A Novel Approach for Path-Directed Source Test Case Generation and Prioritization in Metamorphic Testing Using Python

By

Atif Imran

(Registration No.: 00000330744)

Supervisor: Dr. Wasi Haider Butt

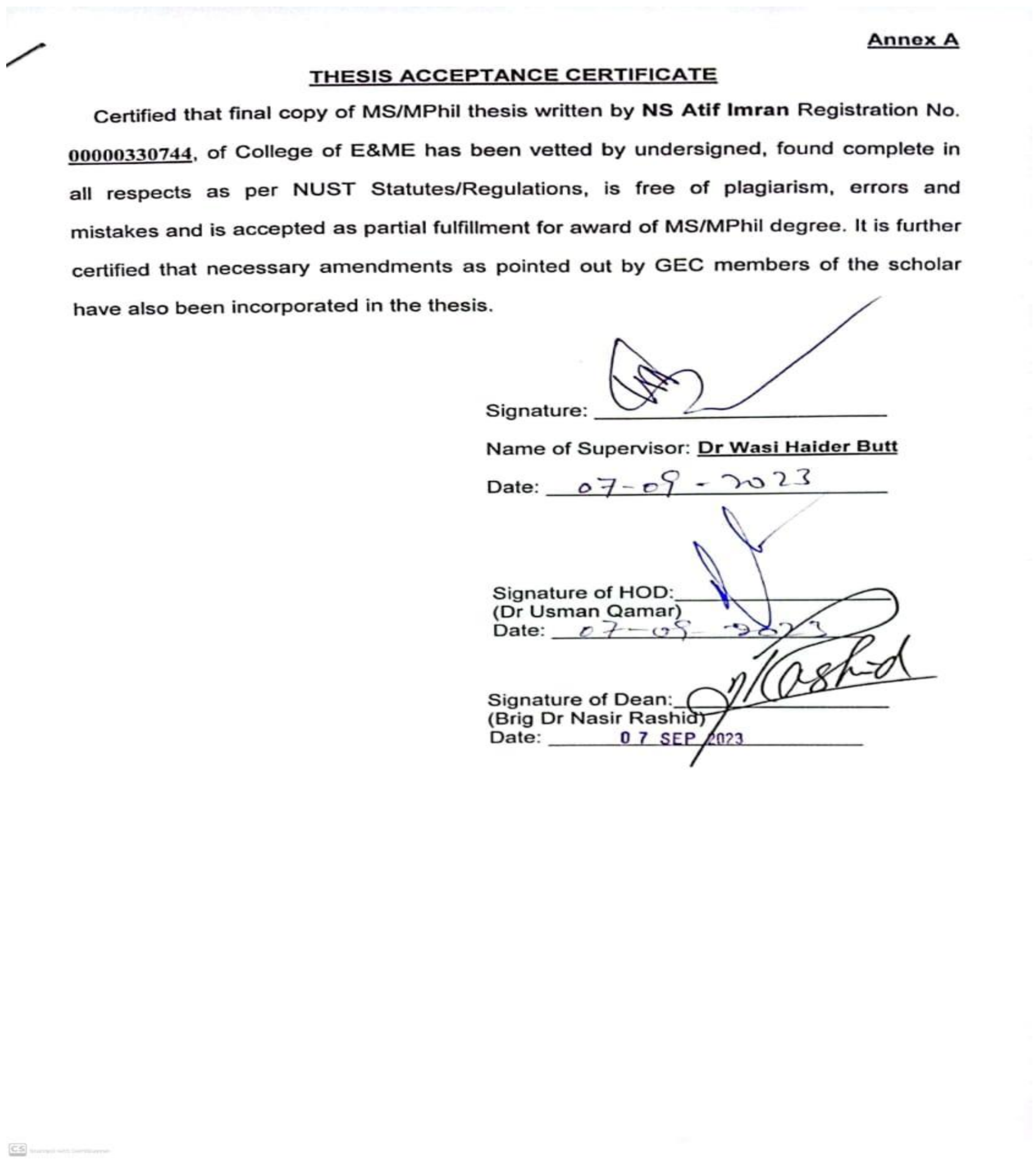
DEPARTMENT OF COMPUTER & SOFTWARE ENGINEERING

COLLEGE OF ELECTRICAL & MECHANICAL ENGINEERING

NATIONAL UNIVERSITY OF SCIENCES AND TECHNOLOGY

ISLAMABAD

September 2023



Annex A

THESIS ACCEPTANCE CERTIFICATE

Certified that final copy of MS/MPhil thesis written by **NS Atif Imran** Registration No. **00000330744**, of College of E&ME has been vetted by undersigned, found complete in all respects as per NUST Statutes/Regulations, is free of plagiarism, errors and mistakes and is accepted as partial fulfillment for award of MS/MPhil degree. It is further certified that necessary amendments as pointed out by GEC members of the scholar have also been incorporated in the thesis.

Signature: [illegible]

Name of Supervisor: **Dr Wasi Haider Butt**

Date: 07-09-2023

Signature of HOD: [illegible]
(Dr Usman Qamar)
Date: 07-09-2023

Signature of Dean: [illegible]
(Brig Dr Nasir Rashid)
Date: 07 SEP 2023

Dedicated to my exceptional parents, Supervisor **: Dr. Wasi Haider Butt** *and adored brother whose tremendous support and cooperation led me to this accomplishment.*

Acknowledgements

All praise to Allah (the omnipotent and the omnipresent) who has bestowed me with ardor, courage, and patience with which I have completed another phase of my academic journey.

I would also dedicate this thesis to my parents, teachers, siblings, and students, who were continuous sources of motivation in my tough times. They always encouraged me to continue higher studies and fully supported me to full fill my dream degree. My parents played a pivotal role in my MS degree by providing moral and financial support.

Most importantly, I want to pay special gratitude to my supervisor “Dr. Wasi Haider Butt” without whom I would not have been able to take this task to fruition. He enlightened my path with continuous support and made me competent during the whole duration of my research.

Finally, I am thankful to all my friends who assisted me in this thesis and throughout the whole research process.

Abstract

Metamorphic testing (MT) represents a robust and innovative methodology that adeptly tackles the challenge of the oracle problem. It supplements traditional testing methods by generating a range of distinct and diverse test cases. However, the generation of effective source test cases, along with their prioritization, continues to be an area of active research interest. In response to this demand, We suggest an innovative and all-encompassing method for generating and prioritizing source test cases. It leverages Python's path tracer and constraint solver to obtain program path constraints, empowering the creation of source test cases with extensive coverage of execution paths, thereby substantially enhancing fault detection effectiveness. Moreover, the proposed approach introduces a sophisticated prioritization technique by assigning higher priority to test cases with higher fault detection capability. Through experimental evaluations on four representative programs, the proposed approach demonstrates exceptional performance and outperforms existing techniques. The incorporation of metamorphic relations enables systematic validation of the behavior of mathematical functions, identifying potential deviations or faults that may arise. Additionally, the integration of mutation testing provides a comprehensive assessment of the approach's effectiveness in fault detection and validation of mathematical functions. This research presents a promising and practical solution to the challenges associated with generating and prioritizing source test cases in metamorphic testing, contributing to the improvement of software testing effectiveness and efficiency. By combining various techniques, we aim to improve fault detection capabilities and provide a practical solution for testing software systems, addressing the specific challenges in the realm of scientific software testing.

Keywords: Metamorphic testing, Fault Detection Effectiveness, Software Testing

Table of Contents

List of Figures

List of Tables

List of Abbreviations

MT	–	Metamorphic Testing
MR	–	Metamorphic Relation
SUT	–	Software Under Test
RT	–	Random Testing
ART	–	Adaptive Random Testing
CSP	–	Constraint Satisfaction Problem
PaDMTP	–	Path Directed Source Test Case Generation and Prioritization in Metamorphic Testing using Python

CHAPTER 1: INTRODUCTION

Traditional testing methods, including black box testing and white box testing, come with inherent limitations. Black box testing centers on the observable behavior of software, often neglecting its internal structure, which can hinder achieving comprehensive coverage. On the other hand, white-box testing requires a deep understanding of the system's internal workings, which may not always be practical or feasible. Consequently, the demand arises for innovative testing techniques that can transcend these limitations and enhance the detection of faults.

In the field of software testing, testers often encounter a significant challenge known as the Oracle problem. The Oracle serves as a crucial mechanism for determining the correctness of test case outcomes. However, there are instances where the availability of an Oracle is limited, or its practical application becomes too costly [1]. Hence, testers are confronted with the challenge of grappling with this issue, which introduces barriers to ensuring the precision of test case executions. Conquering the Oracle problem becomes imperative to elevate the efficacy and efficiency of the software testing process [1] [2] [3]. Metamorphic testing (MT) [4] serves as a purposeful technique meticulously designed to address the challenges brought about by the Oracle problem in the context of software testing. In contrast to relying on explicit input-output behaviors, MT places its focus on meticulously examining the interconnections among program outputs. This approach proves to be a practical alternative when a reliable oracle is unavailable or difficult to establish. Since its introduction in 1998, MT has attracted significant interest within the software testing community, leading to numerous studies exploring different aspects of MT. In recent years, Metamorphic Testing has gained even more attention and demonstrated its effectiveness in detecting a substantial number of real-life faults. The ability of MT to uncover previously unknown faults has surprised the software testing community, showcasing its value as a powerful and innovative testing technique. [5] The field of metamorphic testing has experienced substantial growth in techniques, applications, and assessment studies since its introduction. However, a comprehensive literature review on metamorphic testing is still lacking. Existing overviews often reflect the authors' individual experiences rather than providing a comprehensive analysis of the available research. Therefore, there is a need for a systematic and thorough review that can consolidate and analyze the existing body of research in metamorphic testing. Such a review would greatly contribute to a deeper understanding of the field and help identify areas for further exploration and improvement. [6] [7] [8] [9] and a review of some selected articles [10] [11].

Metamorphic testing (MT) has found extensive application in software testing [12] [13] [14] [15] [16] [17] [18] [19] [20] [21] [22]. Within the realm of software engineering, addressing the oracle problem bears substantial importance in ensuring the caliber and trustworthiness of software systems. By surmounting this challenge, software engineers can construct a reliable framework to authenticate and validate the precision and effectiveness of software systems. This endeavor culminates in an overall enhancement of their holistic quality and dependability [23] [24] [25] [26]. Moreover, Metamorphic testing has been utilized as a validation technique [21] and for quality assessment [27]. Metamorphic testing has showcased its efficacy by proficiently uncovering actual faults in widely used search engines and other software systems.

The proposed approach presented in this research paper extends the work of Liu et al. [28]. While metamorphic testing (MT) proves to be a valuable technique for overcoming the oracle problem and generating effective test cases, it is not without its limitations. A key constraint resides in the process of identifying and choosing appropriate metamorphic relations (MRs) [28]. The proposed path-directed approach for generating source test cases relies on Path constraint solvers, which may impose computational overhead and increase the complexity of the testing process, especially for large-scale programs. Another limitation is that the proposed prioritization technique, although beneficial for improving test case diversity and efficiency, may still require further investigation to explore different prioritization strategies and their impact on fault detection efficiency. Overall, while MT offers promising benefits, addressing these limitations is crucial for enhancing its practical applicability and effectiveness in various software testing scenarios.

Our proposed approach builds upon the foundation of path-directed source test case generation and prioritization in metamorphic testing, introducing innovative techniques to automate test case generation, prioritize based on fault detection capability, utilize a Python path constraint solver, integrate metamorphic relations, and incorporate mutation testing for comprehensive fault detection assessment. By extending the existing methodology, our approach aims to enhance the effectiveness and efficiency of metamorphic testing in testing mathematical functions. It offers advantages such as improved efficiency, coverage, and prioritization, with the accessibility and customization benefits of Python. Empirical studies on benchmark programs demonstrate superior fault detection effectiveness, efficiency, and automation, making our approach a valuable tool for ensuring correctness and reliability, especially in mathematical formulas.

1.1. Motivation

Identifying bugs in certain software types, such as scientific software, can pose specific challenges in the field of software testing. Test oracle problem, which refers to the difficulty in establishing a reliable reference for determining correct outcomes, further complicates bug identification in such scenarios. As a result, addressing the test oracle problem becomes crucial to overcome the challenges associated with testing scientific software effectively [1]The absence of dependable test oracles capable of ascertaining accurate outputs for diverse inputs renders the detection of nuanced faults, isolated errors, or imperfections in scientific software a formidable undertaking. As a result, numerous scientific software systems are frequently categorized as "non-testable programs" due to the complexities involved in devising efficient testing methodologies. [1]

Metamorphic testing was introduced as a promising technique to address the limitations of traditional testing methods and tackle the oracle problem. It offers a unique approach to testing by leveraging metamorphic relations, which define the expected behavior transformations of the software under test (SUT). By utilizing metamorphic relations, MT provides a practical solution to testing scenarios where a reliable oracle is either unavailable or difficult to establish, thereby mitigating the challenges associated with the oracle problem [4]. Metamorphic Testing (MT) tackles the oracle problem by focusing on the relationship between inputs and outputs, rather than directly verifying output correctness for arbitrary inputs. This is achieved using metamorphic relations (MRs), which define the expected impact of input modifications on the outputs. By assessing whether the MRs hold during testing, any deviations detected can indicate the presence of defects within the program.

This thesis employ Metamorphic Testing (MT) as a complementary approach to anticipate Metamorphic Relations (MRs) applicable to mathematical functions. This research aims to facilitate the process by providing automated methods for MR prediction, reducing the burden on experts and programmers involved in the MR identification process. However, despite the potential of metamorphic testing, the identification of suitable metamorphic relations (MRs) often requires domain expertise, making it a labor-intensive task that can challenge both domain experts and programmers involved. Manual identification of MRs can be time-consuming and may limit the practical applicability of metamorphic testing, particularly for complex software systems such as mathematical functions.

By automating MR prediction, this approach aims to enhance the practical applicability and efficiency of metamorphic testing for various software systems, including mathematical functions. This research also focuses on developing an innovative path-directed methodology for generating source test cases, utilizing their associated path constraints extracted from symbolic execution. The separation in path distance among test cases is subsequently harnessed to guide the prioritization of source test cases, leading to heightened efficiency. This approach integrates various techniques, including a path constraint solver implemented in Python, a test case generation algorithm, the establishment of metamorphic relations, and a prioritization technique for test cases. These findings also unveil intriguing research avenues for enhancing metamorphic testing methodologies.

1.2. Problem Statement

Software testing occupies a pivotal role within the software development lifecycle, aiming to detect and rectify flaws present in software systems. Conventional testing methods, including black box and white box testing, while widely used, encounter limitations in achieving comprehensive test coverage and addressing the oracle problem. A promising avenue, termed metamorphic testing (MT), has emerged, not only addressing the oracle problem but also engendering novel test cases through the application of metamorphic relations (MRs). Nonetheless, there remains potential for refinement in the domain of generating and prioritizing source test cases, a research domain that continues to evolve.

The creation of efficient test cases that cover a variety of pathways and behaviors in the system being tested is one of the main issues in software testing. Existing techniques for creating source test cases frequently rely on manual labor, which adds time and increases the possibility of human mistakes. To maximize testing resources and boost the effectiveness of fault detection, it is also essential to choose test cases according to their fault-detection capabilities. Existing prioritization methods, however, do not completely account for the coverage of execution routes and might not integrate path-directed methods.

We propose a unique strategy that uses path-directed methodologies for source test case generation and prioritization in metamorphic testing to overcome these difficulties. The suggested method obtains program path constraints using a Python path tracer [29] and constraint solver [30], allowing for systematic input space exploration and the creation of a wide range of test cases. To increase the variety and effectiveness of testing, we also present a prioritization method based on the distance between test cases. In previous studies [28] [29] [30] [6] [7] [8] [9] path solving was a challenging task that required complex calculations. However, in our research, we have employed machine learning techniques to simplify this process significantly. By harnessing the power of machine learning, we have made path-solving more accessible and efficient, streamlining the overall computation, and achieving more accurate results. The suggested method intends to improve the efficiency of fault identification, automation, and overall metamorphic testing performance, making a significant contribution to the area of software testing. Our method provides a workable alternative to increase the quality and dependability of software systems by automating the test case creation and prioritization procedures as well as adding path-directed techniques.

