Middle East Technical University
Informatics Institute

PathFinder
An Intelligent Algorithm for MCDC Test-Path Generation

Advisor Name: ALTAN KOÇYİĞİT
(METU)

Student Name: İsmail ŞİMŞEKOĞLU
(Software Management)

January 2024

TECHNICAL REPORT
METU/II-TR-2024

Orta Doğu Teknik Üniversitesi
Enformatik Enstitüsü

MCDC Test-Yolu Oluşturmak için Akıllı Bir Algoritma: PathFinder

Danışman Adı: Altan KOÇYİĞİT
(ODTÜ)

Öğrenci Adı: İsmail ŞİMŞEKOĞLU
(Yazılım Yönetimi)

Ocak 2024

TEKNİK RAPOR
ODTÜ/II-TR-2024-

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Internal Use)	2. REPORT DATE 25.01.2024

3. TITLE AND SUBTITLE

PathFinder An Intelligent Algorithm for MCDC Test-Path Generation

4. AUTHOR (S)	5. REPORT NUMBER (Internal Use)
İsmail ŞİMŞEKOĞLU	*1. METU/SM-TR-2024-*

6. SPONSORING/ MONITORING AGENCY NAME(S) AND SIGNATURE(S)
Software Management Master's Programme, Department of Information Systems, Informatics Institute, METU
Advisor: Altan KOÇYİĞİT Signature:

7. SUPPLEMENTARY NOTES

8. ABSTRACT
Introducing Pathfinder, an innovative automated tool designed for generating comprehensive test cases in the realm of C language source codes. The primary objective is to fulfill Modified Condition/Decision Coverage (MC/DC) criteria, and Pathfinder follows a meticulously crafted methodology. The process unfolds in structured phases, commencing with source code parsing, advancing to the creation of a Control Flow Graph (CFG), and culminating in the systematic generation of test paths along with the determination of potential expected results.
Python, a widely used language known for its parsing capabilities and robust libraries, equips Pathfinder to tackle the inherent challenges presented by the intricacies of C language syntax. The project's emphasis on safety-critical industries, such as automotive and aerospace, aligns with the prevalent use of C in these sectors. The report provides a comprehensive exploration of each phase, from foundational source code parsing to the crucial role of identifying expected results in software testing.
Pathfinder's implementation encounters challenges, duly acknowledged and addressed in the report. These include complexities inherent in C language, parsing intricacies, scalability concerns with large codebases, and performance limitations of Python.
The report concludes with a forward-looking perspective on Pathfinder's future evolution. Envisaged enhancements involve broadening language feature support, incorporating external function analysis for more accurate predictions, and exploring the integration of machine learning algorithms. These strides aim to position Pathfinder as a versatile and refined tool adept at addressing the dynamic landscape of software development and testing practices.

9. SUBJECT TERMS	10. NUMBER OF PAGES
Software Testing, MC/DC Coverage, Control Flow Graph, Test-Path	36

TABLE OF CONTENT

LIST OF FIGURES

LIST OF ABBREVIATIONS

AST	Abstract Syntax Tree
CFG	Control Flow Graph
CLTP	Condition Level Test Paths
MC/DC	Modified Condition/Decision Coverage
PBO	Pseudo-Boolean Optimization
UML	Unified Model Language
XSD	XML Schema Definition

CHAPTER 1

INTRODUCTION

1.1 Motivation

Testing code for correctness and reliability is among the important tasks of software engineers. However, it can be challenging a time consuming to achieve Modified Condition/Decision Coverage (MC/DC) standards for testers. Selecting test parameters and expected values that fulfill the MC/DC criteria is usually a tedious and error-prone process, leading to significant resource consumption.

In order to address this challenge, there is a need to create an algorithm that can automate the process of finding test parameters and expected values that satisfy MC/DC coverage. Automating this process will allow to reduce the time and resources required for testing significantly while increasing the effectiveness and reliability of the test suites at the same time.

An algorithm that can automate this process is sought by many software test engineers and could be a valuable tool, allowing them to achieve their testing goals quickly and efficiently. With this, software testing engineers will be able to focus their attention on the more important aspects of software development while relying on automation to handle the tedious and time-consuming tasks associated with software testing.

MC/DC analysis is an important aspect of safety-critical software development. Safety-critical systems, such as those used in aerospace, automotive, and medical industries, must meet safety requirements that are difficult to verify in order to ensure that they operate reliably and safely. MC/DC analysis is a way of verifying that safety-critical software systems meet these requirements by testing each decision and condition in the program with different input values. Thus, MC/DC analysis is important because even a small software error in safety-critical systems can have grave consequences, such as system failure, injury, or loss of life. Furthermore, MC/DC analysis is complex and requires specialized expertise and tools to be performed accurately and effectively. Therefore, the development of an algorithm that can automate the process of finding test parameters and expected values for MC/DC coverage is a critical issue that requires a solution. Such a

tool could greatly improve the efficiency and effectiveness of software testing, thereby enabling software engineers to deliver high-quality software to their customers.

Ensuring the correctness and reliability of code through rigorous testing is a fundamental aspect of software engineering. However, attaining compliance with Modified Condition/Decision Coverage (MC/DC) standards for test coverage presents a daunting and time-intensive endeavor. The meticulous selection of test parameters and expected values that align with MC/DC criteria often proves laborious and prone to errors, resulting in substantial resource consumption.

An algorithm with the capability to automate this intricate process stands to become an invaluable tool for software engineers, providing them with the means to swiftly and effectively achieve their testing objectives. Such an advancement would empower software engineers to concentrate on the pivotal aspects of software development, relegating the mundane and repetitive tasks associated with testing to the realm of automation.

Moreover, MC/DC analysis is a complex process demanding specialized expertise and tools for accurate and effective execution. Consequently, the development of an algorithm or application capable of automating the process of identifying test parameters and expected values for MC/DC coverage emerges as a critical challenge in need of a solution. So, this tool has the potential to substantially enhance the efficiency and effectiveness of software testing.

1.2 Objectives

The primary objective of this project is to develop an algorithm and an application implementing this algorithm that can automatically find test parameters and corresponding expected values to achieve MC/DC coverage. The goal is to enhance the efficiency and effectiveness of MC/DC analysis, a critical component of software testing. By automating the test data generation process, the proposed solution aims to reduce the time and effort required to achieve MC/DC coverage while improving the quality of the generated tests. In addition, the project aims to evaluate the effectiveness of the proposed solution by comparing it with existing MC/DC coverage techniques and assessing its ability to meet software requirements.

In essence, the core objectives of this project are as follows:

- Developing an algorithm that can automatically find test parameters and corresponding expected values to achieve MC/DC coverage.
- Enhancing the efficiency and effectiveness of MC/DC analysis by reducing the time and effort required to achieve MC/DC coverage.

