PROGRAM SLICING TECHNIQUES FOR SOFTWARE TESTING

Thesis Submitted to
Cochin University of Science and Technology
In partial fulfilment of the requirement
for the award of the degree of
Doctor of Philosophy
Under
Faculty of Technology

By
Anupama Surendran
Reg. No: 3807

Under the Supervision of

Dr.Philip Samuel
Supervisor

Dr. K. Poulose Jacob
Co-Supervisor

DEPARTMENT OF COMPUTER SCIENCE
COCHIN UNIVERSITY OF SCIENCE AND TECHNOLOGY
KOCHI - 682 022, KERALA, INDIA

January 2016

Program Slicing Techniques for Software Testing

Ph.D. Thesis

Author:

Anupama Surendran
Reg No. 3807
Department of Computer Science
Cochin University of Science and Technology
Kochi - 682 022, Kerala, India
anupama.deepak@gmail.com

Supervisor

Dr. Philip Samuel
Associate Professor in Information Technology Division, SOE
Cochin University of Science and Technology
Kochi - 682 022, Kerala, India
philips@cusat.ac.in

Co-Supervisor

Dr. K. Poulose Jacob
Pro–Vice–Chancellor
Professor in Computer Science
Cochin University of Science and Technology
Kochi - 682 022, Kerala, India
kpj@cusat.ac.in

January 2016

Certificate

*This is to certify that the work presented in this thesis entitled **"Program Slicing Techniques for Software Testing"** submitted to Cochin University of Science and Technology, in partial fulfilment of the requirement for the award of the degree of Doctor of Philosophy is a bonafide record of research work done by **Ms.Anupama Surendran** in the Department of Computer Science, Cochin University of Science and Technology, under my supervision and co-supervision by Dr. K. Poulose Jacob, Pro-Vice-Chancellor, Cochin University of Science and Technology. All the relevant suggestions and modifications suggested by the audience during the pre-synopsis seminar and recommended by the Doctoral Committee of the candidate have been incorporated in the thesis. The work presented in this thesis has not been included in any other thesis submitted previously for the award of any other degree(s).*

Kochi
January 2016

Dr. Philip Samuel
(Supervisor)
Associate Professor
Information Technology Division, SOE
Cochin University of Science and Technology
Kochi – 682 022

Certificate

This is to certify that the work presented in this thesis entitled ***"Program Slicing Techniques in Software Testing"*** *submitted to Cochin University of Science and Technology, in partial fulfilment of the requirement for the award of the degree of Doctor of Philosophy is a bonafide record of research work done by* ***Ms. Anupama Surendran*** *in the Department of Computer Science, Cochin University of Science and Technology, under the supervision and guidance of Dr. Philip Samuel and myself, and the work has not been included in any other thesis submitted previously for the award of any degree.*

Kochi
January 2016

Dr. K. Poulose Jacob
(Co-Supervisor)
Pro-Vice-Chancellor
Professor in Computer Science
Cochin University of Science and Technology
Kochi – 682 022

Declaration

*I hereby declare that the work presented in this thesis entitled "**Program Slicing Techniques for Software Testing**" submitted to Cochin University of Science and Technology, in partial fulfilment of the requirement for the award of the degree of Doctor of Philosophy under the Faculty of Technology is a record of original and independent research work done by me under the supervision and guidance of Dr. Philip Samuel, Associate Professor, Information Technology Division, SOE, Cochin University of Science and Technology and under the co-supervision of Dr. K. Poulose Jacob, Pro-Vice-Chancellor, Cochin University of Science and Technology. The results presented in this thesis have not been included in any other thesis submitted previously for the award of any degree.*

Kochi **Anupama Surendran**

January 2016

// Acknowledgments

Working on this thesis was a wonderful experience and I am indebted to many people for making this period an unforgettable experience. First of all, I am deeply grateful to my supervisor Dr. Philip Samuel, Associate Professor, Information Technology Division, SOE Cochin University of Science and Technology for giving me an opportunity to work with him as a research scholar. I am very much thankful to him for supporting me and encouraging me to come forward with thoughtful ideas. As a guide, his willingness to discuss and suggest solutions for all the difficulties which I faced during research period and his positive vision and selfless attitude towards my academic development helped me to finish my research work.

I am grateful to Dr. K. Poulose Jacob, my co-supervisor and Pro-Vice-Chancellor of Cochin University of Science and Technology, for being a source of support and encouragement. His sincerity, and supportive attitude enabled the successful completion of this work.

My sincere thanks are also due to Dr. Sumam Mary Idicula, Head of the Department of Computer Science, Cochin university of Science and Technology for her support and guidance.

I would like to thank Dr.G. Santhosh Kumar, Assistant Professor, Department of Computer Science for his valuable suggestions and support. I would also like to thank Mr. Muralidharan, Assistant Professor , Department of Computer Science. I would like to thank Mr Joe Joseph - Librarian, Mr. Renjith, Mr. Shibu and Ms. Manju- technical staff members, Ms. Santhi- Section officer, Ms.Girija- Office staff and all the faculty members and office staff of Department of Computer Science, Cochin university of Science and Technology for providing the much needed help and support in completing the research work.

I would like to extend my sincere thanks to my friends Ms. Preetha Teresa Joy, Ms. Sonia Sunny, Ms. Daleesha Vishwanathan, Ms. Vimina and all the research scholars of my department for their cordiality, support and help.

My sincere thanks are also due to Dr. V.P. Nampoothiri, Ms Priyamvada, Dr. K. V. Pramod, Dr. David Peter, Dr. Varghese Paul, Dr. Shelby Joseph and Dr. Binsu Kovoor for their advice and help.

I have no words to express my gratitude to my ever-loving husband Dr. Deepak R. Nair for allowing me to pursue my research and for his constant encouragement and understanding, which brought this work to frutition. Put simply, he understands me more than anyone and I thank him for making me more than what I am.

My sincere thanks to my little daughter Niharika who adjusted a lot despite of my proper attention and care. I extend my gratitude to my parents Dr. K. Surendran and Dr.C. Sreelatha for their support and blessings. I also extend my gratitude to my father-in-law Dr. R. Ravindran Nair and a special thanks to my mother-in-law Prof. K. R. Sethulakshmi for taking care of my little daughter without any complaints. Inspite of having to adjust her schedule, she never compromised on the care which she gave for my daughter.I also thank my relatives and friends for the concern and support they extended at various stages of my study.

Above all, I thank God Almighty for his abundant blessings.

Anupama Surendran

Abstract

In this century, computers have become an inseparable part of human life. Human beings entrust them with day-to-day activities and as well as highly sensitive data such as credit card information. Therefore, it is highly essential to ensure the proper working of the software before it is handed over to the users. Simple errors in the software may cause billions of dollar loss or even cause harm to human life. Therefore, the software needs to be dependable and reliable. Software testing is one of the most important methods to assure an error free software. This thesis work is centered on software testing.

One of the major concerns in today's practical software testing is the size of the source code which the testers have to deal with. Simple and effective testing methods that can handle the issue of source code length are highly essential. As the length of the program increases, testing activities like test case generation and test execution becomes more complicated. From the literature review it was evident that most of the existing methods for software testing did not address the problem of source code size during testing. Considering these scenarios, the work presented in this thesis tries to handle these challenges in practical software testing so as to make testing easier. With this aim, we have proposed a forward slicing based framework which helps to identify the statements of relevance in a software. Program slicing is used in several fields like program comprehension, debugging, software maintenance, program cohesion, refactoring and reverse engineering. Anyhow, the works that explicitly demonstrate how program slicing may be applied in software testing is extremely rare. In this thesis, we have clearly demonstrated how to perform software testing and trace dependencies in the source code using forward slicing. A formal representation of forward slicing is also presented in this work. As an extension to our forward slicing framework, we have also introduced the concept of partitioned forward slices and partial slices which helps the testers to focus on statements of interest. Partitioned forward slices helps to handle the large size of forward slices whereas partial slices identifies statements of interest with respect to output variables. The research finding finally concludes that, software testing approaches should incorporate slicing methods to make testing more effective and easier.

