:

Definition 2.1: *"A control flow graph is a directed graph which includes nodes representing statements or groups of statements, and edges representing control flow between them. If nodes i and j are part of the control flow graph, there exists an edge from i to j if the statement corresponding to j can be executed immediately after the statement corresponding to i."*[7]

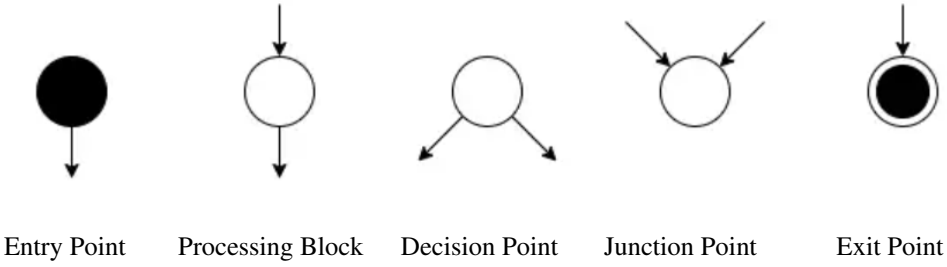

Figure 2.1: Base components in CFG.

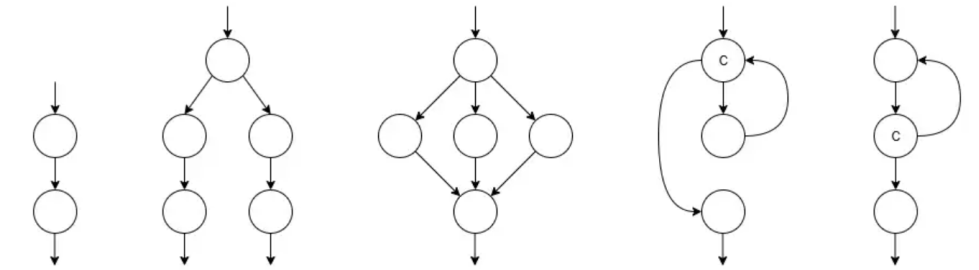

Figure 2.2: Popular structures of CFG.

- **Entry Point:** Marks the starting point of the program.
- **Processing Block:** Represents assignment, declaration, and initialization statements.
- **Decision Point:** Represents conditional statements in control structures, having multiple outgoing edges corresponding to different condition branches.
- **Junction Point:** Represents a statement executed immediately after branch statements, having multiple incoming edges corresponding to converging branches.
- **Exit Point:** Marks the termination of the program.

2.2 Execution Path

An execution path is a set of nodes in a control flow graph (CFG) that are traversed in order. In the context of testing based on CFG, an execution path refers to the set of nodes in the sequence traversed during program execution with a specific input data. The number of execution paths depends on the number of nodes and edges in the control flow graph. As the source code becomes larger and more complex, the number of execution paths increases significantly[7]. In practice, to achieve necessary coverage and optimize costs, automated testing techniques need to identify execution paths corresponding to nodes that have not yet been visited.

In order to generate a test dataset that covers execution paths, dynamic testing methods often collect the constraints along those paths and input them into a Z3 solver to solve. However, not all constraint systems have solutions. For some execution paths, the conditional statements may contain conflicting constraints, making those paths impossible to explore. Additionally, some execution paths include loops that could lead to infinite repetition if the terminating condition is

not satisfied. Therefore, these cases need to be avoided in test data generation to optimize costs and enhance efficiency in the testing process.

2.3 Code Coverage

Code coverage represents the percentage of source code that has been tested compared to the total source code of the program. Higher coverage indicates that more parts of the source code have been executed, thereby making the code more reliable and minimizing potential risks. Coverage is calculated based on different metrics, depending on the requirements for evaluating code quality[7]. In practice, C1, C2, and C3 are three commonly used metrics for assessing code quality in software testing.

- **Statement Coverage (C1):** Ensures that every executable statement in the test unit is executed at least once after the test cases execution.
- **Branch Coverage (C2):** Requires that all possible outputs of each decision point in the control flow graph are executed at least once after the test cases execution.
- **Modified Condition/Decision Coverage (MC/DC or C3):** Demands that all posibile outputs of each individual condition at each decision point in the control flow graph are executed independently at least once after the test cases execution.

2.4 Symbolic Execution

Symbolic execution is the method that aims to explore all possible execution paths in testing units. Because these paths depend on inputs with unknown values, this technique represents each input as a symbolic variable instead of a specific value. The operations are then redefined using these symbolic representations. The method initializes symbolic inputs and executes the program symbolically using symbolic expressions. During execution, the method collects the execution paths which contain sets of constraints. These constraints are used by a constraint solver to determine which paths are feasible. The process continues until all feasible paths have been explored or resource limits are reached[5, 10].

However, symbolic execution still faces several challenges. The most challenge of symbolic execution is the path explosion problem. The number of paths can grow exponentially due to multiple conditions, loops, or nested branches.

This results in increasing computational complexity. Moreover, constraint solving can be resource-intensive and simulating external components such as libraries or system calls is often incomplete or inaccurate. These limitations make it difficult to apply symbolic execution efficiently to large or complex software systems.

2.5 Concolic Testing

Concolic testing is a method that combines symbolic and dynamic testing[6]. The goal of this method is to overcome the limitations of each of the aforementioned approaches. Static testing struggles with complex constraints and execution path explosion. Meanwhile, dynamic testing often achieves limited coverage due to dependence on the scope of test data. First, concolic testing initializes the initial test data using a random generation method. In the second phase, the method executes this test data and analyzes the execution paths within the control flow graph. After the analysis, the executed statements and branches are marked, while the unvisited ones are identified. Next, concolic testing selects an optimal execution path to cover the unvisited branches and statements, and then proceeds to the third phase - generating new test data. Once the new test data is created, the method continues to repeat the execution and path analysis process until maximum coverage is achieved or the available resources are exhausted.

A major advantage of concolic testing is its ability to automatically generate test data with high coverage. Additionally, this method can detect hidden bugs that are difficult to find using previous methods, such as logic errors or boundary issues. However, concolic testing still faces certain challenges. First, concolic testing encounters problems in handling complex constraints arising from large data structures (e.g., C++ Standard Template Library) or user-defined complex data structures. Second, concolic testing may face path explosion issues with complex loop and branch structures.

2.6 Large Language Model

Large Language Models (LLMs) represent a significant breakthrough in artificial intelligence, particularly in natural language processing. LLMs are often based on the Transformer architecture, which allows the model to capture semantic relationships between words in a broad context and generate coherent and meaningful responses. As a result, they can perform a variety of tasks without requiring task-specific training, such as machine translation, text summarization,

question answering, or reasoning support. However, LLMs also pose major challenges including hallucinations, a lack of interpretability and high costs.

Large Language Models (LLMs) demonstrate effectiveness in various fields, including software testing. Thanks to their ability to understand source code context, LLMs can support multiple stages of the testing process: generating test data, detecting bugs, and debugging. LLMs are capable of automatically generating high-coverage test data based on source code context analysis. In addition, LLMs can help in error analysis and detection through log analysis, error messages, or test reports. Finally, LLMs can be combined with traditional methods to generate test data for complex execution paths that are difficult to access.

However, large language models (LLMs) still face many challenges in software testing. LLMs are prone to hallucinations, which can lead to the generation of inaccurate test data or bug detection. Additionally, LLMs rely on the training data. If the training data lacks information about the latest updates of libraries, the model may make incorrect predictions. Moreover, source code often contains hidden dependencies or complex behaviors, which can reduce the accuracy of LLMs' predictions. Finally, due to their reliance on probability, LLMs' predictions may be inconsistent across different uses.

