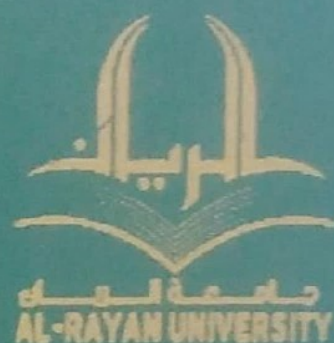


Republic of Yemen
Ministry of High Education
& Scientific Research
Al-Rayhan University
Faculty of Higher Studies



SOFTWARE BUG PREDICTION WITH SMART UNIT TESTS USING STATIC ANALYSIS

**Thesis submitted to Al Rayhan University in fulfillment of the
requirements for the degree of Master in Information Technology**

By

Hisham Abdullah Bin Ateya

Supervised By

Dr. Saeed Mohammed Baneamoon

1442 / 2020

Republic of Yemen
Ministry of High Education
& Scientific Research
Al-Rayyan University
Faculty of Higher Studies



SOFTWARE BUG PREDICTION WITH SMART UNIT TESTS USING STATIC ANALYSIS

**Thesis submitted to Al Rayan University in fulfillment of the
requirements for the degree of Master in Information Technology**

By
Hisham Abdullah Bin Ateya

Supervised By
Dr. Saeed Mohammed Baneamoon

1442 / 2020 م هـ


Approval of the Proofreader

I certify that the master's dissertation titled, (**Software Bug Prediction with Smart Unit Tests Using Static Analysis**) submitted by the student **Hisham Abdullah Bin Ateya** has been linguistically reviewed under my supervision and has become in scientific style and clear from linguistic errors and for that I sign.

Proofreader: Abdullah Amer Alkathiri

Academic Title: Assistant Professor

University: Hadhramout University

Signature: 

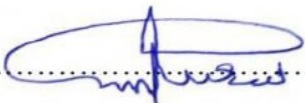
Date: 28 / 08 / 2020

Approval of the Scientific Supervisor

I certify that the master's dissertation titled, (**Software Bug Prediction with Smart Unit Tests Using Static Analysis**) submitted by the student **Hisham Abdullah Bin Ateya** has been completed in all its stages under my supervision and so I nominate it for discussion.

Proofreader: Dr. Saeed Mohammed Baneamoon

Signature:



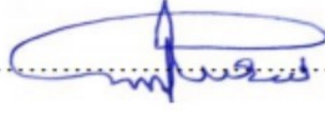
Date: 01 / 09 / 2020

The Discussion Committee Decision

Based on the decision of the President of the University No. (4) in the year 2020 regarding the nomination of the committee for discussing the master's thesis entitled, (**Software Bug Prediction with Smart Unit Tests Using Static Analysis**) for the researcher **Hisham Abdullah Bin Ateya**. We, the head of the discussion committee and its members, acknowledge that we have seen the aforementioned scientific thesis and we have discussed the student in its contents and what related to it.

Chairman of the Committee

Associate Professor Dr. Saeed Mohammed Baneamoon

Signature:

Committee member

Assistant Professor

Dr. Mohammed Abdullah Bamatraf

Signature:

Committee member

Assistant Professor

Dr. Hassan Abdullah Baswad

Signature:

DEDICATION

I dedicate my thesis to my beloved Father and Mother

ACKNOWLEDGMENTS

Praise be to Allah, peace and blessings be upon our prophet Muhammad, his family and his companions.

Firstly, I would like to express my respect and deep gratitude to my supervisor, Associate Prof. Dr. Saeed Mohammed Baneamoon, who helped and supported me in finishing my research.

A special thanks goes to Dr. Jameleddine Hassine, who helped and guided me in all things related to software testing and static analysis.

My sincere thanks goes to colleague Dr. Ali Ben Slim, who assisted me while writing documentations.

I would also like to thank lecturer Amel Hussein Baghaffar who helped me correct the grammar in my documentations, ensuring that they are designed appropriately for the target audience. Also special thanks to assistant professor Abdullah Amer Alkathiri for the final linguistical review.

A special thanks goes to my family, for their encouragement and endless support.

My most sincere thanks and appreciation goes to my beloved wife Amani, for her love, patience and support during my study. Without her patience, understanding, love and prayers, the completion of this work would not have been achieved.

Finally, I am grateful to everyone who helped and supported me during my studies.

ABSTRACT

Software Testing is an important phase in the Software Development Life Cycle, to ensure that the verification and validation of the software meet the requirements. Also it is important to have a lot of testing in this phase, before the software is released, to ensure that the software is bug free. However, shipping a software with tons of bugs is something unintended behavior, and unacceptable from the user point of view. Therefore, predicting the software bugs at the early stage of the software development will reduce the time and cost required for testing the software, which saves a lot of money for the companies, moreover it increases the software quality. Predicting software bugs had some challenges, such as generating test data that had been used into the test, and exploring the method paths. This research addresses these issues, by introducing a hybrid approach proposed to identify the potential bugs in the source code for the method under test by constructing an Abstract Syntax Tree model for the method, then traversing the tree and exploring all paths to find the bugs. Hence, Smart Unit Tests are generated accurately to cover all possible execution paths for the tested method. At the end, the proposed approach using static analysis is able to predict all kinds of static bugs and generate the minimal suite of unit tests which are able to cover all the possible execution paths for the tested code. This indicates that the proposed approach achieves good results compared with other techniques in terms of type of bugs that can be predicted as well as the number of generated unit tests that are required to test the code.

الخلاصة

يعتبر إختبار البرمجيات من أهم المراحل في دورة حياة تطوير البرمجيات، لضمان التحقق وصحة البرمجيات، وتوافقها مع المتطلبات. كذلك من المهم في هذه المرحلة من عمل عدة إختبارات للبرمجيات قبل تصديرها، لضمان خلوها من الأخطاء البرمجية. مع ذلك، يعتبر تصدير البرمجيات التي تحتوي على عدة أخطاء سلوك غير مرغوب فيه، وكذلك غير مقبول من وجهة نظر المستخدم. لهذا فإن التنبؤ بمثل هذه الأخطاء في المراحل الأولى سوف يقوم بتقليل الوقت والتكلفة المطلوبة لإختبار البرمجيات، الأمر الذي سيحفظ الكثير من الأموال للشركات. علاوة على ذلك سوف يزيد من جودة تلك البرمجيات. فعملية التنبؤ بأخطاء البرمجيات يحتوي على بعض التحديات مثل توليد بيانات الإختبار والتي تستخدم في عملية الإختبار، و عملية إستكشاف مسارات الإجراء المراد إختباره. هذا البحث سيتطرق لكل هذه المشاكل، بواسطة تقديم منهجية هجينة تم إقتراحها للتنبؤ بالعلل والأخطاء الموجودة في الشفرة المصدرية للإجراء المراد إختباره وذلك بواسطة تشييد نموذج أشجار الصياغة المجردة للإجراء، ثم التنقل وإستكشاف جميع المسارات لإيجاد هذه الأخطاء. فبالتالي يتم عملية توليد مجموعة من وحدات الإختبار الذكية بدقة، والكافية لفحص وتغطية كافة مسارات التنفيذ الممكنة للشفرة المراد إختبارها. أن المنهجية المقترحة حققت نتائج جيدة بالمقارنة مع التقنيات الأخرى من ناحية أنواع الأخطاء المراد التنبؤ بها وكذلك عدد وحدات الإختبار اللازمة لإختبار الشفرة.

Table of Contents

Dedication.....	A
Acknowledgment.....	B
Abstract.....	C
Arabic Abstract.....	D
Table of Contents.....	E
List of Tables.....	K
List of Figures.....	L
List of Abbreviations.....	N

Chapter 1: Introduction

1.1 Introduction.....	2
1.2 Motivation.....	3
1.3 Problem Statement.....	4
1.4 Objectives.....	4
1.5 Scope & Limitation.....	5
1.6 Research Approach.....	5
1.6.1 Problem Identification.....	5
1.6.2 Analysis of Current Techniques.....	6
1.6.3 Proposed Method.....	6
1.6.4 Implementation (Predicting the Software Bugs).....	6

1.6.5	Testing	6
1.6.6	Evaluation	6
1.7	Research Contribution	7
1.8	Thesis Organization	7

Chapter 2: Background

2.1	Introduction.....	9
2.2	Testing	9
2.2.1	Testing Types.....	10
2.2.2	Testing Methods	11
2.2.3	Testing Approaches	11
2.2.4	Testing Levels	12
2.3	Unit Testing	14
2.3.1	Parameterized Unit Tests	15
2.3.2	Advantages & Disadvantages of Unit Testing.....	16
2.3.3	Unit Testing Frameworks	17
2.4	Summary	17

Chapter 3: Literature Review

3.1	Introduction.....	19
3.2	Static Testing	19
3.3	Static Testing Techniques	20

3.3.1	Informal Reviews.....	20
3.3.2	Technical Reviews.....	20
3.3.3	Walkthrough	20
3.3.4	Inspection.....	20
3.3.5	Static Code Review.....	20
3.4	Dynamic Testing	21
3.5	Dynamic Testing Techniques	21
3.5.1	Unit Testing	21
3.5.2	Integration Testing.....	21
3.5.3	System Testing.....	21
3.5.4	User Acceptance Testing (UAT)	22
3.6	Difference between Static Testing and Dynamic Testing	22
3.7	Static Analysis Techniques	23
3.7.1	Syntactic Pattern Matching.....	23
3.7.2	Data Flow Analysis.....	24
3.7.3	Abstract Interpretation	25
3.7.4	Constraint-Based Analysis.....	26
3.8	Miscellaneous Techniques.....	27
3.8.1	Symbolic Execution.....	27
3.8.2	Theorem Proving	27

3.9	Difference between Static Analysis and Static Testing	27
3.10	Dynamic Analysis Techniques	28
3.10.1	Instrumentation Based	28
3.10.2	VM Profiling Based	28
3.10.3	Aspect Oriented Programming (AOP)	29
3.11	Test Data Generation Techniques	29
3.11.1	Feedback Directed Random Testing (FDRT)	29
3.11.2	Program Exploration (PEX)	29
3.11.3	Directed Automated Random Testing (DART)	29
3.12	Critical Evaluation on Existing Approaches	30
3.13	Remarks	38
3.14	Summary	38

Chapter 4: Methodology

4.1	Introduction	40
4.2	Empirical Study Definition & Design	42
4.2.1	Model Construction	42
4.2.2	Bug Prediction	45
4.2.3	Reports Generation	48
4.3	Program Analysis & Execution	50
4.3.1	Static Analysis	50

4.3.2	Dynamic Analysis	51
4.3.3	Satisfiability Modulo Theories (SMT)	51
4.3.4	Concrete Execution	52
4.3.5	Symbolic Execution	52
4.3.6	Concolic Execution	55
4.4	Summary	58

Chapter 5: Implementation

5.1	Introduction	60
5.2	Model Construction	60
5.2.1	Roslyn	60
5.2.2	Roslyn Core APIs	60
5.2.3	Working with Syntax	63
5.2.4	Working with Semantic	68
5.3	Test Data Generation	69
5.3.1	Test Data Generation Algorithm	69
5.3.2	Path Exploration Algorithm	70
5.3.3	Pattern Matches	71
5.3.4	Z3 Theorem Prover	73
5.4	Result Generation	74
5.4.1	Result Generation Algorithm	74

5.4.2	Visual Studio Unit Testing Frameworks	75
5.5	Experimental Results	76
5.6	Comparison of Results with Previous Works	78
5.7	Testing the Results.....	84
5.8	Results Discussions.....	85
5.9	Summary	85
 Chapter 6: Conclusion & Future Work		
6.1	Conclusion	87
6.2	Limitations of Research Approach	87
6.3	Future Works	88
List of Sources & References		89
List of Publications		95

LIST OF TABLES

Table No.	Title	Page
3.1	Difference between Static & Dynamic Testing	22
3.2	Abstract interpretation rules for the < operator over the domain {-, 0, +, ?}	26
3.3	Comparison of Static Analysis & Testing	28
3.4	Summary of Static Analysis Techniques	31
3.5	Summary of Dynamic Analysis Techniques	34
3.6	Summary of Test Data Generation & Program Exploration Techniques	36
4.1	Examples of Token Values	43
4.2	Concolic Execution Tools	58
5.1	Comparing RANDOOP for .NET with Smart Unit Tests by Bug Type Factor	79
5.2	Comparing RANDOOP for .NET with Smart Unit Tests by Accuracy Factor	79

LIST OF FIGURES

Figure No.	Caption	Page
1.1	Relation between Quality and Amount of Testing	3
1.2	Research Approach	5
2.1	SDLC Phases	10
2.2	Testing Approaches	12
2.3	Testing Levels	13
3.1	Mistaken use of <lhs> = <rhs> in an if-else construct	22
3.2	Intentional use of <lhs> = <rhs> in an if-else construct	22
4.1	Overview of the Empirical Study Design	41
4.2	Grammar for Assignment Statement	44
4.3	AST for Simple Assignment Statement	44
4.4	Flowchart Diagram for AST Model Construction	45
4.5	AST for a simple assignment statement that contains a bug	46
4.6	Flowchart Diagram for Bug Finding	47
4.7	Flowchart Diagram for Result Generation	50
4.8	Simple example using Concolic Execution	57
5.1	Roslyn Compiler Pipeline	61
5.2	Roslyn Compiler API	61
5.3	Roslyn Language Service	62
5.4	Sample Roslyn Syntax Tree	67
5.5	Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types	80
5.6	Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types	80
5.7	Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types	81
5.8	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Generated Unit Tests (1 sec)	81
5.9	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Used Unit Tests (1 sec)	82

List of Figures (Cont'd)

5.10	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Unused Unit Tests (1 sec)	82
5.11	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Generated Unit Tests (10 sec)	83
5.12	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Used Unit Tests (10 sec)	83
5.13	Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Unused Unit Tests (10 sec)	84

LIST OF ABBREVIATIONS

Symbols	Nomenclatures	Page
API	Application Programming Interface	16
AST	Abstract Syntax Tree	3
ATP	Automated Theorem Prover	56
CFG	Control Flow Graph	24
CFG	Context Free Grammar	43
CIL	Common Intermediate Language	60
CPU	Central Processing Unit	21
DART	Directed Automated Random Testing	29
FDRT	Feedback Directed Random Testing	29
IDE	Integrated Development Environment	62
IL	Intermediate Language	61
MUT	Method Under Test	4
OOP	Object Oriented Programming	29
PEX	Program Exploration	29
PUTs	Parameterized Unit Tests	9
QA	Quality Assurance	2
SDLC	Software Development Life Cycle	2
STLC	Software Testing Life Cycle	19
SMT	Satisfiability Modulo Theories	51
SUT	Smart Unit Tests	7
UAT	User Acceptance Testing	22

CHAPTER 1: INTRODUCTION

1.1 Introduction

In Software Engineering, the system development life cycle (SDLC) is the process for planning, creating, testing and deploying an information system (Hlongwane, 2005). There are usually six phases in this cycle: requirement analysis, design, development and testing, implementation, documentation and evaluation.

Software testing plays a vital role in the SDLC, which provides stakeholders with information about the quality of the software product or service under test (Kaner, 2016). Software testing can also provide an objective, independent view of the software to allow the business to appreciate and understand the risks of software implementations. Test techniques include the process of executing a program or application with the intent of finding software bugs (errors or other defects), and verifying that the software product is fit for use ("Software Testing," n.d.).

During the development of a program or software, a range of measures are taken to ensure that the program is tested prior to the release and distribution of the program. These measures are aimed at reducing the number of bugs in the program in order to improve the quality of the program (Pan, 1999; "Software Testing," n.d.). A bug in a software is an unintended state in the executing program that results undesired behavior (Mittal & Aditya, 2015). Regardless of these measures, the program may still contain bugs.

Detecting software bugs are tricky and not easy to accomplish, there are two approaches to detect the software bugs before the developers can fix them: the first one named **Static program analysis** which is the process of analyzing the computer software without executing programs (Nachtigall et al., 2019; Wichmann et al., 1995). The second one is named **Dynamic program analysis** which is the process to analyze the computer software by executing programs on a real or virtual processor, in contrast with static program analysis, whereas the analysis is performed without program execution (Myers et al., 2011).

A bug-free software is the main reason to implement the software testing. Quality Assurance (QA) or Software Testing is crucial because it identifies errors/bugs from a

system at the beginning. Considering the problems in the base helps to turn improvement in the quality of product and brings confidence in it.

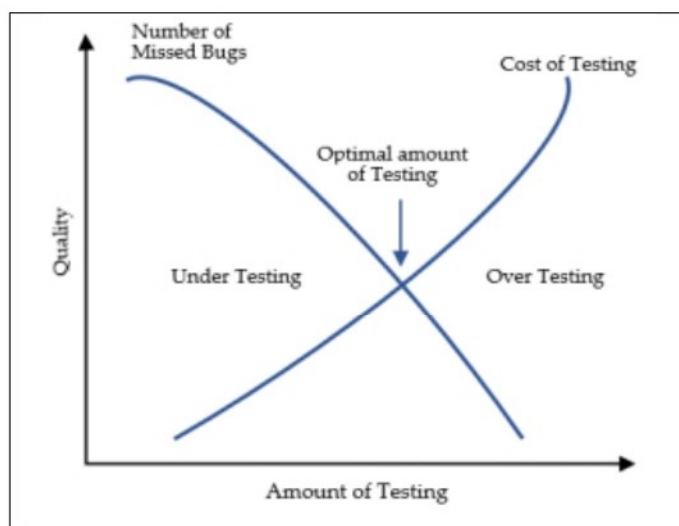


Figure (1.1) Relation between Quality and Amount of Testing (Jamil et al., 2017)

According to Figure 1.1, software testing is an important component of software QA. The importance of testing can be considered from life-critical software (e.g., flight control), testing which can be highly expensive because of risk regarding schedule delays, cost overruns, or outright cancellation (Jamil et al., 2017).

Hence, this research will focus on challenges in predicting the software bugs in the early stage by scanning the Abstract Syntax Tree (AST) for the tested method.

1.2 Motivation

Software maintenance makes the corrective measures needed to fix software bugs after the bugs are reported by end users. Fixing the software bugs after deployment of the program hampers the usability of the deployed program and increases the cost of the software maintenance services. A better solution would be to detect and fix the software bugs prior to releasing the program.

Introducing a developing approach using static program analysis with an AST model will assist to predict the software bugs in the early stage. This will reduce the number of the expected bugs before releasing the software. However, using the AST model will allow exploring all possible paths for the tested software, which implies to

maximize the test code coverage. This leads to discovering all possible bugs, and increases the software quality too.

Furthermore automating the unit tests for a particular software will reduce the human efforts and money required for the software testing, also it increases the quality of the software after release.

1.3 Problem Statement

As mentioned before any software may contain bugs during its life cycle, which of course affects the quality of the software.

Code coverage is one of the critical problems in software testing because the prediction of all different use cases - that program may contain - is very complex.

Increasingly, discovering bugs in software by using the manual process requires a solid and robust software testing that costs the companies a lot of money. Also the time required for writing plenty of unit tests for a software needs an exhaustive human effort to accomplish this process.

1.4 Objectives

The objectives of this research relate to enhancing the quality of the software. In more detail the objectives are as follows:

1. To Increase the software quality, by predicting the software bugs at the early stage of the SDLC.
2. To develop a technique using static analysis and AST model to predict software bugs effectively.
3. Toward to achieve maximum code coverage by traversing the entire AST for the method under test (MUT).
4. To reduce the time required for discovering the software bugs and money cost of the software testing by automating the unit tests.