Contents

List of Tables
List of Figures
Abbreviations

1. Introduction ---------- 1
    - 1.1. Overview ---------- 1
    - 1.2. Motivation ---------- 2
    - 1.3. Research Problem ---------- 3
    - 1.4. Research Objectives ---------- 4
    - 1.5. Methodology ---------- 4
    - 1.6. Outline of the Thesis ---------- 7
    - 1.7. Summary of the Chapter ---------- 8
2. Literature Review ---------- 9
    - 2.1. Introduction ---------- 9
    - 2.2. Literature Review Stages ---------- 10
    - 2.3. Review of Recent Trends in Software Testing ---------- 10
        - 2.3.1 Inferences from Review of Lliterature on RrecentTrends in Software Ttesting ---------- 16
    - 2.4. Review of Genetic Algorithm (GA) based Testing ---------- 17
        - 2.4.1. Brief Overview of Genetic algorithm ---------- 18
        - 2.4.2. GA based Testing-Review Stages ---------- 20
        - 2.4.3. Discussion on GA based Testing Review ---------- 32
        - 2.4.4. An Illustration of using GA in Software Testing ---------- 47
        - 2.4.5. Inferences from the Review on GA Based Software Testing --- 52
    - 2.5. Slicing Based Approaches ---------- 53
        - 2.5.1. Applications of Slicing ---------- 58
        - 2.5.2. Inferences from the Review on Slicing Based Approaches --- 59
    - 2.6. Summary of the Chapter ---------- 59
3. Software Testing using Forward Slicing ---------- 61
    - 3.1. Introduction ---------- 61
    - 3.2. Background ---------- 62
        - 3.2.1. Program Slicing ---------- 63
            - 3.2.1.1. Static Slice ---------- 64

3.2.1.2. Dynamic Slice ---64
3.2.1.3. Backward Slice ---65
3.2.1.4. Forward Slice ---66
3.2.2. Terms & Definitions Related to Program Slice---------------- 67
3.3. Significance of using Forward Slicing in Software Testing------------ 68
3.4. Architecture of Forward Slicing based Testing ------------------------- 70
3.4.1. Input Program selector--- 71
3.4.2. Forward Slicer -- 71
3.4.2.1. Slicing Criterion --71
3.4.2.2. Linked Dependency Method for Identifying Dependencies—72
3.4.2.2.1. Steps in Linked Dependency Method---------------72
3.4.2.2.2. Explanation of Linked Dependency Method Steps -- 73
3.4.2.2.3. Illustration of Identifying Dependencies using Linked Dependency Method------------------------75
3.4.2.3. Forward Slicing Algorithm------------------------------------77
3.4.2.3.1. Algorithm Explanation -------------------------------80
3.4.3. Data Generator --- 81
3.4.3.1. Slice Analyzer ---81
3.4.3.2. Selector ---82
3.4.3.3. Test Data Generator--82
3.4.3.3.1. Random Method of Test Data Generation ---------83
3.4.3.3.2. Gauss Elimination Method of Test Data Generation ---83
3.5. Illustration of Test Data Generation using FST------------------------ 86
3.6. Proof of Correctness and Formalised Representation of Forward Slicing Algorithm --- 89
3.7. Summary of the Chapter -- 99
4. Partitioned Forward Slices--101
4.1. Introduction -- 101
4.2. Motivation--- 101
4.3. Terms & Definitions Related to Partitioned Forward Slice --------- 102
4.3.1. Partition Point Properties-------------------------------------- 103
4.4. Partitioned Forward Slicing Algorithm & Explanation------------- 104
4.4.1. Extended Linked Dependency Method ---------------------- 108
4.5. Illustration of Partitioned Forward Slicing -------------------------- 110

4.5.1 Proof of Correctness of Partitioned Forward Slicing Algorithm 112
4.6. Suitability of Partitioned Forward Slices ---------------------------- 113
4.6.1. Suitability of Partitioned Forward Slices in Testing ------- 113
4.6.2. Suitability of Partitioned Forward Slices in Maintenance 114
4.6.3. Suitability of Partitioned Forward Slices in Program Comprehension -- 115
4.7. Summary of the Chapter -- 117
5. Partial Slices in Program Testing-------------------------------------- 119
5.1. Introduction -- 119
5.2. Motivation--- 120
5.3. Terms & Definitions Related to Partial Slice------------------------- 121
5.4. Program Points Set-up --- 121
5.5. Architecture of Partial Slicer -- 123
5.5.1. Identifying Dependencies using Partial Linked Method -- 124
5.5.2. Partial Slicing Algorithm & Explanation------------------- 126
5.5.3. Illustration of Partial Slicing --------------------------------- 129
5.5.4. Proof of Correctness of Partial Slicing Algorithm----------- 133
5.6. Comparison & Performance Evaluation of Partial Slices and Static Slices --- 134
5.6.1. Test Subjects--- 135
5.6.2. Test Set-up-- 136
5.7. Inference from the Comparison and Evaluation of Partial Slices and Static Slices--- 140
5.8. Suitability of Partial Slices --- 140
5.8.1. Using Partial Slices for Software Reuse --------------------- 140
5.8.2. Using Partial Slices for Program Comprehension ---------- 143
5.9. Summary of the Chapter -- 144
6. Comparison and Performance Evaluation ---------------------------- 145
6.1. Introduction -- 145
6.2. Comparison of Forward Slicing based Testing with Related Testing Approaches-- 146
6.3. Experimental Evaluation & Comparison ---------------------------- 150
6.3.1. Comparison and Evaluation using Statistical Method----- 151

6.3.1.1. Test Subjects---151
6.3.1.2. Test Set-up---152
6.3.1.3. Stage 1- Comparing Forward Slicing based Testing and GA based Testing---155
6.3.1.4. Stage 2- Comparing Forward Slicing based Testing and Random testing --157
6.3.1.5. Summary from Statistical Comparison---------------------160
6.3.2. Metric based Comparison and Evaluation -------------------160
6.3.2.1 Software Testing Technique Metrics-----------------------160
6.3.2.2 Slicing Metrics---165
6.4. Inferences from the Comparison and Evaluation of Related Methods--168
6.4.1. Main Inferences made from the Comparison of Different Testing Approaches -- 168
6.4.2. Inferences from Experimental Evaluation & Comparison ---170
6.5. Summary of the Chapter-- 176
7. **Conclusion and Future Research Direction** --------------------------------177
7.1. Introduction-- 177
7.2. Summary of Achievements-- 177
7.3. Main Contributions -- 181
7.4. Future Directions -- 183
7.5. Conclusion --- 183
References ---185
List of Publications from the Thesis---203
Appendix---207

List of Tables

Table 2.1. *Research questions*------20

Table 2.2. *Keywords used for selecting GA based works from various source repositories*------21

Table 2.3. *Selected category of works*------22

Table 2.4. *Works not included*------22

Table 2.5. *Quality assessment form*------22

Table 2.6. *Data extraction form*------23

Table 2.7. *GA works selected for review*------24

Table 2.8. *Works using variations of GA*------29

Table 2.9. *Limitation & Factors to be resolved in future*------30

Table 2.10. *Observations on works using GA variation for software testing*------32

Table 2.11. *Population representation for structural testing*------34

Table 2.12. *Fitness function design issues & suggested solution*------39

Table 2.13. *Structural testing: Number of works using different types of selection in Table 2.7*------41

Table 2.14. *Parameter setting used in GA based testing*------49

Table 2.15. *Paths to be handled in GA based testing*------49

Table 2.16. *Korel's branch distance function*------49

Table 2.17. *Test data generation steps of F(I)= total credit -10*------50

Table 2.18 *Test data generation steps of F(II) = subject credit-4*------51

Table 3.1. *Example of static slice*------64

Table 3. 2. *Example of dynamic slice*------65

Table 3.3. *Example of backward slice*------66

Table 3.4. *Example of forward slice*------66

Table 5.1. *Partial Slices*------132

Table 5.2. *Subject programs*------135

Table 5.3. *Test subject & the result of applying various testing methods on test subjects*------136

Table 5.4. *Response for Q1 and Q2 (For static slicing and partial slicing)*------138

Table 5.5. *Ranked response of static slicing and partial slicing*------138

Table 6.1. *Recent works on software testing*------146

Table 6.2. *Comparison of different software testing approaches*------149

Table 6.3. *Subject programs*------151

Table 6.4.	*Test subject & the result of applying various testing methods on test subjects*	*152*
Table 6.5.	*Test subject & the result of applying various testing methods on test subjects*	*153*
Table 6.6.	*Response for GA based Testing & FST*	*155*
Table 6.7.	*Ranked response for GA based Testing & FST*	*156*
Table 6.8.	*Response for Random method based Testing & FST*	*157*
Table 6.9.	*Ranked responses for random method based testing and forward slicing based testing*	*158*
Table 6.10.	*TCG values*	*162*
Table 6.11.	*UC values*	*164*
Table 6.12.	*Test subject specification*	*166*
Table 6.13.	*Tightness value*	*166*
Table 6.14.	*Coverage value*	*167*