In practice, businesses often choose to fine-tune LLMs rather than use the APIs of large LLMs such as GPT or DeepSeek. There are several reasons for this choice. First, fine-tuning LLMs helps improve their performance on specific tasks that a business needs. This is something that large general-purpose LLMs like GPT or DeepSeek may not perform well on due to insufficient training data in that particular domain. Additionally, fine-tuning LLMs helps reduce costs for businesses because, in the long run, fine-tuning and deploying a smaller LLM for a specific task is cheaper than calling the API from GPT or DeepSeek, with no significant difference in performance. Finally, there is the issue of data security. Using LLMs such as GPT or Claude requires sending data to a third-party server, which poses risks to business security and privacy. Fine-tuning completely solves this problem since it is deployed within the company's internal environment. Therefore, the thesis also chooses to fine-tune rather than use the GPT or DeepSeek API.

Chapter 3

Methodology

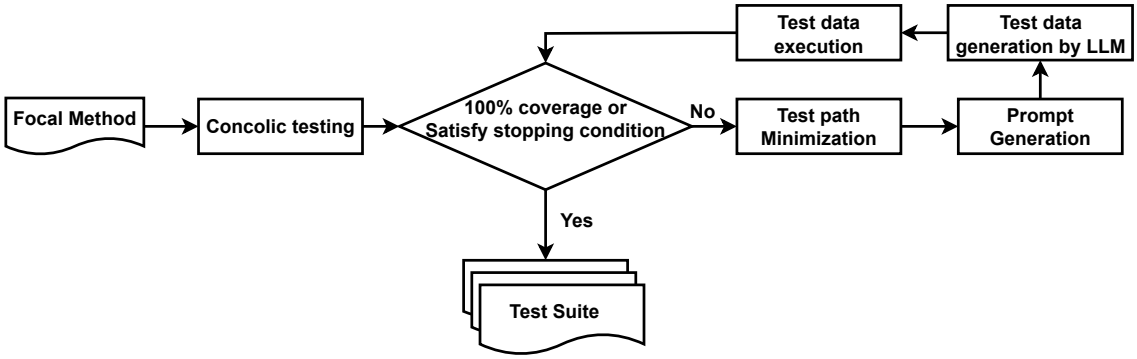

Figure 3.1: The workflow of combining fine-tuned LLM and concolic testing

Figure 3.1 illustrates the workflow of the method combining fine-tuned LLMs and concolic testing. The method begins with the concolic testing phase, during which the system performs concolic testing until maximum coverage is achieved or the resource limit is reached. After completing concolic testing, if the coverage is still insufficient, the method moves to the second phase – path minimization. In this phase, the path minimization algorithm (presented above) is used to identify the optimal set of execution paths while still maximizing coverage. Once the set of execution paths is determined, for each path in this set, AKAAUTO extracts the necessary information to create a prompt according to the presented template, then sends it to the fine-tuned LLM. The fine-tuned LLM then reasons and generates test data suitable for each execution path and sends the results back to AKAAUTO for execution. The process stops when the LLM has generated test data for all execution paths in the set, or when the resource limit is reached.

3.1 Concolic Testing

The concolic testing phase aims to generate test data that achieves maximum coverage for the function under test, while also assessing whether it is necessary to switch to the phase using an LLM. This phase begins by initializing random test data. Then, the AKAAUTO tool executes that test data. When execution is successful, the tool updates coverage information and the nodes that have been executed to support the path analysis process. From the analysis results, the tool identifies the nodes that have not yet been executed and finds execution paths that can cover these nodes. Based on those execution paths, constraints are collected and sent to a constraint solver (such as Z3) to generate new test data. This process repeats continuously - executing and generating new test data - until maximum coverage is achieved or resources are exhausted. After completing concolic testing, the method evaluates the achieved coverage to decide whether further phases need to be performed.

3.2 Test Path Minimization

The number of execution paths in the method can increase exponentially depending on the number of loops and branching conditions in the test unit. This makes generating test data for all execution paths difficult. To address this challenge, the method aims to find a minimized set of execution paths to maximize branch coverage in the source code. The method for finding minimized paths consists of two phases: collecting execution paths and branches, and minimizing the number of paths. First, in the execution paths and branches collection phase, the method uses the AKAAUTO tool to perform a pre-order traversal. During traversal, the tool collects execution paths that contain uncovered branches and uncovered branches. Collection can take a long time if there are complex loop structures in the source code. Therefore, the method limits the number of loops to make collection easier. Once all execution paths and branches are collected, the path reduction algorithm is executed.

Algorithm 1 describes the algorithm for finding the set of minimized execution paths of the method. The inputs of the algorithm are a set of execution paths containing uncovered branches P and a set of uncovered branches B. The algorithm constructs the set of minimized execution paths M using a greedy strategy. It begins by initializing an empty set to store minimized paths (line 1). In each iteration, the algorithm selects an execution path from the set P that covers the most unvisited branches of the set B (line 3). That execution path is added to the set of minimized execution paths M (line 4). After that, the path and its unvisited branches are removed from set P and B (lines 5 - 6). This

Algorithm 1 Path Minimization[4]

Require: set of execution paths P and set of unvisited branches B
Ensure: set of minimized paths M

1: $M \leftarrow \varnothing$
2: **while** $B \neq \varnothing$ **do**
3: $\quad p_{max} \leftarrow \arg\max_{p \in P} \ count \left| \mathsf{getUnvisitedBranches}(p) \cap B \right|$
4: $\quad M \leftarrow M \cup \{p_{max}\}$
5: $\quad P \leftarrow P \setminus \{p_{max}\}$
6: $\quad B \leftarrow B \setminus \mathsf{getUnvisitedBranches}(p_{max})$
7: **end while**

process continues until there are no more unvisited branches in the set B. By applying the greedy strategy, the algorithm significantly reduces the number of execution paths that need to be generated for test data to achieve maximum branch coverage.

3.3 LLM Fine-tuning for Unit Test Data Generation

3.3.1 Dataset

In the community, a number of datasets are widely adopted, such as Cpp Unit Test Benchmark Data. Cpp Unit Test Benchmark Data is a dataset including source code - unit test pairs. It is designed to perform on Google Test - a testing framework developed by Google for C++ but it is incompatible with AKAAUTO - the tool that the method uses. This dataset is excluded due to the fact that multiple samples do not create any deployment environment, specifically 86,25%. During the initial phase, the approach of collecting from open source codes containing C++ solutions of LeetCode algorithm problems on Github. These source codes are selected according to top-rated criteria on this platform. These strategies apply various types of data: primitive types, data from C++ Standard Template Library, ... and also complex control structures such as loop or branch. It reveals that 10 code sources are adequate for this study. The table 3.1 below shows 10 typical source codes the method gathers:

Stt	Project	Star	Size of C++	LoC
1	AhJo53589_leetcode-cn	252	1284	53637
2	Amanhacker_Aman-Barnwal-Leetcode-Solutions	442	595	21895
3	Jack-Cherish_LeetCode	2700	25	879
4	LeadCoding_3-weeks-Google-Prep	1600	36	1678
5	wisdompeak_LeetCode	6100	1769	81297
6	acm-clan_algorithm-stone	2200	398	13614
7	black-shadows_LeetCode-Topicwise-Solutions	274	2065	92518
8	chr1sc2y_LeetCode_Archiver	207	497	15687
9	codedecks-in_LeetCode-Solutions	860	75	3178
10	kamyu104_LeetCode-Solutions	5100	3067	144625