Throughout this research endeavor, we delved into the incorporation of machine learning methods, particularly within the Python framework, with the intent to bolster the efficacy of fault detection using generated source test cases. The objective was to amplify the identification of software system flaws by harnessing the capabilities of machine learning algorithms and techniques within the test case generation phase. This study aspired to appraise the potential superiority of this approach in comparison to established methods of source test case generation, as evidenced by its heightened fault detection capabilities and overall testing prowess. By addressing these core inquiries, we sought to glean invaluable insights to advance the efficiency and effectiveness of software testing methodologies.

The research questions are as follows:

RQ1: Does the integration of machine learning techniques in Python enhance the fault detection effectiveness of generated source test cases?

RQ2: To what extent does this approach outperform existing source test case generation techniques in terms of fault detection capabilities and overall testing effectiveness?

RQ 3: What is the computational overhead associated with the implementation of this approach for generating source test cases, and how does it impact the efficiency and scalability of the testing process?

1.3. Aims and Objectives

The primary objectives of this research activity are described as follows, encapsulating the fundamental aims and goals of the study:

1. Analyze existing approaches and techniques in metamorphic testing, identifying their limitations specifically in path-directed source test case generation and prioritization.
2. Propose a novel methodology, PaDMTP, that integrates path-directed techniques with metamorphic testing principles, aiming to enhance the generation and prioritization of source test cases.
3. Implement the PaDMTP methodology using the Python programming language, leveraging its flexibility and widespread adoption in the software testing community.
4. Examine the efficacy and efficiency of PaDMTP by conducting experiments on different software systems, and then compare the obtained results with existing approaches.
5. Provide comprehensive guidelines and recommendations for practitioners and researchers on the application of the PaDMTP methodology in real-world software testing scenarios.
6. Contribute to the progression of metamorphic testing methodologies by tackling the hurdles linked to path-directed source test case generation and prioritization.

1.4. Metamorphic Testing

Metamorphic Testing(MT) involves modifying the input data of an algorithm within specified constraints to predict the characteristics of the resulting output. This technique utilizes input-output relationships to identify faults and assess the correctness of algorithms. By systematically applying metamorphic transformations and comparing outputs, it provides a powerful approach for testing and validating software systems.

Metamorphic testing has emerged as a promising technique that addresses the limitations of traditional testing methods. Unlike relying solely on a predetermined set of test cases, MT focuses on discovering unexpected relationships between input and output. It operates on the principle that specific properties or relationships should hold for a particular class of inputs, regardless of potential output variations. When deviations from these expected properties occur, it indicates the presence of faults within the system being evaluated.

MT lies the identification and formulation of metamorphic relations(MRs). These MRs define the anticipated relationships between inputs and outputs, providing valuable guidance for generating test cases and validating system correctness. Derived from domain-specific knowledge, system behavior, specifications, or mathematical models, MRs enables the detection of faults that may remain undetected using traditional testing methods. By leveraging MRs, MT offers an effective approach to enhance test case generation and improve fault detection capabilities.

1.4.1. Challenges in Metamorphic Testing

Metamorphic testing brings forth several potential advantages; however, it also introduces distinct challenges that warrant careful consideration. Among these challenges, a significant one lies in devising diverse and effective test cases that span various execution paths. The creation of test cases capable of traversing multiple execution paths and exploring a range of scenarios proves pivotal in effectively harnessing the potential of metamorphic relations (MRs). Since MRs depend on recognizing connections between inputs and outputs, this diverse testing approach is crucial. Nevertheless, manually generating such an extensive array of test cases can become a time-intensive and error-prone process.

Another challenge revolves around the effective ordering of test cases to ensure efficient fault detection. In situations of constrained testing resources, it becomes imperative to allocate them judiciously to maximize the likelihood of detecting issues. The strategic arrangement of test cases based on their capacity to uncover faults can notably enhance the efficiency of the testing process. Nonetheless, devising impactful strategies for prioritization, which take into account multifaceted factors like code coverage and fault detection competence, can pose intricate demands on resources.

1.5. Proposed Approach

This research paper presents a novel approach for path-directed source test case generation and prioritization in metamorphic testing using python(PaDMTP), with a specific focus on utilizing the Python programming language. The main objective of this research is to improve the effectiveness and efficiency of metamorphic testing by automating the generation of test cases that cover diverse execution paths. Additionally, the research aims to prioritize these generated test cases based on their potential to detect faults in the system under test. Through the automation of the test case generation process and the refinement of prioritization techniques, this research aims to elevate the overall quality and dependability of metamorphic testing. The proposed approach (Figure 01) involves a comprehensive framework that integrates various techniques, including Python Path Tracer [29], Constraint Solver [30] and Mutation Testing [31]. The experiments confirm that the suggested method effectively finds issues in typical programs. This shows potential for improving software testing in metamorphic testing (MT).

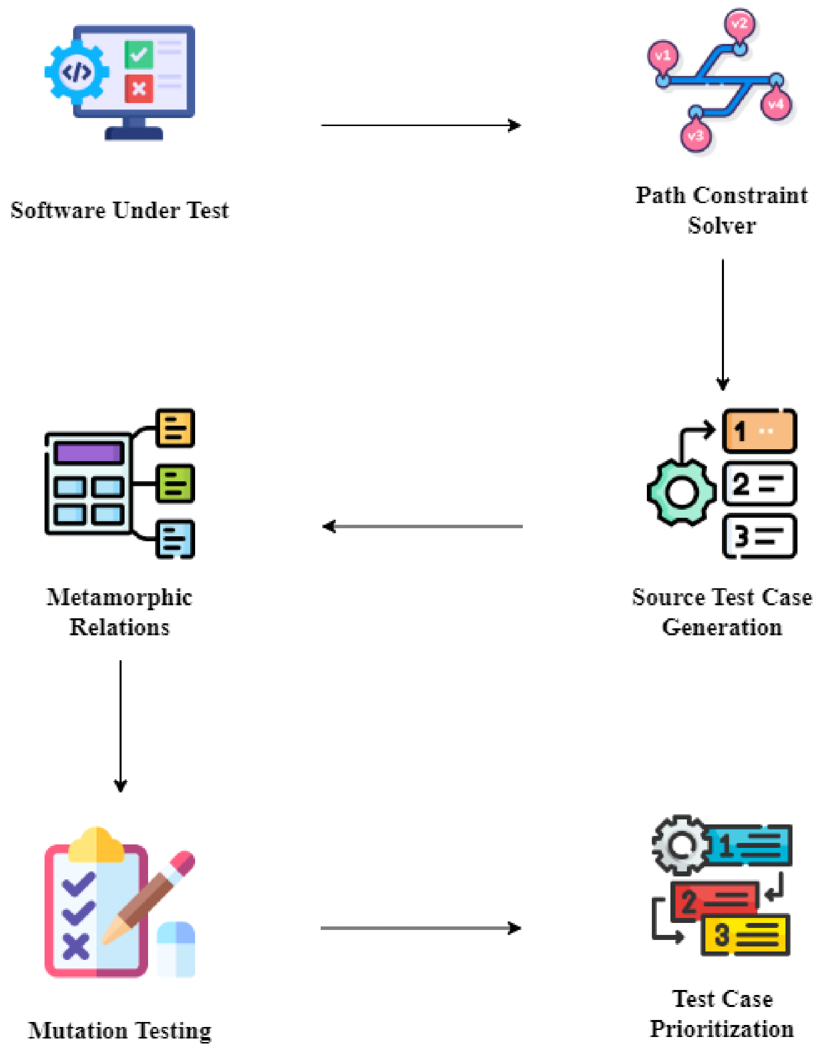


Figure 01: Proposed Approach – Overview

1.5.1. Python Constraint Solver

To explore the complex input spaces of mathematical functions and generate test cases that comply with the given constraints, we employ a Python path constraint solver. The created test cases correctly mimic real-world input conditions thanks to the solver's capability to handle complicated equations and inequalities, making them extremely pertinent for error identification and validation. Our approach substantially enhances the efficiency and success of generating test cases. This guarantees that the created test cases comprehensively cover the behaviors of mathematical functions, simplifying the identification of potential issues with heightened accuracy and precision.

Our suggested method makes use of the skills of a Python constraint solver to deal with the complexity of mathematical functions and their related input restrictions. The development of test cases that fulfill certain requirements imposed by the functions under test is made possible by this solver, which acts as a potent tool for reasoning about mathematical equations, inequalities, and constraints. We may methodically investigate the enormous input space and produce test cases that span a broad variety of potential behaviors of the mathematical functions by utilizing the adaptability and computing power of the route constraint solver. This methodical investigation makes sure that our strategy covers a wide range of test cases, efficiently capturing numerous scenarios and edge circumstances that may potentially uncover concealed errors or inconsistencies inside the mathematical functions. [30]

1.5.2. Generation of Source Test Cases

Harnessing the power of a Python constraint solver, our approach empowers the exploration of the complex input spaces of mathematical functions, enabling the generation of test cases that adhere to given constraints. This solver proves invaluable in dealing with intricate equations and inequalities, allowing us to create test cases that accurately represent real-world input conditions. By leveraging the solver's capabilities, our method significantly enhances the effectiveness and efficiency. The resulting test cases boast comprehensive coverage of the behaviors exhibited by the mathematical functions, making them potent tools for identifying potential errors with a high degree of accuracy and precision. The ability to generate test cases that mimic real-world scenarios and encompass a diverse range of potential behaviors ensures that our approach is well-suited to uncovering hidden flaws and inconsistencies within mathematical functions.

Our proposed methodology further incorporates a test case generation, meticulously designed to leverage the capabilities of the Python constraint solver. Through systematic exploration of the vast input space, the algorithm considers various combinations and ranges of inputs, ensuring a thorough assessment of the SUT’s potential behaviors. The objective is to produce a diverse set of test cases that encapsulate different scenarios and edge cases, effectively exploring various execution paths and shedding light on potential faults lurking within the mathematical functions under scrutiny. By encompassing a broad spectrum of behaviors, the generated test cases become powerful instruments for revealing any underlying issues or discrepancies within the mathematical functions. The systematic and comprehensive nature of our algorithm not only maximizes the fault-detection capabilities but also contributes to the overall effectiveness and efficiency of our metamorphic testing approach, allowing us to achieve a deeper and more nuanced understanding of the SUT's behavior.

1.5.3. Metamorphic Relations

The establishment and utilization of metamorphic relations (MRs) constitute a pivotal aspect of our research, acting as a guiding principle in the evaluation of mathematical functions. These MRs play a vital role in specifying the expected relationships between the inputs and outputs of the functions under examination. Drawing from domain-specific knowledge and system behavior, we formulate a comprehensive set of MRs that encapsulate the inherent properties of the mathematical functions. These metamorphic relations serve as a foundation for assessing the correctness and behavior of the functions by facilitating a comparison of outputs derived from related inputs. By systematically applying MRs to the generated test cases, they function as a validation mechanism, assuring the stability and dependability of the mathematical functions. This process involves confirming that outputs align with the specified MRs, effectively spotting any deviations or flaws that might emerge. This robustly enhances the overall ability of our metamorphic testing approach to detect faults.

To achieve comprehensive testing, we meticulously design and employ a set of metamorphic relations that encompass a wide array of possible input-output relationships within the mathematical functions. These MRs act as essential tools for guiding the evaluation of the functions' behaviors, providing valuable insights into their correctness and performance. Formulated based on a deep understanding of the mathematical functions and their expected properties, the MRs facilitate a rigorous validation process by comparing the outputs produced from related inputs. By systematically applying these MRs to the generated test cases, we ensure that the functions' behavior remains consistent and within the expected bounds. The methodical application of MRs enhances the reliability and precision of the test results, allowing us to identify any deviations or faults within the functions with a high degree of accuracy. Through the integration of MRs into our metamorphic testing approach, we gain a comprehensive and systematic understanding of the functions' behavior, further elevating the effectiveness and efficiency of our testing methodology.

1.5.4. Mutation Testing

Mutation testing is a robust and sophisticated software testing technique that serves as a stringent evaluator of the efficacy of test cases. Unlike traditional testing approaches that measure the success of tests by their ability to pass, mutation testing introduces deliberate changes, or mutations, to the source code. These mutations mimic potential faults or defects that might exist in the software. The core idea is to assess the test suite's capability to identify these altered code segments, thereby gauging its robustness in detecting real-world errors. By applying a range of mutations across the codebase, mutation testing mimics different types of faults, ensuring a comprehensive evaluation of the test suite's effectiveness.

The process of mutation testing entails the creation of numerous mutated versions of the original code. These mutations might involve modifications such as changing operators, altering conditions, or introducing simple logic errors. The test suite then runs these mutated versions and assesses whether the test cases successfully identify and flag the changes. If a mutation is not detected by the test suite, it suggests a deficiency in the testing strategy and highlights a gap in the suite's ability to catch certain types of faults.

Mutation testing provides a valuable quality assessment of the test suite's comprehensiveness. It identifies areas where the test suite might be lacking, offering developers insights into the types of errors that the current suite might miss. While mutation testing can be computationally intensive due to the generation and execution of multiple mutated versions, it offers a rigorous and holistic perspective on the software's testing robustness. In essence, mutation testing enhances the accuracy and reliability of software testing by ensuring that test cases are capable of detecting a broad spectrum of potential defects, leading to higher quality software products. Mutation testing is a crucial step in assessing the robustness and fault-detection capabilities of our metamorphic testing approach. By introducing artificial faults into the mathematical functions, we simulate various scenarios that may potentially lead to errors or inconsistencies in the system. The generated test cases are then executed against these mutants to determine their effectiveness in identifying and detecting the injected faults. The mutation score serves as a quantitative measure of the approach's ability to identify these faults accurately and efficiently. A high mutation score indicates a strong fault-detection capability, suggesting that the generated test cases are effective in uncovering potential issues within the mathematical functions. By analyzing the mutation score, we can gain valuable insights into the reliability and precision of our testing methodology, further validating the effectiveness of our approach in ensuring the correctness and performance of the mathematical functions. Through the integration of mutation testing, our research aims to provide a comprehensive and rigorous evaluation of the proposed approach, enhancing its credibility and applicability in real-world scenarios.

1.5.5. Test Case Prioritization

To optimize resource utilization and streamline the testing process, our approach integrates an advanced test case prioritization technique. This technique is devised to establish a sequence for executing test cases based on diverse factors, encompassing elements like code coverage, input constraints, and the ability to detect faults. By meticulously assigning priorities to test cases, our goal is to concentrate on pivotal sections of the code that hold the potential to expose potential flaws. This prioritization strategy effectively ensures that limited testing resources are allocated strategically, targeting the most critical facets of the mathematical functions. This, in turn, heightens the efficiency of the testing process.

Our test case prioritization technique follows a systematic approach to optimizing the testing process. By incorporating criteria such as code coverage and input constraints, we ensure that test cases are ordered in a way that allows for a comprehensive evaluation of the mathematical functions' behavior. Critical areas of the code that may be prone to faults are given higher priority, allowing us to allocate testing resources more effectively and increasing the chances of detecting and resolving potential issues. The technique is adaptive and flexible, accommodating various complexities of the mathematical functions and their input spaces, providing a dynamic approach to fault detection. With the prioritization technique in place, we can achieve a deeper understanding of the functions' behavior and thoroughly assess their correctness and performance while utilizing testing resources efficiently.

1.6. Significance of the Research

The importance of this study rests in the real-world utility of the suggested method for mathematical functions. By automating the test case generation process and incorporating the Python path constraint solver, researchers and practitioners can efficiently test mathematical functions and identify potential faults. The utilization of Python as the implementation language ensures accessibility and ease of adoption for the wider software testing community.

The research focuses on the practical usability of the proposed approach in the context of mathematical functions. Metamorphic relations provide a systematic means to validate the behavior of mathematical functions and detect deviations or faults. Test case prioritization optimizes the allocation of limited testing resources, maximizing the chances of detecting faults and improving overall test effectiveness.

The proposed approach contributes to the advancement of metamorphic testing methodologies and techniques, particularly in the context of testing mathematical functions. By combining various techniques and leveraging the power of the Python programming language, this research aims to improve fault detection capabilities and provide a practical solution for testing mathematical functions.

1.7. Thesis Outline

In this thesis, we present a comprehensive study on the application of Metamorphic Testing in software testing. The proposed research introduces a novel approach for path-directed source test case generation and prioritization in metamorphic testing using Python. By defining metamorphic relations and leveraging Python's capabilities, this approach aims to automate the generation of test cases with varying input scenarios, ensuring comprehensive and reliable testing of software systems. This research seeks to enhance the efficiency and effectiveness of metamorphic testing, contributing to improved software quality and reliability across various domains.