List of Figures

Figure 1.1.	*Stages of research work*	*5*
Figure 2.1.	*Stages of the review*	*10*
Figure 2.2.	*Rate of publications and research works in search based software testing during the Period 1975 to 2015*	*15*
Figure 2.3.	*GA basic steps*	*18*
Figure 2.4.	*Research direction suggested from the observations on works using GA variation*	*33*
Figure 2.5.	*Research direction suggested from the observations in population representation in GA based software testing*	*35*
Figure 2.6.	*Factors affecting and affected by fitness function design*	*38*
Figure 2.7.	*Fitness function design steps*	*40*
Figure 2.8.	*Suggested research directions in parameter settings*	*43*
Figure 2.9.	*Suggested research directions in convergence criteria*	*45*
Figure 2.10.	*Sample lines of code*	*47*
Figure 2.11.	*Test data generation steps*	*48*
Figure 3.1.	*Architecture of Forward Slicing based Testing (FST)*	*70*
Figure 3.2	*Sample segment of program code*	*75*
Figure 3.3.	*CFG of sample code in figure 3.2*	*76*
Figure 3.4.	*Forward slice for the slicing criterion (4, n)*	*77*
Figure 3.5.	*Steps in slice analyser*	*81*
Figure 3.6.	*Functions of selector*	*82*
Figure 3.7.	*Test data generator*	*83*
Figure 3.8.	*Partial class diagram of brokerage system*	*86*
Figure 3.9.	*Sample code segment*	*87*
Figure 4.1	*Sample CFG*	*109*
Figure 4.2.	*Sample code segment*	*110*
Figure 4.3.	*Partitioned forward Slices*	*111*
Figure 4.4.	*Partitioned forward slices in testing*	*114*
Figure 4.5.	*Partitioned forward slices in maintenance*	*115*
Figure 4.6.	*Partitioned forward slices in program comprehension*	*116*
Figure 5.1.	*Guidelines for setting up program points*	*122*
Figure 5.2	*Architecture of partial slicer*	*123*
Figure 5.3.	*Sample CFG*	*125*

Figure 5.4. *Partial class diagram of payroll software* -------------------------------- 129
Figure 5.5. *Sample program statements* --- 130
Figure 5.6 *Scale corresponding to difficulty levels* ----------------------------------- 137
Figure 5.7. *Using partial slices for program reuse*------------------------------------ 142
Figure 5.8. *Partial slices in program comprehension*---------------------------------- 143
Figure 6.1. *Question outcome and scale* --- 154
Figure 6.2. *Test case generation metric* --- 162
Figure 6.3. *The merits of slicing based test data generation* ------------------------ 169
Figure 6.4. *Identification of errors*-- 172
Figure 6.5. *Dependency level*-- 173
Figure 6.6. *Testing productivity graph* -- 175

Abbreviations

GA	-	*Genetic Algorithm*
PSO	-	*Particle Swarm Optimization*
FST	-	*Forward Slicing based Testing*
RAND	-	*Random Method*
LOC	-	*Lines of Code*

Chapter 1

INTRODUCTION

Contents

1.1 Overview
1.2 Motivation
1.3 Research Problem
1.4 Research Objectives
1.5 Methodology
1.6 Outline of the Thesis
1.7 Summary of the Chapter

1.1 Overview

Software has become an unavoidable part of human life [11]. We do not want the software to fail, as the repercussions that can occur in the event of a software malfunction can be huge, which may include loss of time, money and even life [106]. All these can be prevented by ensuring the quality of the software. In practice, software testing still remains as the primary choice for assessing the quality of the software as it can assess the validity and reliability of the software [85]. Through software testing, we can establish that a particular software contains errors. The main concern of software testers is the practical difficulties faced during the execution of testing activities [87]. During source code testing, size of the source code is one such major concern of software testers. As the number of statements in the software increases, the software becomes more complex [80]. Major testing activities like test case generation and test execution also need high attention in such situations [31]. Performing exhaustive testing may not be possible in such scenarios, as it takes an impractical amount of time and effort to check all combinations of input and output [36]. A clear understanding of the relevant statements in a program will be helpful while

generating test input from the program code. In other words, using some simple and effective methods which can handle the size of the source code in a software will decrease the effort required for software testing. Therefore, the research work presented in this thesis tries to handle the challenging issue of source code size in practical structural testing so as to make testing simple and effective.

1.2 Motivation

Modern man cannot imagine a life without software. As software has pervaded every aspect of human life, ensuring their reliability and correctness needs utmost importance. This confers a cardinal role for software testing in software development [19]. Looking at the current trend in software testing, black box approaches are mostly used in industry [19]. Though the black box approach assures some level of reliability for a software, it cannot be compared with the quality of test result given by structural testing [11]. Though structural testing ensures reliability of the source code, the reluctance to use structural testing in practical software testing is due to the limited time, resources and large number of lines of code (LOC) which the testers have to handle during testing [65]. Unrestricted size of source code is a major issue during source code testing as this affects the scalability, consistency and integrity of software systems. Several testing methods when applied to small software systems may perform up to the user's expectations. The same testing methods when applied to large software systems may not give the expected result because of the inability to handle the voluminous code present in the software. Therefore, testing methods should be able to scale effectively to handle the large size of software [25]. Consistency of software systems is also dependent on the size of the source code [36]. The software may not work when some additional functionalities or constraints are added to the existing software. In other words, the original software may not be able to handle such newly added constraints and this affects the consistency of the whole system.

If the tester is able to get an idea of the program statements which may get affected by the newly added constraints, it may be possible to minimize such inconsistencies. As an increase in source code size makes it difficult to trace the program statements which are affected by a particular variable or test case, the software inconsistency issue gets worsened if not handled with utmost care. Similarly, integrity of software systems is also affected by the size of the software [36]. In software testing, in order to check whether the software satisfies a condition, it is also necessary to see all the dependent statements of the specified condition. Then only the integrity of the whole system may be ensured. Therefore, we can conclude that, testing methods which are unable to handle unrestricted source code size may not be scalable, consistent and stable.

Another problem during source code testing is that, an input value may not be responsible for the execution of every statement present in the program [53, 86]. Checking every bit of program code to find such executed or dependent statements is not at all practical in today's software applications, as this only leads to an increase in cost during software development [53]. Instead, identifying the dependent statements in the software and analysing how these statements affect the software helps to trace out the errors in a program.

Most of the testing work reported in literature does not address the issue of managing the size of the source code [2, 3, 43, 46, 59, 67, 97, 98, 99, 100, 101, 104]. Thus the main motivation behind this thesis work is to introduce an effective method which can handle the problem of source code size during structural testing. This can in turn ensure that the software is scalable, consistent and manageable.

1.3 Research Problem

Having given an idea of practical difficulties in software testing and the advancement of software usage in day to day life, it is high time to develop a simple and effective approach which helps software testers to deal with the

problem of source code length and identification of statements of interest during software testing [65, 77]. Therefore, the research problem formulated in this work is: *To develop an effective method for source code based software testing that can handle the size of source code.*

1.4 Research Objectives

In order to achieve the research goal stated above, some other factors are also considered. Initially, the shortcomings of some of the current software testing methods to handle source code size in testing are analysed. Our work also addresses, several other difficulties related to the source code testing like identifying relevant statements in the program code, tracking dependency in the program and the program statements affecting the output. Hence, the final objectives of this thesis work are:-

- To develop methods that can handle source code size during structural testing
- Identify statements of interest with respect to input variables in software testing
- Identify dependency in the program
- Tracking changes related to output

1.5 Methodology

The methodology adopted in this thesis work for achieving the research objectives stated in section 1.4 is represented in Figure 1.1. The research work presented in this thesis is carried out in five stages which are aimed to reduce the difficulties faced during software testing. For achieving this, the existing trend in software testing was reviewed and studied. After studying the existing trends in software testing, the shortcomings of various software testing approaches were studied.