Table 3.1: Project statistics

After being collected, there are two key paths for them to generate test cases for data collection. The first one is to apply concolic testing on AKAAUTO to create high-coverage test data for those source codes. Inputting public test data of algorithm problems on LeetCode into AKAAUTO to execute and convert these test data into usable formats. The process of constructing test cases is carried out by 3 individuals during a three-month period. Essential details extracted from successful test cases are kept as JSON format as follows:

```
[
    {
        "c": [],
        "fc": "",
        "f": [],
        "fm": "",
        "correspond": [],
        "datatest": [
            {
                "id": 0,
                "dt": {},
                "td" : "",
                "testpath" : "",
                "executed_fm_masked" : "",
                "executed_m_masked": [],
                "m": []
            }
        ]
    }
]
```

Table 3.2: Description of JSON fields

Field	Description
c	Constructors of the focal class
fc	Focal class
f	Fields of the class
fm	Focal method
correspond	List of structures
dt	Data tree
td	Test driver
testpath	Execution path of the test data inputs
executed_fm_masked	Focal method (executed part only)
executed_m	List of executed methods
m	List of methods

Table 3.2 provides information about fields in JSON, representing test case details. First, field c includes data about constructors of focal class of the functions under test. Field fc contains focal class of functions being tested. Field fm demonstrates functions for unit testing. Field correspond stores data structure inputs utilized in these functions. Field dt illustrates data tree - a typical way of data representation of AKAAUTO. Field td shows test drivers used to navigate test cases. Field executed_fm_masked includes the most crucial information - testable function execution parts. Finally, field executed_m depicts methods called by focal methods in the test cases.

```
{
    Source: Generate a test data inputs to maximize
    statement coverage for focal method {fm name}:
    /*FC*/class {fc name} {
        /*FM*/fm_masked;
        /*F*/list of fields;
        /*C*/list of constructors;
        /*M*/list of called methods;
    }
    Target: Key-Value
}
```

Among them, fm name and fm_masked are extracted from field executed_fm_masked, fc name is extracted from field fc while list of fields is isolated from field f, list of constructors is isolated from field c and list of called methods is isolated from field executed_m. In the target part, key-value is the simplified

representation of the data tree, which helps mitigate token count while training and running the model. Key here shows necessary parameter objects to carry out test cases and value is values of these objects. The strategy of using key-value as output instead of test driver decreases token quantities notably in comparison. Figure 3.2 depicts token length distribution of 2 output key-value and test driver on the selected model.

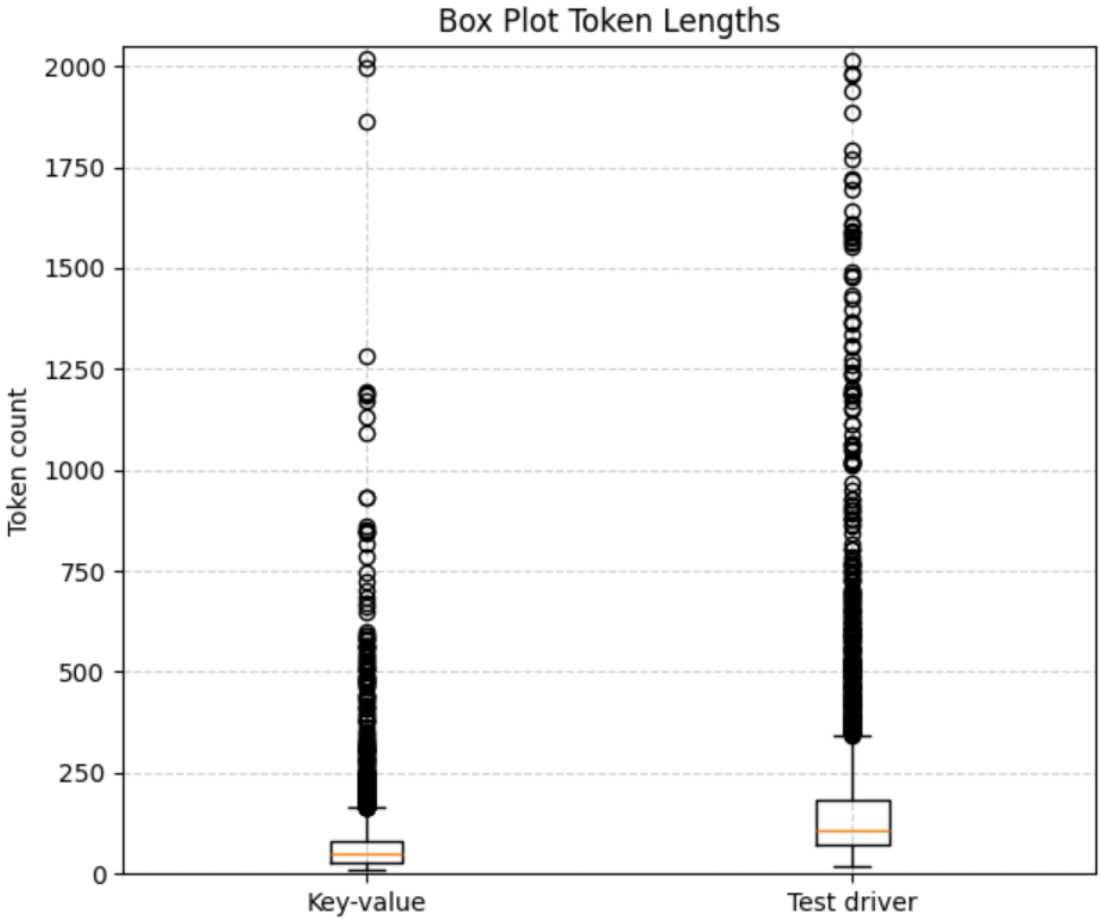

Figure 3.2: Example of one sample in clean dataset

From this figure, it can be concluded that the method should use output key-value for the problem. Deploying a larger model helps test drivers better due to the larger maximum token length, however, in the context of a smaller dataset, using a larger model not only does not provide understanding but also increases the cost in training and integration.

The next one is to preprocess the dataset. The method filters out samples with repeated partial methods in the dataset with the aim of preventing noisy dataset - which is the noise that data samples are duplicated, contributing to data distribution skew. Additionally, this method also removes samples whose token length of source, target exceeds the maximum token length of the model. The

dataset's size decreases from 4685 to 2962 data samples as a result.

After this process, a clean dataset including partial method and context information - test data represented in key-value format. This dataset whose size is 2962 data samples is used for fine-tuning model. Each sample in the dataset is secure of consistency and validity of syntax, semantics and structures. They all have the above format to maintain the consistency of the dataset and are normalized to a common format, helping fine-tune the model more efficiently. Test data in the dataset is executed successfully by AKAAUTO, resulting in syntactic, semantic, and structural validity assurance. Moreover, all samples are filtered out to make sure that each sample in the dataset is unique. The table below depicts types of data and their size. The figure 3.3 describes an example of a sample in the clean dataset.


```
{
    "source": "Generate a test data input to maximize branch coverage for focal method int waysToPartition.
               /*FC*/class Solution {
                   /*FM*/ int waysToPartition(vector<int>& nums, int k) { int n = nums.size(); long sum = accumulate(nums.begin(), nums.end(), 0L);
vector<long>rets(n); vector<long>pre(n); pre[0] = nums[0]; for (int i=1; i<n; i++) pre[i] = pre[i-1]+nums[i]; unordered_map<int,int>count; for (int i=0;
i<n; i++) { int new_sum = sum + k-nums[i]; if (new_sum % 2 == 0) rets[i] += count[new_sum/2]; count[pre[i]]++; } vector<long>suf(n); suf[n-1] = nums[n-1];
for (int i=n-2; i>=0; i--) suf[i] = suf[i+1]+nums[i]; count.clear(); for (int i=n-1; i>=0; i--) { int new_sum = sum + k-nums[i]; if (new_sum % 2 == 0)
rets[i] += count[new_sum/2]; count[suf[i]]++; } long ret = 0; for (int i=0; i<n-1; i++) { if (pre[i]==sum-pre[i]) } for (int i=0; i<n; i++) ret = max(ret,
rets[i]); return ret; };
                   /*F*/
                   /*C*/
                   /*STUB*/
                   /*M*/
               }",
     "target": "{
                   "AKA_INSTANCE___Solution": ["Solution", "Solution()"],
                   "nums": [2],
                   "nums_element0": ["-1076724343"],
                   "nums_element1": ["-1816985020"],
                   "k": ["-880383522"]
               }"
}
```