1.5 Scope & Limitations

The scope of this research is limited to predict the static bugs in the early stage of software development. Because the important thing for now is, to create a proof of concept for the proposed technique and get good results compared with alternative static analysis techniques. So, there are some limitation for this thesis that needs to be addressed in the future work as follows:

- Predict all static bugs that the software may have, so all the bugs that depend on the environment are out of scope.
- Predict the software bugs in the procedural programming languages.

1.6 Research Approach

In this research there are required steps to accomplish the research methodology as shown in Figure 1.2, starts from identifying the problem that the research addresses and ends with the evaluation.

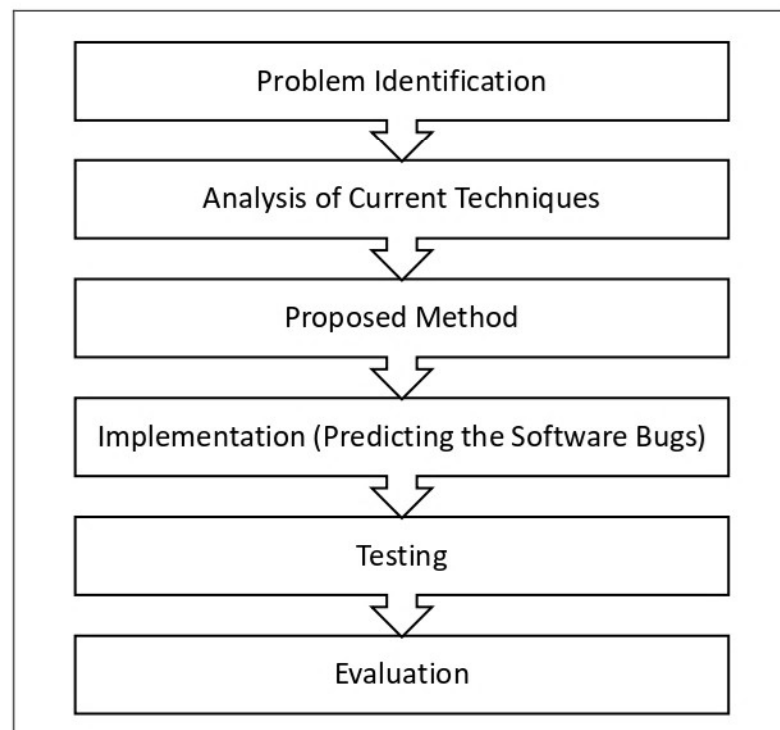


Figure (1.2) Research Approach

1.6.1 Problem Identification

The research will start with identifying the problem, that all the software in industry may contain bugs. The role of the software testing is to avoid such bugs in production. So, predicting such bugs in the early stages of the SDLC will save time and cost that is spent by the companies, also it improves the software quality.

1.6.2 Analysis of Current Techniques

The research will review both static and dynamic analysis techniques that are available, also it will address the techniques that had been used to generate the test data. This includes analyzing each technique, the methodology that has been used, and strengths and weaknesses.

1.6.3 Proposed Method (Research Method)

Based on analysis of the current techniques, this step will be proposed in order to achieve the research objectives. This will address the overall design structure, including all the phases required to predict the software bugs.

1.6.4. Implementation (Predicting the Software Bugs)

In this step, the researcher will discuss the implementation details of applying the proposed method with all the criteria and algorithms that have been described in the research method, including test data generation, program exploration and result generation.

1.6.5 Testing

Last but not least, the research will test the prototype of the proposed method, after the implementation to compute and measure the results according to some factors.

1.6.6 Evaluation

Finally, the research will evaluate the concluded results, compare them with the current techniques and discuss the outcomes to ensure that the proposed technique achieve good results.

1.7 Research Contribution

A hybrid technique has been proposed to overcome the challenges and issues of predicted software bugs without human interference.

The key contributions of this thesis are as follows:

- Proposing hybrid technique for predicting the software bugs using static analysis and AST.
- Improving the selection and generation for the test data that will predict the bugs more accurately.

1.8 Thesis Organization

The rest of thesis is organized into five chapters:

Chapter 2 discusses a background of software testing including testing types, methods, approaches, levels and how it affects the software quality. Also it highlights the research problem, objectives and motivations.

Chapter 3 covers the literature survey of static and dynamic analysis techniques. Including how they work, also their strengths and weaknesses.

Chapter 4 discusses the methodology and design of the proposed Smart Unit Tests (SUTs) using AST. The proposed technique for selecting data under test and the path exploration are introduced and discussed mathematically in this chapter too.

Chapter 5 discusses the implementation for the proposed approach, including the tools that are used, also research experiment results, will be analyzed and discussed.

Chapter 6 summarizes the research findings, research conclusion, and the possible future work for this research.

CHAPTER 2: BACKGROUND

2.1 Introduction

This chapter discusses several definitions and concept techniques that related to the software testing, including the importance of the testing as well as testing methods, approaches and levels.

In addition, unit testing is discussed in detail including the advantages and disadvantages of the unit testing, Parameterized Unit Tests (PUTs) and various unit testing tools and frameworks.

2.2 Testing

In systems engineering, information systems and software engineering, the SDLC, also referred to as the application development life-cycle, is a process for planning, creating, testing, and deploying an information system (Hlongwane, 2005). The SDLC framework provides a sequence of activities for system designers and developers to follow. It consists of a set of phases in which each phase of the SDLC uses the results of the previous one (Everatt & McLeod Jr., 2007; US Department of Justice, 2003).

The SDLC adheres to important phases that are essential for developers such as planning, analysis, design, implementation, testing, deployment and maintenance as shown in Figure 2.1.

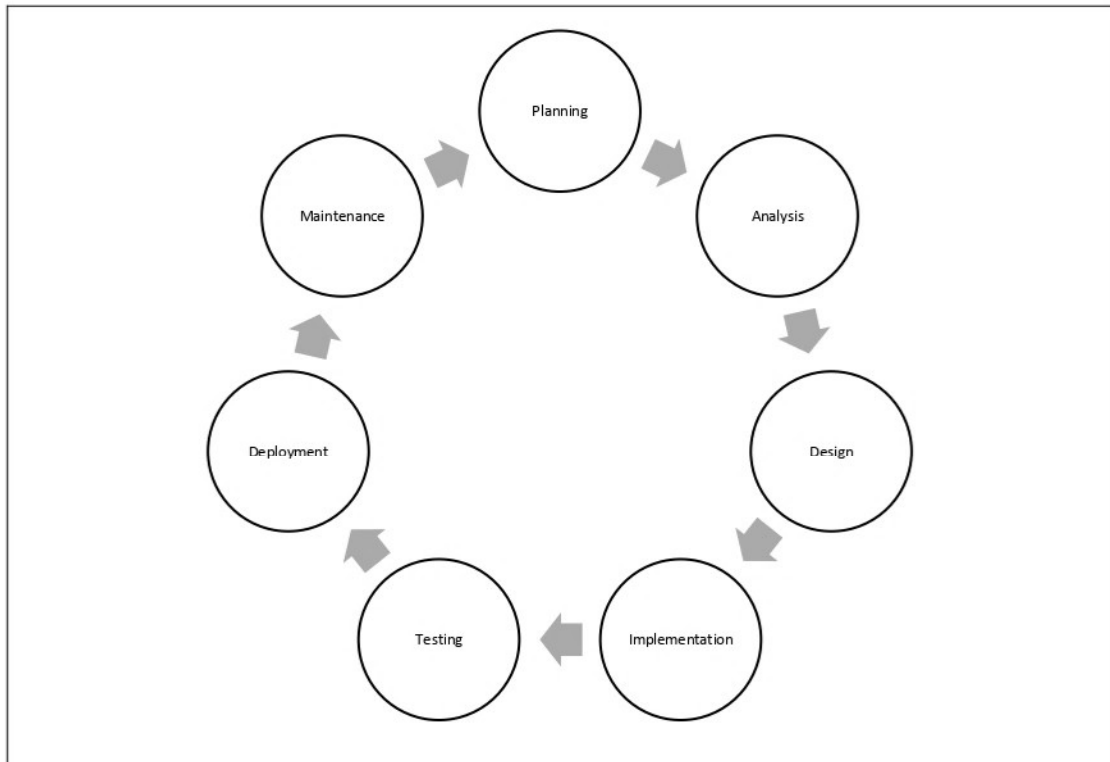


Figure (2.1): SDLC Phases

Software testing is unavoidable part of the SDLC. It is a process, to evaluate the functionality of a software application with an intent to find whether the developed software met the specified requirements or not and to identify the defects to ensure that the product is defect free in order to produce the quality product (Rajkumar, 2021).

2.2.1 Testing Types

Any software could be tested by one of the following testing types:

Manual Testing is the process of testing the software manually to find the defects. Testers should have the perspective of end users and to ensure all the features are working as mentioned in the requirement document. In this process, testers execute the test cases and generate the reports manually without using any automation tools (“Manual Testing,” n.d.).

Automation Testing is the process of testing the software using automation tools to find the defects. In this process, testers execute the test scripts and generate the test results automatically by using automation tools (Huizinga & Kolawa, 2007).

2.2.2 Testing Methods

Whether the software is tested manually or automatically using some tools, is not less important to verify and validate the final product. Here where the testing methods come into play to ensure the software quality.

There are two testing methods are as follows (*Difference between Verification and Validation*, n.d.):

Static Testing (Verification) is a static practice of verifying documents, design, code and program. It includes all the activities associated with producing high quality software: inspection, design analysis and specification analysis. It is a relatively objective process.

Dynamic Testing (Validation) is the process of evaluating the final product to check whether the software meets the customer expectations and requirements. It is a dynamic mechanism of validating and testing the actual product.

2.2.3 Testing Approaches

The first step in the testing process is to generate test cases. The test cases are developed using various testing approaches or techniques, for the effective and accurate testing. The major testing approaches as shown in Figure 2.1 are as follows:

White Box Testing is significantly effective as it is the method of testing that not only tests the functionality of the software but also tests the internal structure of the application. This kind of testing can be applied to all levels including unit, integration or system testing (Limaye Milind G., 2009; Saleh, 2009).

Black Box Testing is a testing technique that essentially tests the functionality of the application without going into its implementation level details (Patton, 2005). This technique can be applied to every level of testing within SDLC.

Gray Box Testing is the combination of the White Box and Black Box Testing techniques serving the advantages of both. The tester will often have access to both "the source code and the executable binary" (Ransome & Misra, 2018).

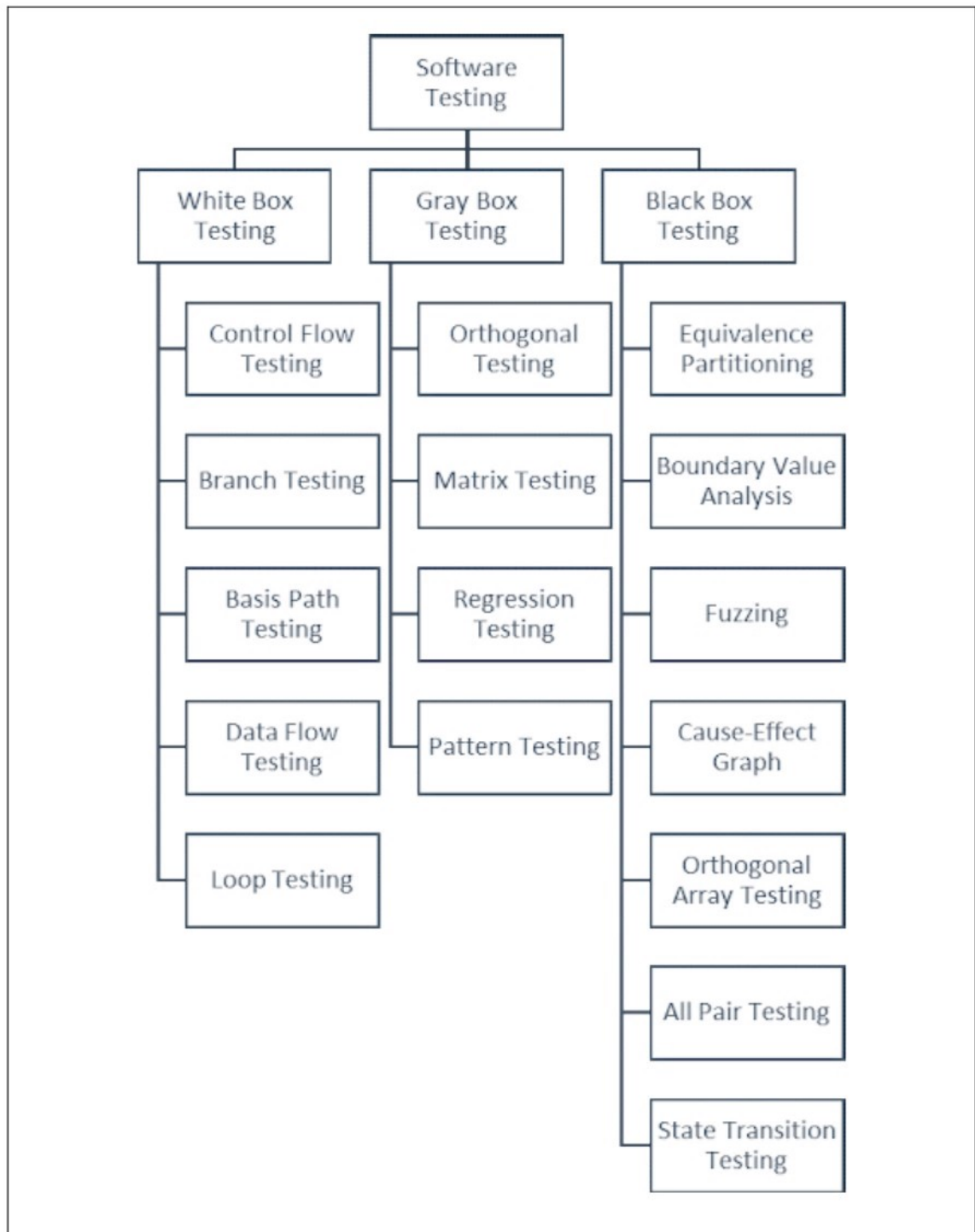


Figure (2.2) Testing Approaches

2.2.4 Testing Levels

Although the testing approaches are needed for test cases development, it is very important to mention the testing levels and how the testing approaches fit with those levels. There are at least three levels of testing as shown in Figure 2.2 listed by chronological order as the following (“Software Testing,” n.d.):

Unit Testing refers to tests that verify the functionality of a specific section of code, usually at the function level. In an object-oriented environment, this is usually at the class level and the minimal unit tests include the constructors and destructors.

Integration Testing is any type of software testing that seeks to verify the interfaces between components (modules) against a software design.

System Testing tests a completely integrated system to verify that the system meets its requirements.

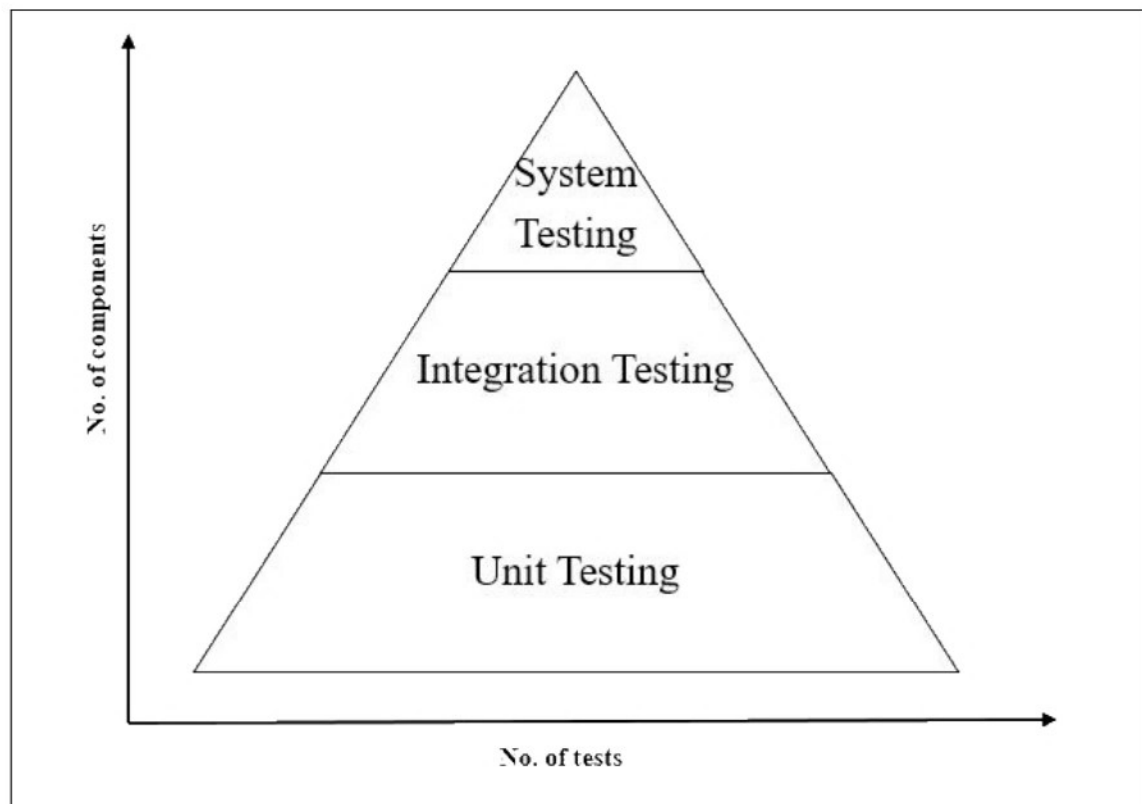


Figure (2.3) Testing Levels

These types of tests are usually written by developers as they work on code (white-box style), to ensure that the specific function is working as expected. One function might have multiple tests, to catch corner cases or other branches in the code.

Unit testing cannot verify the functionality of a piece of software alone, but rather is used to ensure that the building blocks of the software work independently from each other.

2.3 Unit Testing

Unit tests are typically automated tests written and run by software developers to ensure that a unit of an application meets its design and behaves as intended (Hamill, 2004). To isolate issues that may arise, each test case should be tested independently. Stubs, mocks and fakes can be used to assist testing a module in isolation. Examples of unit tests written in C# are as follows:

Example 1:

```
[Test]
public void AddTwoPositiveNumbers()
{
    // Arrange
    int a = 2;
    int b = 3;
    // Act
    int c = a + b;
    // Assert
    Assert.True(c > 0);
    Assert.Equal(5, c);
}
```

Example 2:

```
[Test]
public void MultiplNumberByZeroIsZero()
{
    // Arrange
    int a = 2;
    // Act
    int b = a * 0;
    // Assert
    Assert.True(c == 0);
}
```

Example 3:

```
[Test]
public void DivideByZeroShouldThrowsException()
{
    // Arrange
    int a = 5;
    int b = 0;
    // Act & Assert
    Assert.Throws<DivideByZeroException>(() =>
    {
        var c = a / b;
    });
}
```

2.3.1 Parameterized Unit Tests

Writing and maintaining unit tests can be made faster by using PUTs. These allow the execution of one test multiple times with different input sets, thus reducing the code duplication. Unlike unit tests, which are usually closed methods and test invariant conditions, PUTs have been supported by JUnit, xUnit... etc. In recent years support was added for writing more powerful unit tests, leveraging the concept of theories, test cases that execute the same steps, but using test data generated at runtime, unlike the regular parameterized tests that use the same execution steps with input sets that are predefined (*Getting Started with XUnit.Net (Desktop)*, n.d.; *Parameterized Tests*, n.d.; *Theories*, n.d.). An example written in C# shows PUTs in action is as follows:

Example 4:

```
[DataRow(2, 3, 5)]  
[DataRow(7, 1, 8)]  
[DataRow(5, 6, 11)]  
public void AddTwoPositiveNumbers(int a, int b, int result)  
{  
    // Arrange  
  
    // Act  
    int c = a + b;  
  
    // Assert  
    Assert.True(c > 0);  
    Assert.Equal(result, c);  
}
```

2.3.2 Advantages & Disadvantages of Unit Testing

There are some advantages for using unit testing are as follows:

- Unit testing finds problems early in the development cycle.
- Unit testing allows the programmer to refactor code or upgrade system libraries at a later date, and make sure the module still works correctly.
- Unit testing may reduce uncertainty in the units themselves and can be used in a bottom-up testing style approach. By testing the parts of a program first and then testing the sum of its parts, integration testing becomes much easier.
- Unit testing provides a sort of living documentation of the system. Developers looking to learn what functionality is provided by a unit, and how to use it, can look at the unit tests to gain a basic understanding of the unit's interface (API).