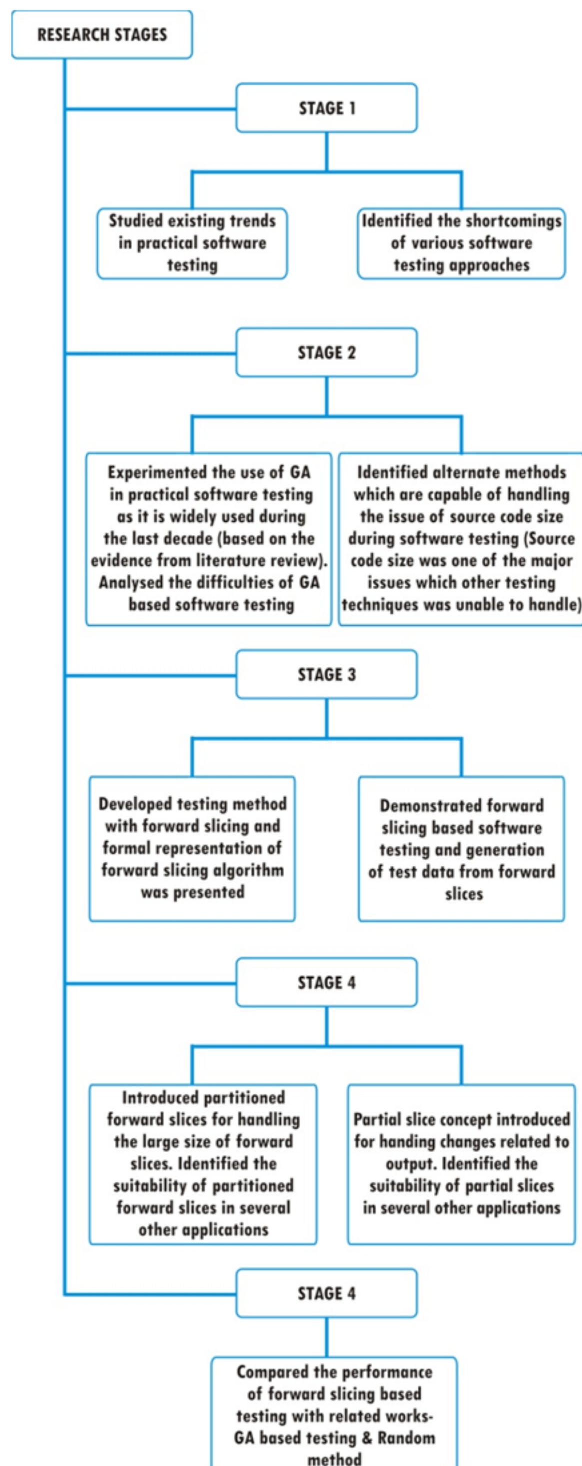


Figure 1.1 Stages of research work

A detailed literature survey on software testing techniques was conducted inorder to find a solution to the practical difficulties faced during software testing. These two works were included in the first stage of the research. In the second stage of the research, Genetic Algorithm (GA) based software testing was experimented. Several practical difficulties were encountered when GA was used for software testing. The literature survey also gave supporting evidence on the shortcomings of GA based testing. One of the main issues which the reported testing methods were incapable of handling was the size of the source code. Alternate methods which were able to handle issue of source code size were also identified in the second stage of the work.

A program slicing based testing approach was introduced in the third stage of this thesis work which was capable of handling the issue of source code size. Testing using forward slicing was developed to handle the source code size and to identify the statements of interest during testing. A formalized representation of slicing algorithm was presented in the same stage. How to generate test data from forward slices was also demonstrated in the third stage of the thesis work. In stage four, an extension to forward slicing based testing is given. Here, the concept of partitioned forward slicing was introduced to handle the large size of forward slice. Apart from partitioned forward slice, the concept of partial slice was also introduced in stage four to trace the changes related to output variables. In the fifth stage of this thesis work, the forward slicing based testing was compared with related testing methods. In this thesis, forward slicing based testing is compared with GA based testing and Random method. In the experimental evaluation, a statistical method and a metric based method were used for comparison of forward slicing based testing with related methods. The evaluation result shows that forward slicing based testing outperforms GA and Random method.

1.6 Outline of the Thesis

The complete outline of the research reported in this thesis consists of 7 chapters. In this outline section, a brief description of the contents in each chapter is provided.

Chapter 1: This chapter provides an introduction to this research work which includes an introduction to software testing, motivation, research problem, research objectives and methodology used in this work.

Chapter 2: This chapter provides a critical survey of reported software testing approaches. Using GA for software testing is also demonstrated in this chapter. The practical difficulties encountered during GA based testing are also highlighted.

Chapter 3: This chapter proposed and demonstrated how to apply the concept of forward slicing in software testing. The relevance of using slicing in testing is explained and test data generation from slices using various methods is also described in this chapter. Formal representation of the forward slicing algorithm is introduced and explained in this chapter.

Chapter 4: The concept of partitioned forward slices is introduced in this chapter. Partitioned forward slices were introduced to make slices focused when the size of forward slice was large. Suitability of partitioned forward slices in different applications is also discussed in this chapter.

Chapter 5: This chapter introduced the concept of partial slices. The idea of partial slices was introduced to handle changes related to output. The potential use of partial slices in different applications is also provided in this chapter.

Chapter 6: A summary of the comparison with related work and performance evaluation of forward slicing based testing with GA based software testing and Random method is described in this chapter.

Chapter 7: This chapter concludes the thesis. The contributions of our work are highlighted. It also discusses the future directions for extending the research work.

References & the list of publications of the author are listed after Chapter 7

1.7 Summary of the Chapter

The introduction, motivation, research problem and objectives of this research work are explained in this chapter. The chapter concludes by giving an overview of the methodology used for carrying out this research work. An outline of the succeeding chapters is also presented.

…….ཙཚ…….

Chapter 2

LITERATURE REVIEW

Contents

2.1 Introduction
2.2 Literature Review Stages
2.3 Review of Recent Trends in Software Testing
2.4 Review of Genetic Algorithm based Testing
2.5 Slicing based Approaches
2.6 Summary of the Chapter

2.1 Introduction

This chapter gives a review of the literature related to the work presented in this thesis. The review is conducted in three stages. The first stage gives a literature review of the recent trends in software testing methods. The shortcomings of different testing techniques are studied thereafter. As discussed in the previous chapter, the main objective of this thesis work is to design an effective method for handling the source code size during source code testing. Following the current trend in software testing from various literature sources, we found that metaheuristic techniques, especially Genetic Algorithm (GA) based techniques is one of the most researched areas in software testing. Therefore, in addition to the review of some relevant testing techniques, a detailed review of Genetic Algorithm based software testing is carried out in the second stage of this work. Certain inferences based on the review are also provided. From the review outcome, it can be noticed that GA based software testing has several shortcomings in practical testing. In the third stage, some solutions are explored for overcoming the shortcomings of testing methods described in stage two. As a solution, a program slicing based testing was introduced in this thesis, with the aim of satisfying the goal of this

thesis work. Therefore, a review of program slicing approaches is given in the third stage of this work.

2.2 Literature Review Stages

The literature review conducted in this work consists of three stages. Figure 2.1 gives the review stages. In the first stage, a review of the recent trends in software testing techniques is provided. Based on the inference from the first stage, a review of GA based testing was carried out in the second stage. Inferences made from stage one and two pointed to some shortcomings in the current testing scenario. Therefore a program slicing based testing was proposed in this thesis work and a review on program slicing is done in stage three. Finally the inferences on program slicing based testing are discussed.

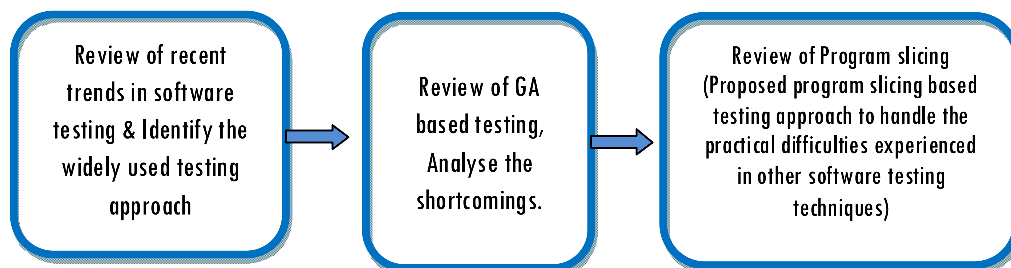


Figure 2.1 Stages of the review

2.3 Review of Recent Trends in Software Testing

Software has an irreplaceable role in our day to day life and this makes software testing a vital cog in the wheel [85, 106]. Even though software testing has evolved tremendously, it still remains both time consuming and personnel dependent. Almost 50% of the cost and time of the software development is taken up by testing [11, 80]. To a great extent, automating the process of software testing could reduce the total cost, time and effort incurred during software testing. The main intention of software testing is to check for

the presence of errors in the software, but a complete testing is many times impractical. To handle the challenges associated with software testing, some strategies are adopted. Black-box testing and white-box testing are the two main strategies of testing.

In Black box testing, the source code of the program is not considered [43]. Only the program functionality is considered in black-box testing. In black-box testing, inorder to ensure that the program is fully error free, the program should be checked with all possible inputs. This is impractical, as this takes a lot of time and money. In white-box testing, the source code of the program is considered. It is also known as structural testing. In structural testing, the fault is revealed by executing the program with selected input and by evaluating the correctness of the program behaviour against the expected ones. In white-box testing, the program paths are executed with test cases. It is not possible to execute all the program paths during white-box testing because the number of paths will be astronomically high [87, 88]. Moreover the program may have missing paths which cannot be identified during path execution. This implies that exhaustive testing is not possible in white-box testing also. One of the most important and difficult tasks in software testing is test data generation. A test case consists of a test data, a set of statements on which the data is applied and the corresponding output [80]. In order to find the test data, the program should be executed with all combinations of input from the program input domain. As already mentioned, this type of exhaustive testing is almost impossible as it takes an impractical amount of time and effort. Therefore, several methodologies of test data generation exist in black-box and white-box testing. Statement coverage, Decision coverage, Condition coverage and Decision-condition coverage are the methodologies used for generating test data in white-box testing. Equivalence partitioning, Boundary value analysis, Cause-

effect graphing and Error guessing are the methodologies used in black-box testing [98, 106]. In white-box testing, each and every statement must be executed at least once for fulfilling statement coverage criterion. If any statement is missing from the program, it may not be detected in statement coverage criterion. Therefore, a stronger criterion called decision coverage is considered. Decision coverage is also known as branch coverage. In decision coverage, test data is generated to cover each branch. A stronger criterion called condition coverage exists to satisfy each condition in the branch. In equivalence partition, the input domain is partitioned into a number of finite equivalent classes. In equivalence partitioning, there are mainly two steps. First step is identifying the equivalence classes and the second step is identifying the test cases. Boundary value analysis is almost similar to equivalence partition. In boundary value analysis, the elements are selected from the edges of the equivalent class rather than selecting from inside the class. This means that boundary value analysis concentrates on and around the edges of the equivalence partitions. One of the main disadvantages of equivalence partition and boundary value analysis is that, the combination of inputs is not checked in these methods. In cause effect method, there will be a cause effect directed graph. There will be a set of causes which is mapped to a set of effects. The causes are considered as input to the program and the effects as output. In the cause effect directed graph, nodes on the left side of the graph represent the cause and the nodes on the right side represent the effect. The causes may be connected by intermediate AND or OR operator. Another methodology used in black box testing is error guessing. This is an intuitive method. No specific rules are used to guess the errors.