The literature review in Chapter 2 introduces the concept of Metamorphic Testing and discusses its principles and techniques and provides an in-depth analysis of the existing research and studies related to Metamorphic Testing. In Chapter 3, we outline the methodology adopted for our research and explain the steps involved in implementing Metamorphic Testing in the context of software testing. Experimental studies and results are presented in Chapter 4, where we describe the design, execution, and results of our experiments to evaluate the effectiveness of Metamorphic Testing in fault detection. Finally, Chapter 5 concludes the thesis by summarizing the key findings, discussing the results of the study, and proposing future research directions for further improvement and application of Metamorphic Testing in software testing.

CHAPTER 2: LITERATURE REVIEW

Metamorphic testing(MT) is a powerful technique for ensuring the quality and reliability of software systems. It utilizes metamorphic relations(MRs) to generate test cases that exhibit expected behavior transformations. However, test case generation and prioritizing in metamorphic testing can be challenging. This literature review focuses on a novel approach that combines path-directed techniques and Python implementation to address this challenge. Metamorphic testing is a powerful method for enhancing the quality of software systems. It addresses the limitations of traditional testing techniques by focusing on the input-output behavior of software and utilizing metamorphic relations to identify potential faults [32]. Various existing techniques for test case generation in metamorphic testing have been proposed, including constraint-based testing (CBT) [33] and evolutionary algorithms.

2.1. Software Testing

Software testing is an indispensable and fundamental process in the software development life cycle (SDLC), utilized to assess and evaluate the quality of developed software products. [34] Its significance lies in guaranteeing that the software aligns with prescribed standards, specifications, and requirements established during the developmental stages. Software testing encompasses an all-encompassing assessment, closely examining the performance of software systems, spanning from discrete individual components to intricate integrated systems. The principal aim is to verify that the software operates as designed, yielding the anticipated results, and effectively satisfying the demands and anticipations of end-users.

The process of software testing involves executing the software with carefully designed test cases and analyzing the actual behavior of the software against the expected outcomes. These expected outcomes are determined based on predefined requirements and specifications, often established using a test oracle. By comparing the actual behavior of the software with the expected behavior, testing aims to identify any discrepancies, bugs, or defects that may arise during execution. The detected issues are then meticulously documented, reported, and addressed through the development and debugging process. The rigorous nature of software testing provides stakeholders with invaluable insights into the quality, reliability, and performance of the software, helping them make informed decisions about its readiness for deployment and use in real-world scenarios. Moreover, it aids in risk assessment, ensuring that potential vulnerabilities and weaknesses are identified and resolved before the software is released to end-users.

Software testing encompasses an array of testing techniques, such as functional testing, performance testing, security testing, and usability testing, among others. Each testing approach fulfills a distinct role, collectively enhancing the quality assurance process and guaranteeing that the software adheres to the utmost standards of quality and dependability. Through systematic and comprehensive testing, organizations can mitigate potential risks, avoid costly software failures, and enhance user satisfaction. Effective software testing is a key factor in achieving customer confidence and loyalty, as it ensures that the software functions as expected, deliver value to end-users, and meets their needs effectively. In conclusion, software testing is a critical aspect of software development that guarantees the overall success of software systems by providing a robust and reliable software product that meets user expectations and industry standards.

2.1.1. Source of Software System Defects

Software bugs or defects can arise from various factors, contributing to the complexity of the software testing process. One prevalent cause of bugs is coding errors, which frequently occur during the development process. While developers often put in considerable effort to review and validate the code, some bugs may persist due to coding mistakes that stem from incorrect initial coding concepts. In certain cases, these errors may go unnoticed during code review, leading to hidden defects in the software [35]. Despite the best intentions of the developers, underlying design flaws or unrecognized coding errors can evade detection, only to manifest later during execution.

Apart from coding errors, requirements gaps are another common reason for software bugs or defects. These gaps occur when programmers unintentionally omit necessary information or fail to fully understand and recognize specific requirements. Incomplete or misunderstood requirements can result in the implementation of incorrect functionalities or the absence of crucial features, leading to unexpected behaviors and defects in the software. Addressing requirements gaps requires effective communication between stakeholders and developers to ensure that all aspects of the software's functionality are well-defined and understood.

Furthermore, software bugs or defects can also be influenced by changing environments, input data, and hardware platforms. As software systems interact with dynamic and diverse environments, variations in the input data or hardware configurations can lead to unexpected behaviors and defects. Ensuring robustness and resilience in the software requires extensive testing across various scenarios and configurations to identify and resolve potential issues arising from environmental variations.

2.1.2. Benefits of Software Testing

The defects mentioned earlier often prove challenging to identify and rectify solely through programmer reviews. Additionally, these defects can have severe consequences, including unexpected failures in software systems, leading to substantial economic losses. For instance, a study conducted by NIST in 2002 revealed that software bugs accounted for approximately $59.5 billion in economic losses annually in the USA. Moreover, the study found that implementing feasible and effective software testing strategies could have potentially prevented more than one-third of these losses. These findings highlight the importance of employing robust software testing approaches to mitigate the risks associated with software defects and minimize economic impacts [36].

Software system failures have significant economic ramifications on a global scale annually, affecting sectors like entertainment, government, finance, transportation, and more. Surprisingly, many of these unexpected software failures could be prevented by implementing appropriate testing techniques. Regrettably, a considerable portion of software undergoes insufficient evaluation or testing procedures before production deployment. This rushed approach undermines the importance of thorough testing, leading to an increased likelihood of encountering severe issues and subsequent economic losses. Emphasizing comprehensive evaluation and testing procedures is crucial to mitigate the risks associated with software failures and their consequential impacts on various aspects of society.

While there are numerous guidelines available for successful software development, it is important to note that following these guidelines does not guarantee absolute success. Software development is a complex and multifaceted process, influenced by various factors such as project requirements [37] [38] Inadequate project post-mortems and limited understanding from past projects restrict the identification of significant success and failure factors, limiting opportunities for improvement and proactive measures.

To prevent economic losses and enhance profitability, technology companies typically maintain professional software testing departments. These departments ensure higher correctness and lower risks of failure in software production.

2.2. Metamorphic Testing

This section aims to provide a comprehensive overview of metamorphic testing. Metamorphic testing, introduced by Chen et al. [4], is a software testing technique designed to address the test oracle problem. We begin by delving into the explanation behind the test oracle problem, the functioning of metamorphic testing, highlighting its reliance on metamorphic relations. Furthermore, we discuss the inherent challenges associated with this technique and elucidate the rationale behind incorporating machine learning methods.

2.2.1. Test Oracle Problem

To enhance the efficiency, convenience, and trustworthiness of software testing, the utilization of a test oracle becomes essential. The test oracle acts as a mechanism for determining the accuracy of software execution. It accomplishes this by evaluating the software's output and comparing it with the anticipated output, thereby validating its correctness. The test oracle has a crucial role in the testing process, offering an impartial evaluation of the software's behavior.

In the realm of software testing, especially concerning scientific software, evaluating the success or failure of a test can be intricate due to the lack of an accessible test oracle. This absence or the challenges associated with constructing one gives rise to the test oracle problem (Figure 02), which represents a significant obstacle in the field of software testing. Addressing this challenge is pivotal to ensuring effective and trustworthy testing methodologies.

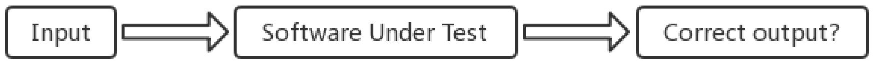


Figure 02: Test Oracle Problem

The presence of the test oracle problem introduces challenges in detecting subtle faults and isolated errors, which can have a significant impact on the accuracy and dependability of software or programs. Historically, resolving this issue has involved domain experts or scientists manually defining test oracles. However, this process is both inefficient and lacks systematic structure. Consequently, there is a demand for an automated testing technique that can effectively evaluate software even in the absence of test oracles. Such an approach would be invaluable in ensuring efficient and reliable testing practices.

In a recent research study, a novel approach was introduced to automatically predict metamorphic relations (MRs) using machine learning techniques. This approach was targeted at programs that lack test oracles. The proposed method employs features extracted from a function's control flow graph to forecast likely MRs. The study demonstrates the effectiveness of this approach, with Support Vector Machines (SVMs) outperforming decision trees. The SVM predictive model achieves high accuracy and Area Under the Curve (AUC), indicating the success of the CFG-based features in predicting MRs. The approach displays potential for practical application, as it can create effective classifiers with reasonably small training sets. Furthermore, the identified MRs exhibit reliability even for faulty programs, as demonstrated through mutation analysis. Overall, this research contributes to automating testing for software without test oracles, thereby enhancing software quality assurance [39].

2.2.2. Metamorphic Testing: Background

Metamorphic Testing (MT) stands as a testing methodology that centers on the interconnections between the inputs and outputs of a software system, as opposed to directly validating the accuracy of individual outputs. Its objective is to surmount the difficulties presented by the test oracle problem, which can impede effective testing due to limitations in the availability or feasibility of a test oracle.

In MT, metamorphic relations (MRs) are defined, which specify the expected relationships between inputs and corresponding outputs. These relations are derived from domain knowledge, system behavior, specifications, or mathematical models. By applying specific transformations to inputs and comparing the outputs against the expected relations, MT can detect potential faults or deviations in the system's behavior.

Metamorphic testing emerges as a captivating software testing method, offering an intriguing and efficient avenue to address the oracle problem often encountered in certain software systems. This innovative technique was initially introduced by Chen et al. [4]. Metamorphic testing has seen remarkable progress since its inception in 1998. Over the last two decades, this technique has rapidly evolved and found extensive application in various research domains, as evident from Figure 03. Clang and GCC [17] and Xie et al. [21] Used MT to validate machine learning algorithms. NASA’s Data Access Toolkit was subjected to applied metamorphic model-based testing by Lindvall et al. [19]

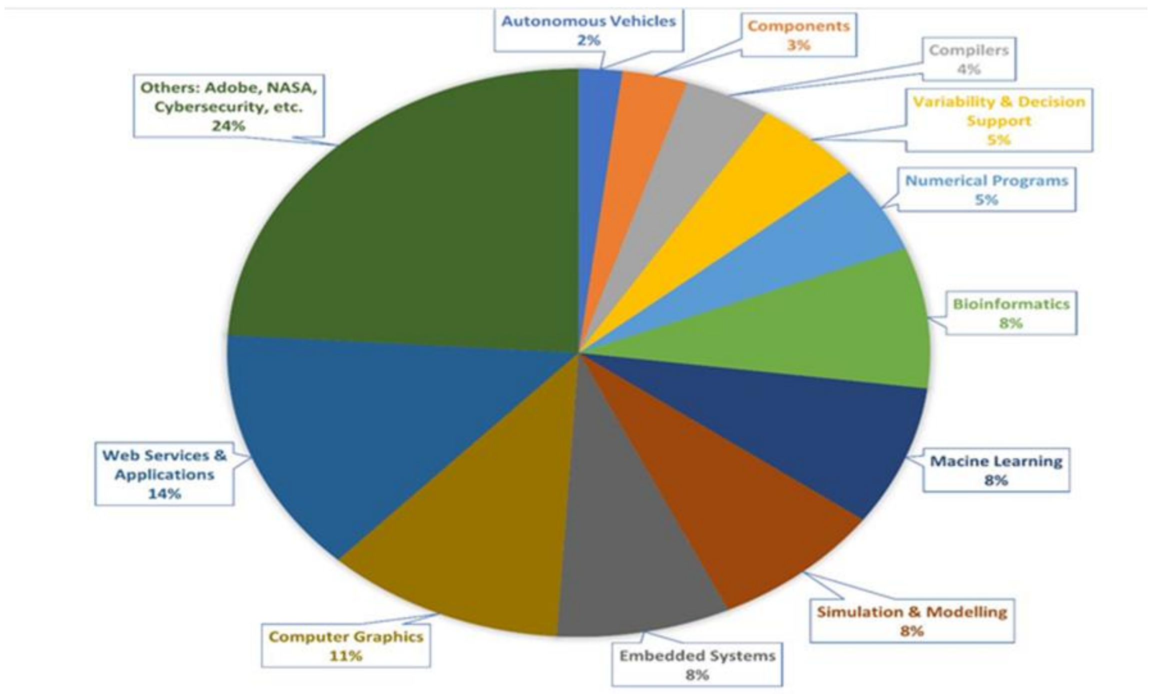


Figure 03: Survey of Application Areas of Metamorphic Testing

Metamorphic testing has also been employed for testing autonomous cars, expanding its application beyond traditional software systems [40]. Most of the research studies have primarily focused on applying metamorphic testing techniques to detect bugs in widely used software and various applications. However, only a minority, approximately 8% of the case studies, have explored the machine learning aspect. Furthermore, among these machine learning studies, only a few have specifically investigated the automatic prediction of metamorphic relations using machine learning methods.

Metamorphic Testing is a property-based testing approach that involves verifying whether a program adheres to predefined properties known as metamorphic relations. These relations specify the expected changes in program output when certain modifications get applied to the input. By comparing the observed outputs with the expected relations, metamorphic testing can detect faults or deviations in the program under evaluation. This technique offers a systematic way to identify potential program defects, even in the absence of a traditional test oracle.

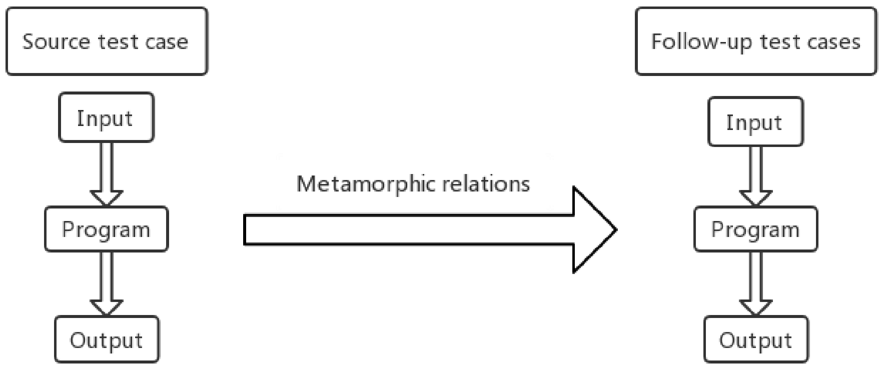


Figure 04: Overview of Metamorphic Testing Process

In general, the implementation of metamorphic testing involves the following steps (as illustrated in Figure 04):

1. Establish a set of metamorphic relations that define the expected relationships between the inputs and outputs of the target program.
2. Generate a set of new source test cases or select existing ones to serve as the initial inputs for the metamorphic testing procedure.
3. Generate follow-up test cases by applying the identified metamorphic relations to the initial source test cases. This step involves transforming the input values of the source test cases based on the specified metamorphic relations, resulting in a set of follow-up test cases.
4. Execute the follow-up test cases on the target program, if there is any violation then the test is failing otherwise the test is passed.

The primary goal of metamorphic testing is to utilize newly generated follow-up test cases, guided by metamorphic relations, to identify potential faults in programs that lack test oracles.

One of the most straightforward examples to illustrate metamorphic testing is by considering the implementation of a SINE function.

Let the equation be $a = b + c$

Applying Sin relation to the above equation

$$sin(a) = sin\ (b + c)$$

$$sin(a) = sin\ (b)\ cos\ (c) + cos\ (b)\ sin\ (c)$$

$$sin(a) = sin\ (b)\ sin\ (90^\circ - c) + sin\ (90^\circ - b)\ sin\ (c)$$

When applying MT to P based on this MR, we generate a source test case x and subsequently create two follow-up test cases, y, and z

$$x = y + z$$

The System Under Test (SUT) will undergo a total of five executions. i.e.,

$$P(x),\ P(y),\ P(z),\ P(90^\circ - y),\ and\ P(90^\circ - z)$$

Now we check the relation between $P(x) = P(y)P(90^\circ - z) + P(90^\circ - y)\ P(z)$ as we did in the above scenario, then if there are any violations of the relationship between the input and expected output, in the SUT, it will indicate that the test leads to failure otherwise the test will be indicated as passed.

2.2.3. Metamorphic Relations

Metamorphic relations encompass predetermined correlations between arbitrary inputs and their respective outputs within a program. To illustrate this, let's delve into an example. (Figure 05) where metamorphic testing is put into practice on a mathematical function responsible for computing the sum of an array. Certain modifications to the input array are expected to have no impact on the output result. For instance, randomly permuting the elements of the input array should not alter the sum. These anticipated relationships, known as metamorphic relations, provide a basis for detecting faults or failures if any violations occur.