- Evaluating the effectiveness of the proposed solution by comparing it with existing MC/DC coverage techniques.
- Determining the effectiveness of the proposed solution in achieving the desired MC/DC coverage.

1.3 Scope

The project mainly concentrates on Boolean expressions as its primary focus with a specific emphasis on programs written in the C language. It will leverage a blend of data analysis, software development, and testing methodologies to accomplish its set objectives. The algorithm's design will prioritize platform independence, ensuring its applicability across a diverse spectrum of software systems.

1.4 Methodology

This project proposes a method to automatically determine test parameters and correct outputs to achieve MC/DC coverage. The steps of the method depicted in Figure 1 are as follows:

1. Parsing the source code to identify the conditions and decisions: The first step is to identify all the conditions and decisions within the code. Conditions refer to the logical expressions that are evaluated to be either true or false, while decisions are the points in the code where the program chooses between two or more paths based on a condition.
2. Constructing the control flow graph: The second step is to construct the control flow graph (CFG) for the code. The CFG is a graphical representation of the program's control flow, showing all the paths that the program can take.
3. Path exploration: The next step is to explore all possible paths in the CFG. This involves generating and solving constraints that satisfy each condition and decision in the program.
4. Test data generation: In this step, the algorithm generates test data that satisfies the MC/DC criteria. This involves selecting input values that satisfy the constraints generated in the previous step and cover all possible paths in the program.
5. Test data optimization: In the final step, the generated test data is optimized using boundary-value analysis. This involves selecting input values that lie on the boundaries of the input domain and are more likely to cause errors.

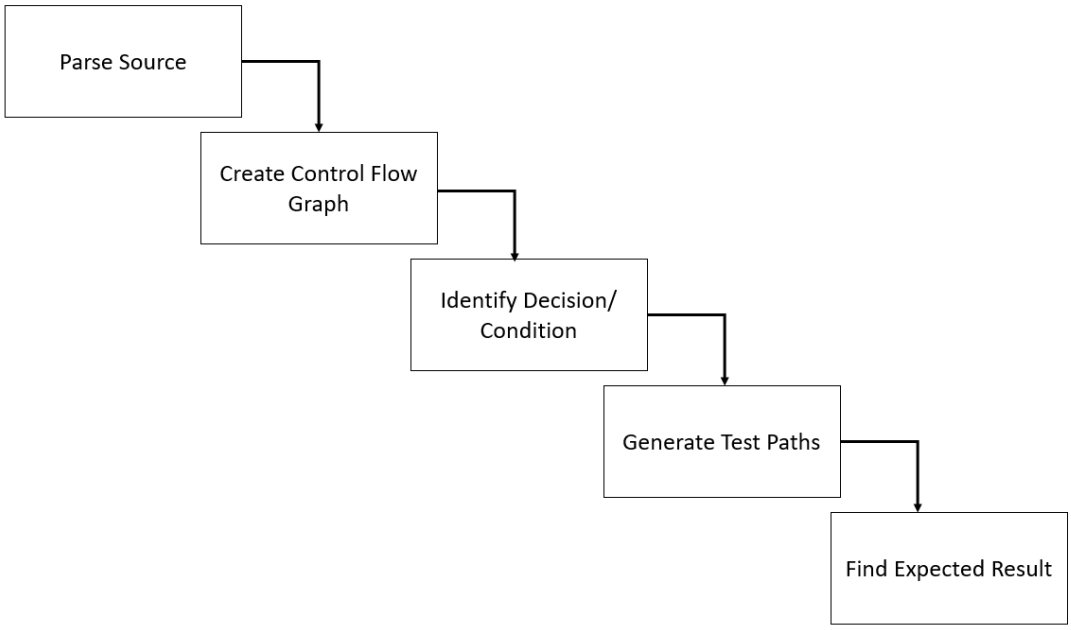


Figure 1 The Outline of PathFinder

In summary, Chapter 1 serves as the foundation for our investigation into automated test parameters and expected value generation for achieving Modified Condition/Decision Coverage (MC/DC) in software testing. Focused on the challenges of manual parameter selection, the chapter articulates project objectives centered on enhancing MC/DC analysis efficiency and effectiveness. The later chapters will delve into the background information, including metrics like Statement Coverage, Decision Coverage, and MC/DC, paving the way for a thorough exploration of related work in Chapter 3. Positioned within the broader academic landscape, this analysis identifies gaps in our innovative approach. Chapter 4 introduces the proposed solution, detailing the algorithm's intricacies, and Chapter 5 concludes the study, summarizing findings and insights derived from the exploration of MC/DC coverage and automated test generation solutions.

CHAPTER 2

BACKGROUND INFORMATION

This chapter provides background information regarding essential concepts and metrics integral to software testing methodologies. By delving into the definitions and intricacies of key metrics such as Statement Coverage, Decision Coverage, Modified Condition/Decision Coverage (MC/DC), Control Flow Graphs (CFGs), and Abstract Syntax Trees (ASTs), we lay the groundwork for a comprehensive understanding of the testing landscape.

2.1 Definitions

2.1.1 Statement Coverage

Statement coverage is a software testing metric that measures the percentage of statements in a program that has been executed at least once during testing. It is a white-box testing technique that focuses on the internal structure of the program.

Statement coverage is a basic metric that is often used to evaluate the effectiveness of test suites. A high statement coverage indicates that a test suite has exercised a large portion of the program code, making it more likely to have uncovered potential defects. However, statement coverage is not a perfect predictor of fault detection, as it does not consider the logical flow of the program.

The metric is straightforward to calculate and comprehend, and it frequently serves as a proxy for more intricate metrics. However, the metric does not fully assess testing effectiveness because it disregards the program's logical flow. Therefore, it is crucial to use the metric in conjunction with other metrics (Whittaker, 2000).

2.1.2 Decision Coverage

Decision coverage is a software testing metric that measures the percentage of decisions in a program that has been executed at least once during testing, considering both the true and false outcomes of each decision. A decision is a point in the program where a choice is made, such as an if statement or a switch statement.

Decision coverage is an important metric for software testing because it ensures that each decision in the program has been exercised, which helps to identify potential defects that may occur due to incorrect logic or data handling. It is a more stringent measure than statement coverage, which only requires that each statement in the program be executed at least once.

A recent study by Chen, Zhou, and Zhang (2018) found that decision coverage is a more effective measure of fault detection than statement coverage. The study evaluated the fault detection effectiveness of decision coverage and statement coverage in a corpus of Java projects. The results showed that decision coverage was able to detect more faults than statement coverage.

2.1.3 MC/DC

MC/DC stands for Modified Condition/Decision Coverage, which is a coverage criterion in software testing that aims at generating test data to cover each independent value of conditions in the program. It was raised as a trade-off between test adequacy and cost and is very important in software testing because it requires only a few test cases to satisfy this coverage criterion, and recent experimental results show that strong MC/DC has better fault detection capability than other criteria (Wu, 2014).