Figure 3.3: Example of one sample in clean dataset

Although the dataset is simulated, it covers most characteristic data types in C++. Therefore, this method can assess the ability to modify small models to utilize and integrate in testing tools such as AKAAUTO for real projects. While fine-tuning models, dataset are usually divided into 3 sets: training set, validation set and test set. However, due to difficulty of having a high-quality validation set,this method uses K-fold cross-validation approach with K = 5. It assists in diminishing risk in using a validation set that does not respresent the distribution of the dataset. Regarding the test set, it filters out uncommon test functions and takes all samples of that test functions' name in turn until it reaches 250 samples. It helps the test set not face the data leakage problems and does not change the distribution of the dataset substantially.

3.3.2 Fine-tuning Method

To fine-tune LLMs for test data generation tasks, the thesis chooses the Low-Rank Adaptation (LoRA) fine-tuning method. LoRA is a technique that significantly

reduces training costs and resources compared to the full fine-tuning method. This method allows fine-tuning of large-scale models without requiring too many resources. The main idea of LoRA is to insert low-rank matrices into the attention layers. During fine-tuning, the model updates new adaptive parameters from these matrices while keeping the original model weights unchanged. The unit test data generation task is a sequence-to-sequence task. Therefore, the method optimizes the following cross-entropy loss function on the target:

$$\mathcal{L} = -\sum_{t=1}^{T} \mathbf{1}_t \cdot \log p(x_t \mid x_{1:t-1}),$$

3.3.3 Evaluation Metrics

The evaluation process focuses on runtime-based metrics that evaluate the correctness and behavior of generated tests during execution. Four metrics are employed to assess performance in the test generation task:

- **Covered Tests:** Test cases that are successfully executed and cover the execution path.
- **Uncovered Tests:** Test cases that execute without errors but do not cover the execution path.
- **Runtime Errors:** Test cases that fail to execute due to runtime exceptions or environment issues.
- **Format Errors:** Tests cases that cannot be executed because they have incorrect formats.

These metrics collectively reflect the effectiveness and reliability of the test data generated by LLMs in practical execution environments.

Chapter 4

Experimental Result

To prove the efficiency of the proposed method, experimental result answer the following research questions:

- **RQ1 - Coverage:** Does the proposed method achieve higher statement and branch coverage compared to the baseline method?
- **RQ2 - Quality of generated test cases:** Does the generated test data cover the right execution path?
- **RQ3 - Performance:** How does the performance of the proposed method compare to the baseline?
- **RQ4 - Token usage:** Does the partial focal method help reduce the number of tokens of the prompts?

All experiments were conducted on a Lenovo IdeaPad 5 Pro laptop which is equipped with RAM: 16GB, CPU: Intel Core i5 12500H with Intel iRIS Xe Graphics. The goal of RQ1 is to evaluate the statement and branch coverage of the proposed method compared to original concolic testing and original LLM. RQ2 aims to evaluate the quality of the generated test data. RQ3 is used to evaluate the performance of the proposed method while RQ4 proves that the partial focal method is effective in reduce prompt's size.

4.1 Configuration

4.1.1 Dataset

To the best of our knowledge, there is currently no publicly available automated unit testing dataset for C++ focal methods. Therefore, in this research, we have created our own dataset based on the source code of LeetCode Solution projects

on GitHub as described in Sec. 3.3.1. To ensure objectivity, the Dataset includes 174 focal methods that were not used in the fine-tuning process. The chosen focal methods have a diverse number of statements and branches to evaluate the performance of the methods.

Table 4.1 shows the complexity of the data. The average number of statements is 13.19 (range 4-42), the average number of branches is 7.51 (range 2-28). Most methods are small to medium in size, a few are highly complex. Table 4.2 lists 327 input parameters (average 1.88 parameters/method). STL types account for 52.6% (`vector`, `string`, `map`), primitive types account for 34.25% (`int`, `double`, `bool`), and self-defined types account for 13.15% (`TreeNode`, `ListNode`). The high proportion of STL types is important because concolic testing often encounters difficulties with this type. This is a good opportunity to evaluate the proposed method. The variety of complexity and data types ensures reliable experimental results.

Table 4.1: Descriptive statistics of focal methods in the experimental dataset (N=174)

Metric	**Statements**	**Branches**
Count (N)	174	174
Min	4	2
Q1 (25th)	9.00	4.00
Median	11.50	4.28
Mean	13.19	7.51
Q3 (75th)	16.00	8.00
Max	42	28
Std	6.38	6.00
P90	22.70	12.00
P95	26.00	16.00

Table 4.2: Distribution of input parameter types in the experimental dataset

Data type	**Count**	**Ratio (%)**
Primitive types	112	34.25
STL types	172	52.60
User-defined types	43	13.15
Total parameters	**327**	**100.00**

From 174 focal methods; avg 1.88 params/method

4.1.2 Selected Large Language Model

The thesis selects the Salesforce CodeT5+ 770M model[1]. This model is pre-trained on CodeSearchNet dataset which consists of 9 programming languages (Python, Java, Ruby, JavaScript, Go, PHP, C, C++, C#). The model is not pre-trained on any dataset about solutions to LeetCode problems on Github. It uses traditional Transformer architecture (Encoder - Decoder) with 770 million parameters, leading to more effective comprehension of language context and helping learn about the relationship between input and output. Through experiences on a pre-trained model, the thesis notices that pre-trained model has no ability to create true and appropriate testing data for an approach-oriented format, so the thesis decided to fine-tune the model.

Se them vi du ve oneshot/few shot cua mo hinh pretrain o day.

In this study, the input is a partial method which represents a execution path of the focal method and the output is test data to cover that execution path. The method experiments with different llms such as DeepSeek Coder 1.3B or Qwen 2.5 Coder 1.5B. The results are not good at learning about the format output that this study aims to. Therefore, the thesis decides to fine-tune CodeT5+ 770M. A limit of this model is the context size of 512 which are much smaller than 1600 of DeepSeek Coder 1.3B or 32768 of Qwen 2.5 Coder 1.5B. However, with the current dataset, the majority of data samples have source and target token lengths smaller than or equal to 512, so this model is suitable with the study. Figure 4.1 represents the distribution of source and target token length on the chosen model.