In addition to black-box and white-box testing, some other types of testing techniques are widely used. Random search is one of the basic and

simplest methods used for finding test data [8, 104]. The program statements are executed using randomly generated input values. Though this is one of the simplest methods, random search cannot guarantee success every time. Therefore, more guaranteed approaches like path oriented approach are considered [13, 31, 32, 40, 83, 90, 101, 125, 126, 129, 130, 132]. In this approach, the problem of test data generation is considered as a 'path' problem. The path for which test data is to be generated is selected automatically [101]. This path in turn leads to the destination. If the selected path doesn't lead to the target path, then another path is considered. This process is continued until the target path is found out or until the required test data is found or until the time specified for data generation is over. Symbolic execution and execution-oriented test data generation is used to generate test data in path oriented approach [13, 35, 98, 131, 133]. In symbolic execution, there will be a set of constraints and these constraints must be satisfied for traversing the paths in path–oriented approach. Symbolic execution faces difficulties when the program is very complex and long. In execution oriented approach, the actual execution of the program occurs. This is a goal oriented approach where the process of finding input is represented as a set of sub goals [7, 37, 93, 101]. Here the program is executed with the randomly generated input. For the generated input, the program execution flow is traced. As the program execution progresses, the search procedure decides whether the execution should proceed or whether an alternate branch is to be considered as the current path may not lead to the goal. If an undesirable execution at a branch occurs, then a real valued function is associated with the branch. A function minimization search algorithm is used to find new input, which may alter the program execution flow at the current branch. This process is continued until the target node or goal is obtained. In path oriented approach,

the main problem occurs when non-executable paths are explored which causes a loss in computation effort [71, 108]

Another method used for test data generation is the chaining approach [41, 101, 102]. This method may be considered as an extension of path oriented approach. In chaining approach, dataflow analysis is used for generating test data. The program statements are represented as nodes. The edges represent the interconnection between the nodes. An input value is randomly generated and the program is executed with this input. If the execution of the program with the generated input leads to a branch which does not lead to the target node, then a real valued function is associated with this node. A function minimization search algorithm is applied to find new input value. The new input alters the program execution flow at the current node. Here the execution flow is altered inorder to find suitable input value. Chaining method also faces difficulties as the program size and complexity increases. Metaheuristic search techniques offer great support in such situations and this may be used for structural, functional and non-functional testing [4, 38, 63, 91, 98, 99, 115]. In metaheuristic techniques, the test data generation problem is amenable to symbolic execution. A fitness function is designed for this purpose and a set of manipulations and operators are applied to optimize this fitness function. Hill-climbing, simulated annealing, genetic algorithms, and swarm-Intelligence techniques are some of the commonly used metaheuristic search techniques for test data generation.

Among the metaheuristic search techniques, hill-climbing is one of the simplest methods. In hill-climbing, an initial solution is chosen randomly [63]. Then the current solution is compared with the neighbours. The new solution is considered if the new solution is better than the current. The main disadvantage of this method is that the solution may get stuck in local maxima. Therefore, hill-climbing is considered as a local search method. Simulated

annealing gives better result compared to hill-climbing. Single space exploration is used in simulated annealing [98]. Therefore, it is considered as a local search method. In simulated annealing also, the solution may get stuck in local maxima. Genetic algorithm is an adaptive global search method based on the principle of evolution. In genetic algorithm, the chances of getting stuck in local maxima are less compared to hill-climbing and simulated annealing. Particle swarm optimization (PSO) is a global search method [72]. Search space exploration is more efficient in genetic algorithm and particle swarm optimization compared to simulated annealing. The main advantage of using PSO compared to genetic algorithm is that PSO is easier to implement and have a fewer parameters to adjust compared to GA. Compared to GA, even though PSO can quickly find the most appropriate area in the solution for a problem, PSO faces difficulty to find the best solution [72].

In the past few years search based software testing, especially evolutionary algorithm based testing (GA), has gained immense popularity [2, 3, 29, 72, 99].

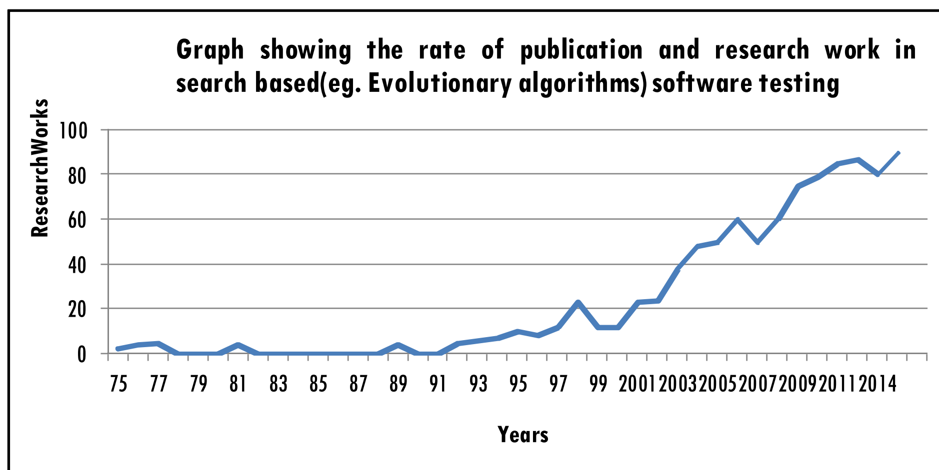


Figure 2.2 Rate of publications and research works in search based software testing during the period 1975 to 2015

A graph is shown in Figure 2.2, which shows an increase in rate of publications and research works in search based software testing during the period 1975 to 2015 [99]. Among evolutionary algorithms, genetic algorithm based software testing has received a wide interest from researchers due to its ability to handle complex problems where an exact solution doesn't exist [99]. Even though genetic algorithm based testing has made a great impact on academic research, only very little attention has been given to understand the complexities of using genetic algorithms in practical software testing. Therefore, an in-depth review of GA based software testing is given in the section 2. 2. 2

2.3.1 Inferences from Review of Literature on Recent Trends in Software Testing

Based on the review of literature on recent trends in software testing, some inferences are drawn. These inferences are given below.

- Though white box testing is more reliable compared to black box testing, black box is mostly used in practical testing. This is due to the practical difficulties experienced in white box testing such as length of source code which is to be handled during testing.
- Random approach of program testing is one of the simplest methods of software testing
- In path oriented approaches, as the length of the path increases, the testing becomes difficult
- In symbolic execution, as the program length increases, testing becomes difficult
- Software testing based on metaheuristic techniques are widely used

- Metaheuristic techniques such as hill-climbing and simulated annealing uses only single-space exploration and therefore they may get stuck in local optima
- Among the metaheuristic techniques, GA is one of the widely used techniques in software testing and has less chance of getting stuck in local optima
- Though PSO uses global search method, it is difficult to find the best solution in PSO

Having got an idea of the recent trends in software testing, the next section deals with an extensive review of GA based testing approach.

2.4 Review of Genetic Algorithm (GA) based Testing

Based on the review of literature on recent trends in software testing, it is evident that genetic algorithm based testing is one of the most extensively research areas in structural testing during the last decade [3, 99]. Therefore, in this section of the review, we have tried to highlight the challenges involved in genetic algorithm based approaches for using it as a practical tool in software testing. The main reason for choosing this is because of the usage of genetic algorithms in software testing without addressing the practical difficulties in genetic algorithm based testing. Since the testing literature is flooded with GA based approaches, we have conducted a detailed examination of GA based testing literature. The details of this study and its inferences are given below.

We can see that none of these works have adopted any general operator setting for testing purpose. This inherent non-deterministic nature of the genetic operators makes program testing a demanding task. The strength of using genetic algorithm mainly depends on setting the genetic parameters to their appropriate values which in turn depends on the problem to be solved [18, 38, 138]. This itself is a major challenge faced by testers. We have mentioned some of these challenges and have also pointed out the factors that

are still not considered by the researchers during GA based software testing. Making an unbiased review like this may help to solve the issues in genetic algorithm based software testing and at the same time help future researchers to explore the untouched research areas in GA based software testing. Before going into the details of GA based testing, a brief overview of Genetic Algorithm is given below in section 2.2.3.1.