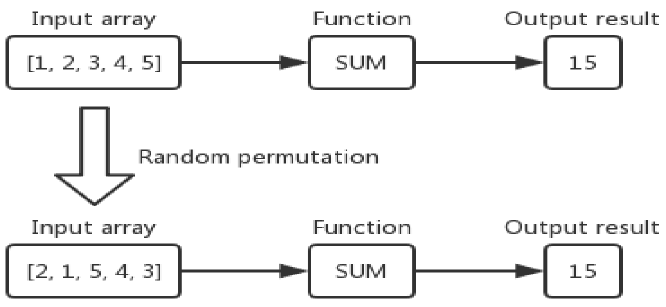


Figure 05: Example of Permutative Metamorphic Relation (MR)

Apart from the Permutative metamorphic relation, various other connections between inputs and outputs can manifest in a program. Previous research by [41] has identified six specific metamorphic relations applicable to mathematical functions that operate on arrays as input.

Table 1: Metamorphic Relations (MRs)

Metamorphic Relations	**R1**	**R2**
Additive	Add a positive constant	Increase or remain
Inclusive	Add a new variable	Increase or remain
Multiplicative	Multiply by a positive constant	Increase or remain
Permutative	Randomly permute a value	Remain

In our strategy, Metamorphic Testing (MT), which is based on the underlying Metamorphic Relations (MRs), plays a key role in aiding the test case production process and integrates a methodical approach for checking test findings. A series of essential steps are involved in applying MT, and they are briefly described as follows, based on [42]:

1. **MR Identification:** A collection of Metamorphic Relations (MRs) that describe the anticipated behavior and connections between the system's inputs and outputs are found and created as the first stage in the MT process. These MRs are formed from the understanding of how the system ought to react to operations or changes.
2. **Test Case Generation:** We must now create follow-up test cases based on the metamorphic relations (MRs), in (Table 1), for this purpose. Follow-up test cases are generated by performing the actions in column R1 on the source test cases.
3. **Test Case Execution:** Upon execution of the follow-up test cases, their respective outputs are systematically compared with the results obtained from the source test case. This comparative analysis serves as a fundamental step in the evaluation process, aiming to identify any disparities or divergences between the output behaviors of the follow-up test cases and the source test case.
4. **Output Comparison:** The output obtained from executing the follow-up test cases is meticulously compared with the output generated by the source test cases.
5. **Result Verification:** Following the comparison of results, the determination of the success or failure of the Metamorphic Relation (MR) is predicated on the evaluation of

the predefined criterion specified in column R2. The outcome of this evaluation serves as the basis for categorizing the MR as either successful or unsuccessful.

When modifying the input results in a change in the program's output following the expected behavior, the program aligns with the relevant metamorphic relation. However, accurately recognizing these metamorphic relations can be challenging, particularly for testers who lack in-depth domain knowledge.

2.2.4. Uses of Metamorphic Testing

Metamorphic testing has emerged as a promising approach to tackle the testing challenges of machine learning (ML) applications, where the correct answers are unknown. By exploiting the properties of the application, metamorphic testing defines transformation functions on input data, enabling the detection of defects when the output deviates from expected results. In a recent study, researchers extensively examined and categorized the metamorphic properties of various ML algorithms. They effectively pinpointed six fundamental properties: additive, multiplicative, permutative, invertive, inclusive, and exclusive. By employing these properties, the researchers showcased how metamorphic testing can effectively expose flaws in particular machine-learning applications. These findings establish a solid basis for future investigations, including the broader application of metamorphic testing in different ML domains, ultimately enhancing the quality assurance of non-testable programs. [43]

Addressing the challenges of reusability and cost in metamorphic testing has been the subject of significant research. Zhang et al. [42] present a novel approach in their paper titled "Predicting Metamorphic Relations Based on Path Features" to enhance the efficiency and effectiveness of this testing methodology. They propose a unique string feature extraction method that leverages common metamorphic relations found in scientific computing programs and their execution paths. They adopt a method that includes training support vector machine models to precisely forecast metamorphic relations during testing. The experimental findings showcased by Zhang et al. substantiate the method's effectiveness in consistently ascertaining the fulfillment of input features concerning metamorphic relations. This study corresponds with our objective of enhancing the efficiency and utility of metamorphic testing within the domains of machine learning and scientific computing applications. [20]

Brailsford et al. [44] provide a comprehensive exploration of constraint satisfaction problems (CSPs) and their applications in operational research in their paper titled "Constraint Satisfaction Problems: Algorithms and Applications." CSPs involve assigning values from a finite domain to variables, ensuring the satisfaction of constraints among the variables. The authors highlight the wide applicability of CSPs in various combinatorial problems, including scheduling and timetabling, commonly encountered in operational research. While CSP approaches are well-established in artificial intelligence research, their adoption among operational researchers is limited. This research serves as an essential resource, introducing CSPs to the operational research community and presenting fundamental techniques for solving CSPs. Furthermore, he compares constraint satisfaction approaches with established operational research techniques like integer programming, branch and bound, and simulated annealing. The insights provided by this study contribute to the understanding and utilization of CSPs in operational research contexts. [45]

A study [46] focused on the application of metamorphic testing (MT) to bioinformatics software, particularly LingPipe, a tool for bio-entity recognition from biomedical literature. The challenge lies in the absence of a well-defined test oracle due to the large number of bio-entities produced. Metamorphic relations (MRs) are proposed as a solution to determine test outcomes. Ten novel MRs are developed to validate bio-entity recognition tools in general, and experimental results demonstrate their effectiveness in identifying faults in LingPipe. By leveraging MT and MRs, this study aims to improve the quality assurance of bioinformatics software, thereby enhancing critical decision-making in medicine and healthcare.

2.3. Python as an Implementation Language

Python has gained widespread popularity in the field of software development, largely attributed to its simplicity, versatility, and extensive libraries. It provides a rich ecosystem of libraries and tools that are particularly valuable for constraint-solving and machine learning-based prioritization. Leveraging these powerful libraries, Python offers efficient solutions for test case generation and prioritization in metamorphic testing [47] [48], making it an ideal choice for software testing tasks.

In our approach, we utilize Python path constraints to tackle the challenge of test case generation. By employing Python's path constraint solver, we can effectively reason about complex mathematical functions and their input constraints. This enables us to systematically explore the vast input spaces of these functions and generate test cases that comply with the given constraints. The resulting test cases accurately mimic real-world input conditions, making them highly relevant and suitable for thorough error identification and validation.

Furthermore, we employ Python's machine learning-based prioritization techniques to enhance the efficiency of our testing process. By harnessing the capabilities of machine learning algorithms, we can intelligently prioritize test cases based on various criteria, such as code coverage, input constraints, and fault detection capability. This prioritization strategy allows us to focus our testing efforts on critical areas of the code, maximizing the potential for fault detection and optimizing the allocation of testing resources.

To evaluate the effectiveness of our approach, we employ mutation analysis and calculate the fault detection rate (FDR). Mutation analysis involves introducing synthetic faults, or mutants, into the mathematical functions and assessing how well the generated test cases can detect these mutations. The FDR provides a quantitative measure of our approach's ability to identify and detect faults within the functions. This comprehensive evaluation ensures that our testing methodology is effective in uncovering potential issues and validating the correctness and performance of the mathematical functions.

2.4. Path-Directed Test Case Generation

Path-directed test case generation techniques, such as symbolic execution and constraint-based testing, have been widely used in software testing. These techniques aim to systematically explore the program's execution paths to generate test cases that cover a wide range of scenarios. Path-directed techniques can be adapted and applied to metamorphic testing to generate effective and diverse test cases that satisfy the defined metamorphic relations. A significant number of previous studies have adopted a random approach to generate source test cases for the existing testing techniques [49] [50].

Similarly, a new technique was proposed for automatically predicting MRs in 2013. In their research, they focused on predicting metamorphic relations (MRs) for Java testing programs that accept arrays as inputs [51]. In 2018, Kanewala et al [52], extended their approach to Java testing programs that utilize matrices as inputs, demonstrating the applicability of their methods to a broader range of testing scenarios. In 2022, Chang-ai Sun proposed a new approach to metamorphic testing through path-directed and prioritization of metamorphic relations. [28]

We extend path-directed source test case generation with the help of Python. In our approach, we use Python path constraints for the generation of source test cases.

The research proposes an innovative graph kernel-based machine learning approach to address the oracle problem in scientific applications utilizing matrices. By automating the prediction of metamorphic relations (MRs) for matrix calculation programs, the study offers a more efficient alternative to labor-intensive manual identification. The successful application of this method not only improves testing effectiveness but also enhances the understanding and visualization of complexity in scientific domains. The integration of advanced machine learning and data visualization techniques holds significant promise for advancing the field of science, enabling more robust and reliable testing practices for complex numerical calculations involving matrices. [53]

2.5. Path-Directed Test Case Prioritization

Test case prioritization (TCP) is a strategy employed to improve efficiency in achieving performance goals. Different researchers used different techniques for prioritizing the test cases. Regression testing is one of the best testing techniques for prioritizing test cases. [54] [55] [56] They used different techniques for prioritizing the test cases.

The research proposes a solution to the test oracle problem in software testing using metamorphic testing. It aims to identify and prioritize metamorphic relations (MRs) based on their fault-finding effectiveness for complex systems. The approach involves using four metrics to measure diversity in test case execution behavior, leading to better MR prioritization. This enhances the efficiency and effectiveness of metamorphic testing in detecting faults and ensuring software reliability. [57]

Our novel approach, presented in this literature review combines path-directed techniques with Python implementation to generate and prioritize test cases. The approach leverages constraint solving, and machine learning techniques available in Python to ensure thorough coverage of execution paths while satisfying metamorphic relations. This approach offers a promising solution for effective test case generation and prioritization in metamorphic testing.

To understand path-directed test case prioritization some definitions are given below:

- **Statement Coverage:** This approach prioritizes test cases based on the number of statements covered in the code. Test cases that cover more statements are given higher priority.
- **Branch Coverage:** In this approach, test cases are prioritized based on the number of branches (or decision points) covered in the code. Test cases that cover more branches are considered more important.
- **Path Coverage:** This technique considers the coverage of individual paths through the code. Each path represents a unique sequence of statements and branches. Test cases that cover more paths are given higher priority.
- **Mutation Score:** This approach involves introducing artificial faults (mutations) into the code and measuring the effectiveness of test cases in detecting these faults. Test cases with higher mutation scores are prioritized. We used the mutation score technique for prioritizing test cases.
- **Control Flow Graph (CFG) Analysis:** By constructing the control flow graph of the code, you can analyze the connectivity and complexity of the paths. Test cases that cover critical paths or paths with complex conditions can be prioritized.

2.6. Experimental Evaluation and Results of Literature Review

To evaluate the effectiveness of the proposed approach, we conducted extensive experimental studies, comparing our methodology with existing techniques in the domain of metamorphic testing for Python programs.

Chang-ai [28] work on test case generation based on metamorphic testing served as a benchmark for our comparison. The results of our experiments demonstrated significant improvements in various aspects, including coverage, fault detection, and prioritization when using Python as the primary tool for test case generation and prioritization.

In our comparative analysis, we observed that Python's capabilities, particularly it's path constraint solver and machine learning-based prioritization techniques, outperformed the traditional Java-based approach proposed by Chang-ai [28]. Python's simplicity and extensive libraries allowed for more efficient test case generation, covering a wider range of potential behaviors in the mathematical functions under test. Additionally, Python's machine learning-based prioritization facilitated smarter allocation of testing resources, leading to improved fault detection and overall testing efficiency.

This study also provided a comprehensive review of the challenges and opportunities in metamorphic testing. The comparison of our approach with existing techniques shed light on the limitations of traditional methods and highlighted the need for innovative approaches like the one presented in this literature review. The challenges in generating effective test cases and achieving comprehensive coverage were addressed through Python's path constraint solver, enabling more in-depth exploration of the input space and the creation of diverse test cases. Moreover, the opportunity to harness machine learning for prioritization allowed us to optimize the testing process and enhance the overall effectiveness of metamorphic testing.

2.7. Applications and Future Directions of Literature Review

The novel approach presented in this study holds significant potential for various domains and scales of software testing. Its application is not limited to specific systems but extends to both small and large-scale software. By leveraging Python's capabilities in constraint-solving and machine learning-based prioritization, our approach can effectively enhance the efficiency and effectiveness of metamorphic testing across a wide range of applications.

As the field of software testing continues to evolve, future research directions can explore several areas to further enhance the approach. One promising avenue is the integration of machine learning techniques into the test case generation and prioritization process. Machine learning algorithms can provide valuable insights into the relationships between input and output behaviors, thereby improving the generation of relevant and diverse test cases. This integration has the potential to optimize the testing process and achieve even higher fault detection rates.

Moreover, automation plays a crucial role in software testing to streamline the testing process and reduce human effort. Future research can focus on automating the test case generation process using our proposed approach. By developing tools that automate the generation of test cases and the application of metamorphic relations, testing teams can save time and resources, making metamorphic testing more feasible and practical for real-world software development projects.

CHAPTER 3: METHODOLOGY

Logically, test cases should be formulated to allow testing techniques, such as MT, to trigger a diverse range of distinct execution behaviors. This encompasses even those execution paths that might be considered difficult to access or less frequently traversed, often requiring significant efforts. The goal is to enhance the probability of identifying a broad spectrum of faults. To address this challenge efficiently, the Python constraint module offers a suitable solution. It provides straightforward and pure Python remedies to constraint satisfaction problems (CSPs) across a finite domain. It uses a range of potential inputs to resolve the program's constraints. The constraints are resolved to produce test cases that run the program. We suggest using the Python Constraint Solver to create source test cases for MT because of its ability to completely cover the input range.

Resources limit all software development activities. The basic objective of software testing is to find errors as many as possible within the constraints of time and money. Concisely, testing must be conducted in an economical method. Test cases often run in a specific sequence. Unfortunately, the test case execution order in all preceding investigations of MT was usually unsystematic; in other words, no systematic prioritization was used for MT's test cases. In this study, we provide a method for prioritizing source test cases so that those with higher priorities are executed first. The method prioritizes source test cases based on how well they contribute to the coverage of program paths and statements. The highest statement coverage-contributing source test cases are executed first. We specifically adopted path-directed prioritization for source test cases because, in general, test case execution paths reflect specific SUT functionalities. As a result, the more functionalities that are evaluated, the higher the likelihood that they will uncover potential flaws. In this work, we create simple prioritization approaches for source test cases since it is exceedingly difficult to regulate the pathways that follow-up test cases (whose creation depends on both source test cases and MRs) will cover.

The proposed solution[1] is made up of the suggested source test case creation and prioritization strategies, which are described in more detail in the next section.

[1] https://github.com/atif-imran-1/MSthesis

3.1. Proposed Approach

The general outline of our strategy (Figure 06) comprises the following phases:

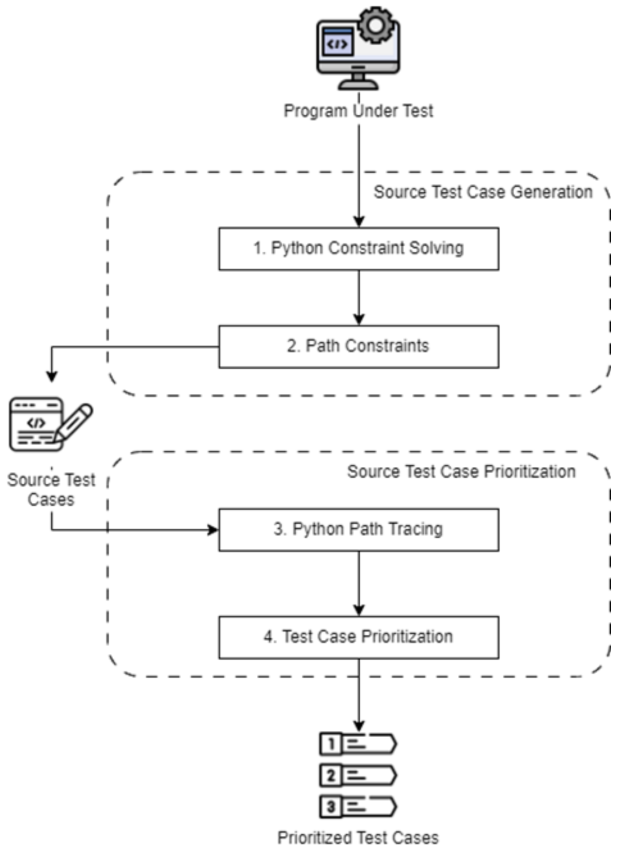

Program Under Test
Source Test Case Generation
1. Python Constraint Solving
2. Path Constraints
Source Test Cases
Source Test Case Prioritization
3. Python Path Tracing
4. Test Case Prioritization
Prioritized Test Cases

Figure 06: Proposed Approach - Detailed Flow Chart

1. ***Python Constraint Solving:*** Python constraint module checks the SUT for required constraints.
2. ***Path Constraints:*** Python constraint solver is employed to generate source test cases that satisfy the constraints obtained from the previous step.
3. ***Python Path Tracing:*** The series of executed statements for each source test case are examined to create the execution path.
4. ***Test Case Prioritization:*** The order of source test cases is determined by the path distance between them.