[1]https://huggingface.co/Salesforce/codet5p-770m

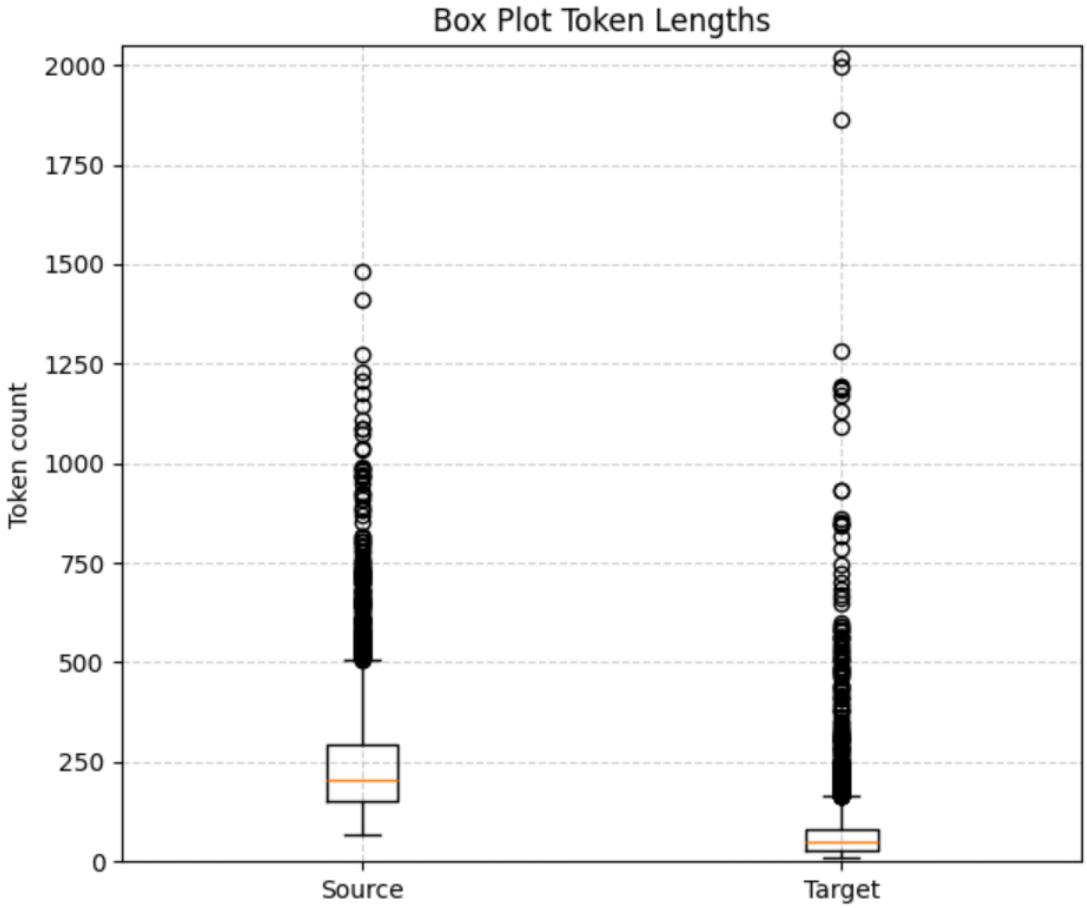

Figure 4.1: Token length distribution

4.1.3 Baseline

To evaluate the effectiveness of the proposed method, different methods are used to generate test data for 174 focal methods in the dataset. The first is the traditional concolic testing method, implemented in Java as published by Nguyen, et al [13], which will be called Concolic in this thesis. The second method is using the model after finetuning, called AI. The last method is the proposed method, combining Concolic and AI. Since the LLM models are unstable, they are run 5 times and evaluate the cumulative source code coverage over the runs.

4.1.4 The Proposed Method

In this experiment, we implement the Concolic Testing, Test Path Minimization, Prompt Generation, and Test data execution modules. When use the model for inference, we set up a 5-time loop for each partial focal method because the model is unstable. The test cases generated by the inference process will have their cumulative coverage calculated by the Test data execution module.

The instability of the model is due to the temperature configuration. According to prior research of using LLM for unit testing, temperature should be set at 0 to minimize instability. However, after trying with different temperatures such as 0, 0.3, etc. the thesis designs to choose the temperature of 0.5 to balance the stability and creativity of model. Table 4.3, 4.4 and 4.5 describe the results of temperatures 0, 0.3 and 0.5 in the first 60 samples of the test set. Low temperatures like 0 and 0.3 produce fairly similar outputs and tend to repeat values. At low temperatures such as 0 and 0.3, the model produces deterministic results that are easy to replicate with similar input data. The generated values may be due to the model trying to remember the values it has learned rather than reasoning about which values satisfy the constraints along the execution path.

Table 4.3: Summary of 60 samples test set results for temperature 0

Category	**Number of Samples**
Covered Tests	16
Uncovered Tests	23
Runtime Errors	12
Format Errors	9
Total	**60**

Table 4.4: Summary of 60 samples test set results for temperature 0.3

Category	**Number of Samples**
Covered Tests	21
Uncovered Tests	17
Runtime Errors	14
Format Errors	8
Total	**60**

Table 4.5: Summary of 60 samples test set results for temperature 0.5

Category	**Number of Samples**
Covered Tests	26
Uncovered Tests	16
Runtime Errors	13
Format Errors	5
Total	**60**

4.1.5 Metric

The thesis uses statement coverage and branch coverage to measure the quality of test case generated by three mentioned methods. They are two common metrics used in high-coverage automated testing data generation strategies.

Statement coverage is calculated using the formula below:

$$\text{Statement Coverage } (\%) = \frac{\text{Number of executed statements}}{\text{Total number of statements}} \times 100$$

The value of statement coverage ranges from 0% to 100%. The maximum value of 100% indicates that all the statements in the focal method are covered when executing the test data. Conversely, the minimum value of 0% means that no statements in the test unit are covered when the test cases are executed.

Branch coverage is calculated using the formula below:

$$\text{Branch Coverage } (\%) = \frac{\text{Number of executed branches}}{\text{Total number of branches}} \times 100$$

The value of branch coverage ranges from 0% to 100%. The maximum value of 100% means that all the branches in the focal method are covered when executing the test data. Conversely, the minimum value of 0% means that no branches in the test unit are covered when the test cases are executed.

4.2 Answer to Research Questions

4.2.1 RQ1 - Coverage

The experimental result shows that the fine-tuned LLM achieves promising results despite the limited size of the train set. Figure 4.2 and 4.3 describes the summary of the focal methods that the Concolic and the Concolic AI have different statement and branch coverage while Figure **??** and Figure **??** show the difference in statement and branch coverage between Concolic AI and AI, respectively. The proposed method achieves higher code coverage than the other two methods in most cases.