On the other hand, there are some disadvantages or limitations for using unit testing as follows:

- Testing will not catch every error in the program, because it cannot evaluate every execution path in any but the most trivial programs.

- It will not catch integration errors or broader system-level errors

- Writing the unit tests is the difficulty of setting up realistic and useful tests (Kolawa, 2009).

- Unit testing embedded system software presents a unique challenge, because the software is being developed on a different platform than the one it will eventually run on (Kucharski, 2011).

2.3.3 Unit Testing Frameworks

There are many unit testing frameworks available that simplify the process of unit testing such as xUnit, NUnit, JUnit, MSTests, Typemock Isolator, JustMock... etc. Some of them are open source and some are commercial solutions (Hamill, 2004).

In general it is possible to perform unit testing without support of a specific framework by writing client code that exercises the units under test and uses assertions, exception handling or other flow control mechanisms to signal failure.

2.4 Summary

This chapter has presented the background of software testing in general, the role that plays in SDLC. It also described how software testing is very important for QA. After that, it discusses related terminologies and concepts including: testing types, methods, approaches and levels. Finally, it ends with the unit testing in more details, including the importance, tools and frameworks.

CHAPTER 3: LITERATURE REVIEW

3.1 Introduction

This chapter reviews the related literature, it begins with introducing both static testing and dynamic testing, and including the techniques used for each. Then, it shows the differences between them. After that, it discusses the static analysis techniques and explains each one of them. Finally, it ends up with the research gap.

3.2 Static Testing

Static testing is a type of a software testing method which is performed to check the defects in software without actually executing the code of the software application. Rather it manually checks the code, requirement and design documents to find errors (“Software Testing | Static Testing,” 2019).

Static testing is performed in the early stage of development to avoid errors as it is easier to find sources of failures and it can be fixed easily. The errors that cannot be found using Dynamic Testing, can be easily found by Static Testing.

The main objective of this testing is to improve the quality of software products by finding errors in the early stages of the SDLC. This type of testing is also known as verification testing.

Static testing involves manual or automated reviews of the documents. This review is done during an initial phase of testing to catch defects early in Software Testing Life Cycle (STLC). It examines work documents and provides review comments. Examples of Work documents as follows:

- Requirement specifications
- Design document
- Source Code
- Test Plans
- Test Cases
- Test Scripts
- Help or User document

3.3 Static Testing Techniques

There are different techniques available for the static testing as the following (Myers et al., 2011):

3.3.1 Informal Reviews

This type of review does not follow any process to find errors in the document. You just review the document and provide informal comments.

3.3.2 Technical Reviews

A team consisting of peers, review the technical specification of the software products and check whether it is suitable for the project. They try to find any conflict in the specifications and standards followed. This review concentrates mainly on the technical documentation related to the software.

3.3.3 Walkthrough

The author of the work product explains the product to his team, participants can ask questions if any, notes of the review comments are written.

3.3.4 Inspection

The main purpose is to find defects and the meeting is led by a trained moderator. Reviewers have a checklist to review the work product, they record the defect and inform the participants to rectify those errors.

3.3.5 Static Code Review

This is a systematic review of the software source code without executing the code. It checks the code syntax, standards, optimization, etc. This is also known as white box testing, and this review can be done at any point during the development.

3.4 Dynamic Testing

Dynamic testing is a type of a Software Testing method which is performed to check the defects in software with executing the code of the software application. It checks for functional behavior of the software system, memory / Central Processing Unit (CPU) usage and overall performance of the system (Myers et al., 2011).

The main objective of this testing is to confirm that the software product works in conformance with the business requirements. This type of testing is also known as validation testing.

Dynamic testing executes the software and validates the output with the expected outcome. It can be performed at all levels of testing and it can be either black or white box testing.

3.5 Dynamic Testing Techniques

There are different techniques available for the dynamic testing as the following (Myers et al., 2011):

3.5.1 Unit Testing

Under Unit Testing, individual units or modules are tested by the developers. It involves testing of source code by developers.

3.5.2 Integration Testing

Individual modules are grouped together and tested by the developers. The purpose is to determine what modules are working as expected once they are integrated.

3.5.3 System Testing

System Testing is performed on the whole system by checking whether the system or application meets the requirement specification document.

3.5.4 User Acceptance Testing (UAT)

Acceptance Testing is performed from user point of view at user's end. Also, Non-functional testing like performance, security testing fall under the category of dynamic testing.

3.6 Difference between Static Testing and Dynamic Testing

However, both Static Testing and Dynamic Testing are important for the software application. There are a number of strengths and weaknesses associated with both types of testing which should be considered while implementing these testing on code as shown in Table 3.1.

Table (3.1) Difference between Static and Dynamic Testing (*Static Testing vs. Dynamic Testing*, n.d.)

Testing Type Property	Static Testing	Dynamic Testing
Execution	Testing was done without executing the program	Testing is done by executing the program
Process	This testing does the verification process	Dynamic testing does the validation process
Defects	Static testing is about prevention of defects	Dynamic testing is about finding and fixing the defects
Outcomes	Static testing gives an assessment of code and documentation	Dynamic testing gives bugs/bottlenecks in the software system
Input	Static testing involves a checklist and process to be followed	Dynamic testing involves test cases for execution
Testing Time	This testing can be performed before compilation	Dynamic testing is performed after compilation
Purpose	Static testing covers the structural and statement coverage testing	Dynamic testing techniques are Boundary Value Analysis & Equivalence Partitioning

Table 3.1: Continued

Cost	Cost of finding defects and fixing is less	Cost of finding and fixing defects is high
Return on investment	Return on investment will be high as this process involved at an early stage	Return on investment will be low as this process involves after the development phase
Recommendations	More reviews comments are highly recommended for good quality	More defects are highly recommended for good quality
No. of Meetings	Requires loads of meetings	Comparatively requires lesser meetings

3.7 Static Analysis Techniques

(Gosain & Sharma, 2015) classified the techniques for the static analysis approach as follows:

3.7.1 Syntactic Pattern Matching

(Pendergrass et al., 2013) discussed the general idea of syntactic pattern matching. This technique is the fastest and easiest technique for static analysis, but it provides a little confidence in program correctness and can result in a high number of false alarms. This technique is based on the syntactic analysis of a program. In terms of bug finding, this technique defines a set of program constructs that are potentially dangerous or invalid and then searching in the program's AST for instances of any of these constructs.

A common example for C# programs is a pattern to prevent the use of an assignment expression, $\langle \text{lhs} \rangle = \langle \text{expr} \rangle$, as the condition of an if-else block illustrated by Figure 3.1, in which a program intends to use the comparison operator “==” to determine whether the two sides of the expression are equal rather than using the assignment operator “=” to assign the value of the right hand side to the variable given by the left hand side.

```

if (n = 4)
{
    ...
}
else
{
    // will never get here
}

```

Figure (3.1) Mistaken use of `<lhs> = <rhs>` in an if-else construct.

However, this pattern does represent perfectly valid C# code and may be used intentionally by a programmer, as shown in Figure 3.2, to assign a variable and branch based on whether the new value is false.

```

if (n = stk.Peek())
{
    // do this if n = true, i.e., if the top element of the stack is true
}
else
{
    // do this if n = false, i.e., if the top element of the stack is false
}

```

Figure (3.2) Intentional use of `<lhs> = <rhs>` in an if-else construct

3.7.2 Data Flow Analysis

(Kam & Ullman, 1976; Kennedy, 1981; Kildall, 1973) discussed the general of the data flow analysis. This technique is the most popular one. It constructs a graph-based representation of the program called control flow graph (CFG), and writes data flow equations for each graph node. These equations are then repeatedly solved to calculate output from input at each node until the system of equations stabilizes or reaches a fixed point. The major data flow analysis used are reaching definitions, live variable analysis, available expression analysis and very busy expression analysis.

At each source code location, data flow analysis records a set of facts about all variables currently in scope. In addition to the set of facts to be tracked, the analysis defines a “kills” set and a “gens” set for each block. The “kills” set describes the set of facts that are invalidated by execution of the statements in the block, and the “gens” set describes the set of facts that are generated by the execution of the statements in the block. To analyze a program, the analysis tool begins with an initial set of facts and updates it according to the “kills” set and “gens” set for each statement of the program in sequence. Although mostly used in compiler optimization (Kam & Ullman, 1976; Kennedy, 1981), data flow analysis has been an integral part of most static analysis tools (Bush et al., 2000; Das et al., 2002).

3.7.3 Abstract Interpretation

(Cousot & Cousot, 1979, 1977) formalized the abstract interpretation technique. It is a theory of semantics approximation of a program based on monotonic functions over ordered sets, especially lattices.

Abstract interpretation is a generic term for a family of static analysis techniques that includes both type systems and data flow analysis, among others. In abstract interpretation, a program’s variables are assigned values from an abstract domain, and the program is executed on the basis of modified semantics for how each language construct applies in this new domain. For example, whereas an *int* variable may typically take on any concrete integer value in the range $[-2^{31}, 2^{31})$, in abstract interpretation the variable may be given a value of -, 0, +, or ? indicating only the sign of the variable (where “?” indicates an unknown or indeterminate value). Operators such as < are given meaning for this new domain, as shown in Table 3.2. Moving away from concrete values and operators allows automated analysis tools to evaluate programs’ meanings in terms of the higher-level abstract domains. This ensures that the analysis will actually terminate on all input programs (Pendergrass et al., 2013).

Table (3.2) Abstract interpretation rules for the < operator over the domain {-, 0, +, ?} (Pendergrass et al., 2013)

Input 1	Operator	Input 2	Is	Result
-	<	-	=	?
-	<	0	=	True
-	<	+	=	True
-	<	?	=	?
0	<	-	=	False
0	<	0	=	False
0	<	+	=	True
0	<	?	=	?
+	<	-	=	False
+	<	0	=	False
+	<	+	=	?
+	<	?	=	?
?	<	-	=	?
?	<	0	=	?
?	<	+	=	?
?	<	?	=	?

Abstract interpretation is a powerful tool in program analysis because it can be used to verify many important program correctness properties, including memory, type, and information-flow safety. The primary challenge to applying abstract interpretation is the design of the abstract domain of reasoning. If the domain is too abstract, then precision is lost, resulting in valid programs being rejected. If the domain is too concrete, then analysis may become computationally infeasible.

3.7.4 Constraint-Based Analysis

(Aiken, 1999) provides an overview of constraint-based program analysis. This technique traverses a program, emitting and solving constraints describing properties of a program. This technique works in two steps. The first step, called constraint generation, produces constraints from a program text that give a declarative specification of the

desired information about the program. The second step is constraint resolution (i.e., solving the constraints) then computes this desired information.

3.8 Miscellaneous Techniques

There are some miscellaneous techniques which are related as follows:

3.8.1 Symbolic Execution

(King, 1976) described the symbolic execution technique. In this method, instead of supplying normal inputs to the program, one uses symbols representing arbitrary values. The execution proceeds as in a normal execution except that the values are now symbolic formulas over input values. As a result, the output values computed by a program are expressed as a function of the input symbolic values. The state of a symbolically executed program includes the symbolic values of program variables, a path condition (PC) and a program counter. The path condition is a quantifier-free Boolean formula over the symbolic inputs; it accumulates constraints which the inputs must satisfy in order for an execution to follow the particular associated path.

3.8.2 Theorem Proving

(Floyd, 1967; Hoare, 1969) proposed a theorem proving which is based on the deductive logic. A program statement S is represented as a triple $\{p\}S\{q\}$, where p (precondition) and q (post-condition) are logical formulas over program states. This triple is valid *iff* for a state t satisfying formula p , executing S on t yields a state t' which satisfies q . Various inference rules are then used to verify system states. One of the most famous theorem provers is Simplify which has been used in tools like ESC/Java (Das et al., 2002; Detlefs et al., 2005; Floyd, 1967; Hoare, 1969).

3.9 Difference between Static Analysis and Static Testing

Although static analysis and static testing are important for the software application, there are some advantages and disadvantages for each one of them, which are summarized in Table 3.3.

Table (3.3) Comparison of Static Analysis and Testing (Gosain & Sharma, 2015)

Analysis Type Property	Static Analysis	Static Testing
Execution	Can be applied without executing the code	Can be applied only by executing the code
Time	Can be applied early in the development process	Is applied late in the development process
Results dependency	Results do not depend on inputs	Results depend on inputs
Results Generalization	Results can be generalized for future executions	Results cannot be generalized for future executions
Cost	Less costly	Very costly
Process	Very fast process	Slow process
False-positive rate	False-positive rate is very high	False-positive rate is very less
Usage	Approximations are used	Exact results are used
Functional correctness	Cannot be used for functional correctness of programs	Can be used for functional correctness of programs

3.10 Dynamic Analysis Techniques

There are many techniques available for the dynamic analysis approach are as follows:

3.10.1 Instrumentation Based

(Larus & Ball, 1994) described that a code instrumenter is used as a pre-processor to insert instrumentation code into the target program. This instrumentation code can be added at any stage of the compilation process.

3.10.2 VM Profiling Based

(Binder et al., 2009) discussed that the dynamic analysis is carried out using the profiling and debugging mechanism provided by the particular virtual machine.

Examples include Microsoft CLR Profiler (Hilyard, 2005) for .NET frameworks and JPDA for Java SDK. These profiles give an insight into the inner operations of a program, specifically related to memory and heap usage.

3.10.3 Aspect Oriented Programming (AOP)

(Kiczales, 1997) described that AOP is a way of modularizing crosscutting concerns much like object-oriented programming (OOP) is a way of modularizing common concerns. With AOP, there is no need to add instrumentation code as the instrumentation facility is provided within the programming language by the built-in constructs.

3.11 Testing Data Generation Techniques

Alongside with the above techniques, it is important to mention the test generation techniques while they are related. The closest ones are as follows:

3.11.1 Feedback Directed Random Testing (FDRT)

(Pacheco, C., et al 2007) introduced a FDRT approach which is creating method sequences incrementally, and using the runtime information to guide the generation. RANDOOP for Java is using FDRT as a tool that automatically generates random, but meaningful unit tests for Java.

3.11.2 Program Exploration (PEX)

(Tao Xie, et al, 2009) discussed PEX which is a white-box test generated tool as a part of Microsoft Research.

PEX in nutshell using a dynamic Symbolic Execution technique with fitness-guided path exploration. The Fitness Function (Fitnex) which was introduced in PEX is a key player in the entire Microsoft PEX Framework as search strategy to reduce the amount of exhaustive search required during the path exploration process.

3.11.3 Directed Automated Random Testing (DART)

(Patrice Godefroid, et al, 2011) introduced DART which combines three techniques: automated interface extraction, automatic generation of a test driver for random testing and dynamic test generation to direct execution along alternative program paths.

3.12 Critical Evaluation on Existing Approaches

According to the reviews that are mentioned previously for static and dynamic analysis techniques, it concludes that there are some advantages and disadvantages for each approach. Tables 3.4, 3.5, 3.6 summarizes all the techniques listed before including their strengths and weaknesses.

Table (3.4) Summary of Static Analysis Techniques

Author(s), Year	Method Name	Methodology	Strength	Limitation
(Pendergrass et al., 2013)	Syntactic Pattern Matching	<ul style="list-style-type: none"> Defines a set of program constructs (pattern) that are potentially dangerous or invalid and then searching in program's AST for instances for such pattern 	<ul style="list-style-type: none"> Easy Fast 	<ul style="list-style-type: none"> Provides a little confidence in program correctness Can result in a high number of false alarms
(Kam & Ullman, 1976; Kennedy, 1981; Kildall, 1973)	Data Flow Analysis	<ul style="list-style-type: none"> Constructing a graph-based representation of the program (CFG) and writing data flow equations for each node of the graph. These equations are then repeatedly solved to calculate output from input at each node locally until the system of equations stabilizes or reaches a fixed point 	<ul style="list-style-type: none"> Most popular Mostly used in compiler optimization 	<ul style="list-style-type: none"> Have sets of data-flow values which can be represented as bit vectors are called bit vector problems, gen-kill problems, or locally separable problems

Table 3.4: Continued

(Cousot & Cousot, 1979, 1977)	Abstract Interpretation	<ul style="list-style-type: none"> • Program's variables are assigned values from an abstract domain and the program is executed on the basis of modified semantics for how each language construct applies in this new domain 	<ul style="list-style-type: none"> • Can be used to verify many important program properties including memory, type and information flow safety 	<ul style="list-style-type: none"> • If the domain is too abstract then precision is lost, resulting invalid programs been rejected • If the domain too concrete then analysis may become computationally infeasible
(Aiken, 1999)	Constraint-Based Analysis	<ul style="list-style-type: none"> • Traverses a program, emitting and solving constraints describing properties of the a program 	<ul style="list-style-type: none"> • Algorithms used in constraint resolution can be written independently of the eventual constraint system used 	<ul style="list-style-type: none"> • The precision of the analysis can be improved by combine it with data flow analysis

Table 3.4: Continued

(King, 1976)	Symbolic Execution	<ul style="list-style-type: none"> • Symbolic values are provided instead of concrete once. The execution proceeds as in normal execution except that the values are now symbolic formulas over input values. As a result the output values computed by a program are expressed as a function of the input symbolic values 	<ul style="list-style-type: none"> • Achieve high program coverage • Provide per-path correctness guarantees 	<ul style="list-style-type: none"> • Path explosion, symbolically executing all feasible program paths does not scale to large programs
(Floyd, 1967; Hoare, 1969)	Constraint-Based Analysis	<ul style="list-style-type: none"> • Based on the deductive logic 	<ul style="list-style-type: none"> • Suited for the solving the complicated program branches 	<ul style="list-style-type: none"> • Difficult to implement • Limited to the features that most of the theorem solvers provide • Not all of the program statements can be satisfied and solved

Table (3.5) Summary of Dynamic Analysis Techniques

Author(s), Year	Method Name	Methodology	Strength	Limitation
(Larus & Ball, 1994)	Instrumentation Based	<ul style="list-style-type: none"> Insert instrument code via pre-processor 	<ul style="list-style-type: none"> Gives unlimited freedom to record any event in the application 	<ul style="list-style-type: none"> Runtime overhead Dynamic instrumentations requires a recompilation Bytecode instrumentation is harder to implement
(Binder et al., 2009)	VM Profiling Based	<ul style="list-style-type: none"> Using profiling and debugging provided by a VM 	<ul style="list-style-type: none"> Simple Easy 	<ul style="list-style-type: none"> High runtime overhead

Table 3.5: Continued

(Kiczales, 1997)	Aspect Oriented Programming	<ul style="list-style-type: none">• Modularization crosscutting concerns such as logging, configurations ... etc.	<ul style="list-style-type: none">• Easy, because the level of abstraction is OOP	<ul style="list-style-type: none">• Design deployment overhead of using the framework• Extensive and contains variety of configuration and deployment options
------------------	-----------------------------	---	---	--

Table (3.6) Summary of Test Data Generation and Program Exploration Techniques

Author(s), Year	Method Name	Methodology	Strength	Limitation
(Pacheco et al., 2007)	FDRT	<ul style="list-style-type: none"> Creates method sequences incrementally Using the runtime information to guide the generation Avoid illegal inputs 	<ul style="list-style-type: none"> Achieved equal or higher code coverage in less time Revealed more errors 	<ul style="list-style-type: none"> Test data are randomly generated The classes under test need to be provided as input Avoid generating illegal and redundant inputs that dominates the space of possible test inputs
(Xie et al., 2009)	PEX	<ul style="list-style-type: none"> Dynamic Symbolic Execution (DSE) Fitness Function to guide path exploration 	<ul style="list-style-type: none"> Fitnex was introduced to reduce the amount of exhaustive search 	<ul style="list-style-type: none"> Complexity

Table 3.6: Continued

(Godefroid et al., 2005)	DART	<ul style="list-style-type: none">• Automated extraction of the interface of a program with its external environment using static source-code parsing• Automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in• Dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths	<ul style="list-style-type: none">• Testing can be performed completely automatically on any program that compiles, there is no need to write any test driver or harness code	<ul style="list-style-type: none">• Test data are randomly generated
--------------------------	------	---	---	--

3.11 Remarks

As described previously and according to the findings, there is lack in tree-based approaches. Also there is no technique used for the AST except the syntactic pattern matching which was discussed earlier. Even though, this technique only defines a set of patterns that are potentially dangerous or invalid, but provides a little confidence in program correctness that can result in a high number of false alarms.