These steps will be categorized into two categories, namely: Step 1 and Step 2 as Source Test Case Generation; Step 3 and Step 4 as Source Test Case Prioritization.

3.1.1. Source Test Case Generation

The proposed approach revolves around a fundamental mathematical framework known as a Constraint Satisfaction Problem (CSP). The CSP provides a formal and systematic method to represent and resolve complex problems involving a collection of variables, their domains, and a set of constraints governing their interactions. In essence, the CSP aims to find suitable values for the variables or assignments that fulfill all the specified requirements and constraints. [44]

The key elements of a CSP are the variables, domains, and constraints. Variables act as placeholders for the entities for which we are seeking to determine values. Each variable is associated with a domain, which represents the range of potential values that can be assigned to it. The domains define the scope of acceptable assignments for all the variables in the problem. The constraints, on the other hand, play a critical role in setting the restrictions or prerequisites that the variable assignments must satisfy. These constraints describe the relationships, dependencies, or limitations between the variables, providing essential guidance for finding valid solutions.

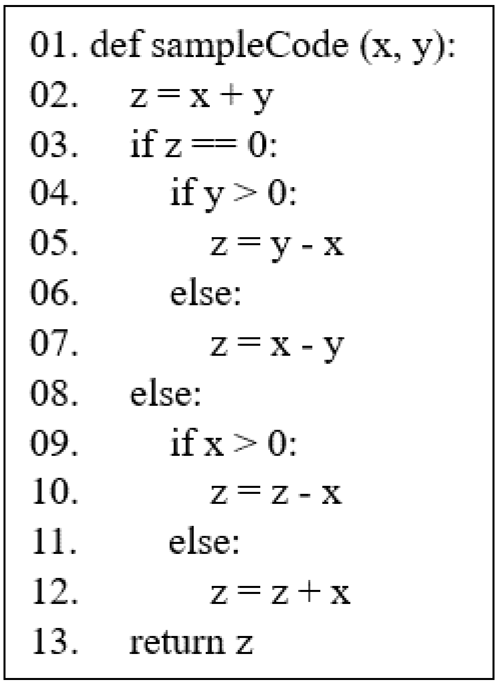
The goal of solving a CSP is to identify a variable assignment that simultaneously satisfies all the specified constraints while adhering to the predetermined requirements. This process often involves a meticulous analysis of the vast search space of potential assignments to eliminate inconsistent or undesirable choices. By thoroughly exploring and evaluating the potential solutions, we aim to derive an optimal assignment that meets all the necessary criteria.

In the proposed approach, we leverage the power of the Python Constraint module to address the Constraint Satisfaction Problems (CSPs) that arise during the testing process. The Python Constraint module is a comprehensive package that offers a wide range of features and tools specifically tailored for handling CSPs in Python. It facilitates the effective modeling and resolution of constraint-based issues, providing a user-friendly interface to define variables, domains, and constraints. [30] [58]

At the core of the Python Constraint module lies the concept of constraints, which represent conditions that must be met for the problem to be solved successfully. The module empowers us to create, manipulate, and manage these constraints with ease, providing a powerful means to express complex relationships and dependencies between variables.

Furthermore, the Python Constraint module takes on the critical task of finding solutions that satisfy all the constraints and requirements specified by the CSP. It employs a variety of constraint propagation strategies and search algorithms to efficiently navigate the vast solution space and identify feasible assignments for the variables. The ability to effectively explore the solution space and identify valid solutions is paramount to the success of the proposed approach, as it ensures that we can intelligently address the challenges posed by the CSPs and optimize the test case generation process.

In conclusion, the incorporation of the Constraint Satisfaction Problem and the Python Constraint module in this approach provides a robust and systematic foundation for generating effective test cases and optimizing the testing process. By leveraging the power of CSPs and harnessing the capabilities of the Python Constraint module, we aim to enhance the efficiency and effectiveness of test case generation, thereby improving the overall quality and reliability of software testing. The intelligent integration of this mathematical framework and Python-based tools offers a promising solution to the challenges posed by complex testing scenarios, opening new opportunities for advancing the field of software testing and metamorphic testing.



```
def sampleCode (x, y):
    z = x + y
    if z == 0:
        if y > 0:
            z = y - x
        else:
            z = x - y
    else:
        if x > 0:
            z = z - x
        else:
            z = z + x
    return z
```

Figure 07: SampleCode Function

Let's look at an example method called *SampleCode* (Figure 07) to demonstrate the suggested source test case generating process.

During the initial step (Figure 06) of our approach, the execution of *SampleCode* is facilitated utilizing the Python Constraint module, thereby generating a diverse range of outputs that adhere to the specified constraints of the problem being addressed.

In the subsequent step, step 2 (Figure 06), the constraints generated from the previous stage are consolidated into a comprehensive list, which serves as a foundation for subsequent processing and refinement of our approach. This compilation of constraints enables further analysis, manipulation, and exploration to enhance the effectiveness and efficiency of our methodology.

3.1.2. Source Test Case Prioritization

Before moving on, let's examine what the Python Trace Module is. The Python Trace module is a software library [29] that provides capabilities for tracing and analyzing the execution of Python programs. This module allows developers to gain insights into the runtime behavior of their code by generating detailed information about function calls, line execution, and program flow.

The primary purpose of the Python Trace module is to facilitate program analysis, debugging, and performance profiling. By using the trace module, developers can gather information about which functions are called, in what order, and how much time is spent on each function during program execution. This information can be crucial for identifying bottlenecks, detecting errors, and optimizing code performance.

The Python Trace module offers various tracing modes, including function-level and line-level tracing. Function-level tracing provides information about the calls made to different functions, while line-level tracing captures the execution of each line of code. Developers can choose the appropriate tracing mode based on their specific analysis needs.

In the present approach, we have leveraged the capabilities of line-level tracing as part of step three (Figure 06). By employing line-level tracing, we have been able to meticulously capture the sequence of executed statements, thereby obtaining a comprehensive path for a specific test case. This step is reiterated iteratively until the execution of all the test cases has been completed.

Throughout this process, line-level tracing facilitates a detailed examination of the program's runtime behavior. It enables the identification and logging of each statement executed within the codebase, ensuring an accurate representation of the path taken by a given test case. This fine-grained tracing mechanism empowers us to thoroughly analyze the program flow, comprehend the sequence of statements executed within the context of each test case, and gain a holistic understanding of the program's behavior. By systematically applying line-level tracing to every test case, we systematically construct a comprehensive collection of paths that correspond to the various scenarios covered by the test suite.

Moving on to Step Four (Figure 06) of the process, we undertake the task of systematically processing the list of paths that have been generated in the preceding step. In this phase, we assign priority to the paths based on their respective lengths, thus establishing an order of significance.

By prioritizing the paths according to their length, we aim to derive insights into the complexity and coverage of each path. The length of a path, in this context, refers to the number of executed statements within that specific path. Longer paths are generally indicative of more intricate program flows and potentially encompass a greater range of functionalities, thereby possessing heightened relevance for the analysis.

The act of prioritizing the paths enables us to focus our attention on the most extensive and comprehensive paths first, ensuring that we thoroughly examine the intricate portions of the program's execution. This approach allows for a systematic and organized exploration of the code, whereby the more intricate and lengthy paths are given precedence in the subsequent stages of analysis.

CHAPTER 4: EXPERIMENTAL STUDIES & RESULTS

To assess the effectiveness of our strategy, empirical experiments were done, including source test case development and prioritization. The settings of experimental research and the results are presented in this section.

4.1. Experimental Studies

In our study, we engage in practical experiments to assess the efficacy of our novel strategy for path-directed source test case generation and prioritization, implemented using Python. These experiments involve a comprehensive evaluation of our approach in comparison to established techniques. By subjecting it to various datasets and meticulously analyzing the results, our aim is to showcase the simplicity and efficiency of incorporating machine learning into path-solving processes. These empirical studies offer valuable insights into both the strengths and limitations of our approach, thereby contributing to the progression of metamorphic testing and the methods employed for calculating path constraints.

4.1.1. Research Questions

The present research delves into the integration of PaDMTP to heighten the efficacy of fault detection via the utilization of generated source test cases. By infusing machine learning algorithms and methodologies into the test case generation process, the intention is to amplify the identification of faults or defects within software systems. This study seeks to evaluate the superiority of our approach, bolstered by machine learning techniques, in contrast to prevailing methods of generating source test cases. The assessment will encompass fault detection capabilities and the overall effectiveness of testing. Through addressing these research queries, valuable insights can be garnered, subsequently elevating the efficiency and efficacy of software testing methodologies.

RQ1. How does the integration of machine learning techniques, specifically in Python, enhance the fault detection effectiveness of using the generated source test cases?

RQ2. To what extent does this approach outperform existing source test case generation techniques in terms of fault detection capabilities and overall testing effectiveness?

RQ3. What is the computational overhead associated with the implementation of this approach for generating source test cases, and how does it impact the efficiency and scalability of the testing process?

4.1.2. Object Programs

Four diverse object programs, each with its special qualities and importance in the context of our study, were carefully picked for our selection procedure. We have provided thorough information about these chosen object programs below:

1. **SampleCode:** It is a crucial element of our investigation. We took on the responsibility of translating this code into the Python programming language for the sake of our research and inquiry (Figure 06). This code was originally designed as an example program in the Java programming language [28]. The code snippet's *SampleCode* function is defined to take two parameters. Its major goal is to use these input values as inputs in a sequence of computations and conditional operations to determine the result.
2. **Highest Common Factor (HCF):** The Highest Common Factor (HCF) of two supplied integers may be determined using the Python program that we utilized. It stands for the biggest positive integer that can be used to divide the two input integers without producing a residual. The two integers are first accepted by the program through predefined input. This data is positive integers with non-zero values. The calculated HCF is shown as the calculation's outcome by the program.
3. **Least Common Multiple (LCM):** The Python method under consideration was created primarily to find the Least Common Multiple (LCM) between two provided values. The smallest positive integer that can be divided by both input integers without leaving a residual is represented by the LCM. The two integers are the first parameters that the function accepts. This data is positive integers with non-zero values. The function provides a dependable and effective method for calculating the LCM of the specified input values by returning the computed LCM as the output.
4. **Positive Difference Calculation Function (DIFF):** The under-consideration Python technique determines the positive difference between two supplied numbers. The function's initial two parameters are the two numbers x and y. These numbers may be zero, positive, or negative. It starts by determining if x is greater than y. If this is the case, the outcome is determined by deducting y from x. Conversely, the outcome is

calculated by deducting x from y if x is not bigger than y. The function returns the calculated outcome.

4.1.3. Generation of Test Cases

For all four object programs, we used generated source test cases as the initial step in our methodology. Using Python Constraint Solver [30], we produced all potential inputs within a given range for each object program, then tried to resolve the relevant route constraints for each one. We produced 141, 225, 225, and 156 source test cases for *SampleCode*, *HCF*, *LCM*, and *DIFF*, respectively, after removing the unsolvable route restrictions. Be aware that not all Metamorphic Relations (MRs) may be appropriate for all source test instances. Consequently, we initially identified a subset of suitable MRs for a certain source test case before creating follow-up test cases. The subsequent test cases can then be created using the chosen MRs.

4.1.4. Baseline Techniques

In this study, specific methodologies were selected based on the previous study [28], also known as baseline techniques, to serve as reference points for comparison in addressing our research questions. These baseline techniques have been chosen to provide a standard against which we can assess the performance and effectiveness of the methodologies under investigation.

The selected baseline techniques include Random Testing (RT) and Adaptive Random Testing (ART). These techniques will be used as benchmarks to evaluate the superiority and fault detection capabilities of the methodologies being studied. By comparing the results and outcomes of these baseline techniques with our proposed approach, we aim to gain valuable insights into the strengths and weaknesses of the different methodologies, thus contributing to a comprehensive understanding of their potential applications in software testing.

4.1.4.1. Random Testing (RT)

Random testing is a technique for creating and running test cases at random without following any systematic methodology. The underlying structure of the program under test is not considered because it is a "black box" testing approach. The basic goal of random testing is to examine various regions of the input space of the program by supplying random inputs and evaluating the results.

Typically, inputs are chosen from the input domain of the software being tested. A variety of methods, including random number generators and sampling from predetermined sets of values, can be used to provide unpredictability. To test programs, test cases from the input domain must be selected, executed, and the outcomes must then be compared to an oracle. Given that the size of the input domain is often arbitrarily huge, testing every conceivable input is not practical. As a result, only a (little) portion of the potential test cases may be assessed. Random testing is, perhaps, the simplest method for choosing test cases. [59] [60]

Random testing does not ensure that all aspects of the software are covered. Nevertheless, it can still be useful for investigating various regions of the input space, particularly when used in conjunction with other methods or under the direction of predetermined coverage criteria. It seeks to find software flaws or errors. It may disclose unexpected behaviors, border situations, or corner cases that may uncover flaws by offering a large variety of random inputs.

Based on various distributions, test instances are randomly selected from the input domain. A uniform distribution is frequently used for verification reasons to prevent biases. Numerical inputs make it simple to evenly select random test examples from an input domain. But when more complicated test case types are utilized, it is not always obvious how to achieve them.

Although selecting random inputs may appear simple, some issues must be resolved. Although these may be handled in practice, the ultimate effect is that uniform random testing is either impractical or not advised in a purely mathematical sense. In general, it would be preferable to choose a distribution that is as uniform as feasible. However, utilizing various forms of sample distribution is a sensible option when this is not feasible or when there is domain information that may be used to add bias in picking test cases that are probably to be more effective. [60]

4.1.4.2. Adaptive Random Testing (ART)

By dynamically modifying the distribution of test cases by the behavior of the program, the adaptive random testing (ART), software testing approach, seeks to increase the efficacy and efficiency of random testing. Contrary to conventional random testing, which chooses test cases evenly and at random, ART makes decisions about the next set of test cases based on input from the program's execution. [61]

The main concept underlying ART is to give priority to test cases that are more likely to find bugs or investigate untried program routes. This is accomplished by keeping an eye on how the program is running and obtaining data on the level of coverage attained and defects found. Based on this feedback, ART modifies the test case selection probability distribution, favoring uncharted territory or regions where faults are more likely to develop.

ART uses a variety of adaptive algorithms to change the test case selection on the fly. These approaches might be probabilistic ones, such as proportionate selection based on coverage or fault detection rates, or more complex ones that include program dependencies and past test case execution data. [62]

The advantages of ART include the capacity to deploy more efficient testing resources, concentrating on the areas of the program that are either less studied or more likely to have flaws. ART can raise the possibility of finding flaws early in the testing process and boost testing efficiency by adaptively modifying the test case selection.