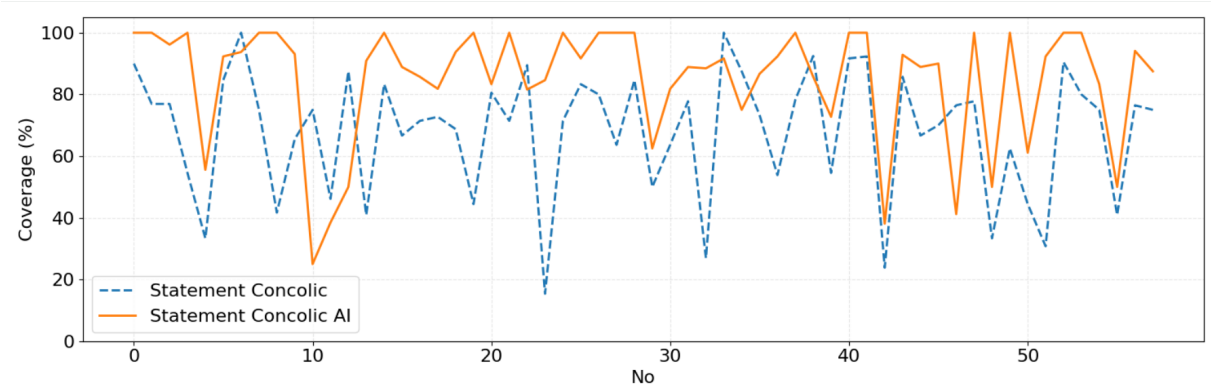

Figure 4.2: Statement coverage comparsion between Concolic and Concolic AI

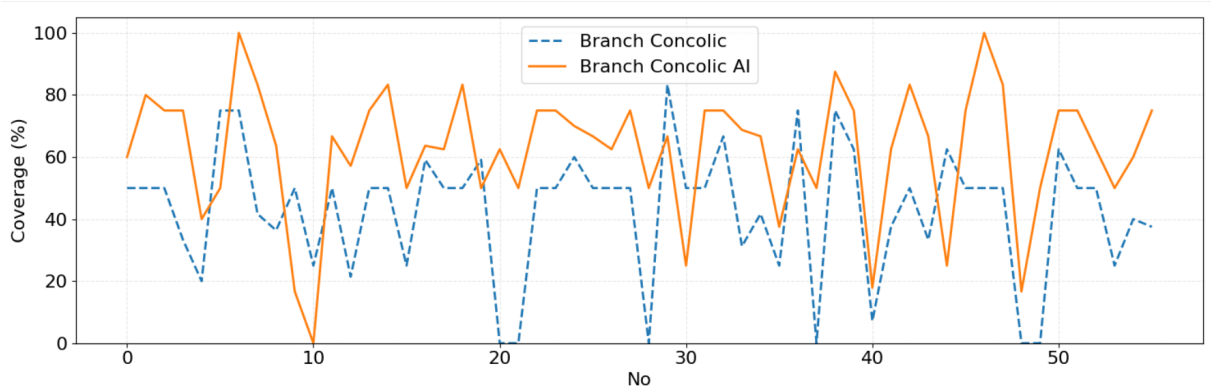

Figure 4.3: Branch coverage comparsion between Concolic and Concolic AI

The proposed method generates test data that achieve more coverage than the Concolic with common C++ STL input data types in the train set. The majority of the cases where the proposed method achieves higher coverage have input data types including string, vector. The data types such as string, vector need more data input to declare a variable than the primitive data types. Their constraints are also more complex than those of the primitive data types. In addition to that, these data types have many functions that constraint solvers do not understand how they work. These reasons lead to the generation of test data that do not satisfy these constraints and execution paths. This results in reduced coverage of the focal methods. Leveraging LLMs helps with handling complex and large constraints to overcome the drawbacks of Concolic testing. *The figure* **??** *and* **??** *describe an example of a case where the coverage of the proposed method is higher than Concolic. It can be seen that Concolic testing cannot handle complex constraints like ... in the example.* Meanwhile, thanks to its ability to understand context and C++ STL functions, the model can generate test data that satisfy those constraints. This helps significantly increase the overall coverage of the focal method.

However, the proposed method struggles to generate test data for user-defined data types. The majority of test cases where the proposed method achieves less coverage relates to user-defined data types such as TreeNode, ListNode, Node. There are some reasons for these cases. Firstly, the train set contains little of samples that has user-defined data type inputs. This leads to LLM generating test data for these types inaccurately. In addition to that, the generated test data for these types are less accurate because the prompts do not contain the information on class inputs. This is more evident when combining the model and random strategy instead of concolic. Figure 4.4 and 4.5 describes the summary of the focal methods that Concolic and AI have different statement and branch coverage. When not integrated with concolic, the coverage achieved by Concolic AI is significantly reduced in cases with class inputs. *The figure* **??** *and* **??** *describe an example of a case where the coverage of the random plus ai method is lower than the baseline.* This focal method has the input parameters that contains class inputs. The information about class inputs is not provided to the model, so the model can only infer based on the class inputs in the train set. Therefore, the generated results may be inaccurate if those input classes are completely different or partially different from those in the dataset.

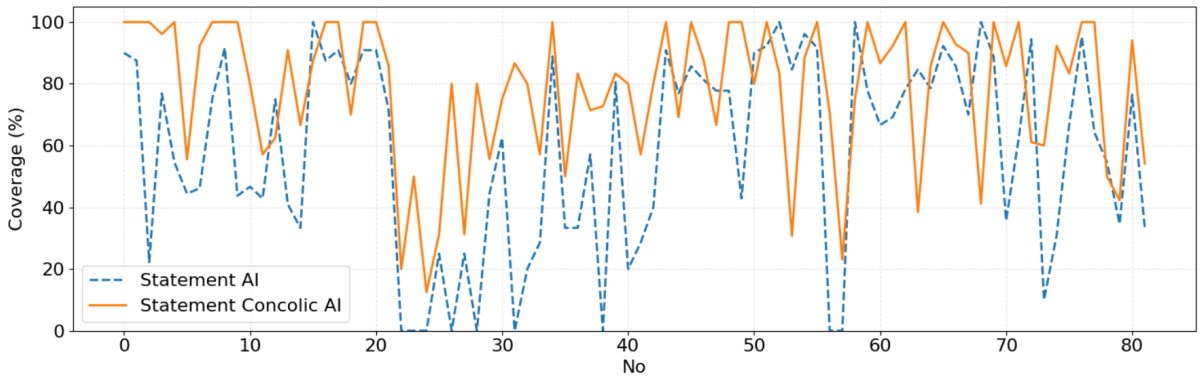

Figure 4.4: Statement coverage comparsion between Concolic AI and AI

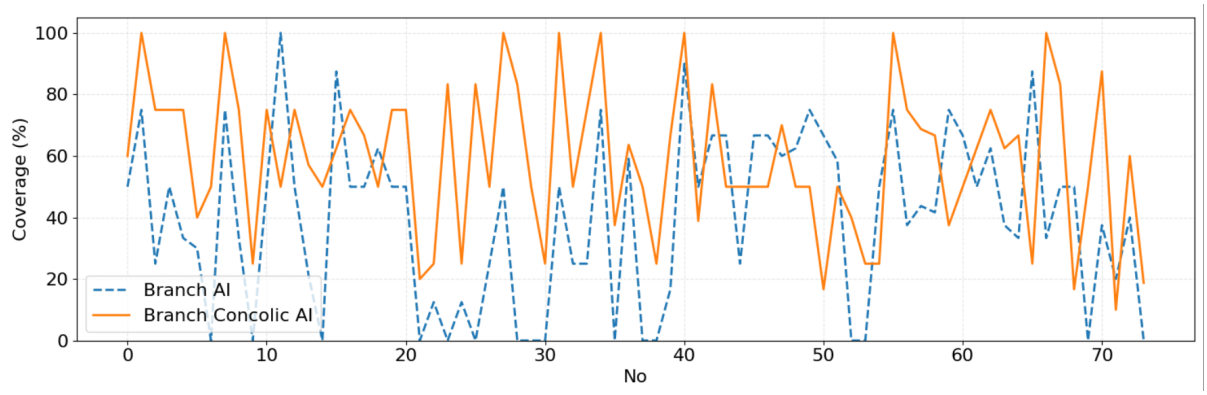

Figure 4.5: Branch coverage comparsion between Concolic AI and AI

Answer to RQ1: In conclusion, the proposed method works better than the concolic testing with the common data types in train set. However, the method struggles to generate test data for user-defined data types.

4.2.2 RQ2 - Quality of generated test cases

To evaluate the quality of generated test cases, the thesis evaluates on the test set of fine-tuning the model. The generated test cases are executed and summarized in Table 4.6. The result show that the method generates highly executable test cases.