2.4.1 Brief Overview of Genetic Algorithm

Genetic Algorithm is a type of evolutionary algorithm and is considered as one of the best of all evolutionary algorithms [58]. It is a type of meta heuristic search technique developed by John Holland and works on Darwin's principle of survival of the fittest [74]. The basic steps of genetic algorithm are given in figure 2.3.

procedure Genetic Algorithm
begin
GET THE (Initial Population);
CALCULATE THE FITNESS FUNCTION (Initial Population)
loop
SELECT THE FINAL POPULATION FOR CROSSOVER (Parent population)
PERFORM CROSSOVER ON THE PARENT (Parent population, child)
APPLY MUTATION (Child)
CALCULATE FITNESS (Child)
GET THE NEXT GENERATION (Parent population, Child)
Stop the process when TERMINATION CRITERA
exit loop
end

Figure 2.3 GA basic steps

Genetic algorithm uses the technique of natural genetics, representing a computer model of biological evolution. Genetic algorithms have the ability to solve a variety of optimization and search problems. Several testing techniques use genetic algorithms believing that testing may be carried out in a better way using the natural evolutionary process present in them.

Genetic algorithm identifies an optimal solution for a problem by applying natural evolutionary techniques to a group of possible solutions referred to as "population" [57, 124, 129]. After each generation, a new generation is formed which is better than the previous generation. In figure 2.3, it can be noticed that the series of steps involved in genetic algorithm are population initialization, selection, crossover, mutation and termination [48]. A string of digits called chromosomes are present and each individual of the string is called a gene. Each individual in the population has a fitness value which decides the quality and performance of that individual. Greater the fitness value, better will be the problem solving capacity of an individual [104]. Collection of chromosomes makes up a population. The initial population is created randomly and the fitness of the individuals in the population is calculated. This information is used to select the best candidates for forming the next generation parents. After selecting parents of the successive generation, the next step is to combine these candidates to form the offspring. Crossover operation is used to perform this step. Crossover enables the selection of good features from parents to form the offspring. Mutation is applied to the offspring to create better quality individuals. Mutation is defined as the process of altering the genes in the chromosome. A new generation is chosen from the offspring based on the fitness of the individuals. These individuals are considered as parents of the next generation. This cycle is repeated until a global solution for the problem is obtained.

2.4.2 GA based Testing-Review Stages

Inorder to perform an impartial review of GA based testing, a research question (RQ) is framed which is given below:

RQ: "Can GA based software testing evolve as an attractive technique in software testing industry? If so, what are issues to be sorted out in GA based testing?"

In order to answer the question RQ, we have to consider several factors which improve the quality of GA based software testing. Therefore, we have refined this question into six research questions RQ1, RQ2, RQ3, RQ4, RQ5 AND RQ6 which are given below in Table 2.1. These research questions (RQ) helps to emphasize the strengths and significance of GA based testing review.

Table 2.1 Research Questions

RQ1: In spite of the large volume of works in genetic algorithm based testing, why have some works considered variations of genetic algorithms?

RQ2: What is the effect of population representation and size in software testing?

RQ3: Is there any common method to design fitness function during software testing?

RQ4: What is the general strategy adopted in operator selection and parameter setting during software testing?

RQ5: What is the significance of computing time and convergence in GA based testing?

RQ6: What is the role of coverage in GA based testing?

Research question RQ1 is expected to find out the works which use variations of GA in test data generation and is aimed at finding whether the variation of GA proves to be better than GA for test data generation. RQ2 is expected to find out the population settings used in software testing and their impact on the final outcome during testing. RQ3 analyses the different methods used for designing the fitness function. RQ4 analyses the operator and parameter setting used for software testing and the impact caused by the same during testing. RQ5 stresses the significance of computing time and convergence and RQ6 highlights the role of coverage in GA based testing. The 'Keyword selection criteria' for source selection from the source repository (Transactions, Journals, Conferences, Books) is given below in table 2.2

Table 2.2 Keywords used for selecting GA based works from various source repositories

Genetic algorithm
AND
Software
AND
(Test Data OR Test Case)
AND
(Generate OR Find OR Get)

Using the keyword selection criteria given in table 2.2, we selected nearly 31 papers which clearly fall within our review scope. The category of works selected and works not considered in this review are given below in table 2.3 & 2.4:-

Table 2.3 Selected category of works

Works considered for review

- Using GA for software test data generation
- Using GA for software test case generation
- Using GA for Path testing
- GA based structural testing (mainly considered works in which coverage criteria is set as branch, statement, dataflow and control flow)
- Study effect of GA parameter settings during testing
- Study effect of GA operator settings during testing

Table 2.4 Works not included

Works which are not selected

- Using GA for hardware testing, circuit testing etc.
- Using GA for test data augmentation
- Using GA for software test case minimization, test case prioritization, augmentation, regeneration etc.
- Using GA for Mutation, temporal, service oriented etc.
- Black box/Model based testing using GA, Mutation,

The category of selected works given in table 2.3 defines the condition for the selection of works on GA based testing whereas the category of works in table 2.4 defines under what condition the works are excluded [82]. After defining the inclusion and exclusion criteria, in order to assess the quality of the selected works in literature, a quality assessment check was made in this review [112]. A quality assessment form was used which is given in table 2.5.

Table 2.5 Quality assessment form

Quality Factors	Description	Outcome/Result of quality factors
Q1	Is the aim of the work clearly defined?	Yes/No/Average
Q2	Whether the method is clearly explained?	Yes/No/Average
Q3	Whether the outcome/result of the method clearly explained?	Yes/No/Average
Q4	Whether the limitations of the work/method specified?	Yes/No/Average
Q5	Does the study contribute to our review?	Yes/No/Average

There were several factors in the quality assessment form. In table 2.5., there were 5 factors (Q1 to Q5). Each factor is given a score can have either a value of 0, 1 or 0.5 which corresponds to 'No', 'Yes' or 'Average'. Each work was analysed and a score was assigned for each of these factors [112]. For all the works included in the study, the values got for each of these factors were noted and the total value was calculated. If the total value was below a certain limit, such works were not considered.

After assessing the quality of the works considered in this review, the required data was extracted from each work according to the data extraction form criteria given in table 2.6. The extracted data is supposed to contain all relevant information inorder to answer the research question. The entries of our data extraction form are given in table 2.6. Each work that is selected for the review is verified and the entries mentioned in the data extraction form are noted for each work. The works selected after the quality assessment check is listed here in table 7. All the values corresponding to the entries in the data extraction form were noted.

Table 2.6 Data extraction form

Data Extraction form Entries

- Work Identifier (Author)
- Title of the work
- Description of the work
- Reference of the work
- Works using variations of GA for testing
- Population, selection, crossover &mutation operator
- Parameter settings
- Fitness function design
- Coverage criteria
- Future research directions
- Whether the observations from the works address the research question?

Table 2.7 GA works selected for review

Work	Type of Testing	Purpose	Population Representation	Population Size & Generations	Selection	Crossover Operator & rate	Type of Mutation & rate	Fitness Function	Coverage
Xue-ying et. Al [2005].	Structural testing	Reduce cost associated with test suite reduction using GA	Subset of test cases	Not specified	Roulette wheel selection	One point	1/L where L is the number of bits in the gene	Measure of coverage and cost	Test suite coverage
Xiao, J. et al. [2010]	Structural testing	Bug fixation	Bug id represented as binary	Size of population as 100 & 500 generations	Ratio method	0.8	0.01	"Value(B)=α* priority+β* severity* Has finished(B), where B is the bug and α and β are the weights for priority and severity , Hasfinished (B) may have vale 1 or 0 if the bug is fixed before deadline and if the bug cannot be fixed"	Test data for covering bugs
Latiu, G. I. [2012]	Structural testing	Software path testing using GA	Array of bits	Size set as 40 and generations as 100	Best fitness	One point and rate as 0.75	0.1	"Function of branch distance and approximation level (Branch distance designed based on Korel's branch distance function)"	Path coverage
Ahmed, M. A et al. [2008]	Structural testing	Test data generation for multipath testing using GA	Base 10 representation of alleles	size set 30 and generations as 100	Roulette wheel	Single point and rate set as 0.5 or 0.9	0.1 or 0.3	"FF= Sum of normalized intermediate fitness functions of multiple paths"	Multipath coverage
Pachure, A.et al. [2013]	Structural testing	Automated approach for branch testing using GA	Binary representation,	Size set as 6,10, 16, 20, 26....110 and Generations as 107	Binary Tournament	Two point and rate set 1.0	0.01	"FF= Approximation level + Normalised branch distance"	Branch coverage
Roper, M. et al. [1995]	Structural testing	Used GA for testing C programs for attaining the branch coverage criteria.	Character string,	Size set as 30, 80 and generations as 6, 25	Random selection	Single, double and uniform crossover point	Simple mutation.	"Two fitness function- First one is the weighted hamming distance of predicate value, Second one is the reciprocal of the branch predicate value"	Branch coverage
Jones, B. et al. [1996]	Structural testing	Branch coverage using GA	Binary representation	Size set as 45, generations not specified	Random selection	One point	Simple mutation. Mutation rate decide by user	Measure of coverage	Branch coverage
Pargas, R. P. et al. [1999]	Structural testing	Path coverage & Branch coverage using TGen tool	String of characters	Size set as 100 and the number of generations considered until the specific branch coverage is obtained	Random selection	One point	Simple mutation & and rate as 0.10	Measure of number of common branch predicated in the control flow graph of the program	Path coverage & Branch coverage
Michael, C. C. et al. [2001]	Structural testing	Developed a tool called GADGET which uses genetic algorithms for generating test data for branch coverage of C programs	Binary string	Population size set as 24 and 100 and the number of	Random selection	One point	Simple mutation & rate set as 0.001	"Expressed as Predicate function based on Korel's fitness function"	Branch coverage