It is crucial to remember that ART is not a panacea and cannot ensure the identification of every potential flaw. It still depends on the caliber of the test cases produced and is restricted by the limits of random testing. Additionally, the accuracy of the feedback systems and the suitability of the applied adaptive methods have a significant impact on ART's success. [63]

4.1.5. Evaluation Metrics

To compare PaDMTP with our baseline techniques for defect identification, we employed the following metrics:

- **Fault Detection Rate:** The trend of fault detection was demonstrated using the fault detection rate as the test case count increased. We chose the first k% of test cases from a complete test suite (TS) for each technique on every object program, and we calculated the ratio of the number of defects these test cases found to the overall number of faults. It makes sense to assume that the first k% of test cases will perform better with a greater fault detection rate. [28]
- **Mutation Score (MS):** It is defined as the ratio of the total number of non-equivalent mutants (or all faults) to the number of killed mutants (or exposed faults).

$$MS(P, TS) = \frac{N_K}{N_T - N_E}$$

In the above equation, P is the SUT, TS is a test suite, and N_K is the number of mutants that were killed (i.e., exposed) by the TS. N_T is the total number of mutants, and N_E is the number of non-equivalent (i.e., all faults) mutants. When an MR is broken during testing (that is, when an MR does not hold among the results of its associated metamorphic test group), it is said that a mutant has been killed. Generally, a testing technique is considered more effective when it has a higher value of MS. [28]

- **Prioritization Overhead (PO):** Prioritization overhead in terms of refers to the time and effort required for certain activities or processes within a project. In the context of test case generation, prioritization overhead specifically refers to the extra time needed to prioritize and order test cases based on certain criteria or priorities. A smaller prioritization overhead indicates a more efficient process with minimal time spent on prioritization compared to the overall test case generation time. [28]

4.2. Results

We review and share the results of our studies in this section. We present a thorough analysis of the data gathered and derive important conclusions from the findings. The emphasis is on evaluating the results, seeing trends, and making inferences based on the results that were seen. We hope to create a greater understanding of the experimental findings and their consequences by presenting and analyzing the data.

4.2.1. Fault Detection Effectiveness (RQ1)

To respond to RQ1, we used mutation testing to objectively assess how well our PaDMTP-generated test cases discover faults. The average Mutation Score (MS) for the four object programs varied from 70.0% to 84.6%, with a mean value of 76.8% (Table 2). In other words, on average, our method was able to identify approximately 77% of the flaws introduced by mutation analysis.

More significantly, MT relied on MRs rather than a test oracle to confirm test outcomes. If such flaws could not be represented by MR breaches, it is not that unexpected. According to earlier research [64], a modest collection of various MRs may be adequate on their own to identify most flaws that an oracle reveals. Our evaluation's findings suggest that there is still more to be done to identify suitable and varied MRs that can account for the majority of SUT functions and execution behaviors, as well as a wide range of defects.

Table 2: Mutation Scores of Object Programs using PaDMTP

Object Programs	**Mutation Score (MS)**
SampleCode	84.6%
HCF	70.0%
LCM	72.7%
DIFF	80.0%
Average	76.8%

4.2.2. Comparative Fault Detection (RQ2)

To address RQ2, we evaluated the average fault detection rates of the top k% test cases (k = 10, 20, ..., 100) generated by our proposed method (PaDMTP) in comparison to the baseline approaches (Random Testing and Adaptive Random Testing). By examining the fault detection rates at different levels of test case prioritization, we aim to determine the effectiveness and superiority of our approach over the baseline techniques. This analysis provides valuable insights into the fault detection capabilities and overall testing effectiveness of PaDMTP, allowing us to assess its performance in comparison to the traditional testing methods. The trend of the defect detection rate for each item program is shown in the figures below.

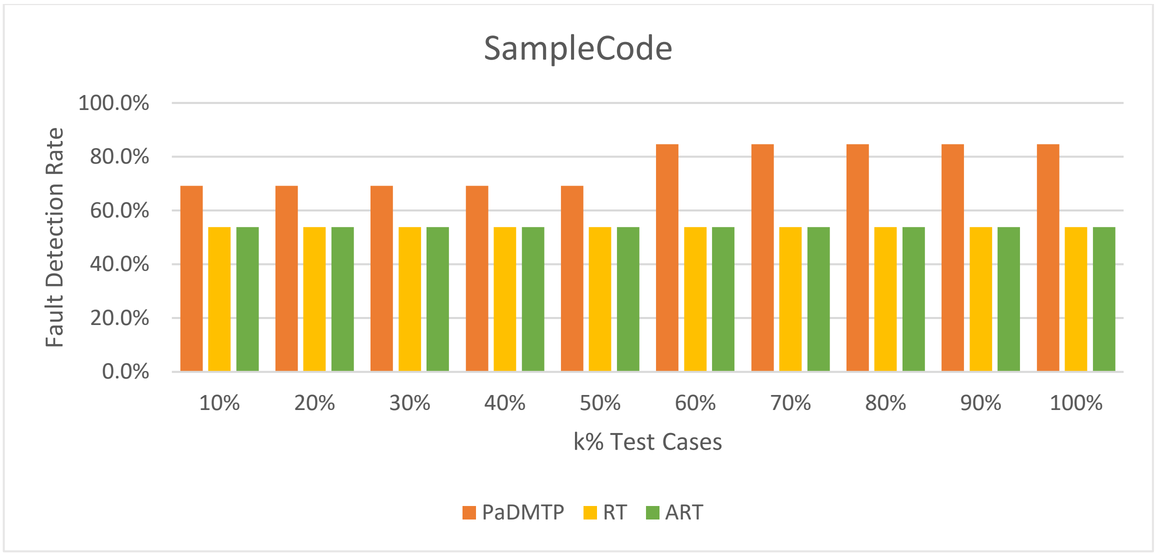


Figure 08: Fault Detection Rate of SampleCode Program

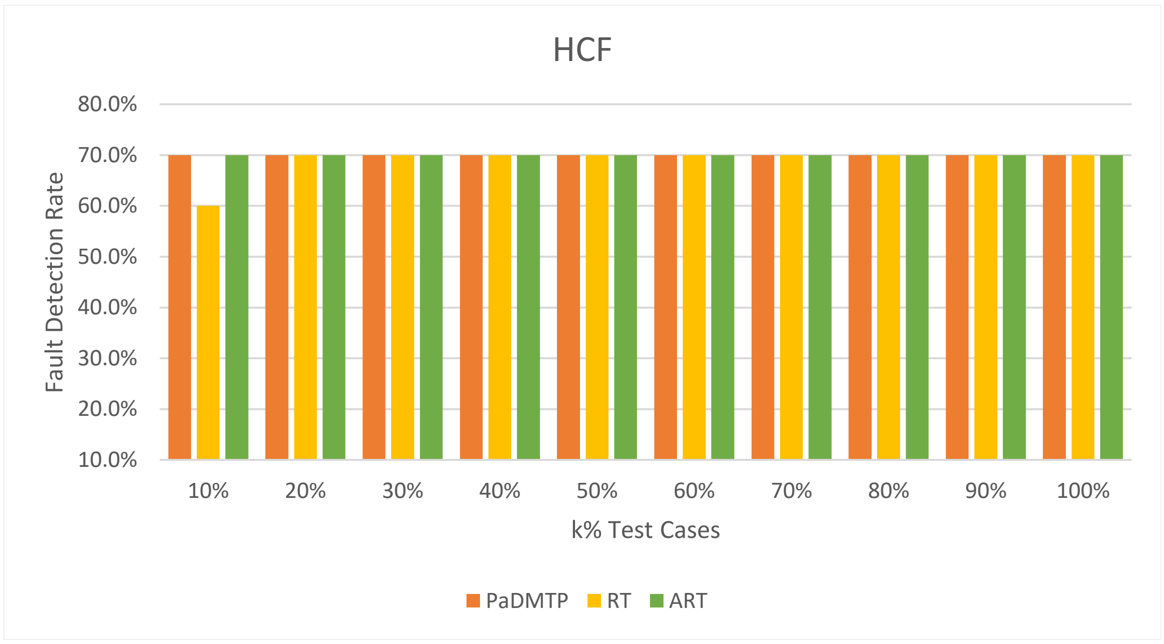


Figure 09: Fault Detection Rate of the HCF Program

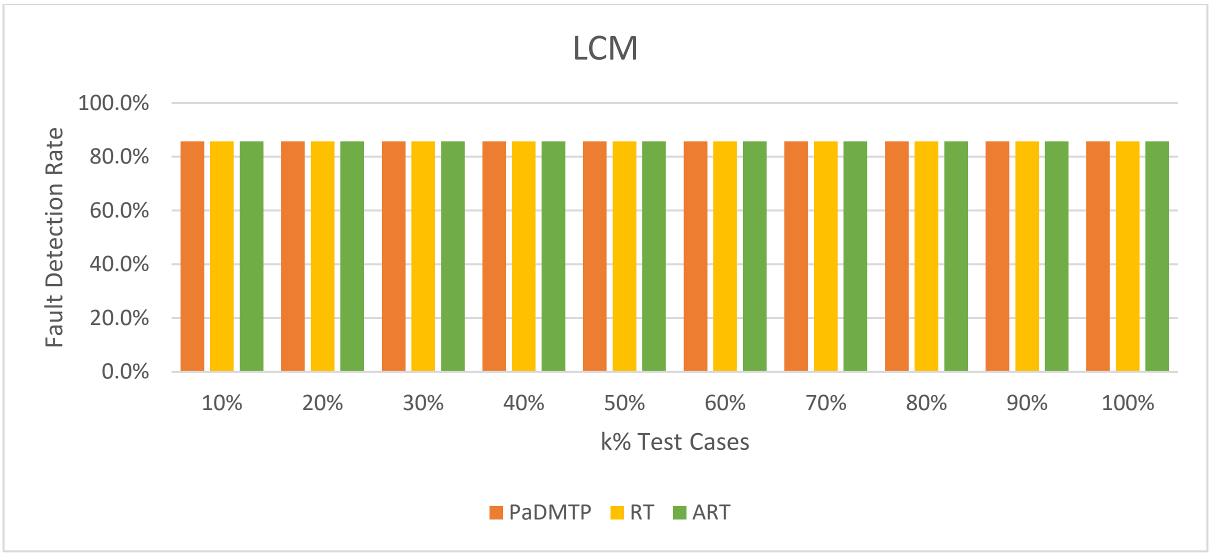


Figure 10: Fault Detection Rate of the LCM Program

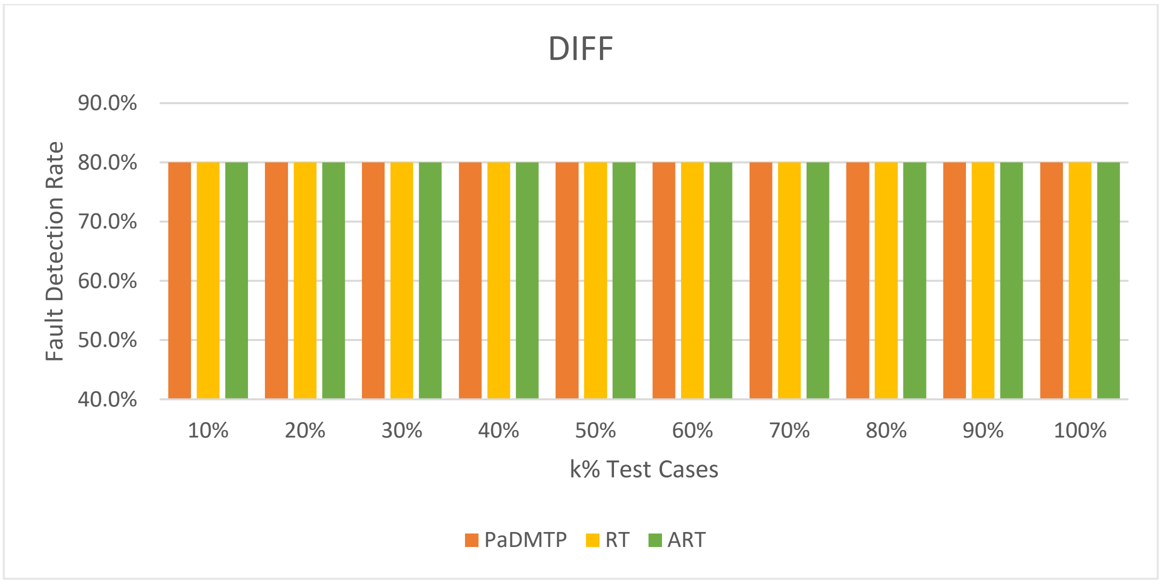
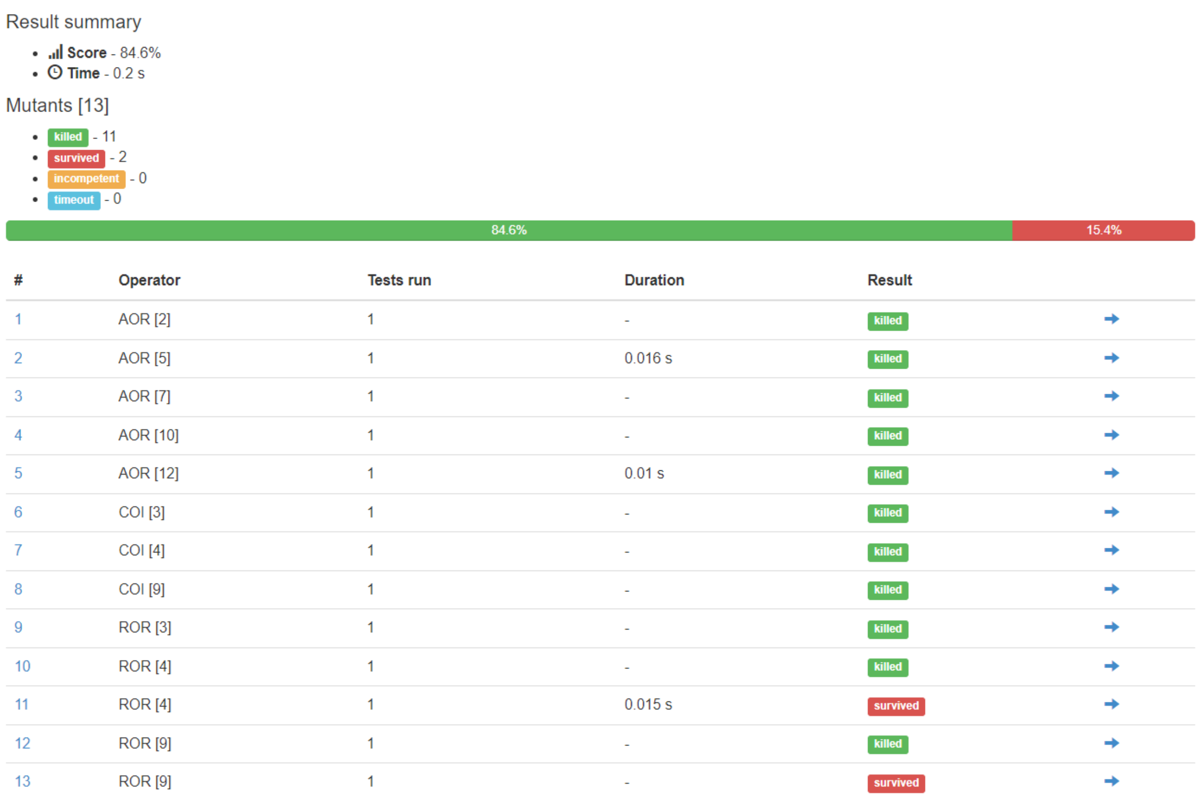


Figure 11: Fault Detection Rate of the DIFF Program



Result summary

- Score - 84.6%
- Time - 0.2 s

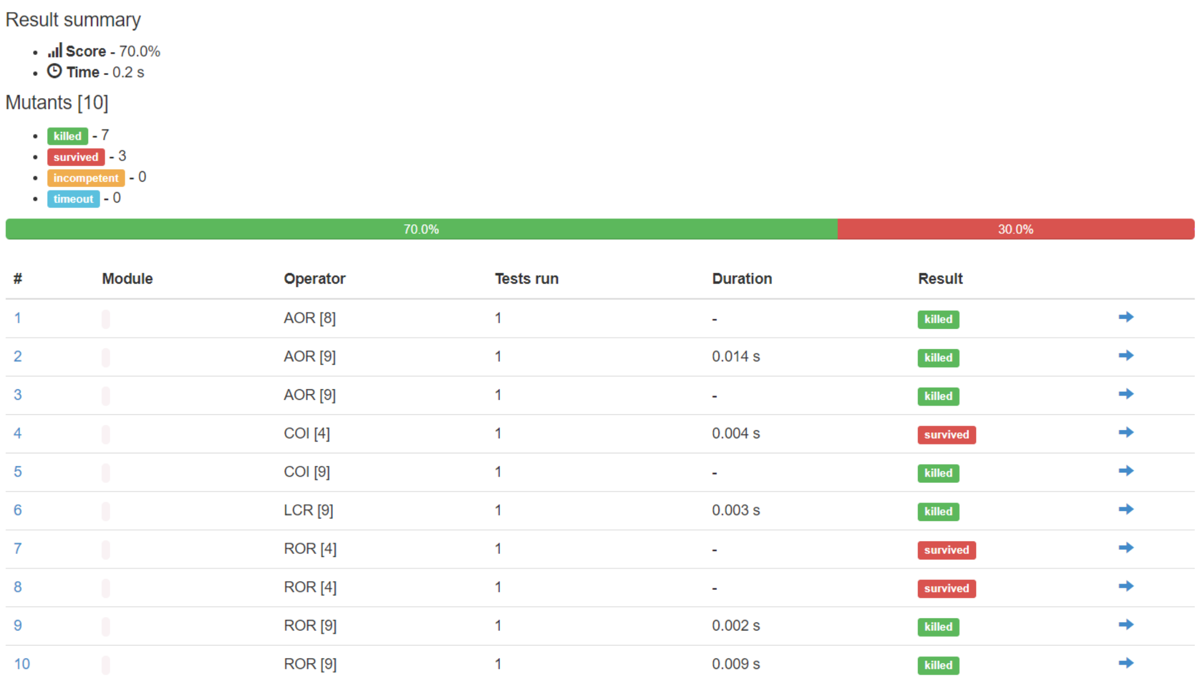
Mutants [13]