This research proposed a developing approach for path analysis for given methods that need to be tested by using an AST to explore all the branches effectively and predict all the bugs. The proposed technique is a tree-based that solves two major issues: 1. Test data generation, 2. Program path exploration by combining the power of concolic execution - that will be explained in detail in the next chapter - and AST model representation for the program. So, the proposed approach gives the advantages of the syntactic pattern matching and more.

3.12 Summary

In software development methodology, both verification and validation processes are carried out to certify that the final software has all the requirements implemented correctly.

On other hand, static analysis techniques are quite often used in many code analysis tools in the industry, for many reasons including: code refactoring, code inspection and much more. Consequently, realized that there is a need to introduce a developing technique for static code analysis that is based on the AST model which can be constructed in almost all programming languages.

CHAPTER 4: RESEARCH METHODOLOGY

4.1 Introduction

This chapter introduces the proposal technique for bug's prediction using a tree-based approach. At the beginning, it gives an overview about the entire design for the proposed technique. Then, it explains each phase of the overall design in more detail. After that, it discusses program analysis and execution techniques. Finally, it ends up with path exploration and code coverage.

4.2 Empirical Study Definition and Design

Based on all the problems and issues that mentioned previously, this research proposed a hybrid approach for predicting the software bugs that may miss or occur for certain circumstances by making a static analysis for the source code of the methods that need to be tests, after constructing an AST model.

A suite of unit tests will automatically be generated, after exploring all the possible paths and finding out where are the places that lead the program to crash or raise a bug.

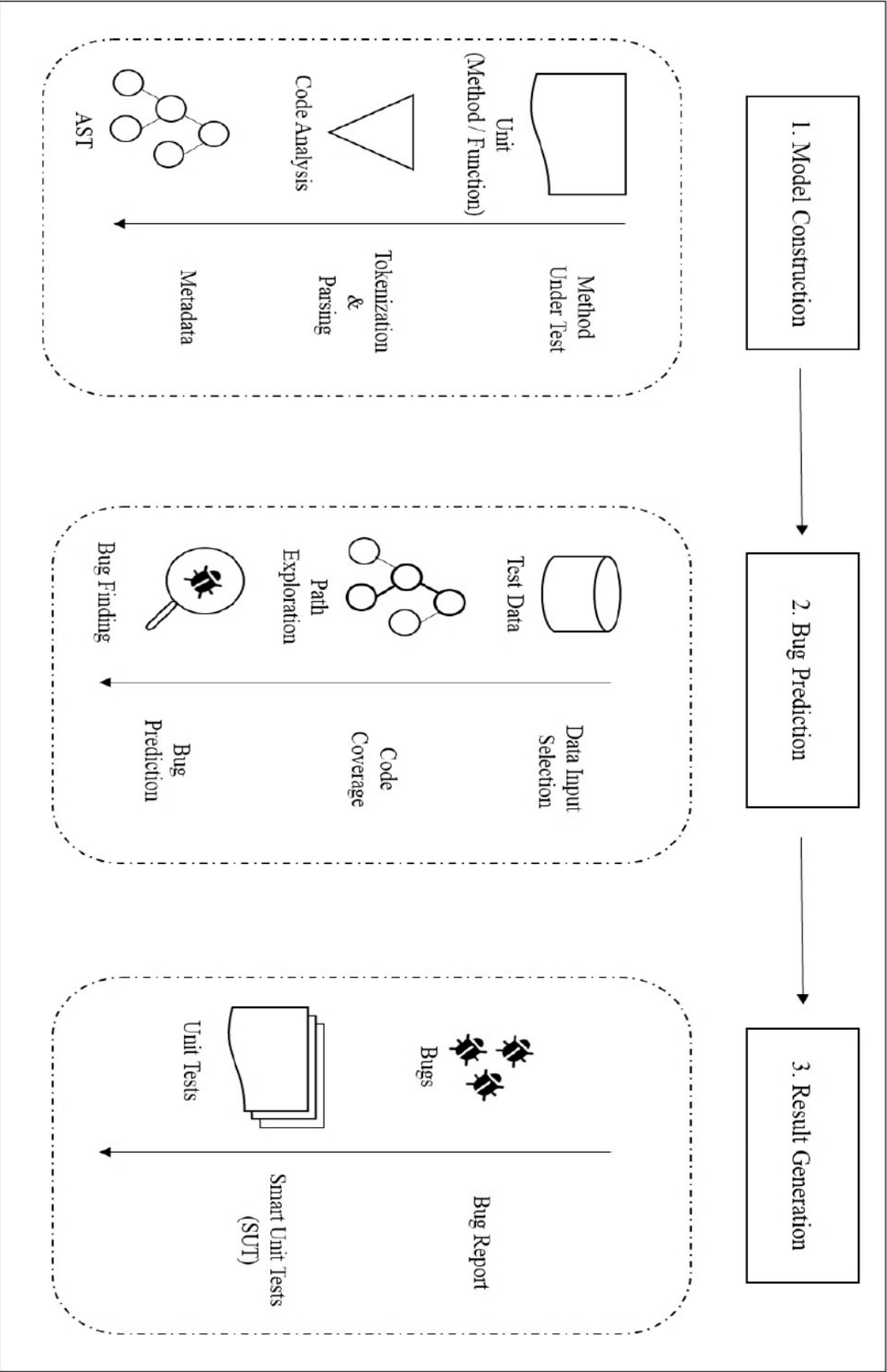


Figure (4.1) Overview of the Empirical Study Design

The empirical study design shown in Figure 4.1 consists of three phases described as the following:

4.2.1 Model Construction

The first phase is concerned about the model creation and construction, which is a very important process that assists the bug predictor to find the bugs.

The model is basically in the form of tree representation – which is in this case an AST - for the MUT. Constructing an AST needs a sort of compiler like tool to create the syntax tree for a given source code.

Analyze certain program code has two essential operations in order to construct an AST which are lexical and syntax analysis respectively.

A. Lexical Analysis (Lexing)

Lexical analysis is the process of converting a source code of a program into a sequence of tokens (lexemes), each token is a data structure that represents a particular type such as identifiers, keywords, and operators (“Lexical Analysis,” n.d.). A program that performs lexical analysis usually named lexer, tokenizer or scanner (*Anatomy of a Compiler and The Tokenizer*, n.d.).

Tokens is basically a string with identified meaning. It is structured as a pair that consists of a token name and optional token value (Aho et al., 2007; Thain, 2020).

Common token names are as follows:

- Identifier: names chosen by the programmers;
- Keyword: names reserved in the programming language;
- Separator: punctuation characters and paired delimiters;
- Operator: symbols that operate on arguments and produce results;
- Literal: numeric, logical, textual, reference literals;
- Comment line, block

Table 4.1 shows some examples of the tokens and their values.

Table (4.1) Examples of Token Values

Token Name	Token Value
Identifier	x, color
Keyword	if, while
Separator	}, (
Operator	+, -
Literal	true, "hello"
Comment	// Simple comment

Example 1: Consider the following C# expression:

```
a = b + 4 / c;
```

The lexical analysis of this expression yields the following sequence of tokens:

```
[(identifier, a), (operator, =), (identifier, b), (operator, +), (literal, 4), (operator, /),  
(identifier, c), (separator, ;)]
```

B. Syntax Analysis (Parsing)

Syntax analysis is the process of analyzing a string of symbols conforming to the rules of a formal grammar. A program that performs syntax analysis is usually named parser ("Parsing," n.d.).

Continuing with the previous example the parser will group the tokens which are generated by the lexer and construct a syntax tree for each statement in the source code with respect of the Context Free Grammar (CFG) which is a list of rules that formally describe the allowable sentences in a language, Figure 4.2 and Figure 4.3 show the CFG and an AST for the example that mentioned earlier.

```

<A - expression> ::= <A - expression> + <A1> | <A - expression> - <A1> | <A1>
<A1> ::= <A1> * <A2> | <A1> / <A2> | <A2>
<A2> ::= <identifier> | <number>
<identifier> ::= letter (letter | digit | '_' )
<number> ::= sign? digit+
<letter> ::= [a-zA-Z]
<digit> ::= [0-9]
<sign> ::= '+' | '-'

```

Figure (4.2) Grammar for Assignment Statement

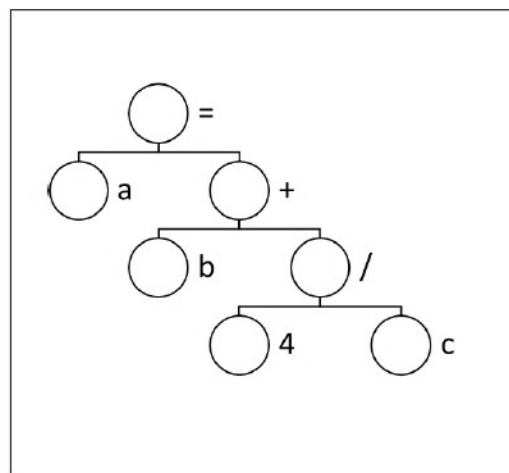


Figure (4.3) AST for Simple Assignment Statement

Example 1 shows a single assignment statement, but in reality the method may contains tons lines of code that have a giant AST, but the process is still the same. Figure 4.4 illustrates the entire process of AST model construction that explained previously.

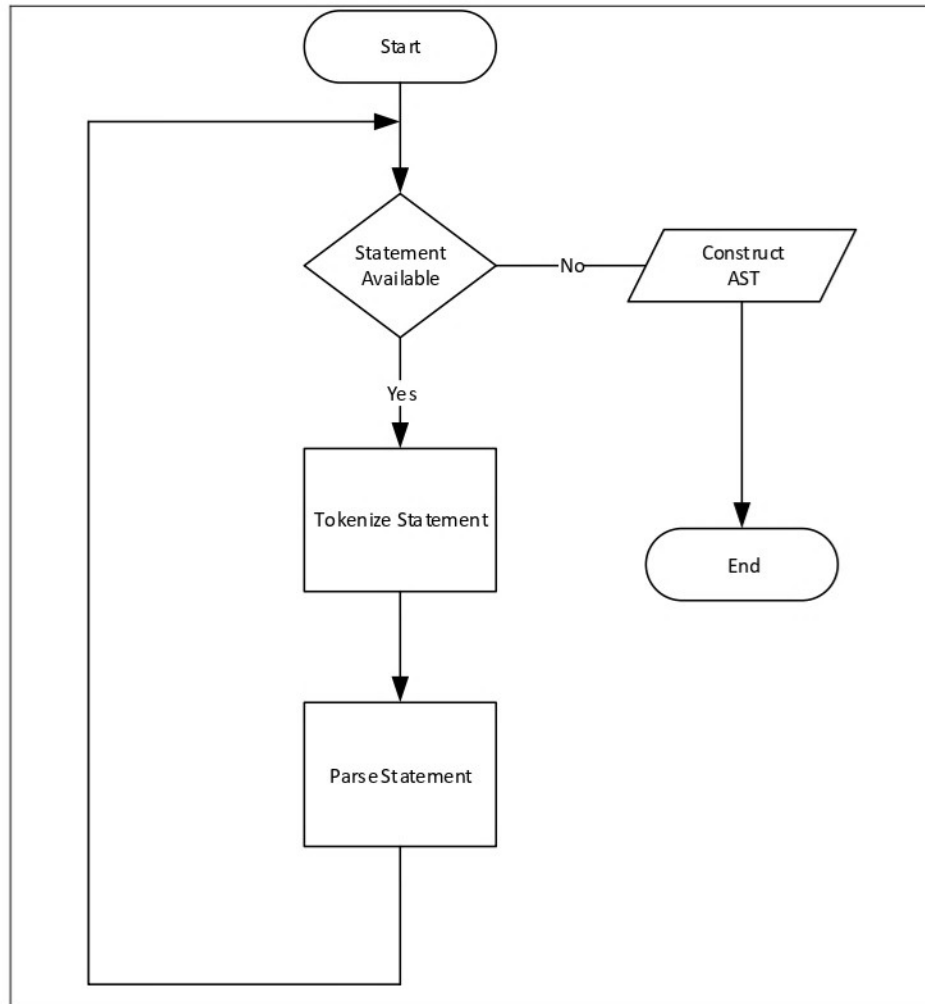


Figure (4.4) Flowchart Diagram for AST Model Construction

4.2.2 Bug Prediction

After the AST has been constructed, it is the time to start the second phase which is concerned about evaluating the syntax tree with test data and looking forward to predicating any sort of bugs that MUT has.

Choosing test data is one of the challenges in this research. As mentioned previously there are many techniques available for generating the test data, so the proposed method suggested in this research is to use concolic execution technique – that will be explained in more details in the next section - over the others because it has the advantages of the static analysis and symbolic execution technique. After the data have been generated, each path is examined with the test input to explore all the possible paths in the syntax tree.

Path exploration process indeed is another challenge, because not all the paths are feasible in almost all cases, nothing but the path exploration will be guided with the test data and path constraints to achieve a high code coverage and expected results.

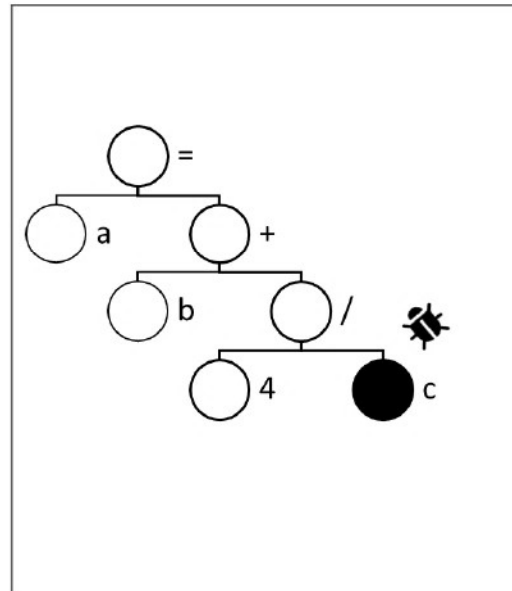


Figure (4.5) AST for a simple assignment statement that contains a bug

Continuing with Example 1, an obvious bug has been detected in $4 / c$ as shown in Figure 4.5, because all knows if the $c = 0$ the program will crash at this point and will throw *DivideByZeroException*.

The path exploration process will be discussed later including the technique that has been used, and how we can achieve a very good code coverage to ensure we addressed almost if not all the paths in a given AST. Figure 4.6 illustrates the entire process for finding the bugs in the tested method.

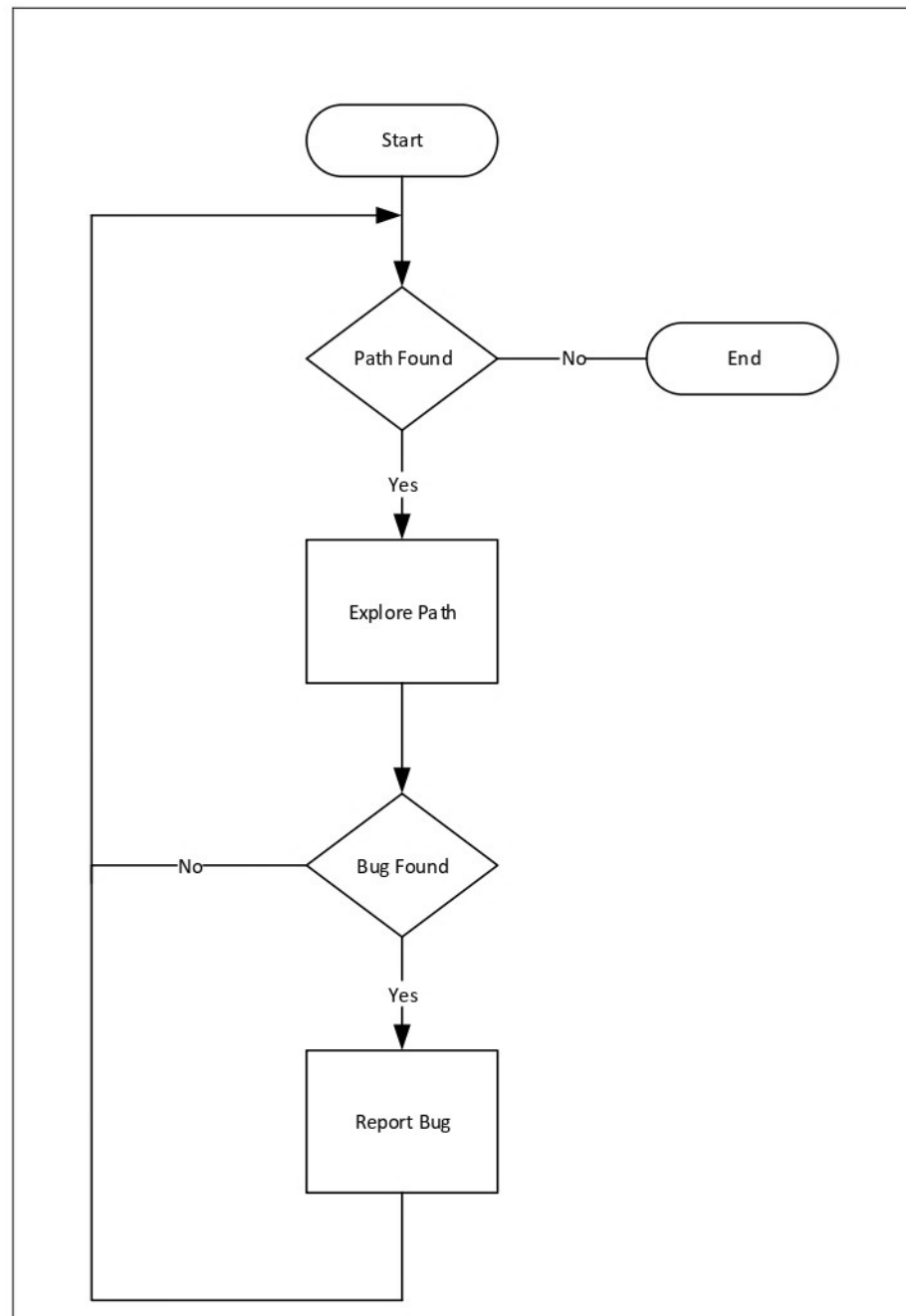


Figure (4.6) Flowchart Diagram for Bug Finding

4.2.3 Report Generation

Finally, after almost – if not all - the code branches have been discovered, and all the bugs have been detected, it is time report those bugs and generate a proper suite of unit tests that are smart enough to examine all the source code branches that have already been discovered that lead to path explosion and test fails.