MC/DC addresses this limitation by demanding that each condition or decision point be exercised under both its true and false outcomes. This comprehensive approach ensures that the program's logical flow is adequately tested, significantly increasing the likelihood of detecting faults that might otherwise go unnoticed.

```
if (condition1 && condition2) {
    // Code block A
}
else {
    // Code block B
}
```

Figure 2 MC/DC Example

Consider a simple example given in Figure 2. Testing for statement coverage, executing either Code block A or Code block B would suffice. However, MC/DC demands that both true and false outcomes of the condition are tested, requiring the execution of both Code block A and Code block B.

This thoroughness extends beyond simple binary decisions, encompassing complex conditions involving multiple variables and nested statements. MC/DC ensures that

each condition or decision is exercised under all possible combinations of true and false outcomes, providing a more rigorous assessment of the program's logic.

The effectiveness of MC/DC in fault detection stems from its ability to uncover faults that lie in the program's decision-making processes. These faults can occur due to errors in condition evaluation or incorrect handling of different outcomes.

By exercising each condition or decision under both true and false outcomes, MC/DC increases the likelihood of triggering these faults, allowing them to be identified and addressed during the testing phase. This proactive approach prevents faults from propagating to subsequent stages of software development, reducing the risk of costly and potentially disastrous errors in the final product.

2.1.4 Control Flow Graph

A Control Flow Graph (CFG) serves as a visual depiction encapsulating all potential pathways that a program may traverse during its execution. This directed graph systematically illustrates the program's control flow, where individual nodes correspond to basic blocks, and edges signify the transition of control between these blocks. Basic blocks, within the CFG context, denote sequences of instructions executed sequentially without any jumps or targeted transfers. The directed edges capture control flow jumps within the program. Notably, the CFG designates two distinct blocks: the entry block, facilitating the initiation of control into the flow graph, and the exit block, signifying the point where all control flow exits. This graphical representation is indispensable for numerous compiler optimizations and static-analysis tools, providing a comprehensive overview of a program's control dynamics (Koppel, 2020).

CFGs serve as a fundamental visual representation of a program's control structure. CFGs play an important role in identifying conditions and decisions within the code. The graphical depiction provided by CFGs allows for clear and systematic visualization of the program's logical flow, including loops, conditions, and branching points. This visual insight is essential for pinpointing specific elements requiring test coverage to fulfill MC/DC criteria. CFGs facilitate the identification of critical paths and decision points, aiding in the creation of comprehensive test cases that cover all possible combinations of conditions.

2.1.5 Abstract Syntax Tree

An Abstract Syntax Tree (AST) functions as a tree-shaped data structure that mirrors the abstract syntactic arrangement of source code composed in a programming language. It meticulously captures the hierarchical composition of the program and its

syntax, although it does not encapsulate semantic nuances. Within the AST, nodes correspond to language constructs such as expressions, statements, and declarations, while edges symbolize the interconnections between these constructs. This structural representation proves instrumental in numerous software development tools, including compilers, interpreters, static analyzers, and refactoring tools, as it facilitates a comprehensive understanding and manipulation of the program's syntactic organization without delving into its semantic intricacies (Liang, 2022).

CHAPTER 3

RELATED WORKS

In this chapter, we explore the existing body of knowledge and research that serves as the foundation for our work. We looked for the previously proposed solutions and found out the justification of our project. It is important to have a thorough understanding of the current state of the field in order to properly appreciate and put into context the advancements presented in this study. We delve into previous research and key findings that have helped shape and develop the subject matter over time. By critically examining relevant literature, we aim to identify gaps, challenges, and opportunities that motivate and guide the novel contributions presented in our work. This comprehensive review serves as a roadmap for readers, offering insights into the broader academic landscape and positioning our research within the larger context of ongoing scholarly discussions.

3.1 Validating Object-Oriented Software At The Design Phase By Achieving MC/DC (Barisal, 2019):

The method proposed in this paper is a technique for validating object-oriented software at the design phase by achieving MC/DC (Modified Condition/Decision Coverage). The method consists of the following steps:

- Constructing a UML activity diagram for the given system using ArgoUML, a tool that supports various UML diagrams.
- Generating XML code from the UML activity diagram using ArgoUML's export function.
- Converting the XML code to XSD (XML Schema Definition) code, which is a more precise and readable representation of the XML elements.
- Generating a skeletal Java code from the XSD code using JAXB (Java Architecture for XML Binding), a tool that provides methods for binding XML schema and Java objects.
- Customizing the Java code according to the syntax of jCUTE, a tool that performs concolic testing, which combines concrete and symbolic execution to generate test cases.
- Applying jCUTE to the Java code to obtain test cases that cover all possible paths and outcomes of the program.
- Calculating MC/DC percentage from the test cases and the Java code using COPECA (COverage Percentage Calculator), an in-house developed tool that uses

an Extended Truth Table to find independent conditions and a formula to compute the coverage score.

The paper suggests a hybrid software verification technique that combines symbolic execution and concrete execution to generate test cases that aim to maximize code coverage. They have achieved 56.31% MC/DC coverage in their experiment. This solution is dependent on UML diagrams generated by third-party tools which may not be available for software engineers who work for automotive or aerospace industries.

3.2 MCDC-Star A White-Box Based Automated Test Generation For High MC/DC Coverage (Wong, 2018):

This paper proposes a white-box-based automated test case generation technique for achieving high modified condition/decision coverage (MC/DC) criterion using greedy-based symbolic execution.

The paper describes the following steps of the method:

- Program Instrumentation and Compilation: The paper instruments the subject program to measure MC/DC and compiles it to obtain an executable.
- Path Direction Generation: The paper analyzes the control flow of the subject program and constructs a path direction for each decision, which consists of a condition combination, an MC/DC improvement, and an assembly code execution sequence.
- Test Generation Using Symbolic Execution: The paper uses Triton, a dynamic symbolic executor, to generate test input values that can follow the path directions with the highest MC/DC improvements. The paper also handles the issues of unreachable decisions and constraint conflicts.
- Test Execution and MC/DC Measurement: The paper executes the generated test input values against the instrumented executable and updates the MC/DC coverage information using a branch-independent effect (BIE)-based approach.

The paper claims that the method can achieve high MC/DC coverage faster and more effectively than a random method.

The downside of their method is that it might change the program's behavior, which limits its practicality. This is because they use code transformation-based symbolic execution techniques to generate test cases for achieving MC/DC. Code transformation is a process of modifying the source code of a program to make it easier to analyze or test. However, this process can introduce errors or alter the semantics of the original program, which can

affect the validity of the test results. Therefore, their method is not suitable for testing programs that have strict requirements on correctness and reliability.