- killed - 11
- survived - 2
- incompetent - 0
- timeout - 0

84.6% 15.4%

#	Operator	Tests run	Duration	Result	
1	AOR [2]	1	-	killed	➡
2	AOR [5]	1	0.016 s	killed	➡
3	AOR [7]	1	-	killed	➡
4	AOR [10]	1	-	killed	➡
5	AOR [12]	1	0.01 s	killed	➡
6	COI [3]	1	-	killed	➡
7	COI [4]	1	-	killed	➡
8	COI [9]	1	-	killed	➡
9	ROR [3]	1	-	killed	➡
10	ROR [4]	1	-	killed	➡
11	ROR [4]	1	0.015 s	survived	➡
12	ROR [9]	1	-	killed	➡
13	ROR [9]	1	-	survived	➡

Figure 12: SampleCode PaDMTP Mutation Testing Result



Result summary

- Score - 70.0%
- Time - 0.2 s

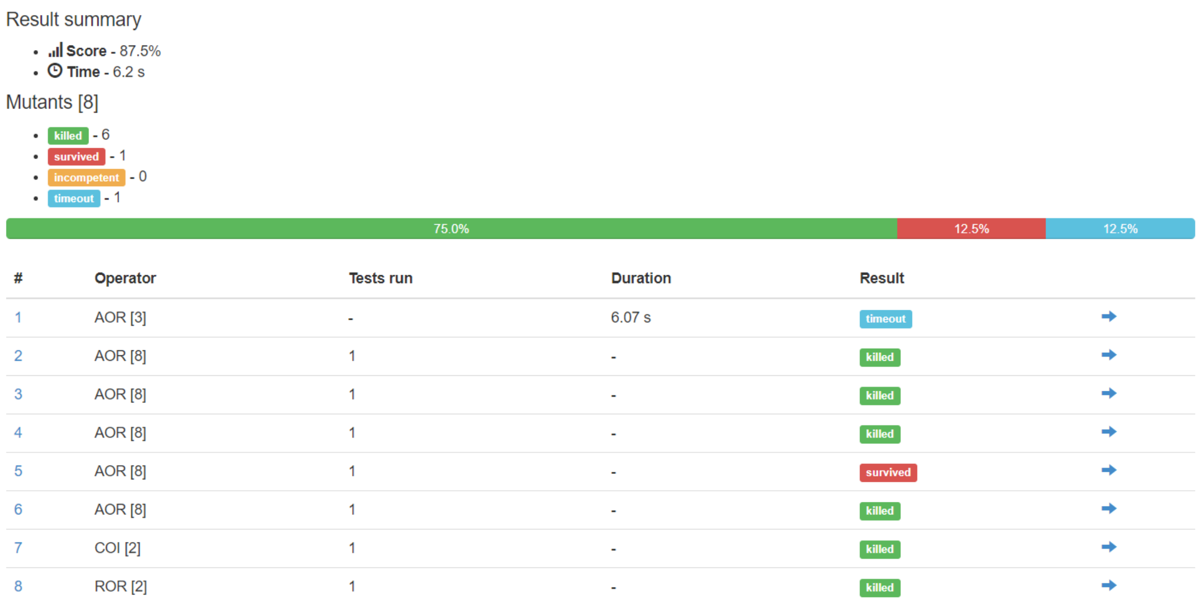
Mutants [10]

- killed - 7
- survived - 3
- incompetent - 0
- timeout - 0

70.0% 30.0%

#	Module	Operator	Tests run	Duration	Result	
1		AOR [8]	1	-	killed	➡
2		AOR [9]	1	0.014 s	killed	➡
3		AOR [9]	1	-	killed	➡
4		COI [4]	1	0.004 s	survived	➡
5		COI [9]	1	-	killed	➡
6		LCR [9]	1	0.003 s	killed	➡
7		ROR [4]	1	-	survived	➡
8		ROR [4]	1	-	survived	➡
9		ROR [9]	1	0.002 s	killed	➡
10		ROR [9]	1	0.009 s	killed	➡

Figure 13: HCF PaDMTP Mutation Testing Result


Result summary

- Score - 87.5%
- Time - 6.2 s

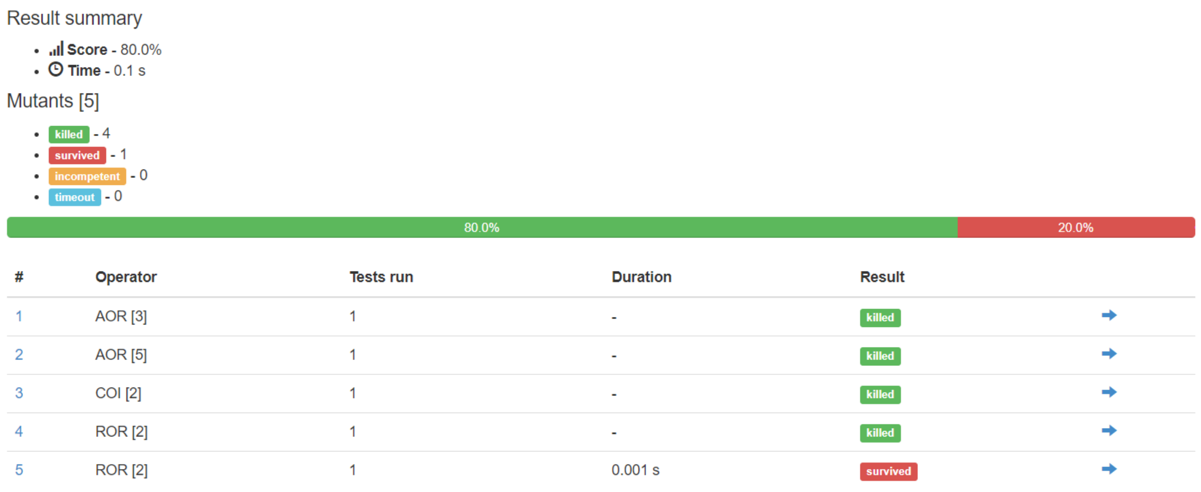
Mutants [8]

- killed - 6
- survived - 1
- incompetent - 0
- timeout - 1

75.0% 12.5% 12.5%

#	Operator	Tests run	Duration	Result	
1	AOR [3]	-	6.07 s	timeout	➡
2	AOR [8]	1	-	killed	➡
3	AOR [8]	1	-	killed	➡
4	AOR [8]	1	-	killed	➡
5	AOR [8]	1	-	survived	➡
6	AOR [8]	1	-	killed	➡
7	COI [2]	1	-	killed	➡
8	ROR [2]	1	-	killed	➡

Figure 14: LCM PaDMTP Mutation Testing Result


Result summary

- Score - 80.0%
- Time - 0.1 s

Mutants [5]

- killed - 4
- survived - 1
- incompetent - 0
- timeout - 0

80.0% 20.0%

#	Operator	Tests run	Duration	Result	
1	AOR [3]	1	-	killed	➡
2	AOR [5]	1	-	killed	➡
3	COI [2]	1	-	killed	➡
4	ROR [2]	1	-	killed	➡
5	ROR [2]	1	0.001 s	survived	➡

Figure 15: Positive Difference PaDMTP Mutation Testing Result

In the context of our first program, *SampleCode*, the PaDMTP technique demonstrated the highest performance in terms of fault detection, outperforming the other techniques. The following positions were occupied by a tie between the RT and ART techniques, indicating comparable fault detection capabilities.

Moving to the second program, *HCF*, the ART technique exhibited superior fault detection performance, securing the top position. It was closely followed by a tie between the PaDMTP and RT techniques, indicating similar effectiveness in fault detection for this program.

For the third and fourth programs, *LCM* and *DIFF*, respectively, all three techniques (PaDMTP, RT, and ART) demonstrated comparable fault detection performance. No technique stood out as superior, resulting in a tie among them.

These findings highlight the varying effectiveness of fault detection techniques across different programs. The results indicate that the performance ranking of the techniques is program-dependent, with each technique showcasing its strengths and limitations in different contexts.

The tables below (Table 3 and Table 4) present the outcomes of T-Tests conducted to compare the fault detection effectiveness of two different techniques: PaDMTP (proposed approach) and RT (Random Testing). Each row in the tables corresponds to a distinct object program, while the columns display the t-statistic, p-value, and effect size for each comparison.

The t-statistic signifies the difference between the means of the fault detection rates for the two techniques. A higher t-statistic indicates a larger disparity in effectiveness. On the other hand, the p-value assesses the likelihood of observing such a difference due to random chance. Smaller p-values suggest a statistically significant difference between the techniques, while larger p-values indicate a lack of statistical significance.

The effect size provides a measure of the magnitude of the difference between the means of the fault detection rates. A higher effect size implies a more substantial difference and indicates the practical significance of the statistical disparities found by the T-Test.

Table 3: T-Test values of Object Programs (PaDMTP vs RT)

Object Program	**PaDMTP vs RT**		
	t-stat	**p-value**	**Effect Size**
SampleCode	9.000	8.538×10^{-06}	42.426
HCF	0.042	9.677×10^{-01}	0.205
LCM	0.892	3.958×10^{-01}	4.639
DIFF	2.106	6.452×10^{-02}	5.346

Upon analyzing the results, we find that for the object program *SampleCode*, PaDMTP demonstrates superior performance over RT with a significantly higher t-statistic and a very low p-value. The effect size further supports the practical significance of this difference, indicating a substantial advantage for PaDMTP.

Table 4: T-Test values of Object Programs (PaDMTP vs ART)

Object Program	**PaDMTP vs ART**		
	t-stat	**p-value**	**Effect Size**
SampleCode	9.000	8.538×10^{-06}	11.767
HCF	0.795	4.472×10^{-01}	3.797
LCM	1.125	2.899×10^{-01}	5.282
DIFF	1.685	1.263×10^{-01}	7.639

However, for the object program *HCF*, the T-Test results suggest no significant difference between the fault detection rates of PaDMTP and RT. The effect size further confirms that any difference, if present, is minimal.

For the object programs *LCM* and *DIFF*, the T-Test results also indicate no significant difference between PaDMTP and RT. However, the effect sizes suggest that PaDMTP may have a slightly higher fault detection effectiveness, though the practical significance may not be as pronounced.

Similarly, the comparison between PaDMTP and ART follows a comparable pattern. In most cases, PaDMTP exhibits a higher t-statistic, a lower p-value, and a larger effect size, signifying its superiority in fault detection effectiveness compared to ART.

In conclusion, the T-Test outcomes indicate that PaDMTP generally outperforms both RT and ART in terms of fault detection effectiveness across the tested object programs. The effect sizes further reinforce the practical significance of the observed differences, highlighting the superiority of PaDMTP in detecting faults in these programs.

4.2.3. Computational Overhead (RQ3)

In response to RQ3, we analyzed the average time required for generating a fixed number of test cases using PaDMTP, RT, and ART across four object programs: *SampleCode*, *HCF*, *LCM*, and *DIFF*. The number of source test cases generated for each program was 141, 225, 225, and 156, respectively. The results, presented in Table 5, show that the prioritization overhead ranges from 0.008s to 0.063s. Notably, this indicates that the time spent on source test case generation is negligibly small compared to the time required for source test case prioritization. PaDMTP incurred considerably lower overhead than RT and ART, highlighting its efficiency in terms of time cost for source test case generation. Additionally, we observed that the source test case generation time for RT was consistently higher than that of PaDMTP and ART, further demonstrating the potential benefits and effectiveness of PaDMTP for source test case generation.

Table 5: Overhead of Source Test Case Generation

Object Program	**Overhead (s)**		
	PaDMTP	**RT**	**ART**
SampleCode	0.008	0.022	0.022
HCF	0.015	0.033	0.020
LCM	0.016	0.063	0.033
DIFF	0.015	0.036	0.020

4.3. Threats to Validity

The threats to the validity of our study are outlined as follows:

- **Correctness of Implementation:** Our approach involved integrating various open-source packages to support the main steps of our method, including Python constraint solver and Python path tracer. These packages are widely used and regularly updated, which ensures the reliability of our implementation.
- **Representativeness of Object Programs:** To enhance the validity of our experimental results, it would have been beneficial to include more complex object programs. However, the selection of object programs was based on availability and the number of MRs for conducting experiments. We diversified the sources of object programs to mitigate the impact of this threat on the experimental outcomes. Although our experiment did not involve large-size programs with millions of lines of code, we believe that our approach can be extended and applied to larger real-world subjects.
- **Selection of Baseline Techniques:** For the comparison of source test case generation, we used two baseline techniques that are commonly employed in the field of Metamorphic Testing (MT). However, since there were no existing prioritization techniques specifically designed for source test cases in MT, we compared our approach

with random prioritization. Future research could explore the application of existing prioritization techniques from the general context of software testing into MT.

- **Representativeness of Evaluation Metrics:** The evaluation metrics used in our experiments, such as mutation score, have been extensively utilized in previous studies to assess the fault detection effectiveness of testing techniques. These metrics are well-established and widely accepted in the field.

By acknowledging these potential threats to the validity of our study, we aim to ensure the credibility and reliability of our experimental findings and encourage further research around Metamorphic Testing.

CHAPTER 5: CONCLUSION

Metamorphic testing (MT) has emerged as a highly effective technique in the field of software testing. It not only addresses the challenging oracle problem by utilizing metamorphic relations (MRs) but also complements traditional testing methods by generating unique test cases. As software systems become more complex, the generation of effective source test cases has gained significant attention in research. In this study, we present a novel and path-directed approach for source test case generation using the power of Python. By leveraging Python's path tracer and constraint solver, our method obtains program path constraints, enabling the creation of source test cases that achieve extensive coverage of execution paths and significantly enhance the effectiveness of fault detection. The integration of Python's path tracer and constraint solver provides us with a potent and versatile toolset for tackling complex testing challenges, making our approach well-suited for real-world software testing scenarios.

Our proposed approach utilizes the Python constraint solver to efficiently handle complex mathematical equations, inequalities, and constraints associated with the software under test. By methodically exploring the vast input space, our test case generation algorithm considers various combinations and ranges of inputs, ensuring a thorough assessment of the system's potential behaviors. The resulting test cases mimic real-world scenarios and encompass a diverse range of potential behaviors, making them highly effective tools for identifying potential errors with a high degree of accuracy and precision. The systematic and comprehensive nature of our algorithm ensures thorough coverage of the behaviors exhibited by the software, making it a powerful instrument for revealing any underlying issues or discrepancies.

In addition to the source test case generation, we propose a sophisticated test case prioritization technique to optimize resource allocation and improve fault detection efficiency. By assigning higher priority to test cases that have a higher likelihood of revealing faults, our prioritization strategy ensures that critical areas of the software are thoroughly tested, considering input constraints, and maximizing the potential for fault detection. The adaptive and flexible nature of our technique allows us to handle various complexities of the software and its input spaces, providing a dynamic approach to fault detection. The integration of our prioritization technique enhances the reliability and precision of the test results, allowing us to identify any deviations or faults within the software with a high degree of accuracy. Through the systematic application of our approach, we gain a comprehensive and detailed understanding of the software's behavior, further elevating the effectiveness and efficiency of our testing methodology.

In the experimental evaluations, we conducted extensive studies on four representative programs to demonstrate the exceptional performance of our proposed techniques. The results showcased the superiority of our approach in fault detection effectiveness, outperforming traditional testing methods in multiple scenarios. Our method not only effectively addressed the oracle problem through the utilization of MRs but also showcased its ability to generate unique and diverse test cases that explore different execution paths. The integration of Python's path tracer and constraint solver provided us with a competitive advantage in generating source test cases with extensive coverage and enhanced fault detection capabilities. These findings highlight the potential of our approach in significantly improving software testing effectiveness and provide valuable insights for further advancements in the field of metamorphic testing.