Those SUTs will be automatically generated without human interference, this is very handy to the developers and saves a lot of time to discover many software bugs that many developers do not think about.

According to the Example 1 the following unit tests will be generated as follows:

```
[Test]
public void TestCase1()
{
    // Arrange
    int a;
    int b = 0;
    int c = 0;
    // Act & Assert
    Assert.Throws<DivideByZeroException>(() =>
    {
        a = b + 4 / c;
    });
}
```

```
[Test]
public void TestCase2()
{
    // Arrange
    int a;
    int b = 0;
    int c = 2;
    // Act
    a = b + 4 / c;
    // Assert
    Assert.Equal(2, a);
}
```

As seen from the above unit tests, the first unit test is the one that predicts the bug especially when $c = 0$, while the other ensures the correctness of the tested code, and doing what is supposed to do. Figure 4.7 illustrates the entire process of results generation.

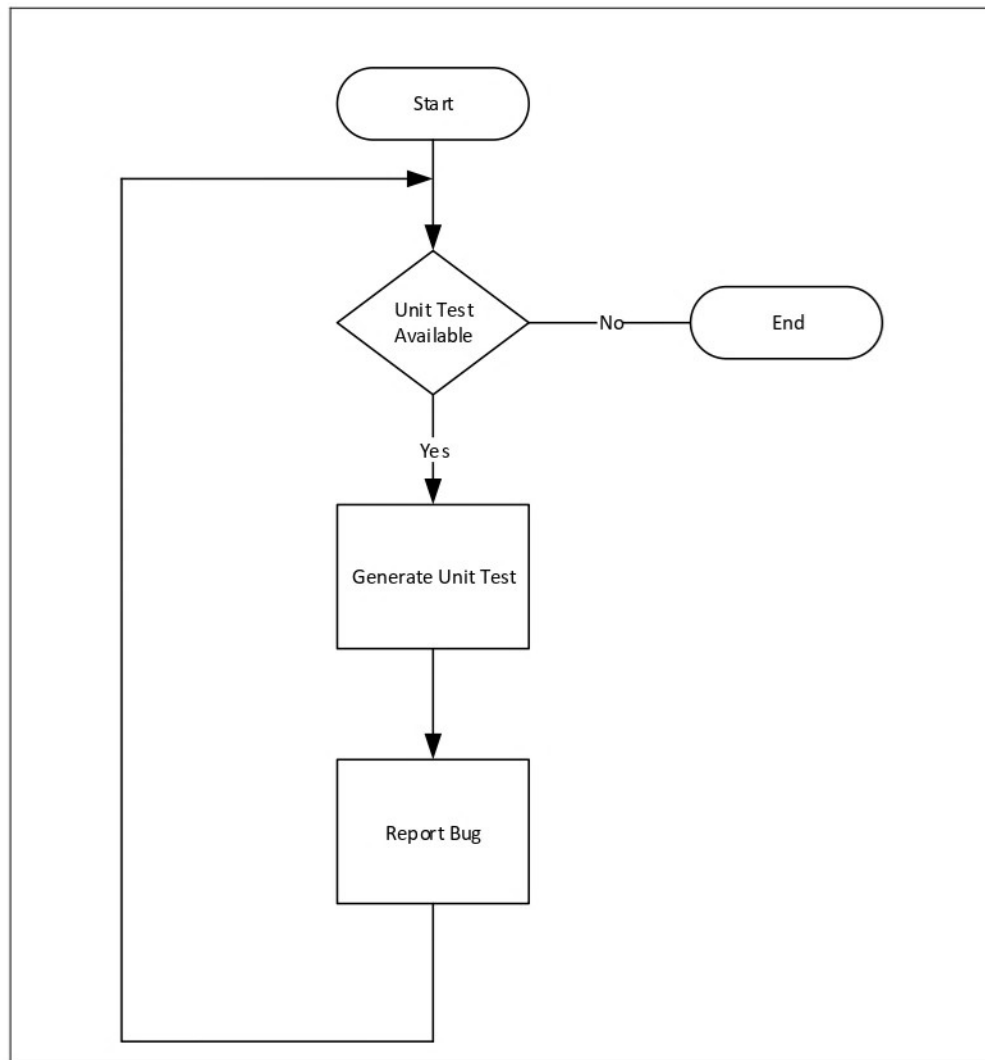


Figure (4.7) Flowchart Diagram for Result Generation

4.3 Program Analysis and Execution

The program analysis is the process of automatically analyzing the behavior of computer programs regarding a property such as correctness, robustness, safety and liveness. Program analysis focuses on two major areas: program optimization and program correctness. The first focuses on improving the program's performance while reducing the resource usage while the latter focuses on ensuring that the program does what it is supposed to do.

4.3.1 Static Analysis is the analysis of computer software that is performed without actually executing programs (Wichmann et al., 1995).

4.3.2 Dynamic Analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor (Myers et al., 2011).

4.3.3 Satisfiability Modulo Theories (SMT)

In computer science and mathematical logic, the SMT problem is a decision problem for logical formulas with respect to combinations of background theories expressed in classical first-order logic with equality. In computer science, the theory of integers, real numbers and various data structures (array, lists, and bit vectors) are examples of such theories (Barrett et al., 2009).

First Order Theories

Restrictions on first order logic and the interpretation of formulas, theories allow us to capture structures which are used by programs (e.g., arrays, integers, etc.) and enable reasoning about them (*First Order Logic*, 1995).

The formulas in the first order logic theory are constructed with a specific set of function, predicate and constant symbols, this signature called Σ .

A first order formula in the theory is then built from elements of Σ together with variables, logical connectives such as \wedge , \vee , \rightarrow , \neg and quantifiers \forall , \exists

An SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations, and SMT is the problem of determining whether such a formula is satisfiable. In other words, imagine an instance of the boolean satisfiability problem in which some of the binary variables are replaced by predicates over a suitable set of non-binary variables. A predicate is a binary-valued function of non-binary variables. The example predicates include linear inequalities (e.g., $3x + 2y - z \leq 4$) or equalities involving uninterpreted terms and function symbols (e.g., $f(f(u, v), v) = f(u, v)$ where f is some unspecified function of two arguments). These predicates are classified according to each respective theory assigned. For instance, linear inequalities over real variables are evaluated using the rules of the theory of linear real arithmetic, whereas predicates involving uninterpreted terms and function symbols are evaluated using the rules of the theory of uninterpreted functions with equality aka empty theory. Other theories include the theories of arrays and list structures (useful for modeling and

verifying computer programs), and the theory of bit vectors (useful in modeling and verifying hardware design) (Barrett et al., 2009).

SMT solvers are widely used in many analyses including synthesis (as in the project), verification, program analysis and software testing based on symbolic execution which will be discussed in later (Barrett et al., 2009).

4.3.4 Concrete Execution

Concrete execution refers to actually running the program on specific inputs and seeing what happens.

4.3.5 Symbolic Execution

Symbolic execution is a means of analyzing a program to determine what inputs cause each part of a program to execute. An interpreter follows the program, assuming symbolic values for inputs rather than obtaining actual inputs as normal execution of the program would. It thus arrives at expressions in terms of those symbols for expressions and variables in the program, and constraints in terms of those symbols for the possible outcomes of each conditional branch (King, 1976).

Consider the program below, which reads in a value and fails if the input is 6.

```
void Test(int a)
{
    b = 2 * a;

    If (b == 12)
    {
        throw new Exception();
    }
    else
    {
        Console.WriteLine("OK");
    }
}
```

In normal execution aka concrete execution, the program would accept a concrete input value (e.g., 5) and assign it to a . Execution would then proceed with the multiplication and the conditional branch, which would evaluate to false and print OK.

In symbolic execution, the program will assign a symbolic value (e.g., λ) to a . The program would then proceed with the multiplication and assign $\lambda * 2$ to b . When reaching the *if* statement, it would evaluate $\lambda * 2 == 12$. At this point of the program, λ could take any value, and symbolic execution can therefore proceed along both branches, by forking two paths. Each path gets assigned a copy of the program state at the branch instruction as well as a path constraint. In this example, the path constraint is $\lambda * 2 == 12$ for the *then* branch and $\lambda * 2 != 12$ for the *else* branch. Both paths can be symbolically executed independently. When paths terminate (e.g., as a result of throwing the exception or exit), symbolic execution computes a concrete value for λ by solving the accumulated path constraints on each path. These concrete values can be thought of as concrete test cases that can, e.g., help developers reproduce bugs. In this example, the `_constraint` solver would determine that in order to reach the exception statement, λ would need to equal 6.

At any point during program execution, symbolic execution keeps two formulas: symbolic store and a path constraint. Therefore, at any point in time the symbolic state is described as the conjunction of these two formulas (Vechev, n.d.).

a) Symbolic Store

The variables values at any time are given by a function $\sigma_s \in \text{SymbolStore} = \text{Var} \times \text{Sym}$ where, Var is the set of variables, Sym is a set of symbolic values and σ_s is called a symbolic store.

Let $\sigma_s : x \rightarrow x_0, y \rightarrow y_0$, then $z = x + y$ will produce the symbolic store $x \rightarrow x_0, y \rightarrow y_0, z \rightarrow x_0 + y_0$ that we keep symbolic expression $x_0 + y_0$

b) Path Constraint

The analysis keeps a path constraint which records the history of all branches taken so far. The path constraint is simply a formula which is typically in decidable logical fragments without quantifiers.

Let $\sigma_s : x \rightarrow x_0, y \rightarrow y_0, pct = x_0 > 10$, let us evaluate: $\text{if } (x > y + 1) \{ 5: \dots \}$

At label 5, we will get symbolic store σ_s , which does not change, but we will get an updated path constraint $pct = x_0 > 10 \wedge x_0 > y_0 + 1$

Limitations

There are some limitations for symbolic execution:

- Path explosion

Symbolically executing all feasible program paths does not scale to large programs, because the number of feasible paths in a program grows exponentially with an increase in program size and can even be infinite in the case of programs with unbounded loop iterations (Anand et al., 2008). The solution for this problem is to use either heuristics for path finding to increase code coverage (Ma et al., 2011), reduce execution time by parallelizing independent paths (Staats & Păsăreanu, 2010), or by merging similar paths (Kuznetsov et al., 2012).

- Program-dependent efficiency

Symbolic execution is used to reason about a program path by path which is an advantage over reasoning about a program input by input as other testing paradigms use (e.g. Dynamic program analysis). However, if few inputs take the same path through the program, there is little savings over testing each of the inputs separately.

- Memory aliasing

Symbolic execution is harder when the same memory location can be accessed through different names (aliasing). Aliasing cannot always be recognized statically, so the symbolic execution engine cannot recognize that a change to the value of one variable also changes the other (DeMillo & Offutt, 1991).

- Arrays

Since an array is a collection of many distinct values, symbolic executors must either treat the entire array as one value or treat each array element as a separate location. The problem with treating each array element separately is that a reference such as "A[i]" can only be specified dynamically, when the value for i has a concrete value (DeMillo & Offutt, 1991).

- Environment interactions

Programs interact with their environment by performing system calls, receiving signals, etc. Consistency problems may arise when execution reaches components that are not under control of the symbolic execution tool (e.g., kernel or libraries).

4.3.6 Concolic Execution

Concolic execution is a mix between **Concrete** execution and **symbolic** execution with the purpose of feasibility. Concolic execution is a hybrid software verification technique that performs symbolic execution, a classical technique that treats program variables as symbolic variables, along a concrete execution (testing on particular inputs) path. Symbolic execution is used in conjunction with an automated theorem prover (ATP) or constraint solver based on constraint logic programming to generate new concrete inputs

(test cases) with the aim of maximizing code coverage. Its main focus is finding bugs in real-world software, rather than demonstrating program correctness (“Concolic Testing,” n.d.).

The concolic approach is also applicable to model checking. In a concolic model checker, the model checker traverses states of the model representing the software being checked, while storing both a concrete state and a symbolic state. The symbolic state is used for checking properties on the software, while the concrete state is used to avoid reaching unreachable states.

Consider the following simple example, written in C#:

```
void Test(int x, int y)
{
    int z = 2 * y;
    if (x == 100000) {
        if (x < z) {
            throw new Exception();
        }
    }
}
```

Simple random testing, trying random values of x and y , would require an impractically large number of tests to reproduce the failure.

This can begin with an arbitrary choice for x and y , for example $x = y = 1$. In the concrete execution, line 2 sets z to 2, and the test in line 3 fails since $1 \neq 100000$. Concurrently, the symbolic execution follows the same path but treats x and y as symbolic variables. It sets z to the expression $2y$ and notes that, because the test in line 3 failed, $x \neq 100000$. This inequality is called a *path condition* and must be true for all executions following the same execution path as the current one.

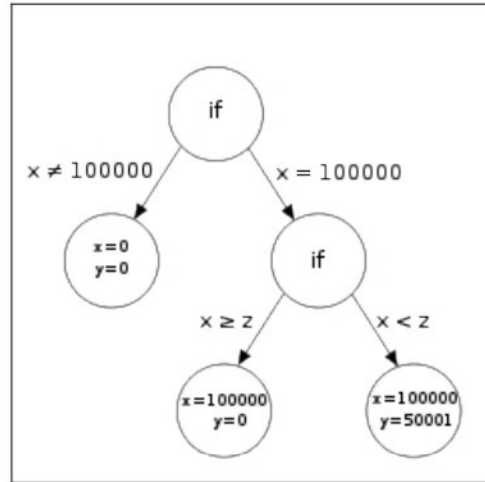


Figure (4.8) Simple example using Concolic Execution (“Concolic Testing,” n.d.)

Figure 4.8 illustrates the program needs to follow a different execution path on the next run, so the last path taken, condition encountered, $x \neq 100000$, and negate it, giving $x = 100000$. An ATP is then invoked to find values for the input variables x and y given the complete set of symbolic variable values and path conditions constructed during symbolic execution. In this case, a valid response from the theorem prover might be $x = 100000, y = 0$.

Running the program on this input allows it to reach the inner branch on line 4, which is not taken since 100000 (x) is not less than 0 ($z = 2y$). The path conditions are $x = 100000$ and $x \geq z$. The latter is negated, giving $x < z$. The theorem prover then looks for x, y satisfying $x = 100000, x < z \wedge z = 2y$ for example, $x = 100000, y = 50001$. This input reaches the exception.

A. Limitations

Concolic testing has a number of limitations:

- If the program exhibits non deterministic behavior, it may follow a different path than the intended one. This can lead to non-termination of the search and poor coverage.
- Even in a deterministic program, a number of factors may lead to poor coverage, including imprecise symbolic representations, incomplete theorem proving, and failure to search the most fruitful portion of a large or infinite path tree.

- Programs which thoroughly mix the state of their variables, such as cryptographic primitives, generate very large symbolic representations that cannot be solved in practice. For example, the condition $if(\text{sha256_hash}(\text{input}) == 0x12345678) \{ \dots \}$ requires the theorem prover to invert SHA256, which is an open problem.

B. Tools

Table 4.2 shows some of the tools that are using the concolic execution in different languages and frameworks.

Table (4.2) Concolic Execution Tools (“Concolic Testing,” n.d.)

Name	Target	Url	Source
jCUTE	Java	https://github.com/osl/jcute	Open
CREST	C	http://www.burn.im/crest/	Open
KLEE	LLVM	https://klee.github.io/	Open
CATG	Java	https://github.com/ksen007/janala2	Open
Jalangi	JavaScript	https://github.com/SRA-SiliconValley/jalangi	Open
PEX	.NET Framework	http://research.microsoft.com/en-us/projects/pex/	Close
Triton	x86 and x86-64	http://triton.quarkslab.com	Open
CutEr	Erlang	https://github.com/aggelgian/cuter/	Open
PathCrawler		http://pathcrawler-online.com/	Close

4.4 Summary

This chapter summarizes the overall design of the proposal technique which uses AST for software bugs prediction. It explains in detail all the phases of the proposed technique from model construction of the AST, path exploration and finally result generation which will generate a bunch of SUTs that are enough to cover all the corner cases for the MUT.

CHAPTER 5: IMPLEMENTATION

5.1 Introduction

This chapter discusses the implementation details of the proposed methodology to conduct the objectives of this research. This includes both lexical and syntax analysis to construct the AST model using Roslyn, then the SMT solver that is used in conjunction with concolic execution to generate the test data. After that, it discusses the implementation of the syntactic pattern matching – that described in Chapter 3 – which will be used in the path exploration for a given AST. Finally it will illustrate some examples that show how all the things fit together.

5.2 Model Construction

The construction of the AST model mainly needs a compiler-like tool, in terms of the implementation Roslyn APIs have been used, instead of reinventing the wheel.

5.2.1 Roslyn

.NET Compiler Platform, also known by its nickname Roslyn (Turner, 2014), is a set of open-source compilers and code analysis APIs for C# and Visual Basic.NET languages from Microsoft (“Roslyn (Compiler),” n.d.). The project notably includes self-hosting versions of the C# and VB.NET compilers – compilers written in the languages themselves. The compilers are available via the traditional command-line programs but also as APIs available natively from within .NET code. Roslyn exposes modules for syntactic (lexical) analysis of code, semantic analysis, dynamic compilation to Common Intermediate Language (CIL), and code emission (Manish Vasani, 2017; McAllister, 2011).

5.2.2 Roslyn Core APIs

The Roslyn exposes C# and Visual Basic compiler’s code analysis to you as a consumer by providing an API layer that mirrors a traditional compiler pipeline (Ng et al., 2012).

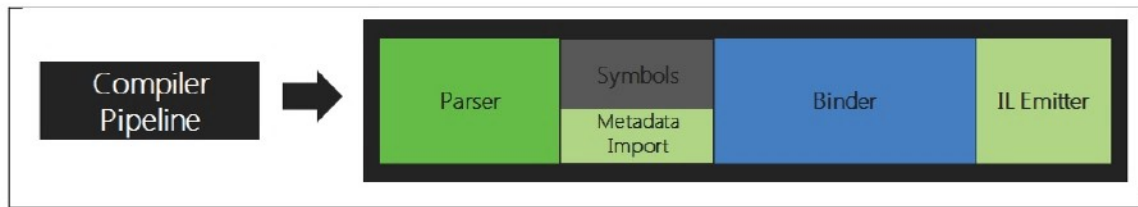


Figure (5.1) Roslyn Compiler Pipeline (Ng et al., 2012)

Each phase of this pipeline is a separate component as shown in Figure 5.1. First the parse phase, where source is tokenized and parsed into syntax that follows the language grammar. Second the declaration phase, where declarations from source and imported metadata are analyzed to form named symbols. Next the bind phase, where identifiers in the code are matched to symbols. Finally, the emit phase, where all the information built up by the compiler is emitted as an assembly.