3.3 Automatic Test Data Generation For Unit Testing To Achieve Mc/Dc Criterion (Tianyong Wu, 2014):

This method generates test data for MC/DC, a complex coverage criterion that considers the logical expressions in the branch statements of a program. This solution relies on third-party tools to create test paths.

A four-step process that involves:

- Translating the target program into a Control Flow Graph (CFG).
- Extracting Condition Level Test Paths (CLTPs) from the CFG using a greedy strategy to select the next condition vector for each decision.
- Also, checking the feasibility of CLTPs and generating test data for the feasible and complete CLTPs using symbolic execution and constraint solving.
- Reducing the CLTP set without decreasing the coverage using a Pseudo-Boolean Optimization (PBO) solver.
- A prototype tool that implements the method and uses Clang, Z3, and clasp as external tools.

The condition vector selection strategy in the proposed method is a greedy strategy that selects the next condition vector for each decision. The strategy selects the condition vector that covers the most uncovered conditions in the current CLTP (Condition Level Test-Path) and has the least number of constraints. The strategy also considers the feasibility of the selected condition vector and the coverage of the remaining CLTPs. The goal of the strategy is to reduce the number of constraint-solving calls and the size of the CLTP set without decreasing the coverage.

The effectiveness of the greedy strategy is evaluated in the experiments, and the results show that the greedy strategy can improve the efficiency and cost of the test data generation process.

3.4 An Automated Tool for MC/DC Test Data Generation (Ariful Haque, 2014):

The objective of this paper is to design and implement an automated tool for MC/DC test data generation, called MC/DC GEN. The tool takes a Boolean expression as input and produces a set of test cases that satisfy the MC/DC criterion. The tool uses a local search

algorithm to find all possible MC/DC pairs for each predicate in the expression and then removes the redundant pairs to generate the final test data. The tool also compares the effectiveness of MC/DC with a pairwise testing technique using a case study.

The design and framework for MC/DC Gen is as follows:

- MC/DC Gen is a web-based tool that can automatically generate test data for structural testing based on the Modified Condition/Decision Coverage (MC/DC) criterion.
- MC/DC Gen is developed using PHP programming language and hosted in a Linux-based Virtual Private Server.
- MC/DC Gen consists of five main components: input processing and analysis, generating a list of solutions, MC/DC pairs with predicates, generating the final test data, and case study.
- Input processing and analysis: The user inputs the Boolean expression of the predicate to be tested according to the MC/DC Gen standard notation. The tool separates the predicates, operators, and grouping notation from the expression and keeps a pointer to match them correctly.
- Generating a list of solutions: The tool starts with a local search to find all possible solutions for the separated predicates. It randomly chooses an initial solution and computes its result. It then seeks for a neighbor solution that differs only by the value of one predicate but gives a different result for the expression. This is called an MC/DC pair and is stored in an array. The search continues until all the predicates are searched completely.
- MC/DC pairs with predicates: The tool populates a table with predicates in columns and a list of identified MC/DC pairs for each predicate in rows. This representation gives an overall view of the identified solutions.
- Generating the final test data: The tool removes the duplication test data from the previous step and generates the actual MC/DC pairs from the combination of pairs.

MC/DC GEN's web-based nature may hinder accessibility in restricted environments, and its exclusive reliance on PHP limits language support, posing challenges for projects in other languages. These downsides underscore the importance of considering infrastructure dependencies and language flexibility when opting for automated MC/DC test data generation tools like MC/DC GEN.

3.5 Comparing Pathfinder With Related Works

In this section, we draw comparisons between Pathfinder, our automated test case generation tool, and the existing works discussed in Chapter 3. Each tool contributes to

the realm of achieving Modified Condition/Decision Coverage (MC/DC) in software testing, but distinctive features and methodologies set them apart.

3.5.1 Validating Object-Oriented Software at the Design Phase (Barisal, 2019):

Similarities:

- Both approaches aim to achieve MC/DC coverage in the software testing process.
- Both tools utilize a combination of techniques, including symbolic execution, to generate test cases.

Differences:

- Pathfinder focuses on the automation of test case generation specifically for C language source codes, while Barisal's method targets object-oriented software using UML diagrams.
- Pathfinder employs a comprehensive process from source code parsing to expected result generation, whereas Barisal's method relies on UML diagrams and tools like ArgoUML and jCUTE.

3.5.2 MCDC-STAR (Wong, 2018):

Similarities:

- Both tools are designed to achieve high MC/DC coverage.
- Both utilize symbolic execution in their methodologies.

Differences:

- Pathfinder concentrates on C language source codes, while MCDC-STAR is a white-box-based automated test case generation tool written in Java.
- MCDC-STAR introduces the possibility of altering the program's behavior due to code transformation, a concern not present in Pathfinder.

3.5.3 Automatic Test Data Generation for Unit Testing (Tianyong Wu, 2014):

Similarities:

- Both tools aim for MC/DC coverage in unit testing.
- Both employ symbolic execution and constraint-solving in their processes.

Differences:

- Pathfinder is tailored for C language source codes, whereas Tianyong Wu's method is more generic and can be applied to various programming languages.

- Tianyong Wu's method uses a greedy strategy for condition vector selection, which is not explicitly employed in Pathfinder.

3.5.4 Automated Tool for MC/DC Test Data Generation (Ariful Haque, 2014):

Similarities:

- Both tools are designed to generate test data for achieving MC/DC coverage.
- Both emphasize automated processes in test data generation.

Differences:

- Pathfinder specifically targets C language source codes, while MC/DC GEN is developed for PHP.
- MC/DC GEN utilizes a local search algorithm, contrasting with Pathfinder's approach, which employs a combination of parsing, CFG generation, and test path generation.

In summary, while all the discussed tools, including Pathfinder, share the common goal of achieving MC/DC coverage, the differences in their target languages, methodologies, and underlying techniques highlight the unique contributions of each tool. Pathfinder stands out for its tailored approach to C language source codes and its comprehensive, automated process from source code parsing to expected result generation.

CHAPTER 4

PATHFINDER

In this section, we explain the details of PathFinder, our tool designed to automate the generation of comprehensive test cases for C language source codes, with a particular focus on fulfilling the Modified Condition/Decision Coverage (MC/DC) criteria. The methodology unfolds in a series of well-defined steps, each explained within dedicated sections. Beginning with the foundational step of parsing the source code (Section 4.1), we progress through the creation of a Control Flow Graph (CFG) (Section 4.2), the identification of decisions and conditions (Section 4.3), and the systematic generation of test paths (Section 4.4). The significance of finding expected results in software testing is explored in Section 4.5, underscoring its crucial role. A comprehensive summary of the entire PathFinder process is presented in Section 4.6, highlighting the interconnected modules that collectively enhance the efficiency and effectiveness of software testing. However, before delving into these detailed steps, we acknowledge and address the challenges encountered during the implementation of PathFinder in Section 4.7.