Sofokleous, A. A. et al. [2008]	Structural testing	Test data generation framework using genetic algorithms to provide edge/partition coverage	Integers	Size set as 100 and the generations set as 600	Tournament	Inter and Intra crossover	Inter and intra mutation	"For BO-GA , Fitness function F= w1(#edges exec) + w2(#predtrue+ #predfalse)/w1+w2, where w1 and w2 are the weights in the range [0,1] and #edges exec is the number of executed edges and #predtrue and #predfalse is the number of simple predicates evaluated at least once to true or once to false" "For CU-GA, fitness function Fclose-up-= Fvert + 1/ Fdist(c) , where Fvert is the sum of exercised vertices and Fdist(c) focus on the condition that prohibits test case to visit next vertex"	Edge/partition coverage
Chen, C. et al. [2009]	Structural testing	Test data generation for branch coverage	Not specified	Size set as 100	Random	0.4	0.1	Set by analyzing the pre-dominator tree used to construct the EPDG.	Branch coverage
Berndt, D. J. et al. [2004]	Structural testing	Studied the effect of response time in GA based testing	Not specified	Not specified	Not specified	Not specified	Not specified	Not mentioned	Not mentioned
Khor, S. et al. [2004]	Structural testing	Used the concept of GA and formal concept analysis to generate test data for branches	Real value representation	Size set as 250, Generations set as 50	Selection based on the ranking of individuals according to the concepts	Uniform crossover, rate=0.8	Mutation rate = 0.4	"Uses concept analysis Concept pair=(o, a), where o is the object set and a is the attribute set"	Branch coverage
Cao, Y. et al. [2009].	Structural testing	Test data generation of a specific path using GA	Binary representation	Size=80, generations=800	Roulette wheel	0.80	0.15	"Measure of similarity between target path and execution path with sub path overlapped."	Path coverage
Malburg, J.[2011]	Structural testing	Introduced a hybrid method which combines GA based and constrained based test data generation approach	Bit vector	Not specified	Random	Several setting	Several settings	"Integrates approach level (number of unsatisfied control dependencies) and branch distance"	Branch coverage
Zhang, W. et al. [2007]	Structural testing	Test data generation of many paths using GA	Binary	Size set as 90 for one experiment and size set as 130 for another experiment	Roulette wheel	One point, rate is 0.9	One point mutation and rate = 0.3	"Measure of approach level and branch distance level"	Path coverage

Fraser, G.et al. [2012]	Structural Testing	Studied the effect of seeding in test data generation of object oriented programs	Method calls	Size set as 80 and the number of generations set as 10 minute time out (or 1000000 statements)	Rank selection based on fitness	"Special type of one point crossover where the first part of the sequence of statements of the 1st parent is merged with the second part of the second parent and vice versa"	"Mutation probability for test suite = 1/T, where T is the test suite" "Mutation probability for test case= 1/3, where the operations applied are remove, change & insert"	"fitness (T)=,um of abs value of M and M_T and summation of $d(b_k, T)$, where T is the test suite, M is the methods in the program and M_T is the number of executed methods and $d(b_k, T)$ is the branch distance for branch b on test suite T"	Branch coverage
Arcuri, A. et. al [2011]	Structural Testing	Studied the effect of parameter settings for object oriented programs using Evosuite tool	Method calls	Size set as 4, 10, 50, 100, and 200. Generations set as 10 minute time out (or 1000000 statements)	Roulette wheel, Tournament & Rank selection	"Special type of one point crossover where the first part of the sequence of statements of the 1st parent is merged with the second part of the second parent and vice versa"	"Mutation probability for test suite = 1/T, where T is the test suite" "Mutation probability for test case= 1/3, where the operations applied are remove, change & insert"	"fitness (T)= Sum of abs value of M and M_T and summation of $d(b_k, T)$, where T is the test suite, M is the methods in the program and M_T is the number of executed methods and $d(b_k, T)$ is the branch distance for branch b on test suite T "	Branch coverage
Bueno, M.P et al. [2000]	Structural Testing	Path coverage testing using GA	Binary string	Size set as 80	Individuals selected based on Previous knowledge	Single point	Simple & rate set as 0.03	"FT=NC-EP/MEP"	Path coverage
Wegner, J. et al. [2002]	Structural Testing	Test data generator for structural testing of real-world embedded software systems using GA	Integer representation	Generation set as 200	Individual selected based on fitness	One point	Discrete recombination (Different mutation range for each subpopulation)	"Measure of approximation level and normalized predicate level distance"	Statement & Branch coverage
Miller, J. et al.[2006]	Structural Testing	Test data generation for Branch coverage using GA	Integer representation	Population size set as 100 and generations set as 300	Tournament	One point	Uniform random mutation, non-uniform random mutation and Muhlenbein's mutation, is randomly chosen	"Rules to build fitness function for basic relational operations Example Fitness function (where p is a preset penalty value and a & b are expressions):-" "if (a), F=0 for true & F= p for false" "if (a = b), F= 0 for(a=b) & F= abs(a-b)+p for a≠b" "if (a < > b), F =0 for(a=b) & F =p for a≠b"	Branch coverage

								"if (a < b), F=0 for (a< b) & F= abs(a-b)+p for a ≥b" "if (a < =b), F=0 for (a≤ b) & F= abs(a-b)+p for a >b"	
McMinn, P. [2013]	Structural Testing	Studied Impact of cross over	Input vector	Size set as 300	Best fitness	One point, Uniform & Discrete recombination	Breeder genetic algorithm mutation operator applied at an inverse of chromosome length		Branch coverage
Fraser, G.et al. [2012]	Structural Testing	Test suite generation for object oriented programs which cover multiple goals simultaneously	Method calls	Size set as 80 and the number of generations set as 10 minute time out (or 1000000 statements)	Rank selection based on fitness	"Special type of one point crossover where the first part of the sequence of statements of the 1st parent is merged with the second part of the second parent and vice versa"	"Mutation probability for test suite = 1/T, where T is the test suite" "Mutation probability for test case= 1/3, where the operations applied are remove, change & insert"	"fitness (T)=Sum of abs value of M and M_T and summation of $d(b_k, T)$, where T is the test suite, M is the method in the program and M_T is the number of executed methods and d(bk, T) is the branch distance for branch b on test suite T "	Branch coverage
Gong, D. et al. [2011]	Structural Testing	Test data generation for many paths using GA	Integer	Generations set as 70 for one experiment and as 130 for another experiment	Roulette wheel	One point, rate is 0.9	One point, rate is 0.3	Represented as an n-dimensional vector, where each dimension is related with a target path & the fitness function of a target path is given as, FF= Approximation level + Normalised branch distance	Multipath coverage
Pocatilu, P. et al. [2013]	Structural testing	Test data generation for embedded systems based on GA using control flow graph construction	Sbyte (Integer)	Size set as 100 and generations as 100	Roulette wheel	0.1	0.8	"Fitness function is measured as inverse similarity coverage (ISC) , F=len(a)/sym(a), where len(a) gives the length of the path given by chromosome a and sym(a) is the similarity function"	Path coverage
Mao, C. et al.[2013]	Structural testing	Used variation of GA called Quantum inspired GA for test data generation to improve program coverage	Q-bit (Quantum chromosome)	Size set as 30 and generations as 100	Gambling roulette selection	One point, rate is 0.90	Quantum rotation gate technique, rate is 0.05	"Measure of branch distance function for the ith in a program (bchi) and the corresponding branch weight (wi) "	Branch coverage
Liu, D. et al. [2013]	Structural testing	Test data generation using modified GA to avoid premature convergence	Real numbers	Size set as 15 and generations as 30	Not specified	0.8	0.4	Ratio of branch (f) or path of practical coverage of test case to the total branches (z) or paths, F=f/z	Branch & Path coverage
Suresh, Y. et al.[2013]	Structural testing	Test data generation for basis path testing using GA	Binary	Size set as 100 and generations as 500	Fitness based selection	Two point, rate is 0.5	Bit wise, rate is0.05	Fitnes function is a measure of branch distance function based on Korel's branch distance function	Path coverage