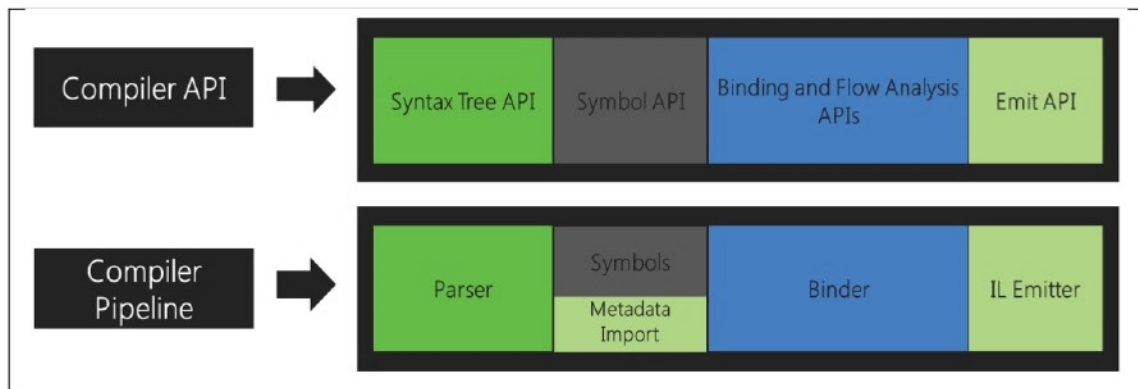


Figure (5.2) Roslyn Compiler API (Ng et al., 2012)

Corresponding to each of those phases, an object model is surfaced that allows access to the information at that phase as shown in Figure 5.2. The parsing phase is exposed as a syntax tree, the declaration phase as a hierarchical symbol table, the binding phase as a model that exposes the result of the compiler's semantic analysis and the emit phase as an API that produces Intermediate Language (IL) byte codes.

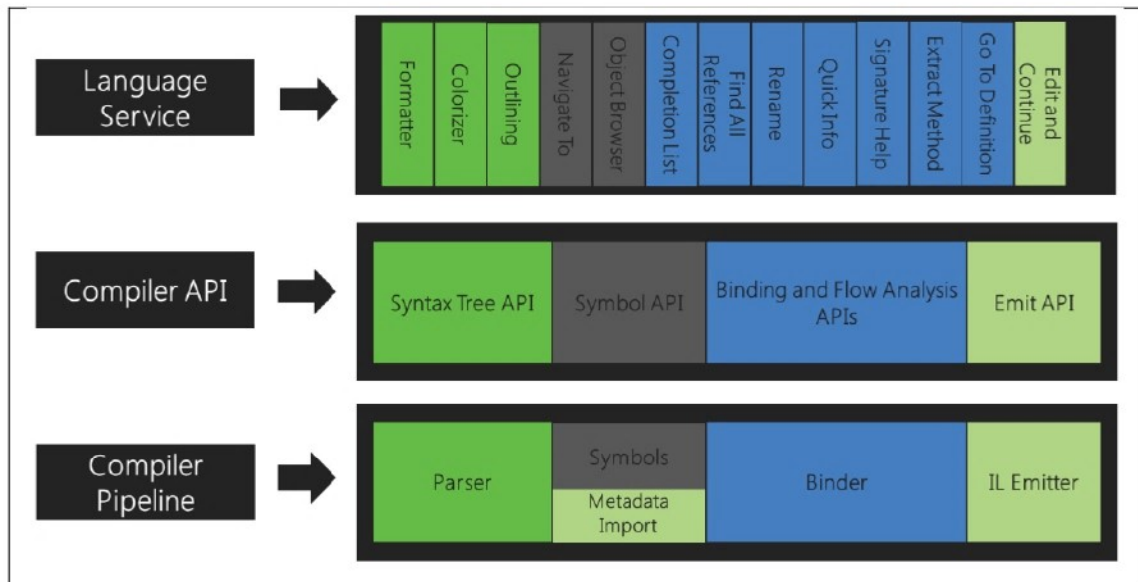


Figure (5.3) Roslyn Language Service (Ng et al., 2012)

Each compiler combines these components together as a single end-to-end whole.

To ensure that the public Compiler APIs are sufficient for building world-class Integrated Development Environment (IDE) features, the Roslyn language services which illustrated in Figure 5.3 are built using them. For instance, the code outlining and formatting features use the syntax trees, the Object Browser and navigation features use the symbol table, refactoring and Go to Definition use the semantic model, and Edit and Continue uses all of these, including the Emit API.

A. Compiler APIs

The compiler layer contains the object models that correspond with information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains a representation of a single invocation of a compiler, including assembly references, compiler options, and source code files.

B. Scripting APIs

As a part of the compiler layer, a set of the scripting APIs are exposed that represents a runtime execution context for C# or Visual Basic snippets of code. It contains a scripting engine that allows evaluation of expressions and statements as top-level constructs in programs.

5.2.3 Working with Syntax

The most fundamental data structure exposed by the Compiler APIs is the syntax tree. These trees represent the lexical and syntactic structure of source code. This section will discuss the key concepts regarding the Syntax API.

A. Syntax Trees

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without being first identified and categorized into one of many well-known structural language elements.

Syntax trees have three key attributes. The first attribute is that syntax trees hold all the source information in full fidelity. This means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between including whitespace, comments, and preprocessor directives. For example, each literal mentioned in the source is represented exactly as it was typed. The syntax trees also represent errors in source code when the program is incomplete or malformed, by representing skipped or missing tokens in the syntax tree.

This enables the second attribute of syntax trees. A syntax tree obtained from the parser is completely round-trippable back to the text it was parsed from. From any syntax node, it is possible to get the text representation of the sub-tree rooted at that node. This means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have created the equivalent text, by implication and by editing a syntax tree, making a new tree out of changes to an existing tree, you have effectively edited the text.

The third attribute of syntax trees is that they are immutable and thread-safe. This means that after a tree is obtained, it is a snapshot of the current state of the code, and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so the new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of nodes, tokens, and trivia.

B. Syntax Nodes

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from *SyntaxNode*.

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a parent node that can be accessed through the *Parent* property. Because nodes and trees are immutable, the parent of a node never changes. The root of the tree has a null parent.

Each node has a *ChildNodes* method, which returns a list of child nodes in sequential order based on its position in the source text. This list does not contain tokens. Each node also has a collection of *Descendant** methods - such as *DescendantNodes*, *DescendantTokens*, or *DescendantTrivia* - that represent a list of all the nodes, tokens, or trivia that exist in the sub-tree rooted by that node.

In addition, each syntax node subclass exposes all the same children through strongly typed properties. For example, a *BinaryExpressionSyntax* node class has three additional properties specific to binary operators: *Left*, *OperatorToken*, and *Right*. The type of *Left* and *Right* is *ExpressionSyntax*, and the type of *OperatorToken* is *SyntaxToken*.

Some syntax nodes have optional children. For example, an *IfStatementSyntax* has an optional *ElseClauseSyntax*. If the child is not present, the property returns null.

C. Syntax Tokens

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation.

For example, an integer literal token represents a numeric value. In addition to the raw source text the token spans, the literal token has a *Value* property that tells you the exact decoded integer value. This property is typed as *Object* because it may be one of many primitive types.

The *ValueText* property tells you the same information as the *Value* property; however this property is always typed as *String*. An identifier in C# source text may include Unicode escape characters, yet the syntax of the escape sequence itself is not considered part of the identifier name. So although the raw text spanned by the token does include the escape sequence, the *ValueText* property does not. Instead, it includes the Unicode characters identified by the escape.

D. Syntax Trivia

Syntax trivia represents the parts of the source text that are largely insignificant for normal understanding of the code, such as whitespace, comments, and preprocessor directives.

Because trivia is not part of the normal language syntax and can appear anywhere between any two tokens, they are not included in the syntax tree as a child of a node. Yet, because they are important when implementing a feature like refactoring and to maintain full fidelity with the source text, they do exist as part of the syntax tree.

Trivia can be accessed by inspecting a token's *LeadingTrivia* or *TrailingTrivia* collections. When source text is parsed, sequences of trivia are associated with tokens. In general, a token owns any trivia after it on the same line up to the next token. Any trivia after that, line is associated with the following token. The first token in the source file

gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Unlike syntax nodes and tokens, syntax trivia do not have parents. Yet, because they are part of the tree and each is associated with a single token, you may access the token associated with using the `Token` property.

E. Spans

Each node, token, or trivia knows its position within the source text and the number of characters it consists of. A text position is represented as a 32-bit integer, which is a zero-based Unicode character index. A *TextSpan* object is the beginning position and a count of characters, both represented as integers. If *TextSpan* has a zero length, it refers to a location between two characters.

F. Kinds

Each node, token, or trivia has a `Kind` property, of type *SyntaxKind*, that identifies the exact syntax element represented.

The `Kind` property allows easy disambiguation of syntax node types that share the same node class. For tokens and trivia, this property is the only way to distinguish one type of element from another.

For example, a single *BinaryExpressionSyntax* class has *Left*, *OperatorToken*, and *Right* as children. The `Kind` property distinguishes whether it is an *AddExpression*, *SubtractExpression*, or *MultiplyExpression* kind of syntax node.

G. Errors

When the parser encounters code that does not conform to the defined syntax of the language, it uses one of two techniques to create a syntax tree.

First, if the parser expects a particular kind of token, but does not find it, it may insert a missing token into the syntax tree in the location that the token was expected. A missing token represents the actual token that was expected, but it has an empty span, and its *IsMissing* property returns true.

Second, the parser may skip tokens until it finds one where it can continue parsing. In this case, the skipped tokens that were skipped are attached as a trivia node with the kind *SkippedTokens*.

Until now everything related to the Roslyn syntax trees has been covered. Let us have a look at example 1 to see Roslyn tree in action:

Example 1:

```
Class SimpleClass
{
    public void SimpleMethod()
    {
    }
}
```

Using Roslyn's syntax visualizer, which can be add-in to Visual Studio.NET as extension, the class syntax tree for example 1 will be shown as follow:

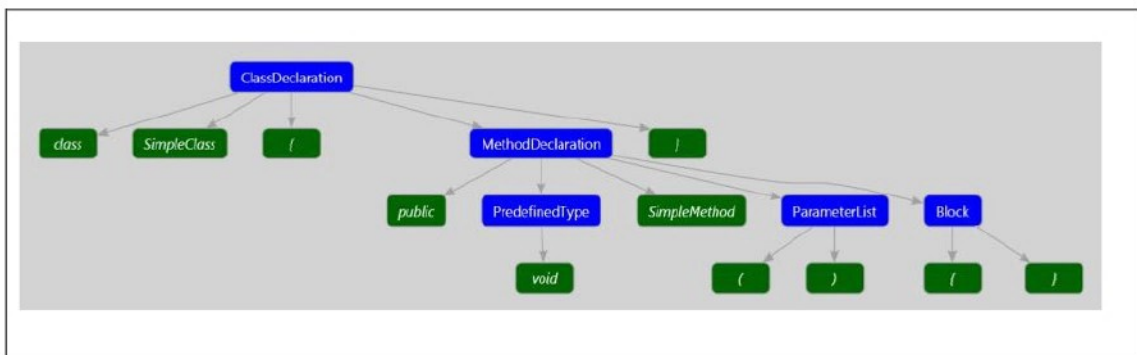


Figure (5.4) Sample Roslyn Syntax Tree (Varty, 2014)

As shown in Figure 5.4 the syntax nodes are the blue one, while the syntax tokens are the green one. Syntax Nodes are *ClassDeclaration*, *MethodDeclaration*, *ParameterList* and *Block*. Syntax tokens are *class*, *SimpleClass*, *Punctuation*, *void* and *SimpleMethod*.

5.2.4 Working with Semantic

Syntax trees represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced.

For example, many types, fields, methods, and local variables with the same name may be spread throughout the source. Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules.

A. Compilation

A compilation is a process that represents everything needed to compile a C# or Visual Basic program including all assembly references, compiler options and source files.

The compilation represents each declared type, member or variable as a symbol and contains a variety of methods that help you find and relate symbols that have either been declared in the source code or imported from another assembly.

B. Symbols

A symbol represents a distinct element declared by the source code or imported from an assembly as metadata.

As mentioned in the previous section a variety of methods and properties on the `Compilation` type help you find symbols. For example, you can find a symbol for a declared type by its common metadata name.

Symbols present a common representation of namespaces, types, and members, between source code and metadata. For example, a method that was declared in source code and a method that was imported from metadata are both represented by a *MethodSymbol* with the same properties.

C. Semantic Model

A semantic model represents all the semantic information for a single source file. It is useful for discovering the following:

- The symbols referenced at a specific location in source.
- The resultant type of any expression.
- All diagnostics, which are errors and warnings.
- How variables flow in and out of regions of source.
- The answers to more speculative questions.

5.3 Test Data Generation

As discussed earlier in Chapter 4, one of the challenges of creating a SUTs is generating a test data for MUT, these input data are critical to any generated unit test because it leads to a path explosion.

5.3.1 Test Data Generation Algorithm

Test data generation algorithm is responsible for generating the test data, which is used as an input for the MUT. This can be done Algorithm 5.1, which shows all the steps required for generating the test data as the following:

1. Loops through all the MUT parameters.
2. Loops through all the pattern matching visitors, which will be described in the 5.3.3 section.
3. Apply each pattern matching algorithm for the previous parameters, in case to generate the suitable inputs that triggers an exception – if any – in the tested method.

Algorithm 5.1: Test Data Generation

```
For  $\forall p \in M_{pl}$ 
  For  $\forall m \in PM$ 
    Compute  $m(p)$ 
  End For
End For
```

Where M_{pl} is a method parameters list, PM are pattern matches and $m(x)$ is a pattern matching algorithm, where x is the parameter in which assists to generate the test data.

5.3.2 Path Exploration Algorithm

Another challenge that was already discussed in the previous chapter is path exploration, a SUTs should examine all the paths in the AST if it is possible, for a given method. This will increase the code coverage for a particular method.

Interestingly it might be that some paths in AST are safe (bug free), eliminating such paths will gain some performance for the proposed algorithm. This could be done by filtering the entire AST with the patterns that are implemented to catch the bugs.

The path exploration algorithm is responsible for traversing all the feasible AST paths for the MUT. This can be done by Algorithm 5.2, which shows all the steps required for exploring the paths as the following:

1. Loops through all the feasible paths for the MUT.
2. Loops through all the pattern matching visitors.
3. Checks if the particular path not contains any pattern matching, then go to step 2, otherwise
4. Apply the matching algorithm for the given execution path, with the generated inputs.

Algorithm 5.2: Path Exploration

```
For  $\forall p \in M_p$ 
  For  $\forall m \in PM$ 
    If  $\neg m(p)$  Then
      ‘ No action is needed, when there is no match is found for a given path
    Continue For
  End If
  Compute  $m(S_p, g(M))$ 
End For
End For
```

Where M_p are method paths, PM are pattern matches, S_p is a statement for a certain execution path, $m(x, y)$ is pattern matching algorithm, where x is the statement to be computed against the pattern matcher, y is the input data to be applied for x and $g(x)$ is a generated test data function, which is shown in Algorithm 5.1.

5.3.3 Pattern Matches

There are some pattern matches that have been used to detect the most common static bugs as the following:

A. Null Reference Visitor Pattern

This pattern scans the entire AST and looks for a declared variables that are not assigned to any values, especially variables who have reference types or null able types.

```
string name;
int? amount;
```

Both of the above variables may cause *NullReferenceException* if they are assigned somewhere else before they assigned to any values.

B. Divide By Zero Visitor Pattern

This pattern match algorithm scans the entire AST and looks for a division operator '/', which exists into two types of expressions: *DivideExpression* and *DivideAssignmentExpression*.

```
a = b / c;  
a /= b;
```

Both of the above expressions may cause *DivideByZeroException* if the divisor is equal to zero.

C. Array Out of Bound Visitor Pattern

This pattern match algorithm scans the entire AST and looks for an array element accessor, and checks whether the array index is out of range or not.

```
a[i] = 7;  
c = b[2];
```

Both of the above expression may cause *IndexOutOfRangeException* if the array index is less or greater than the array size.

D. Overflow Visitor Pattern

This pattern matches an algorithm scans the entire AST and looks for a mathematical assignment expression that may exceed the range of the variable types.

```
int a = b + c;
```

The above expressions may cause *OverflowException* if the value of both b and c exceed the limit of the integer size.

D. Z3 Theorem Prover

Z3 Theorem prover is an open source cross-platform SMT solver - hosted on GitHub (Z3, n.d.) - that was developed by Microsoft in the Research in Software Engineering group at Microsoft Research (*Using the SMT Solver Z3*, n.d.).

It aims to solve problems that arise in software verification and software analysis. Z3 supports arithmetic, fixed-size bit-vectors, extensional arrays, data types, uninterpreted functions, and quantifiers. Its main applications are extended static checking, test case generation, and predicate abstraction (“Z3 Theorem Prover,” n.d.).

Z3 has bindings for various programming languages including C, C++, Java, Haskell, OCaml, Python and .NET.

Now let us illustrate two examples to show how Z3 can be used in both (propositions and predicate logic) and solving equations.

Example 2:

In this example propositional logic assertions are checked using functions to represent the propositions a and b . The following Z3 script checks to see whether the $\neg(a \wedge b) \equiv (\neg a \vee \neg b)$:

```
(declare-fun a () Bool)

(declare-fun b () Bool)

(assert (= (not(and a b)) (or (not a)(not b))))

(check-sat)
```

Result:

```
Sat
```


Example 3:

In this example two equations are given, Z3 finding suitable values for the variables a and b :

```
(declare-const a Int)
(declare-const b Int)
(assert (= (+ a b) 20))
(assert (= (+ a (* 2 b)) 10))
(check-sat)
(get-model)
```

Result:

```
sat
(model
  (define-fun b () Int
    -10)
  (define-fun a () Int
    30)
)
```

5.4 Result Generation

Generating the SUTs is the last phase that needs to be implemented, this simply generates all the reported bugs from the previous phase and writes them in the form of unit tests. There are many unit testing frameworks out there, some are already mentioned in Chapter 2.

5.4.1 Result Generation Algorithm

The result generation algorithm is responsible for generating the required for the MUT aka SUTs. This can be done by Algorithm 5.3, which shows all the steps required for generating the result are as follows:

1. Loops through all predicted bugs for the MUT.

2. Generate a unit test for a given bug, with the selected inputs.

Algorithm 5.3: Result Generation

```
For  $\forall b \in M_{pb}$   
  Print  $f(g(b))$   
End For
```

Where M_{pb} is a predicted bug, $f(g(b)) = f \circ g$ which is a function to generate a unit test for a particular bug in the tested method, by providing the test data that makes the method throws an exception.

5.4.2 Visual Studio Unit Testing Framework

The Visual Studio Unit Testing Framework describes Microsoft's suite of unit testing tools as integrated into some of Visual Studio 2005 and later. The unit testing framework is defined in `Microsoft.VisualStudio.QualityTools.UnitTestFramework.dll`. Unit tests created with the unit testing framework can be executed in Visual Studio or, using `MSTest.exe`, from a command line (*Get Started with Unit Testing*, 2020).

There are some elements used by MSTest tool such as *TestClassAttribute*, *TestMethodAttribute*. The following example shows how a unit test can be written by using MSTest:

```
using Microsoft.VisualStudio.TestTools.UnitTesting;

[TestClass]
public class TestClass
{
    [TestMethod]
    Public void AddTwoNumbersTest()
    {
        // Arrange
        int a = 3;
        int b = 4;
        // Act
        int c = Add(a, b);
        // Assert
        Assert.Equals(7, c);
    }
}
```

The above unit test examines that the result of the method $Add(a, b)$ in the given context returns 7, where $a = 3$, $b = 4$.

5.5 Experimental Results

The research did some experiments, to measure the efficiency of the proposed approach with some criteria: the discovered bug types and accuracy factors.