4.1 Parse Source Code

The proposed method ParseSource's implementation uses the Python programming language to parse C language source codes to extract decision structures, conditions, and related information. Once parsed, the tool generates comprehensive test cases that fulfill the MC/DC criteria. Leveraging Python's flexibility and powerful parsing libraries, we aim to automate the generation of test cases, reducing the manual effort involved in creating exhaustive test suites.

We chose the C programming language because C is one of the most used programming languages in safety-critical industries such as automotive and aerospace. Many embedded systems and firmware in these domains are written in C. Hence, this project directly addresses the testing needs of a significant portion of safety-critical applications. Moreover, several safety-critical systems have legacy codebases written in C. By supporting C, our project caters to the need for compatibility with existing systems. Moreover, industry standards often prescribe the use of C in safety-critical software development, reinforcing the relevance of our tool in such environments.

Parsing is essential for the extraction of decision structures, conditions, and related information, crucial elements for fulfilling the MC/DC criteria in test case generation. For the parsing algorithm a sample code, ParseSource, is given in Figure 3.

The ParseSource class assumes a central role within PathFinder by specializing in the critical task of parsing C code into a structured representation. Its primary responsibility lies in extracting information from the Abstract Syntax Tree (AST), a fundamental data structure that captures the hierarchical structure of the parsed code. The class performs this operation through its core method, parse(), which parses the C source code and populates the AST with relevant details. By focusing on the extraction of decision structures, conditions, and other essential elements, the ParseSource class transforms raw C code into an interpretable structured representation.

```python
import pycparser.c_parser
import pycparser.c_ast

class ParseSource:
    def __init__(self, file_path):
        self.file_path = file_path
        self.ast = None

    1 usage (1 dynamic)
    def parse(self):
        with open(self.file_path, 'r') as f:
            try:
                ast = pycparser.c_parser.CParser().parse(f.read())
                self.ast = ast
            except pycparser.c_parser.ParseError as e:
                print(f"Parsing error: {e}")
                return None

    1 usage (1 dynamic)
    def get_statements(self):
        if self.ast is None:
            return None
        return [stmt for stmt in self.ast.ext if isinstance(stmt, pycparser.c_ast.stmt)]

    1 usage (1 dynamic)
    def get_conditions(self):
        if self.ast is None:
            return None
        return [expr for expr in self.ast.ext if isinstance(expr, pycparser.c_ast.Expr) and expr.op in ('>', '<', '==', '!=', '>=', '<=')]

    1 usage (1 dynamic)
    def get_control_structures(self):
        if self.ast is None:
            return None
        return [stmt for stmt in self.ast.ext if isinstance(stmt, (pycparser.c_ast.If, pycparser.c_ast.While, pycparser.c_ast.For))]
```

Figure 3 ParseSource Class – Parsing Algorithm

The sample code given in Figure 3, starts with import statements and includes the following methods:

- parse(): *Parses the C source code and stores the AST*
- get_statements(): *Extracts all statements from the AST*
- get_conditions(): *Extracts all conditional expressions from the AST.*

- get_control_structures(): Extracts all control structures (if, while, for) from the AST.

4.2. Create Control Flow Graph

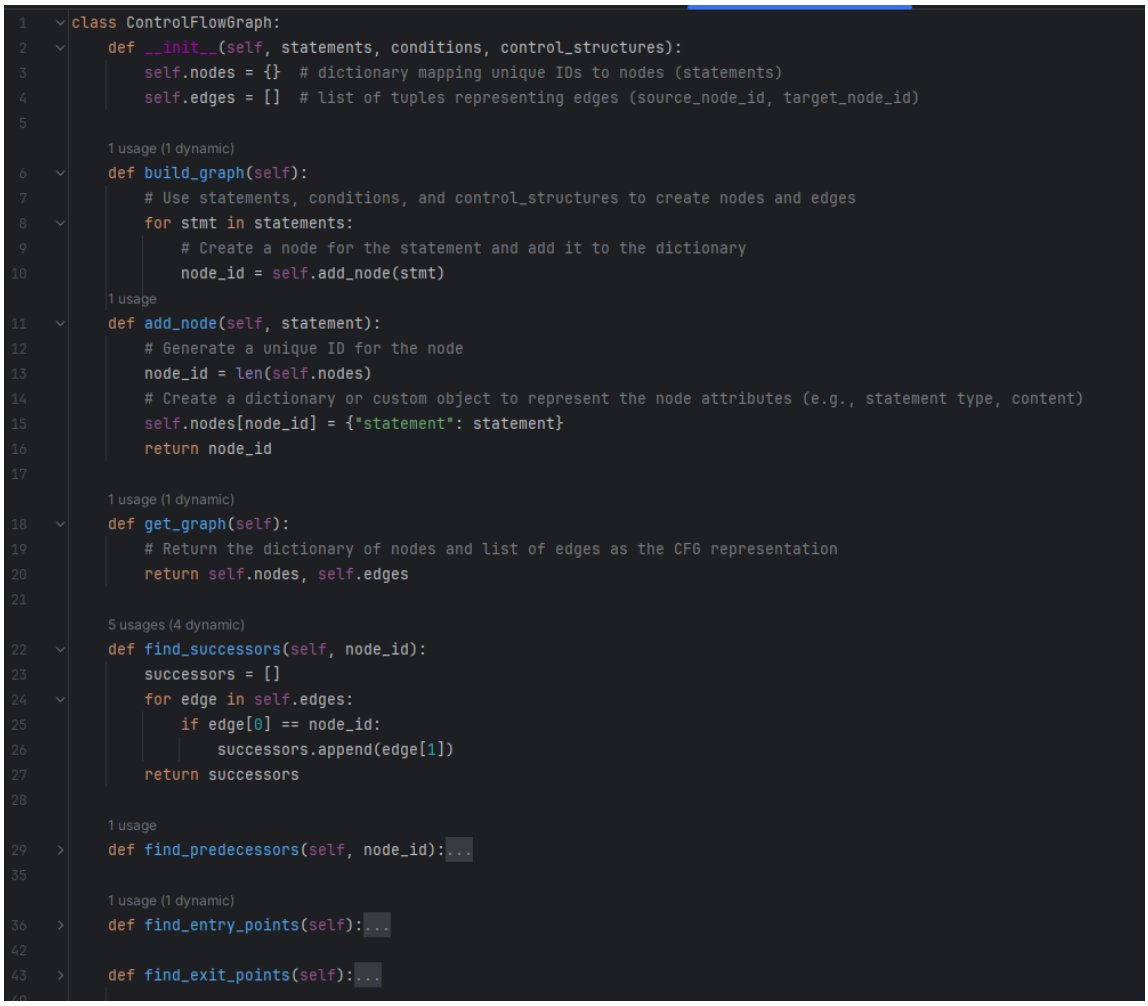
Creating a Control Flow Graph (CFG) is important for several reasons. A CFG provides a visual representation of the flow of control within the code. This is valuable for understanding the overall structure of the program, including loops, conditions, and branching.