5.1. Results Discussion

In this section, we present a comprehensive analysis of the experimental findings, addressing the research questions and deriving significant conclusions. Firstly, in response to RQ1, we evaluated the fault detection effectiveness of our PaDMTP-generated test cases using mutation testing. The average Mutation Score (MS) across the four object programs ranged from 70.0% to 84.6%, with a mean value of 76.8%. This demonstrates that, on average, our method identified approximately 77% of the faults introduced by mutation analysis. The reliance on Metamorphic Relations (MRs) for test outcomes may limit the detection of certain flaws that cannot be represented by MR violations, emphasizing the need for a more diverse and appropriate MRs to cover a wider range of software behaviors and defects.

In addressing RQ2, we compared the fault detection rates of the top k% test cases generated by PaDMTP with those of Random Testing (RT) and Adaptive Random Testing (ART). We observed varying effectiveness among the techniques for different object programs. For the *SampleCode* program, PaDMTP exhibited superior fault detection performance over RT and ART. In contrast, for the *HCF* program, ART outperformed PaDMTP and RT. For the *LCM* and *DIFF* programs, all three techniques showed comparable fault detection performance, resulting in a tie. The T-Test results confirmed that PaDMTP generally outperformed RT and ART in terms of fault detection effectiveness, with substantial practical significance in some cases.

Regarding RQ3, we analyzed the computational overhead associated with test case generation using PaDMTP, RT, and ART. The results demonstrated that the prioritization overhead for PaDMTP was considerably lower than that of RT and ART. The time spent on source test case generation was minimal compared to the time required for prioritization. Additionally, the source test case generation time for RT was consistently higher than that of PaDMTP and ART, further highlighting the efficiency of PaDMTP for source test case generation.

In summary, our PaDMTP approach showed promising results in fault detection effectiveness, outperforming traditional testing techniques in some cases. The reliance on MRs may limit the detection of certain flaws, necessitating further research into diverse and comprehensive MRs. The computational overhead analysis revealed PaDMTP's efficiency in generating source test cases, making it a valuable and effective approach for software testing.

5.2. Future Work

As we look towards future investigations, one of the primary objectives is to assess the performance of our approach in real-world scenarios by applying it to industrial large-size programs. Conducting more extensive and in-depth empirical studies on a broader range of software systems will be crucial to further evaluate the effectiveness and practicality of the proposed path-directed technique. By testing our approach on complex and industrial-scale programs, we can gain valuable insights into its scalability, adaptability, and fault detection capabilities, enabling us to better understand its potential benefits and limitations in real-world software testing contexts.

While the current study focused on using path constraints for source test case generation, there is an exciting avenue for further research in exploring the applicability of this concept in constructing follow-up test cases. Follow-up test cases are essential for evaluating the stability and robustness of the software under various scenarios and potential user interactions. By investigating how path constraints can be utilized to generate follow-up test cases that cover a wide range of execution paths and behaviors, we can extend the scope of our approach and enhance its effectiveness in identifying and addressing potential faults and defects.

Furthermore, we aim to delve deeper into the potential enhancement of fault detection efficiency achieved by different prioritization strategies in comparison to random prioritization. By analyzing and comparing various prioritization techniques, we can gain insights into how different strategies impact fault detection rates, testing efficiency, and overall software reliability. Identifying the most effective and efficient prioritization strategy will significantly contribute to the practical application of our approach in real-world testing scenarios, where resource allocation and time constraints are critical factors.

In conclusion, our future research directions aim to advance the understanding and application of our approach in software testing. By conducting more comprehensive and rigorous empirical studies, exploring the applicability of path constraints in generating follow-up test cases, and investigating different prioritization strategies, we seek to contribute to the field of Metamorphic Testing and drive forward the evolution of effective and efficient software testing methodologies. The insights gained from these future investigations will help us further refine and optimize our approach, making it a valuable and practical tool for ensuring the reliability, quality, and effectiveness of software systems.

REFERENCES

[1] E. T. Barr, M. Harman, P. McMinn and M. Shahbaz, "The Oracle Problem in Software Testing: A Survey," *IEEE Transactions on Software Engineering,* 2014.

[2] . C. Zhang and M. Pezze, "Automated test oracles: A survey," in Advances in Computers," *A. Memon, Ed. Waltham, MA: Academic press,* vol. 95, pp. 1-48, 2014.

[3] H. R. M. and . K. Patel, "A mapping study on testing non-testable system," *Softw. Quality J.,* vol. 26, no. 4, p. 1373–1413, 2018.

[4] T. Y. Chen, S. C. Cheung and S. M. Yiu., "Metamorphic testing: a new approach for generating next test cases.," DepartmentofComputerScience, HongKongUniversityofScienceandTechnology, HongKong, 1998.

[5] N. Wang, T. s. Y. Chen, Z. Zheng and P. Rao, "Impacts of test suite's class imbalance on spectrum-base dfault localization techniques," *13th International Conference on Quality Software (QSIC'13). IEEE Computer Society,* pp. 260-267, 2013.

[6] Z. Q. Zhou, T. Tse, F.-C. Kuo and T. Y. Chen, "Metamorphic testing and beyond," *11th Annu. Int. Workshop Softw.Technol. Eng. Practice,* p. 94–100, 2003.

[7] T. H. Tse, "Research directions on model-based metamorphic testing and verification," *in Proc. 29th Annu. Int. Comput. Softw.Appl.,* vol. 1, p. 332, 2005.

[8] T. Y. Chen, "Metamorphic testing: A simple method for alleviating the test oracle problem," " in Proc. 10th Int. Workshop Autom. software.test, 2015.

[9] Z. Q. Zhou, D. T. owey, F.-C. Kuo and T. Y. Chen, "Metamorphic testing: Applications and integration with other methods," *12th Int. Conf. Quality Softw.,* pp. 285-288, 2012.

[10] S. Huang and Z. Hui, "Achievements and challenges of meta," *in Proc. 4th World Congr. Softw. Eng.,* pp. 73-77, 2013.

[11] J. M. Bieman and . U. Kanewala, "Techniques for testing scientific programs without an oracle," in *5th International Workshop on Software Engineering for Computational Science and Engineering (SE-CSE)*, San Francisco, CA, USA, 2013.

[12] S. Beydeda, "Self metamorphic testing components," *30th Annual International Computer Software and Applications Conference .IEEE Computer Society,* vol. 1, pp. 265-272, 2006.

[13] T. Tse, L. Heng, T. Y. Chen and W. K. Chan, "Integration testing of context-sensitive middle ware-based application: a metamorphic approach," *International Journal of Software Engineering and Knowledge Engineering,* pp. 677-703, 2006.

[14] K. R. Leung, S. C. Cheung and W. K. Chan, "A metamorphic testing approach for online testing of service-oriented software application," *International journal of web research,* vol. 4, no. 2, pp. 60-80, 2007.

[15] T. Y. Chen, J. W. Ho, H. Liu and X. Xie, "An innovative approach for testing bioinformatics programs using metamorphic testing," *BMC Bioinformatics,* 2009.

[16] W. Susilo, W. j. Ma, F.-C. Kuo and T. s. Y. Chen, "Metamorphic testing for cyber security," vol. 6, no. 49, pp. 48-55, 2016.

[17] . Z. Su., V. Le and M. Afshari, "Compiler validation via equivalence modulo inputs," *Compiler validation via equivalence modulo inputs,* vol. 49.6, no. ACM, pp. 216-226, 2014.

[18] A. F. Donaldson, N. Chong, A. Lascu and C. Lidbury, "Many-core compiler fuzzing," *In Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15).,* pp. 65-76, 2015.

[19] M. Lindvall , "Metamorphic Model-Based Testing Applied on NASA DAT–An Experience Report," *37th IEEE,* vol. 2, pp. 129-138, 2015.

[20] S. Sergio , . R. M. Hierons, D. Benavides and A. Ruiz-Cortés, "Automated metamorphic testing on the analyses of feature models," *Information and Software Technology,* vol. 53, no. 3, pp. 245-258, 2011.

[21] X. Xie, J. W. K. Ho, C. Murphy, G. Kaiser, B. Xu and T. Y. Chen, "Testing and validating machine learning classifiers by metamorphic testing," *Journal of Systems and Software,* vol. 84, no. 4, pp. 554-558, 2011.

[22] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo and T. Y. Chen, "Automated functional testing of online seaech services," *Journal of Software: Testing, Verification and Reliability,* vol. 22, no. 4, pp. 221-243, 2012.

[23] Z. Q. Zhou, T. Tse and T. Y. Chen, "Semi-proving: An integrated method based on global symbolic evaluation and metamorphic testin," *ACM SIGSOFT International Symposium on Software Testing and Analysis,* vol. 2, pp. 191-195, 2002.

[24] T. Y. Chen, Z. Q. Zhou. and T. se, "Semi-proving: An integrated method for program proving,testing, and debugging," *IEEE,* vol. 37, no. 1, pp. 109-125, 2011.

[25] D. Towey, F.-C. Kuo, T. Y. Chen and M. Jiang, "A metamorphic testing approach for supporting program repair without need for a test oracle," *journal of syste software,* pp. 127-140, 2017.

[26] X. o. Ma, C. Xu, N. Liu, Y. Jiang and H. Jin, "Concolic metamorphic debugging," *IEEE 39th annual International proceeding,* vol. 2, 2015.

[27] T. Y. Chen., S. Xiang and Z. Q. Zhou, "Metamorphic testing for software quality assessment:A study of search engines.," *IEEE Transactions on Software Engineering,* pp. 264-284, 2016.

[28] C.-a. Sun a, B. Liu a and Y. Li, "Path-directed source test case generation and prioritization in metamorphic testing," *The Journal of Systems & Software,* 2022.

[29] Anonymous, "trace - Trace or Track Python Statement Execution," [Online]. Available: https://docs.python.org/3/library/trace.html. [Accessed 16 May 2023].

[30] G. Niemeyer, "Python-Constraint 1.4.0," 5 November 2018. [Online]. Available: https://pypi.org/project/python-constraint/. [Accessed July 2023].

[31] K. Hałas and P. Hossner, "MutPy," 17 November 2019. [Online]. Available: https://github.com/mutpy/mutpy. [Accessed 20 July 2023].

[32] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse and Z. Q. Zhou, "Metamorphic Testing: A Review of Challenges and Opportunities," *ACM Computing Surveys,* vol. 51, no. 1, pp. 1-27, 2014.

[33] B. Sébastien, . B. Bernard and D. Fré, "Constraint-Based Software Testing," 2009.

[34] M. Ehmer Khan , "Different Forms of Software Testing Techniques for Finding Errors," *IJCSI International Journal of Computer Science Issues,* vol. 7, no. 3, 2010.

[35] D. Huizinga and A. Kolawa, Automated defect prevention: best practices in software management, John Wiley & Sons, 2007.

[36] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," 2002.

[37] B. L. and D. D., " Successful IT projects Thomson Learning," in *Middesex University Press*, 2007.

[38] K. R. and Linberg, "Software Developer Perceptions About Software Project Failure: A Case Study," *Journal of Systems and Software,* pp. 177-192, 1999.

[39] U. Kanewala and J. M. Bieman, "Using machine learning techniques to detect metamorphic relations for programs without test oracles," in *IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[40] Y. Tian et al. , "DeepTest: Automated testing of deep-neuralnetwork-driven autonomous cars," *arXiv preprint arXiv:1708.08559,* 2017.

[41] C. Murphy et al., "Properties of Machine Learning Applications for Use in Metamorphic Testing," *SEKE,* vol. 8, pp. 867-872, 2008.

[42] H. Zhang, L. Liu and P. Zhang, "Predicting Metamorphic Relations Based on Path Features," *Journal of Physics: Conference Series,* 2020.

[43] C. Murphy, K. Shen and G. Kaiser, "Automatic System Testing of Programs without Test Oracles," *Dept. of Computer Science Columbia University New York,* 2009.

[44] S. C. Brailsford, C. N. Potts and B. M. Smith, "Constraint satisfaction problems: Algorithms and applications," *European Journal of Operational Research,* vol. 119, no. 3, pp. 557-581, 1999.

[45] U. Kanewala and J. M. Bieman, "Testing scientific software: A systematic literature review," *Information and Software Technology,* vol. 56, no. 10, pp. 1219-1232, 2014.

[46] M. Srinivasan, M. P. Shahri, I. Kahanda and U. Kanewala, "Quality Assurance of Bioinformatics Software : A Case Study of Testing a Biomedical Text Processing Tool Using Metamorphic Testing," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, 2018.

[47] H. Stallbaum, A. Metzger and K. Pohl, "An automated technique for risk-based test case generation and prioritization," *3rd international workshop on Automation of software test,* pp. 67-70, 2008.

[48] Y. Qi, Y. Lei and X. Mao, "Efficient Automated Program Repair through Fault-Recorded Testing Prioritization," in *IEEE International Conference on Software Maintenance*, China, 2013.

[49] Y.-H. Tung, S.-S. Tseng and T.-J. L, "A Novel Approach to Automatic Test Case Generation for Web Applications," in *10th International Conference on Quality Software*, ROC, 2010.

[50] D. Corradini, A. Zampieri, M. Pasqua and . M. Ceccato, "Empirical Comparison of Black-box Test Case Generation Tools for RESTful APIs," in *IEEE 21st International Working Conference on Source Code Analysis and Manipulation (SCAM)*, University of Verona – Verona, Italy, 2021.

[51] W. Wang, M. Kessentini and w. Jiang, "Test Cases Generation for Model Transformations from Structural Information from Structural Information," 2013.

[52] F. Jafari , A. Nadeem and Q. u. Zaman, "Evaluation of Metamorphic Testing for Edge Detection in MRI Brain Diagnostics," *Department of Computer Science, Capital University of Science and Technology, Islamabad 44000, Pakistan,* 2022.

[53] U. Kanewala and K. Rahman, "Predicting Metamorphic Relation for Matrix Calculation Programs," in *Proceedings of the 3rd International Workshop on Metamorphic Testing*, 2018.

[54] S. Biswas, A. Bansal, R. Mall and P. Mitra, "Fault-Based Regression Test Case Prioritization," in *IEEE Transactions on Reliability*, 2022.

[55] D. Gläser, T. Koch, S. Peters, S. Marcus and B. Flemisch, "fieldcompare: A Python package for regression testing simulation results," *The journal of open source software,* vol. 8, 2023.

[56] S. Elbaum, j. Penix and G. Rothermel, "Techniques for Improving Regression Testing in Continuous Integration Development Environments," 2014.

[57] M. Srinivasan, "Prioritization of Metamorphic Relations Based on Test Case Execution Properties," in *IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2018.

[58] A. Griffis and S. Celles, "Python-Constraint: Constraint Solving Problem resolver for Python," 10 April 2019. [Online]. Available: https://github.com/python-constraint/python-constraint. [Accessed 16 May 2023].

[59] R. Hamlet, "Random Testing," in *Encyclopedia of Software Engineering*, John Wiley & Sons, Ltd, 2002.

[60] A. Arcuri, M. Z. Iqbal and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Software Engineering,* vol. 38, no. 2, pp. 258-277, 2012.

[61] T. Y. Chen, H. Leung and I. K. Mak, "Adaptive Random Testing," in *Advances in Computer Science - ASIAN 2004. Higher-Level Decision Making*, Berlin, Heidelberg, 2004.

[62] T. Y. Chen, F.-C. Kuo, R. G. Merkel and T. H. Tse, "Adaptive Random Testing: The ART of test case diversity," *Journal of Systems and Software,* vol. 83, no. 1, pp. 60-66, 2010.

[63] R. Huang, W. Sun, Y. Xu, H. Chen, D. Towey and X. Xia, "A Survey on Adaptive Random Testing," *IEEE Transactions on Software Engineering,* vol. 47, no. 10, pp. 2052-2083, 2019.

[64] Anonymous, "Path Constraints," [Online]. Available: https://jckantor.github.io/CBE30338/07.08-Path-Constraints.html.

[65] H. Liu, F.-C. Kuo, D. Towey and T. Y. Chen, "How Effectively Does Metamorphic Testing Alleviate the Oracle Problem?," *IEEE Transactions on Software Engineering,* vol. 40, no. 1, pp. 4-22, 2014.