After executing the prototype for the proposed approach, in multiple runs the results show that the prototype is able to discover all known static bugs that were mentioned previously. Also the number of generated unit tests are based on the possible cases for the test data of the MUT, which increases the accuracy for the proposed method on predicting the bugs.

Let us consider the following method to be tested:

```
public int Method(int a, int b)
{
    int c = b + 2;
    int d = a / c;

    return d;
}
```

As shown from the previous code snippet, the code will throw an exception if $c = 0$, this will occur when $b = -2$, so the prototype will generate two SUTs that will examine all the possible execution paths for the above method as follows:

```
public void TestMethod1()
{
    // Arrange
    int a = 30;
    int b = -2;
    // Act & Assert
    Assert.Throws<DivideByZeroException>(() => Method(a, b));
}
```

The previous unit test is the only one, which makes the given MUT fail, and the next unit test is just to ensure that the given method works as expected.

```

public void TestMethod2()
{
    // Arrange
    int a = 10;
    int b = 2;
    // Act
    int c Method(a, b);
    // Assert

    // Assert.Equals(5, c);
}

```

This concludes that the proposed method is able to predict all of the static bugs, also its accurate in terms of the generated only the minimal required unit tests which cover all the execution paths for a given method.

5.6 Comparison of Result with Previous Works

Based on the reviews and some of the preliminary experiments results, it shows that using static analysis with AST improves the time required for generating the test data which are used as input for the MUT. Also it increases the code coverage, so all the paths for the MUT will be traversed to cover all the possibilities during the program exploration.

To measure some result is suggested the closest tool that can be used for the upcoming comparison. After a lot of research RANDOOP for .NET was the most suited tool to compare with (Industrial Software Systems at ABB Corporate Research, n.d.).

Tables 5.1 and 5.2 shows some experimental results that were made against RANDOOP for .NET which is using FDRT technique - that is discussed in Chapter 3 - using bug type and accuracy factors.

Table (5.1) Comparing RANDOOP for .NET with Smart Unit Tests by Bug Type Factor

Bug Type	RANDOOP for .NET	Smart Unit Tests
Null Reference	No	Yes
Divide By Zero	Yes	Yes
Overflow	No	Yes
Array Out of Bound	Yes	Yes

Table 5.1 shows the ability to discover all of the common static bugs. In addition Table 5.2 shows the comparison between the two approaches with accuracy as a factor.

In order to run this comparison, a time parameter needs to be configured in RANDOOP for .NET, so the underneath comparison shows different values for the time parameter.

Table (5.2) Comparing RANDOOP for .NET with Smart Unit Tests by Accuracy Factor

Bug Type	RANDOOP for .NET				Smart Unit Tests
	1 sec		10 sec		
Null Reference	0	0	0	0	1
Divide By Zero	11	4	89	30	1
Overflow	0	0	0	0	1
Array Out of Bound	2	3	13	49	1

The results that are highlighted above indicate that these unit tests are not necessary at all, of course this will consume the time required to generate all the unit tests. Moreover the number of generated unit tests compared with SUTs are too much, that is why it is named smart.

The Figures 5.5, 5.6, 5.7, 5.8, 5.9, 5.10, 5.11, 5.12 and 5.13 show more analytical data about the comparison about both techniques, also it shows how the generated tests by using RANDOOP for .NET increased dramatically over the time.

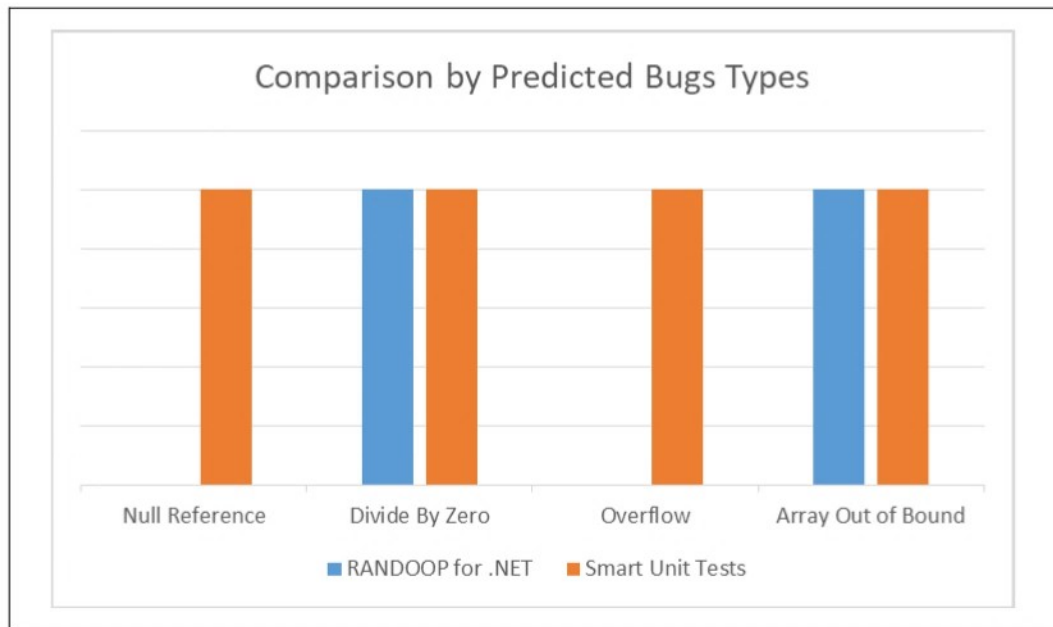


Figure (5.5) Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types

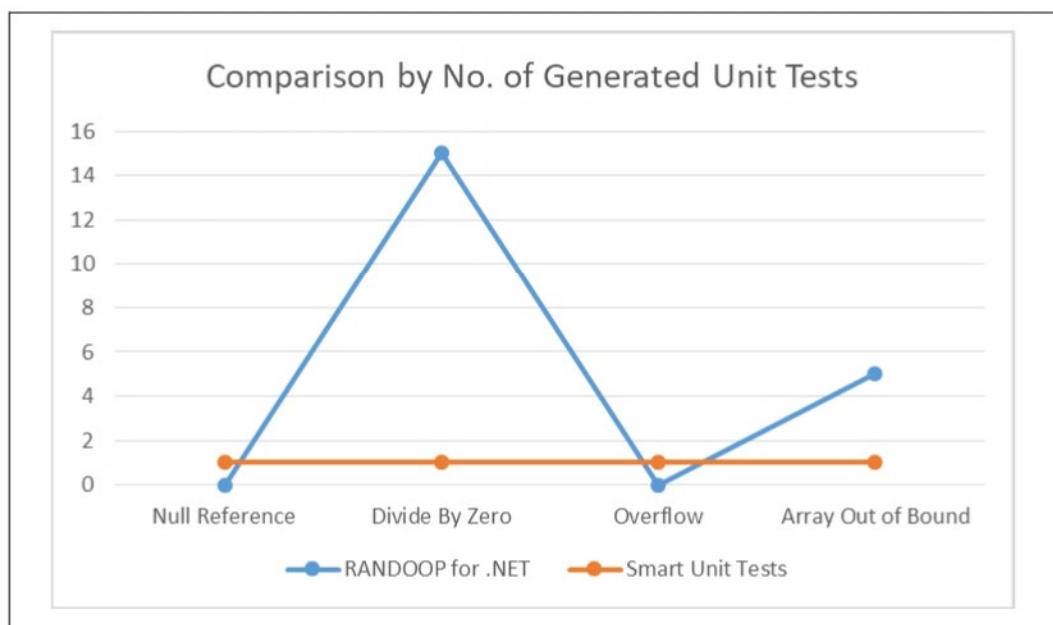


Figure (5.6) Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types

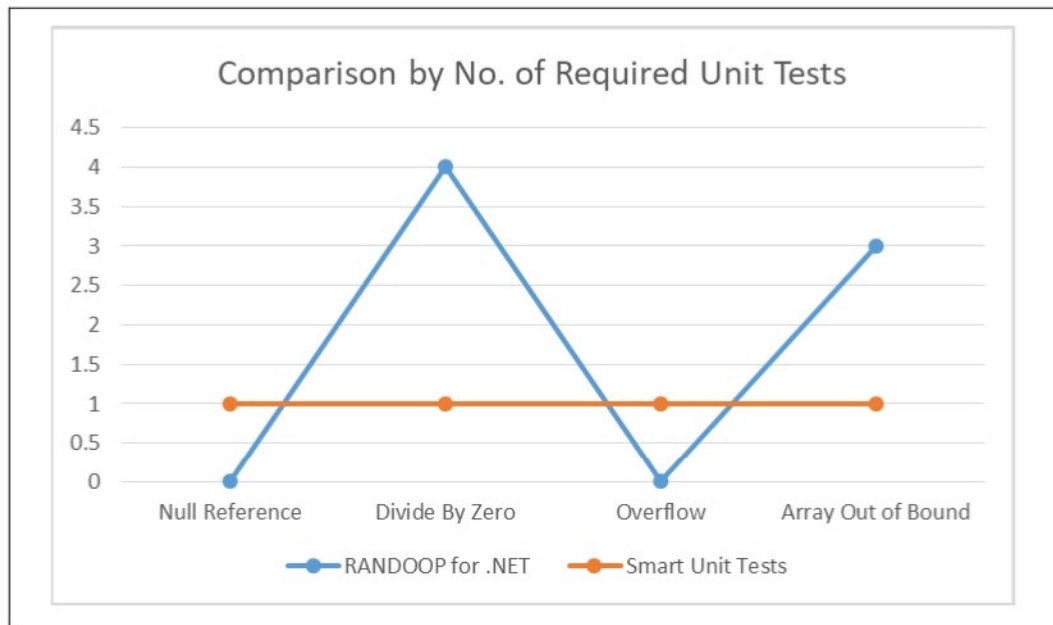


Figure (5.7) Comparison between RANDOOP for .NET and Smart Unit Tests by Predicted Bugs Types

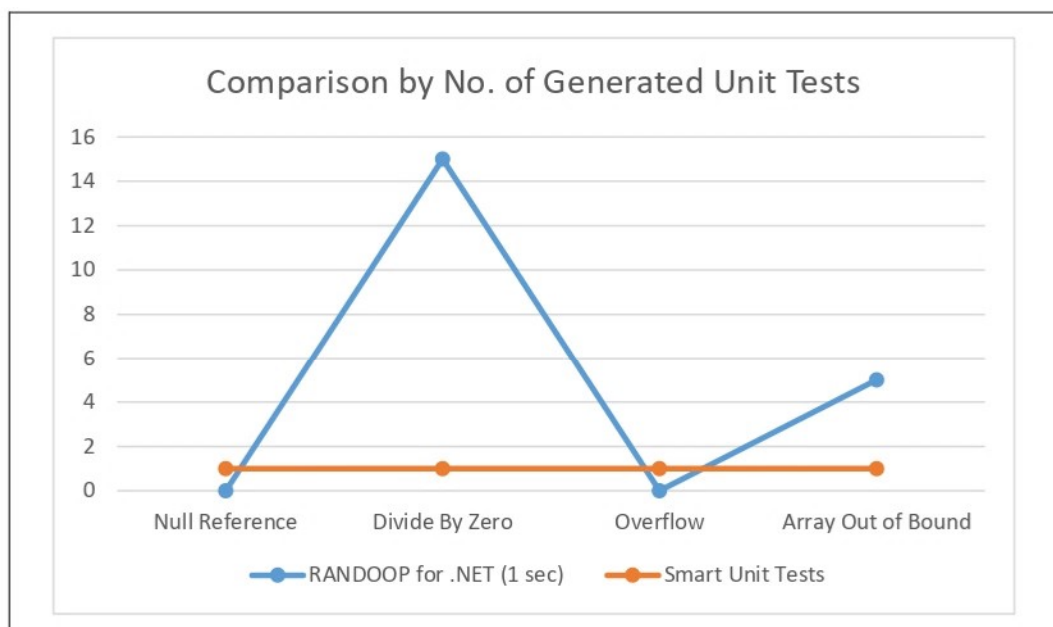


Figure (5.8) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Generated Unit Tests (1 sec)

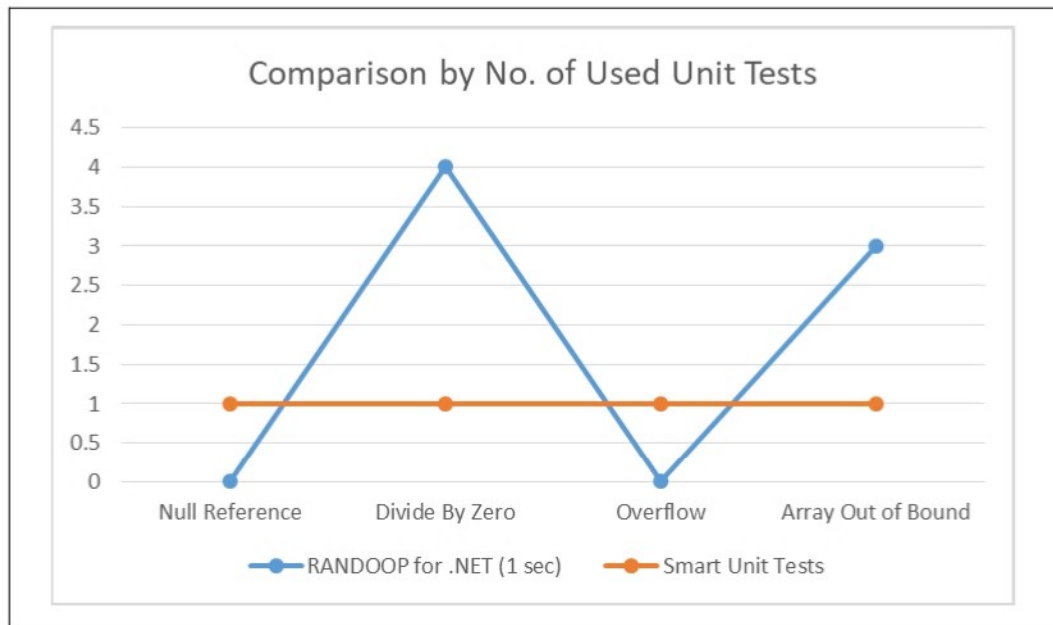


Figure (5.9) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Used Unit Tests (1 sec)

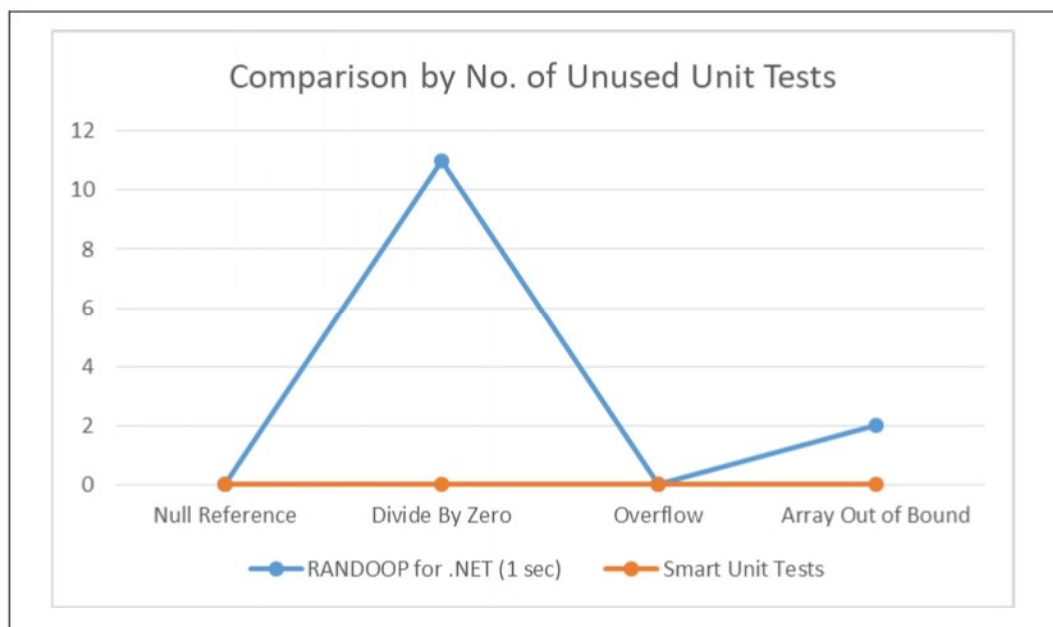


Figure (5.10) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Unused Unit Tests (1 sec)

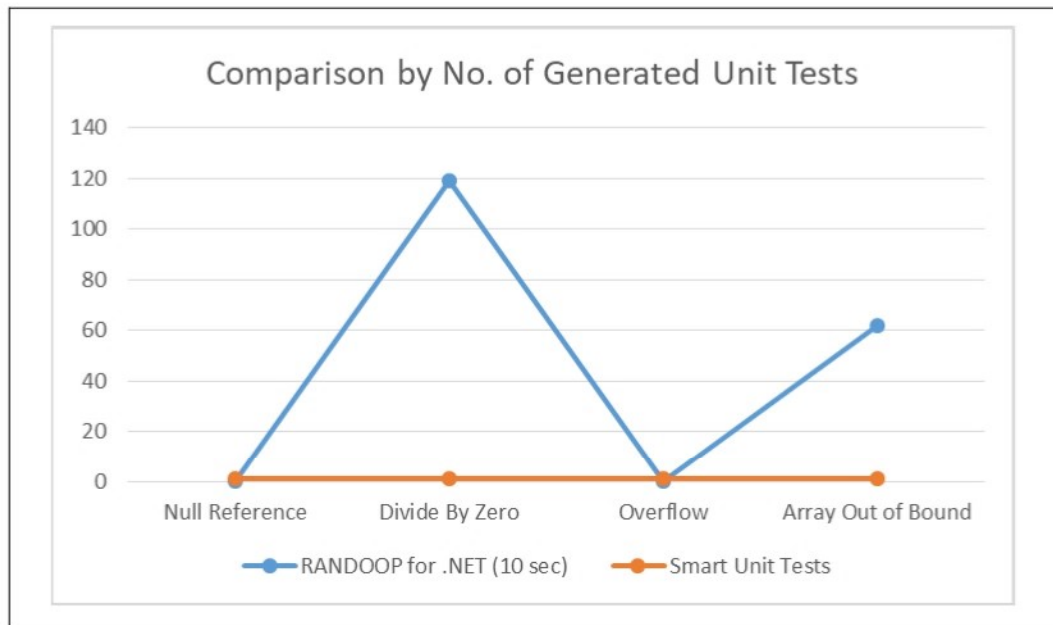


Figure (5.11) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Generated Unit Tests (10 sec)