MC/DC coverage requires identifying conditions and decisions in the code. The CFG makes it easier to pinpoint these elements, aiding in the creation of test cases that cover all possible combinations.

MC/DC coverage mandates thorough testing of conditions and decisions in the code. The CFG serves as a powerful tool for achieving this coverage by making it easier to pinpoint these elements. Test cases derived from CFG analysis can systematically cover different paths, ensuring that all possible combinations of conditions are exercised.

CFG aids in identifying and visualizing potential paths through the code, enabling testers to analyze and verify the correctness of the program's logic. It also facilitates the detection of unreachable or redundant code, contributing to code quality and maintainability.

The code excerpt given in Figure 4, ControlFlowGraph, illustrates the CFG generation algorithm. The ControlFlowGraph class assumes a pivotal role in PathFinder by standing for a control flow graph (CFG) of the source code. The primary responsibility of this class is to visually capture and articulate the flow of control within the code. Achieved through the build_graph() method, the CFG serves as a graphical representation, elucidating the program's overall structure, encompassing loops, conditions, and branching.



```python
class ControlFlowGraph:
    def __init__(self, statements, conditions, control_structures):
        self.nodes = {}  # dictionary mapping unique IDs to nodes (statements)
        self.edges = []  # list of tuples representing edges (source_node_id, target_node_id)

    1 usage (1 dynamic)
    def build_graph(self):
        # Use statements, conditions, and control_structures to create nodes and edges
        for stmt in statements:
            # Create a node for the statement and add it to the dictionary
            node_id = self.add_node(stmt)
    1 usage
    def add_node(self, statement):
        # Generate a unique ID for the node
        node_id = len(self.nodes)
        # Create a dictionary or custom object to represent the node attributes (e.g., statement type, content)
        self.nodes[node_id] = {"statement": statement}
        return node_id

    1 usage (1 dynamic)
    def get_graph(self):
        # Return the dictionary of nodes and list of edges as the CFG representation
        return self.nodes, self.edges

    5 usages (4 dynamic)
    def find_successors(self, node_id):
        successors = []
        for edge in self.edges:
            if edge[0] == node_id:
                successors.append(edge[1])
        return successors

    1 usage
    def find_predecessors(self, node_id):...

    1 usage (1 dynamic)
    def find_entry_points(self):...

    def find_exit_points(self):...
```

Figure 4 ControlFlowGraph Class - CFG Generation Algorithm

The excerpt code given in Figure 4, includes the following methods that build a comprehensive CFG representation, aiding in the generation of exhaustive test suites that adhere to MC/DC criteria:

- build_graph(): *Builds the CFG by creating nodes and edges based on the provided statements, conditions, and control structures.*
- add_node(): *Adds a node to the CFG with the given statement.*
- get_graph(): *Returns the CFG representation as a tuple of (nodes, edges)*
- find_successors(): *Returns a list of node IDs that are directly reachable from the given node.*
- find_predecessors(): *Finds the predecessor nodes that can directly reach a given node.*

- find_entry_points(): *Identifies nodes without any incoming edges (potential starting points).*
- find_exit_points(): *Identifies nodes without any outgoing edges (potential ending points).*

4.3 Identify Decisions and Conditions

Identifying and thoroughly testing decision points and conditions within the source code is a crucial responsibility of PathFinder, facilitated by the functionalities within the Identify Decisions and Conditions phase. This phase serves as a bridge between the parsing capabilities of the ParseSource class and the graphical representation provided by the ControlFlowGraph class.

The identification and parsing of branches contribute to the creation of a map of possible execution paths, enabling the PathFinder to cover all conceivable scenarios and ensuring the reliability and robustness of safety-critical applications in the face of diverse operational conditions.

The primary responsibility in this phase is to identify decision points (branches) within the Control Flow Graph (CFG). The class definition given in Figure 5, DecisionIdentifier, ensures that decision points and conditions are accurately identified and analyzed. The systematic traversal of paths and parsing of branches contribute to the creation of a detailed understanding of the program's decision-making structures.

The sample code given in Figure 5, includes the following methods:

- identify_branches(): *Identifies branches within the CFG and returns a dictionary mapping decision points to their associated branch path.*
- parse_branches(): *Parses branches for a given decision point and returns a list of branch paths.*
- traverse_path(): *Recursively traverses a path through the CFG, adding statements to the branch_path list.*

```python
class DecisionIdentifier:
    def __init__(self, cfg):
        self.cfg = cfg  # ControlFlowGraph object
        self.decision_points = self.cfg.analyze_branch_conditions()

    1 usage (1 dynamic)
    def identify_branches(self):
        branches = {}
        for decision_point in self.decision_points:
            # Analyze branches based on condition and outgoing edges
            # Group statements into branch paths (true, false, etc.)
            # Store branches in a dictionary with decision point key and list of paths values
            branches[decision_point] = self.parse_branches(decision_point)
        return branches

    1 usage
    def parse_branches(self, decision_point):
        branches = []
        next_nodes = self.cfg.find_successors(decision_point)
        for next_node in next_nodes:
            branch_path = [self.cfg.nodes[decision_point]["statement"]]
            self.traverse_path(branch_path, next_node)
            branches.append(branch_path)
        return branches

    2 usages
    def traverse_path(self, branch_path, node_id):
        # Recursively explore the path through successors until reaching an exit point
        # Add statements to the branch_path list
        next_nodes = self.cfg.find_successors(node_id)
        if not next_nodes:
            return
        for next_node in next_nodes:
            branch_path.append(self.cfg.nodes[next_node]["statement"])
            self.traverse_path(branch_path, next_node)
```

Figure 5 DecisionIdentifie Class - Decision Identifying Algorithm

4.4 Generate Test Paths

The core of the PathFinder tool lies in its ability to systematically generate test paths that fulfill the MC/DC criteria. Once the decision points and conditions are identified through the ControlFlowGraph and DecisionIdentifier phases, the Generate Test Paths phase takes center stage. This phase is responsible for exploring and generating all possible test paths within the Control Flow Graph (CFG) to ensure comprehensive coverage.

The algorithm in Figure 6, TestPathGenerator, outlines the process of generating test paths. The TestPathGenerator class encapsulates the functionality for generating test paths, utilizing the depth-first search (DFS) approach to systematically traverse the Control Flow Graph (CFG). The recursive nature of the DFS ensures that all possible paths are explored, considering decision points and conditions at each step. The generated test

paths represent diverse scenarios, fulfilling the MC/DC coverage criteria and providing a comprehensive set of test cases for software testing.