Table 4.6: Summary of fine-tuning evaluation results

Category	Number of Samples
Covered Tests	118
Uncovered Tests	69
Runtime Errors	35
Format Errors	28
Total	**250**

The fine-tuned LLM are able to generate highly executable test cases. The ratio of executable test cases is 74.8%. There are 47.2% of test cases that covers the right execution paths. These results show the usability of the method despite the small size of model. Before fine-tuning, the model cannot generate test data with correct format. After fine-tuning, the ratio of correct format test data is 88.8%. This is a good development of the model in spite of the small train set of 2962 samples. However, there are several cases that the generated test data cannot be executed due to runtime error or format error. The majority of these cases is related to class input which the model is not provided with in the prompts. In addition to that, a few cases fail because the attributes of the focal classes and the global variables are not declared correctly.

Answer to RQ2: In conclusion, the quality of generated test cases is good despite the small LLM and train set sizes. The generated test cases cover the execution path quite well and can be better when the size of the model is larger.

4.2.3 RQ3 - Performance

Although the Concolic AI method achieves fairly positive results in terms of increasing coverage, it comes at the cost of a longer execution time. The AI-only method has the best speed because it doesn't take time for the initial concolic phase. This significantly reduces the execution time of the method compared to Concolic and Concolic AI. Table 4.7 provides statistics on the average execution time of the three methods.

Table 4.7: Summary of average time execution

Method	**Average time (s)**	**Average AI call time (s)**
Concolic	38,83	0
AI	23,75	9.23
Concolic AI	53,28	7.35

Compared to Concolic, **the average execution time of the Concolic AI increases by 37.21%**. The increase in time is mainly due to the processing time of algorithm Test Path Minimization and the time spent calling the LLM to generate test data for the execution paths. This is a trade-off between execution time and coverage. Meanwhile, the average execution time of AI decreases by 38.84%. Because the Test Path Minimization algorithm reduces the number of execution paths and does not need to find new execution paths after each test case execution, the execution time of AI is the lowest and decreases quite significantly compared to Concolic. Concolic AI trades off a loss in runtime for better code coverage than AI and Concolic, which is worth it.

Answer to RQ3: In summary, the Concolic AI method, although it has a longer execution time, provides higher coverage compared to Concolic and AI. In addition, AI-only method, while having a shorter execution time, has unstable coverage effectiveness.

4.2.4 RQ4 - Token usage

To reduce the number of tokens for the prompts, the method uses the partial method instead of the focal method to represent the execution path. The partial method is a focal method that removes the statements that is not executed. This helps remove redundant information about other execution paths and decrease the token length of the prompts. Figure 4.6 describes the distribute of the token lengths when using partial methods and focal methods.

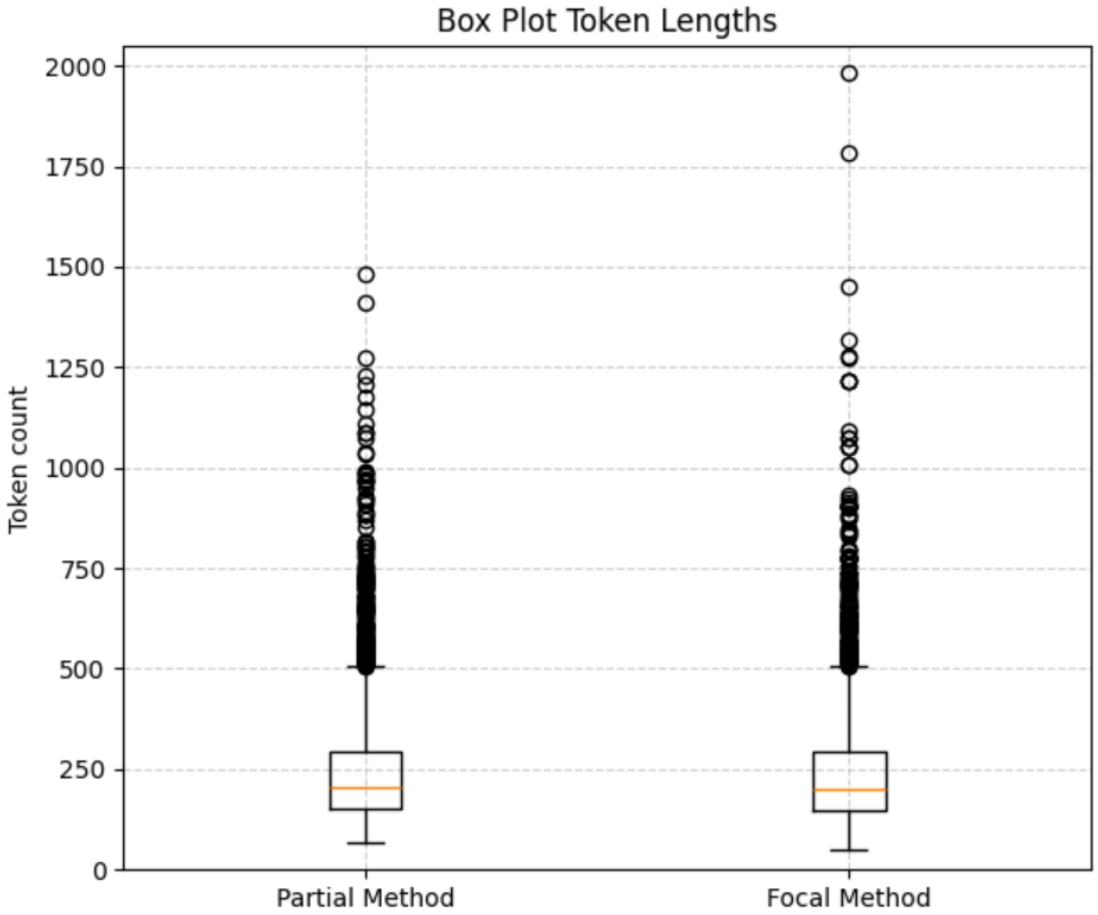

Figure 4.6: Token length distribution

The token lengths of the prompts reduce ...% when using the partial methods and fit the context length of the LLM. When the prompt length exceeds the context length of the model, the prompt gets cut off, resulting in missing information when input into the model. Additionally, the cut off prompt can cause the model to misunderstand the context, leading to inaccurate output. The token lengths decrease but do not affect the quality of the generated test cases and the coverage of the test units, presented in RQ1 and RQ2.

Answer to RQ4: In summary, using partial methods not only helps visually represent the execution path but also significantly reduces the size of the prompt, avoiding unnecessary information about other execution paths and exceeding the context length of the LLM.

Chapter 5

Discussion

5.1 Threats to Validity

5.1.1 External Threat

LLM selection is an external threat that can be mentioned. The study cannot test all LLMs to find the most suitable model. Therefore, this study focuses on LLMs that are well known and highly regarded for code such as DeepSeek Coder, Qwen Coder and CodeT5. This makes finding a LLM that is suitable with the study easier.

5.1.2 Internal Threat

One of the internal threats is data leak. The partial methods of a focal method can be in both train set and test set. The thesis overcomes this threat by countering the names of focal methods in the train set. The names are taken from smallest to largest and put those focal methods into the test set until exceed the size of the test set. This helps all the partial method of a focal method only be in train set or test set. An other internal threat is the value of hyperparameters. The test data generated by LLM can be unstable due to sampling strategy. To reduce the instability of the LLM, the method reduces temperature to 0.5 and top k to 0.9. These hyperparameters are to balance the stability and diversity of the generated test data.