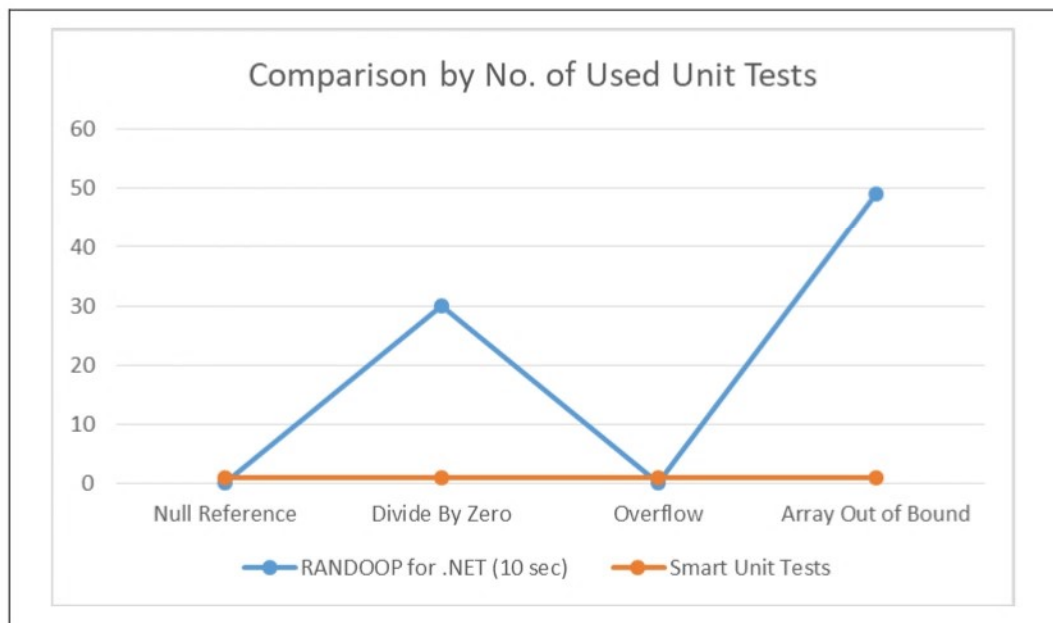


Figure (5.12) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Used Unit Tests (10 sec)

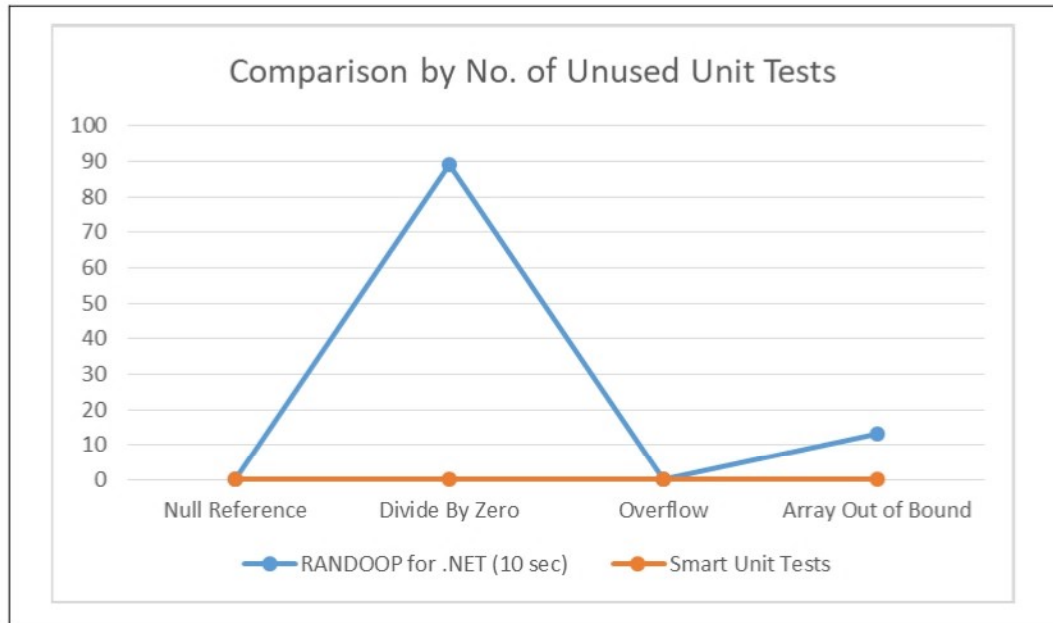


Figure (5.13) Comparison between RANDOOP for .NET and Smart Unit Tests by No. of Unused Unit Tests (10 sec)

5.7 Testing the Results

To test the experimental results that discussed previously for RANDOOP.NET, there are some steps need to be done as the following:

1. Go to the GitHub website, more specifically to the RANDOOP.NET repository <https://github.com/abb-iss/Randoop.NET>.
2. Clone the above repository in your machine.
3. Open the cloned repository via Visual Studio.NET 2008 or above.
4. Build the RANDOOP.NET solution.
5. Open the command line CMD, and point into the path `Randoop.NET\randoop-NET-src\Randoop\bin\Release`
6. Run the command `randoop.exe /timelimit:[t] [p]`, where t is the time that is required to run the tests for (in seconds) and p is the assembly path to be tested.

To run the same tests against the suggested approach, please refer to the appendices section.

5.8 Results Discussion

Based on the preliminary results that are made, comparing with FDRT technique it shows that the proposed technique is promising. Also it proves that using the AST model is suited for maximizing the test code coverage, while it implies predicting various types of the static bugs using static program analysis technique.

Furthermore, during the experiments that made it is clearly seen that the RANDOOP implementation for .NET generated many unit tests, which are valid, but not necessary, this is because of the technique that had been chosen for the test data generation. In contrast, the proposed approach generated a SUTs that is enough to test all the possible paths for the MUT. That means the proposed approach will generate the suitable cases as fast as possible, no time consumed for unnecessary unit tests even if they are valid.

5.9 Summary

.NET Compiler Platform (Roslyn) is a powerful tool that helps to construct the AST for all methods that need to be tested instead of reinventing the wheel and creating a new tool from scratch. This helps to implement a proof of concept on top of Roslyn, to create a tree-based model, then use both a concolic execution with Z3 theorem prover to obtain the test data, alongside with the syntactic pattern matching the implementation to explore the AST paths that not only feasible, but accurate for a particular method.

CHAPTER 6: CONCLUSION & FUTURE WORK

6.1 Conclusion

This thesis contributes to the literature from empirical, theoretical and practical points of view. Empirically, it provides understanding of the importance of unit testing to software developers and QA, and how SUTs are used in their work and help them to accomplish their tasks in a good manner by generating unpredictable suites of unit tests.

From a theoretical point of view, the findings of this research support several theoretical perspectives that have been introduced in relation with the QA in Software Testing & Software Quality.

Practically, this research provides a proof of concept to the developing approach that has been proposed. So, the implemented prototype is not mature enough to be used in production, but hopefully this will be addressed in the upcoming future.

6.2 Limitations of Research Approach

SUTs that are proposed in this thesis have addressed the challenges that had been considered in the existing approaches for test automation and bug finding. The experiments are done for different syntactic pattern matching that are discussed in Chapter 5.

In this section, few issues are briefly discussed, and general research directions are provided to address them as follows:

- Predict the software static bugs, so bugs such as: memory leaks, performance issues and exceptions that rely on the software environment cannot be predicted, because this by nature is a static program analysis limitation.

- Predict the software bugs that may occur in the procedural programming languages, excluding the nested conditions and loops.

6.3 Future Work

Three avenues of future research are identified in light of findings of this research and limitations discussed in Section 6.2.

First, future research could be able to include a wide range of statements, including control flow and loop statements. Also supports object oriented languages afterward.

Second, future research could support adding time factors to measure the performance for the proposed technique.

Lastly, future research could improve a tool that could be integrated with various unit testing adapters within Visual Studio IDE such as MS Test, xUnit.

List of Sources & References

- Aho, A., Lam, M., Sethi, R., & Ullman, J. (2007). *Compilers Principles, Techniques & Tools* (2nd ed.). WorldCat.
- Aiken, A. (1999). Introduction to set constraint-based program analysis. *Science of Computer Programming*, 35(2). [https://doi.org/10.1016/S0167-6423\(99\)00007-6](https://doi.org/10.1016/S0167-6423(99)00007-6)
- Anand, S., Godefroid, P., & Tillmann, N. (2008). Demand-driven compositional symbolic execution. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 4963 LNCS. https://doi.org/10.1007/978-3-540-78800-3_28
- Anatomy of a Compiler and The Tokenizer*. (n.d.). www.cs.man.ac.uk
- Barrett, C., Sebastiani, R., & Tinelli, S. C. (2009). Handbook of Satisfiability. In *New York* (Vol. 185, Issue 3).
- Binder, W., Hulaas, J., Moret, P., & Villazón, A. (2009). Platform-independent profiling in a virtual execution environment. *Software - Practice and Experience*, 39(1). <https://doi.org/10.1002/spe.890>
- Bush, W. R., Pincus, J. D., & Sielaff, D. J. (2000). Static analyzer for finding dynamic programming errors. *Software - Practice and Experience*, 30(7). [https://doi.org/10.1002/\(SICI\)1097-024X\(200006\)30:7<775::AID-SPE309>3.0.CO;2-H](https://doi.org/10.1002/(SICI)1097-024X(200006)30:7<775::AID-SPE309>3.0.CO;2-H)
- Concolic testing. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Concolic_testing
- Cousot, P., & Cousot, R. (1979). Systematic design of program analysis frameworks. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*. <https://doi.org/10.1145/567752.567778>
- Cousot, P., & Cousot, R. (1977). Abstract interpretation: “A” unified lattice model for static analysis of programs by construction or approximation of fixpoints. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages, Part F130756*. <https://doi.org/10.1145/512950.512973>
- Das, M., Lerner, S., & Seigle, M. (2002). ESP: Path-sensitive program verification in polynomial time. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

- <https://doi.org/10.1145/543552.512538>
- DeMillo, R. A., & Offutt, A. J. (1991). Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9).
<https://doi.org/10.1109/32.92910>
- Detlefs, D., Nelson, G., & Saxe, J. B. (2005). Simplify: A theorem prover for program checking. *Journal of the ACM*, 52(3). <https://doi.org/10.1145/1066100.1066102>
- Difference between verification and validation*. (n.d.). Tools QA.
<https://www.toolsqa.com/software-testing/difference-between-verification-and-validation/>
- Everatt, G. D., & McLeod Jr., R. (2007). The Software Development Life Cycle. In *Software Testing: Testing Across the Entire Software Development Life Cycle* (pp. 29–58). John Wiley & Sons.
- First Order Logic*. (1995). Saint Joseph's University.
<http://people.sju.edu/~jhodgson/ugai/1order.html>
- Floyd, R. W. (1967). *Assigning meanings to programs*.
<https://doi.org/10.1090/psapm/019/0235771>
- Get started with unit testing*. (2020, July 4). Microsoft Docs.
<https://docs.microsoft.com/en-us/visualstudio/test/getting-started-with-unit-testing?view=vs-2019&tabs=mstest>
- Getting Started with xUnit.net (desktop)*. (n.d.). <https://xunit.github.io/docs/getting-started-desktop.html>
- Godefroid, P., Klarlund, N., & Sen, K. (2005). DART: Directed automated random testing. *ACM SIGPLAN Notices*, 40(6). <https://doi.org/10.1145/1064978.1065036>
- Gosain, A., & Sharma, G. (2015). Static analysis: A survey of techniques and tools. *Advances in Intelligent Systems and Computing*, 343. https://doi.org/10.1007/978-81-322-2268-2_59
- Hamill, P. (2004). Unit Test Frameworks: Tools for High-Quality Software Development. In *O'Reilly Media*.
- Hilyard, J. (2005, January). No Code Can Hide from the Profiling API in the .NET Framework 2.0. *MSDN Magazine*.
<http://msdn.microsoft.com/msdnmag/issues/05/01/CLRProfiler/>

- Hlongwane, A. (2005). *SELECTING A DEVELOPMENT APPROACH*.
https://www.academia.edu/22583615/SELECTING_A_DEVELOPMENT_APPROACH
- Hoare, C. A. R. (1969). An axiomatic basis for computer programming.
Communications of the ACM, 12(10). <https://doi.org/10.1145/363235.363259>
- Huizinga, D., & Kolawa, A. (2007). Automated Defect Prevention: Best Practices in Software Management. In *Automated Defect Prevention: Best Practices in Software Management*. <https://doi.org/10.1002/9780470165171>
- Industrial Software Systems at ABB Corporate Research. (n.d.). *Randoop.NET*. GitHub.
<https://github.com/abb-iss/Randoop.NET>
- Jamil, M. A., Arif, M., Abubakar, N. S. A., & Ahmad, A. (2017). Software testing techniques: A literature review. *Proceedings - 6th International Conference on Information and Communication Technology for the Muslim World, ICT4M 2016*.
<https://doi.org/10.1109/ICT4M.2016.40>
- Kam, J. B., & Ullman, J. D. (1976). Global Data Flow Analysis and Iterative Algorithms. *Journal of the ACM (JACM)*, 23(1).
<https://doi.org/10.1145/321921.321938>
- Kaner, C. (2016). *Exploratory Testing*.
- Kennedy, K. (1981). A survey of data flow analysis techniques. In *Program Flow Analysis: Theory and Applications*.
- Kiczales, G. . et al. (1997). Aspect-oriented programming. *Proc of the 11th European Conference on Object-Oriented Programming*, 220–242.
- Kildall, G. A. (1973). A unified approach to global program optimization. *Conference Record of the Annual ACM Symposium on Principles of Programming Languages*.
<https://doi.org/10.1145/512927.512945>
- King, J. C. (1976). Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7). <https://doi.org/10.1145/360248.360252>
- Kolawa, A. (2009). *Unit Testing Best Practices*. <http://www.parasoft.com/unit-testing-best-practices>
- Kucharski, M. (2011). *Making Unit Testing Practical for Embedded Development*.
<http://electronicdesign.com/article/embedded/Making-Unit-Testing-Practical-for->

Embedded-Development

- Kuznetsov, V., Kinder, J., Bucur, S., & Candea, G. (2012). Efficient state merging in symbolic execution. *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.
<https://doi.org/10.1145/2254064.2254088>
- Larus, J. R., & Ball, T. (1994). Rewriting executable files to measure program behavior. *Software: Practice and Experience*, 24(2). <https://doi.org/10.1002/spe.4380240204>
- Lexical Analysis. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Lexical_analysis
- Limaye Milind G. (2009). *Software Testing*. Tata McGraw-Hill Education.
- Ma, K. K., Yit Phang, K., Foster, J. S., & Hicks, M. (2011). Directed symbolic execution. *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 6887 LNCS.
https://doi.org/10.1007/978-3-642-23702-7_11
- Manish Vasani. (2017). *Roslyn cookbook : compiler as a service, static code analysis, code quality, code generation, and more*. Packt Publishing.
- Manual Testing. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Manual_testing
- McAllister, N. (2011, October 20). *Microsoft's Roslyn: Reinventing the compiler as we know it*. InfoWorld. <https://www.infoworld.com/article/2621132/microsoft-s-roslyn--reinventing-the-compiler-as-we-know-it.html>
- Mittal, V., & Aditya, S. (2015). Recent developments in the field of bug fixing. *Procedia Computer Science*, 48(C). <https://doi.org/10.1016/j.procs.2015.04.184>
- Myers, G. J., Thomas, T. M., & Sandler, C. (2011). The Art of Software Testing 3rd Edition. In *Booksgooglecom* (Vol. 1, Issue 3).
- Nachtigall, M., Nguyen Quang Do, L., & Bodden, E. (2019). Explaining static analysis-a perspective. *Proceedings - 2019 34th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASEW 2019*.
<https://doi.org/10.1109/ASEW.2019.00023>
- Ng, K., Warren, M., Golde, P., & Hejlsberg, A. (2012). *The Roslyn Project: Exposing the C# and VB compiler's code analysis*.
[https://download.microsoft.com/download/E/A/D/EADDEC33E-FBA3-43BF-9226-427BDAC27610/Roslyn Project Overview.docx](https://download.microsoft.com/download/E/A/D/EADDEC33E-FBA3-43BF-9226-427BDAC27610/Roslyn%20Project%20Overview.docx)

- Pacheco, C., Lahiri, S. K., Ernst, M. D., & Ball, T. (2007). Feedback-directed random test generation. *Proceedings - International Conference on Software Engineering*. <https://doi.org/10.1109/ICSE.2007.37>
- Pan, J. (1999). Software Testing. *Dependable Embedded Systems*, 5(2006), 1–9.
- Parameterized tests*. (n.d.). <https://github.com/junit-team/junit4/wiki/Parameterized-tests>
- Parsing. (n.d.). In *Wikipedia*. <https://en.wikipedia.org/wiki/Parsing>
- Patton, R. (2005). *Software Testing* (2nd ed.). Sams Publishing.
- Pendergrass, J. A., Lee, S. C., & McDonnell, C. D. (2013). Theory and practice of mechanized software analysis. *Johns Hopkins APL Technical Digest (Applied Physics Laboratory)*, 32(2).
- Rajkumar. (2021). *What Is Software Testing | Everything You Should Know*. Software Testing Material. <https://www.softwaretestingmaterial.com/software-testing/>
- Ransome, J., & Misra, A. (2018). Core Software Security. In *Core Software Security*. <https://doi.org/10.1201/b16134>
- Roslyn (compiler). (n.d.). In *Wikipedia*. [https://en.wikipedia.org/wiki/Roslyn_\(compiler\)](https://en.wikipedia.org/wiki/Roslyn_(compiler))
- Saleh, K. A. (2009). *Software Engineering*. Ross Publishing, Incorporated, J.
- Software Testing. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Software_testing
- Software Testing | Static Testing. (2019, May). *GeeksForGeeks*. <https://www.geeksforgeeks.org/software-testing-static-testing/>
- Staats, M., & Păsăreanu, C. (2010). Parallel symbolic execution for structural test generation. *ISSTA'10 - Proceedings of the 2010 International Symposium on Software Testing and Analysis*. <https://doi.org/10.1145/1831708.1831732>
- Static Testing vs. Dynamic Testing*. (n.d.). Software Testing Fundamentals. <https://softwaretestingfundamentals.com/static-testing-vs-dynamic-testing/>
- Thain, D. (2020). *Introduction to Compilers and Language Design* (2nd ed.).
- Theories*. (n.d.). <https://github.com/junit-team/junit4/wiki/Theories>
- Turner, A. (2014). C# and Visual Basic - Use Roslyn to Write a Live Code Analyzer for Your API. *MSDN Magazine*. <https://docs.microsoft.com/en-us/archive/msdn->

- magazine/2014/special-issue/csharp-and-visual-basic-use-roslyn-to-write-a-live-code-analyzer-for-your-api
- US Department of Justice. (2003). Introduction. In *INFORMATION RESOURCES MANAGEMENT*.
- Using the SMT solver Z3. (n.d.). Reunion University. <http://lim.univ-reunion.fr/staff/fred/Enseignement/AlgoAvancee/Exos/Z3-exercices.pdf>
- Varty, J. (2014, July 11). *Learn Roslyn Now: Part 3 Syntax Nodes and Syntax Tokens*. <https://joshvarty.com/2014/07/11/learn-roslyn-now-part-3-syntax-nodes-and-syntax-tokens/>
- Vechev, M. (n.d.). *Program Analysis*. VERIMAG. http://www-verimag.imag.fr/~mounier/Enseignement/Software_Security/ConcolicExecution.pdf
- Wichmann, B. A., Canning, A. A., Clutterbuck, D. L., Winsborrow, L. A., Ward, N. J., & Marsh, D. W. R. (1995). Industrial perspective on static analysis. *Software Engineering Journal*, 10(2). <https://doi.org/10.1049/sej.1995.0010>
- Xie, T., Tillmann, N., De Halleux, J., & Schulte, W. (2009). Fitness-guided path exploration in dynamic symbolic execution. *Proceedings of the International Conference on Dependable Systems and Networks*. <https://doi.org/10.1109/DSN.2009.5270315>
- Z3. (n.d.). GitHub. <https://github.com/Z3Prover/z3>
- Z3 Theorem Prover. (n.d.). In *Wikipedia*. https://en.wikipedia.org/wiki/Z3_Theorem_Prover

List of Publications

- [1] Bin Ateya, H. A. and Baneamoon, S. M. (2020), Software Bug Prediction Using Static Analysis with Abstract Syntax Trees, *In International Journal of Engineering and Artificial Intelligence*, Vol.1, Issue.4, pp. 66-73.