The TestPathGenerator has the following methods:

- generate_expected_results(): Generates expected results for all test paths.
- calculate_results_for_path(): Calculates expected results for a single test path.
- Satisfies_coverage(): Checks if a given path satisfies the specified coverage criteria.

```python
class TestPathGenerator:
    def __init__(self, cfg, branches):
        self.cfg = cfg
        self.branches = branches

    1 usage (1 dynamic)
    def generate_paths(self, coverage_criteria):
        paths = []
        for entry_point in self.cfg.find_entry_points():
            paths.extend(self.explore_paths(entry_point, coverage_criteria))
        return paths

    3 usages
    def explore_paths(self, node_id, coverage_criteria, path=[]):
        if node_id in self.branches:
            for branch_path in self.branches[node_id]:
                new_path = path + branch_path
                if self.satisfies_coverage(new_path, coverage_criteria):
                    yield new_path
                self.explore_paths(self.cfg.find_successors(node_id)[-1], coverage_criteria, new_path)
        else:
            next_node = self.cfg.find_successors(node_id)[0]
            if next_node is None:
                yield path
            else:
                self.explore_paths(next_node, coverage_criteria, path + [self.cfg.nodes[next_node]["statement"]])

    1 usage
    def satisfies_coverage(self, path, coverage_criteria):
        covered_conditions = set()

        for statement in path:
            if statement in self.branches:
                condition_id = self.cfg.nodes[statement]["id"]
                covered_conditions.add(condition_id)

        if coverage_criteria == "mcdc":
            # Check MC/DC coverage
            return len(covered_conditions) == len(self.branches)
```

Figure 6 TestPathGenerator Class - Test Path Generation Algorithm

In summary, the TestPathGenerator class serves as a pivotal component in the PathFinder, offering a systematic approach to generate test paths that adhere to the specified coverage criteria, with a particular focus on MC/DC Coverage. Leveraging recursive exploration

and decision-point analysis, the algorithm navigates through the Control Flow Graph (CFG), producing diverse test paths that comprehensively cover the program's logical flow. The satisfaction of MC/DC coverage criteria is rigorously checked, ensuring that each decision point and condition undergoes thorough testing.

4.5. Find Expected Results

The expected result is crucial in software testing because it serves as a predefined benchmark against which the actual outcomes of test cases can be compared. It provides a definitive target or anticipated behavior for a given set of inputs, allowing testers to verify whether the software functions as intended. Testers compare the actual outcomes of test cases with the expected results to identify any discrepancies or deviations from the intended behavior.

Any disparities between actual results and expected outcomes indicate potential defects or issues in the software.

Especially, in the absence of formal requirements, finding expected results during testing becomes a critical navigational tool for software testers. Without predefined specifications, the process of determining expected outcomes involves a careful examination of the software's behavior under various inputs and scenarios. The act of finding expected results in such scenarios aids in uncovering defects.

```python
class ExpectedResultGenerator:
    def __init__(self, cfg, test_paths):

        self.cfg = cfg
        self.test_paths = test_paths

    1 usage (1 dynamic)
    def generate_expected_results(self, include_intermediate=False):

        expected_results = {}
        for path in self.test_paths:
            expected_results[path] = self.calculate_results_for_path(path, include_intermediate)
        return expected_results

    1 usage
    def calculate_results_for_path(self, path, include_intermediate):

        results = []  # Store intermediate or final results for the path
        current_state = {}  # Initialize variables and their values

        for statement in path:
            # 1. Evaluate variable assignments and updates
            assignments = self.parse_assignments(statement)
            current_state.update(assignments)

            # 2. Evaluate conditional expressions and branch outcomes
            if statement in self.cfg.decision_points:
                condition = self.cfg.nodes[statement]["condition"]
                result = self.evaluate_condition(condition, current_state)
                results.append(result)  # Track branch outcome

            # 3. Track intermediate results based on preferences
            if include_intermediate:
                intermediate_value = self.calculate_intermediate_value(statement, current_state)
                results.append(intermediate_value)

        # 4. Calculate final result for the path
        final_result = self.calculate_final_result(path, current_state)
        results.append(final_result)
        return results

```

Figure 7 ExpectedResultGenerator Class: Expected Result Algorithm

In this context, ExpectedResultGenerator class shown in Figure 7, systematically generates expected results for a given set of test paths. This critical functionality is based on the analysis of the Control Flow Graph (CFG), ensuring that the expected outcomes for each test path are accurately determined.

ExpectedResultGenerator has the following methods:

- generate_expected_results(): Generates expected results for all test paths.
- calculate_results_for_path(): Calculates expected results for a single test path.

In essence, the "Find Expected Result" section plays a pivotal role in enhancing the depth and accuracy of software testing. Through systematic analysis and calculation, it ensures that the expected results align with the specified coverage criteria, providing software

engineers with valuable insights into the performance and reliability of their code under diverse scenarios.

4.6. Summary Of Pathfinder Process

The test automation process within the PathFinder comprises several interconnected modules, each playing a distinct role in enhancing the efficiency and effectiveness of software testing. From the initial parsing of the source code to the systematic generation of expected results, the process unfolds in a structured manner.

The journey begins with the Parse Source Code module (4.1), where the C language source code is analyzed to extract decision structures, conditions, and related information. This parsed information forms the foundation for subsequent analysis.

Following this, the Create Control Flow Graph module (4.2) steps in to construct a visual representation of the program's control flow. The Control Flow Graph (CFG) serves as a roadmap for identifying conditions, decisions, and potential paths, laying the groundwork for comprehensive test coverage.

In the Identify Decisions and Conditions phase (4.3), the tool systematically identifies decision points and conditions within the CFG, creating a map of possible execution paths. This critical step facilitates a detailed understanding of the program's decision-making structures.

The Generate Test Paths module (4.4) takes center stage as it dynamically explores the CFG, systematically generating diverse test paths that adhere to the Modified Condition/Decision Coverage (MC/DC) criteria. Through recursive depth-first search, the algorithm ensures comprehensive coverage of decision points and conditions.

Transitioning to the Find Expected Result section (4.5), the tool calculates and generates expected results for each test path. This process involves both the systematic calculation of outcomes for individual paths and the holistic generation of expected results for the entire set of test paths.

In summary, the summarized test automation process encapsulates the journey from code parsing to the generation of expected outcomes. This organized approach improves the effectiveness of software testing, offering a strong structure for identifying problems, confirming accuracy, and guaranteeing the dependability of software in various situations. The interconnected nature of these modules contributes to the overall effectiveness of the PathFinder in empowering software engineers to deliver high-quality, reliable software products.

4.7. Challenges of Pathfinder

The implementation of the PathFinder encountered several challenges, each posing distinctive hurdles in achieving seamless and efficient test automation. These challenges spanned various aspects of the development process and are outlined below:

- Complexity of C Language:

  The inherent complexity of the C programming language posed a significant challenge in code analysis. Navigating through the multitude of language constructs, intricate syntax, and addressing edge cases demanded a sophisticated approach. Ensuring that the tool effectively handles diverse coding styles and constructs was a critical challenge, requiring attention to detail during the code analysis phase.