Chapter 6

Related Works

6.1 Concolic Testing

Concolic testing [6, 2, 16] is original proposed in DART by Godefroid and colleagues. This is an automated white-box testing method that aims to generate test data achieving high code coverage. The main idea of Concolic testing is to combine symbolic execution and concrete execution. Inheriting this approach, Sanghoon Rho and colleagues introduced Coyote C++: An Industrial-Strength Fully Automated Unit Testing Tool. This tool achieves promising results on popular open-source C++ projects and Hyundai KEFICO's projects. However, Concolic Testing sill faces several drawbacks. These drawbacks are mostly from the limits of constraint solvers. These solvers are difficult to handle highly complex constraints. This makes the accuracy of generated test data and the coverage of test units decrease.

6.2 LLM-based Unit Test Data Generation

Many studies leverage the rapid development of LLM to overcome the drawbacks of Concolic Testing [22, 1, 3]. One of typical studies is Code-Aware Prompting: A Study of Coverage-Guided Test Generation in Regression Setting using LLM [17]. Gabriel Ryan and his colleagues utilize symbolic execution to analyze the execution paths of the focal method and LLM to generate test data to cover these paths. This method results in the high ratio of the test data that covers the right execution path. This helps the coverage of the focal methods increase significantly. However, using third-party LLMs are more expensive and less secure. In addition, these LLms can be not pre-trained with latest data of libraries or frameworks. This leads to the lower accuracy of the generated test data.

Chapter 7

Conclusion

Software testing is an important phase in the software quality assurance process. With the rapid development of software systems, automated testing is necessary to increase efficiency and reduce costs in the testing process. Automated testing has two approaches: generating test data to detect errors and generating test data to maximize code coverage. In the community, there are many studies on the generation of test data with high code coverage. However, these methods have several drawbacks, such as difficulty in handling complex constraints of Concolic Testing and large prompt sizes, a lack of security of LLM-based methods. Therefore, the thesis proposes a method that fine-tunes LLM and integrates it with Concolic Testing to solve these problems. The main idea of the method is to utilize the fine-tuned LLM to generate test data to cover the execution paths that Concolic Testing does not handle. This approach helps to increase code coverage of test units.

The thesis installs and experiments in AKAAUTO tool to compare the performance of the proposed method and Concolic Testing. The results show that the proposed method achieves higher coverage in the majority of the different coverage cases. The average coverage of the proposed method in these cases is 90% statement coverage and 80% branch coverage. Meanwhile, Concolic Testing achieves the average coverage of 80% statement coverage and 70% branch coverage. The results demonstrate the practical usage of the proposed method.

However, the method has several drawbacks. The following study will overcome these drawbacks. Firstly, the study increases the size and diversity of the training set. This helps LLM understand and generate better test data for more different data types. Secondly, the study finds a suitable LLM with a larger size and context size. The prompts can contain class input information without the limit of context size. This helps LLM understand class inputs and generate better test data for these data types.

References

[1] Juan Altmayer Pizzorno and Emery D Berger. Coverup: Effective high coverage test generation for python. *Proceedings of the ACM on Software Engineering*, 2(FSE):2897–2919, 2025.

[2] Cristian Cadar, Patrice Godefroid, Sarfraz Khurshid, Corina S Păsăreanu, Koushik Sen, Nikolai Tillmann, and Willem Visser. Symbolic execution for software testing in practice: preliminary assessment. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 1066–1071, 2011.

[3] Arda Celik and Qusay H Mahmoud. A review of large language models for automated test case generation. *Machine Learning and Knowledge Extraction*, 7(3):97, 2025.

[4] Bei Chu, Yang Feng, Kui Liu, Hange Shi, Zifan Nan, Zhaoqiang Guo, and Baowen Xu. Palm: Synergizing program analysis and llms to enhance rust unit test coverage, 2025. Accepted to ASE 2025 (Research Paper Track).

[5] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3):215–222, 1976.

[6] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[7] Pham Ngoc Hung, Truong Anh Hoang, and Dang Van Hung. Giao trinh kiem thu phan mem. 2014.

[8] Tran Nguyen Huong, Hoang-Viet Tran, Pham Ngoc Hung, et al. A hybrid method for test data generation for unit testing of c/c++ projects. *VNU Journal of Science: Computer Science and Communication Engineering*, 39(2), 2022.

[9] Timo Hynninen, Jussi Kasurinen, Antti Knutas, and Ossi Taipale. Software testing: Survey of the industry practices. In *2018 41st International convention on information and communication technology, electronics and microelectronics (MIPRO)*, pages 1449–1454. IEEE, 2018.

[10] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[11] Wei Ma, Shangqing Liu, Zhihao Lin, Wenhan Wang, Qiang Hu, Ye Liu, Cen Zhang, Liming Nie, Li Li, and Yang Liu. Lms: Understanding code syntax and semantics for code analysis. *arXiv preprint arXiv:2305.12138*, 2023.

[12] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *29th International Conference on Software Engineering (ICSE'07)*, pages 416–426. IEEE, 2007.

[13] Duc-Anh Nguyen and Pham Ngoc Hung. A test data generation method for c/c++ projects. In *Proceedings of the 8th International Symposium on Information and Communication Technology*, pages 431–438, 2017.

[14] Lam Nguyen Tung, Nguyen Vu Binh Duong, Khoi Nguyen Le, and Pham Ngoc Hung. Automated test data generation and stubbing method for c/c++ embedded projects. *Automated Software Engineering*, 31(2):52, 2024.

[15] Tomas Potuzak and Richard Lipka. Current trends in automated test case generation. In *2023 18th Conference on Computer Science and Intelligence Systems (FedCSIS)*, pages 627–636. IEEE, 2023.

[16] Sanghoon Rho, Philipp Martens, Seungcheol Shin, Yeoneo Kim, Hoon Heo, and SeungHyun Oh. Coyote c++: An industrial-strength fully automated unit testing tool. *arXiv preprint arXiv:2310.14500*, 2023.

[17] Gabriel Ryan, Siddhartha Jain, Mingyue Shang, Shiqi Wang, Xiaofei Ma, Murali Krishna Ramanathan, and Baishakhi Ray. Code-aware prompting: A study of coverage-guided test generation in regression setting using llm. *Proceedings of the ACM on Software Engineering*, 1(FSE):951–971, 2024.

[18] Koushik Sen. Concolic testing. In *Proceedings of the 22nd IEEE/ACM international conference on Automated software engineering*, pages 571–572, 2007.

[19] Divyani Shivkumar Taley and Bageshree Pathak. Comprehensive study of software testing techniques and strategies: a review. *Int. J. Eng. Res*, 9(08):817–822, 2020.

[20] Haoxin Tu, Seongmin Lee, Yuxian Li, Peng Chen, Lingxiao Jiang, and Marcel Böhme. Large language model-driven concolic execution for highly structured test input generation. *arXiv preprint arXiv:2504.17542*, 2025.

[21] Junjie Wang, Yuchao Huang, Chunyang Chen, Zhe Liu, Song Wang, and Qing Wang. Software testing with large language models: Survey, landscape, and vision. *IEEE Transactions on Software Engineering*, 50(4):911–936, 2024.

[22] Yaoxuan Wu, Xiaojie Zhou, Ahmad Humayun, Muhammad Ali Gulzar, and Miryung Kim. Generating and understanding tests via path-aware symbolic execution with llms. *arXiv preprint arXiv:2506.19287*, 2025.

[23] Dovydas Marius Zapkus and Asta Slotkienė. Unit test generation using large language models: A systematic literature review. *Lietuvos magistrantu informatikos ir IT tyrimai: konferencijos darbai, 2024 m. gegužės 10 d.*, pages 136–144, 2024.