- Parsing Challenges:

  The presence of preprocessor directives, macros, and complex declarations in C code introduced parsing challenges. The variability introduced by these elements required the development of robust parsing mechanisms to accurately extract decision structures and conditions. Overcoming the intricacies of preprocessor directives and handling complex macro expansions emerged as a formidable task in achieving precise code analysis.

- Handling Large Codebases:

  As the tool aimed to be applicable to real-world scenarios, handling large codebases became a pivotal challenge. The complexities introduced by multiple files, interdependencies, and diverse project structures necessitated the development of scalable mechanisms. Ensuring that the tool maintains efficiency and accuracy in the face of extensive codebases posed a constant challenge throughout the implementation process.

- Python Performance Limitations:

  The performance limitations of the Python programming language became evident when dealing with extensive codebases. Parsing and generating test paths for large codes presented time-intensive processes, impacting the overall efficiency of the tool.

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this report, we introduced Pathfinder, an automated tool designed to streamline the generation of comprehensive test cases for C language source codes, with a specific focus on fulfilling the Modified Condition/Decision Coverage (MC/DC) criteria. The methodology presented in Pathfinder unfolds through a structured series of steps, including source code parsing, control flow graph (CFG) creation, decision and condition identification, test path generation, and determination of expected results.

Pathfinder's foundation lies in its ability to parse C language source codes, extract decision structures, and systematically generate test paths that adhere to MC/DC criteria. Leveraging Python's parsing capabilities and powerful libraries, the tool aims to reduce manual effort and enhance the efficiency of test case generation.

The creation of a CFG provides a visual representation of the code's control flow, aiding in the identification of decision points and conditions crucial for MC/DC coverage. The systematic generation of test paths, facilitated by the TestPathGenerator, ensures thorough coverage of decision points and conditions. Additionally, the identification and parsing of branches, as handled by the DecisionIdentifier, contribute to the creation of a detailed map of possible execution paths.

The Find Expected Results module plays a pivotal role in software testing, systematically generating expected outcomes for each test path. This process is crucial in evaluating the correctness and reliability of the code under diverse scenarios.

As Pathfinder evolves, several avenues for future enhancements emerge to augment its capabilities and address potential limitations. Looking ahead, Pathfinder's future work involves enhancing support for more C language features, including complex expressions, function calls, and arrays. Additionally, external function analysis is proposed to analyze calls and potential impacts on variable values, contributing to more accurate result prediction. The integration of machine learning models is also considered, aiming to improve prediction accuracy and handle more complex code constructs.

Enhanced Support for C Language Features:

While Pathfinder is currently capable of parsing and analyzing decision structures in C language source codes, future development could extend its capabilities to encompass a broader range of language features. This includes the incorporation of more complex expressions, function calls, and arrays, enabling Pathfinder to generate test cases that account for a wider spectrum of C language constructs. This expansion would contribute to a more comprehensive and adaptable testing solution.

External Function Analysis:

To enhance the precision of Pathfinder's test case generation, an area of potential improvement involves the analysis of external functions. If the parsed code interacts with external functions, future iterations of Pathfinder could incorporate mechanisms to analyze these calls and assess their potential impact on variable values within the program. This additional layer of analysis would contribute to a more accurate prediction of results, particularly in scenarios where external functions influence the program's behavior.

Machine Learning Integration:

To harness the power of predictive modeling and further refine its test case generation, Pathfinder could explore the integration of machine learning models. By leveraging machine learning algorithms trained on extensive datasets of program behavior, Pathfinder might enhance its prediction accuracy. This integration could prove especially beneficial when handling intricate code constructs and adapting to diverse programming styles. The utilization of machine learning could contribute to a more intelligent and adaptive test case generation process.

The envisioned future work for Pathfinder revolves around expanding its language feature support, delving into external function analysis for more precise predictions, and exploring the integration of machine learning models to enhance overall accuracy and adaptability. These potential advancements aim to position Pathfinder as a more versatile and sophisticated tool, catering to the evolving landscape of software development and testing practices.

In conclusion, Pathfinder represents a valuable contribution to the field of automated test case generation, offering a systematic and adaptable approach. Its structured methodology, combined with potential future enhancements, positions Pathfinder as a versatile tool to aid software engineers in delivering high-quality, reliable software products. As software development continues to evolve, Pathfinder stands as a testament to the ongoing pursuit of effective and intelligent testing solutions.

REFERENCES

F. Ahishakiye, J. I. Requeno Jarabo, L. M. Kristensen, V. Stolz, "MC/DC Test Cases Generation Based on BDDs," in Dependable Software Engineering: Theories, Tools, and Applications, vol. 13071, pp. 178, 2021.

I. K. Ariful Haque, "An Automated Tool for MC/DC Test Data Generation," in Proceedings of the Australian Software Engineering Conference, pp. 152-157, 2014.

J. A. Whittaker, "What Is Software Testing? Why Is It So Hard? Practice Tutorial," in IEEE Software, vol. 17, pp. 70-79, 2000.

J. Koppel, "Automatically Deriving Control-Flow Graph Generators from Operational Semantics," in Proceedings of the ACM on Programming Languages, pp. 742-771, 2020.

L. H. Wong, "MCDC-Star: A White-Box Based Automated Test Generation for High MC/DC Coverage," in 5th International Conference on Dependable Systems and Their Applications (DSA), pp. 102-112, 2018.

Linghuan Hu, W. Eric Wong, D. Richard Kuhn, Raghu Kacker, "MCDC-Star: A White-Box Based Automated Test Generation for High MC/DC Coverage", 2018 5th International Conference on Dependable Systems and Their Applications (DSA), pp.102-112, 2018.

R. Liang, "AstBERT: Enabling Language Model for Code Understanding with Abstract," in Proceedings of the Fourth Workshop on Financial Technology and Natural Language Processing, 2022, pp. 70-77.

S. Hallé, "Test Suite Generation for Boolean Conditions with Equivalence Class Partitioning," 2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering (FormaliSE), Pittsburgh, PA, USA, 2022, pp. 23-33, doi: 10.1145/3524482.3527659.

S. K. Barisal, "Validating Object-Oriented Software at Design Phase by Achieving MC/DC," in International Journal of System Assurance Engineering and Management, pp. 811-823, 2019.

S. Kangoye, A. Todoskoff, and M. Barreau, "Practical methods for automatic MC/DC test case generation of Boolean expressions," 2015 IEEE AUTOTESTCON, National Harbor, MD, USA, 2015, pp. 203-212, doi: 10.1109/AUTEST.2015.7356490.

T. Wu, J. Yan, "Automatic Test Data Generation for Unit Testing to Achieve MC/DC Criterion," in Eighth International Conference on Software Security and Reliability, pp. 118-126, 2014.