Arcuri, A. et al. [2013]	Structural testing	Studied the effect of parameter settings for object oriented programs using Evosuite tool and proved that parameter tuning may or may not give good result. If search budget and time is a constraint, then default value of parameters may be used for such problems rather than going for parameter tuning	Method calls	Size set as 4, 10, 50, 100, and 200. Generations set as 10 minute time out (or 1000000 statements)	Roulette wheel, Tournament & Rank selection	"Special type of one point crossover where the first part of the sequence of statements of the 1st parent is merged with the second part of the second parent and vice versa"	"Mutation probability for test suite = 1/T, where T is the test suite" "Mutation probability for test case= 1/3, where the operations applied are remove, change & insert"	Sum of the normalized branch distances of all branches in the program under test	Branch coverage
Fraser, G.et al. [2013 & 2014]	Structural testing	Extended the GA based Evosuite tool to a memetic algorithm based approach to improve the performance of GA during test data generation	Sequence of statements of length l	Size set as 5, 25, 50, 75, 100 and Generations set as 10 minute time out	Rank selection based on fitness	"Special type of one point crossover where the first part of the sequence of statements of the 1st parent is merged with the second part of the second parent and vice versa. If P1 & P2 are parents then O1= αP1. (1-α)P2 & O2= αP2. (1-α)P1 , where O1 & O2 are offspring and α is a random value chosen from [0,1]"	"Mutation probability for test suite = 1/T, where T is the test suite" "Mutation probability for test case= 1/3, where the operations applied are remove, change & insert"	Sum of normalized minimal branch distance	Branch coverage

In table 2.7, using the data extraction form entry criteria, works are listed [2, 5, 6, 14, 21, 26, 28, 29, 47, 49, 50, 59, 79, 83, 89, 95, 96, 97, 100, 104, 105, 113, 114, 116, 117, 123, 126, 136, 139, 140, 141]. Fields like the author and year of work, type of testing, purpose, population representation, population size & generation, type of selection, crossover operator & rate, type of mutation & rate, fitness function and coverage are given in table 2.7. These works are analyzed during the review process. The next step was to consider the factors in data extraction form entries which were not included in table 2.7. Table 2. 8 give a list of works which use some variation of GA for software testing. Table 2.8 includes the works formed based on the data extraction form criteria which is not included in table 2.7.

Table 2.8 Works on variations of GA

Work	GA VARIATION	RESULT
Xue-ying et al. [2005]	New type of GA called GeA for reducing cost associated with test suit reduction	GeA is better than ordinary GA
Latiu, G. I. [2012]	New approach based on simulated annealing for path testing	Simulated annealing based approach is better than GA based approach
Malburg, J.et al. [2011]	Hybrid method which combines GA based and constrained test data generation approach	Hybrid method proved to have better performance
Khor, S. et al. [2004]	Used concept analysis along with GA for structural test data generation	Concept analysis with GA is better than simple GA
Wegner, J. et al. [2001]	Applied GA and multi-population GA for structural testing of real-world embedded software systems	Proved multi-population to be better than simple GA
Fraser, G.et al. [2013 & 2014]	Extended the GA based Evosuite tool to a memetic algorithm based approach to improve the performance of GA during test data generation	Proved that memetic algorithm based GA approach gives better branch coverage than GA
Galeotti, J. P.et al. [2014]	Integrating DSE(dynamic symbolic execution) approach with GA based Evosuite tool for test data generation of programs	Gives better code coverage than GA
Mao, C. et al.[2013]	Used variation of GA called Quantum inspired GA (QIGA) for test data generation to improve program coverage	As the search space is enlarged, the new QIGA avoid local optimal solution compared to GA
Liu, D. et al. [2013]	Used a modified GA	Modified approach gives higher test data efficiency and avoids premature convergence compared to GA.
Lin, P. et al. [2012]	Used adaptive GA for test case generation	Adaptive GA gives better performance compared to simple GA

List of works using variation of GA given in table 2.8 are used to check whether GA based testing or works using variation of GA performs better[49, 50, 56, 83, 89, 92, 95, 96, 97, 136, 140]. In table 2.9, given below, the limitations and future research directions of the works considered in table 2.7 are given. This factor is also an element of the data extraction form.

Table 2.9 Limitation & Factors to be resolved in future

Work	Limitations & Factors to be resolved in future
Xue-ying et al. [2005]	- Effect of different operators and parameter setting may be studied -Optimal/Best parameter setting may be found by conducting more experiments - Dependency between different test cases may be discussed and the effect of applying metaheuristic on the same
Xiao, J. et al. [2010]	-Operator and Parameter setting may be studied in detail - The side effects of fixing the bugs may be discussed. In other words, solving the dependency issue between the bugs
Latiu, G. I. [2012]	- To find the reason why GA based method is less competing than PSO and SA - Study parameter settings
Ahmed, M. A et al. [2008]	- How to identify target paths, Handle source code size - Study the dependency issue between multiple paths during testing - Study operator and parameter settings
Pachure, A.et al. [2013]	- How to identify target paths, Handle source code size - Study the dependency issue between multiple paths during testing - Study operator and parameter settings
Roper, M. et al. [1995]	-Solving dependency issues - Operator and Parameter setting variations
Jones, B. et al. [1996]	- Scalability of the approach, Handle source code size -Solving dependency issues - Operator and Parameter setting variations
Pargas, R. P. et al. [1999]	- The authors claim that their approach is scalable. Experiments are needed to prove their claim(Handle source code size) -Covering full dependency in the program using a program dependence graph instead of control flow graph - Operator and Parameter setting variations
Michael, C. C. et al. [2001]	-Method to identify the predicates should be found out -Methods to prioritize branches during testing -Solving the dependency issues in the program -Operator and Parameter setting variations -Handle source code size -As the tool is developed for C program, extend the tool for other languages
Sofokleous, A. A. et al. [2008]	- The authors claim that their approach is scalable. Experiments are needed to prove their claim -Covering full dependency in the program using a program dependence graph instead of control flow graph - Operator and Parameter setting variations - Handle source code size

Chen, C. et al. [2009]	-Solving dependency issues and finding rules for setting parameters and measures needed to improve fitness function
Bernadt, D. J. et al. [2004]	- May study operator setting variations
Khor, S. et al. [2004]	Even though, the authors claim that their approach does not use any flow graph, they have also mentioned that it is difficult to cover nested predicates using their approach. This issue is to be solved in the future.
Cao, Y. et al. [2009].	-Automatic selection of path from control flow graph -Study the influence of different type of operators and parameter setting
Malburg, J.[2011]	-Methods to improve the coverage using different options of fitness function
Zhang, W. et al. [2007]	-Full automation -Finding Suitable value of Subpopulation size
Fraser, G.et al. [2012]	-Study different settings for parameters and operators during test suite generation - Study the effect of seeding for other search based techniques
Arcuri, A. et al. [2011]	-May extend the work to different languages
Bueno M. P. et al. [2000]	- Study operator and parameter settings - Better methods to solve random variations
Wegner, J. et al. [2002]	-Assumes that the target is already given -Multipath coverage not discussed -Study the dependency issue in the program when using multiple subpopulation during testing - Study operator and parameter settings -Handle source code size
Miller, J. et al. [2006]	-Handle complex data structures like arrays, improve scalability and improve path coverage -Study operator and parameter settings
McMinn, P. [2013]	-May conduct study on various types of systems to study the impact of crossover, so that the method may be generalized
Fraser, G.et al. [2012]	-Methods to handle collateral coverage of individuals (A the overhead increases when collateral coverage is considered) -Study the effects of different coverage -Test the approach using other type of algorithms like simulated annealing in addition to genetic algorithms
Gong, D. et al. [2011]	-Full automation of the approach -As the initial population is split-up into subpopulation, assigning correct size to the subpopulation remains an issue -Study operator and parameter settings
Pocatilu, P. et al. [2013]	-Fitness function improvement -Compare the performance with other genetic algorithm based test data generation methods
Mao, C. et al.[2013]	-Fitness function improvement -Improve coverage of the method, Handle source code size
Liu, D. et al. [2013]	-Improve time efficiency (Parameter tuning & parameter selection)
Suresh, Y. et al.[2013]	-Improve path coverage, Handle source code size -Handle complex program
Arcuri, A. et al. [2013]	-Study the effect of parameter tuning to different kind of problems [As parameter tuning may/may not result in worse result compared to the result obtained without parameter tuning]
Fraser, G.et al. [2013 & 2014]	- Find optimal parameter configurations as the result of the suggested work depends on the class on which test data generation is applied

From table 2.9. given above, by analysing the factors to be resolved in future in GA based testing, an idea of practical difficulties related to using GA in practical software testing is highlighted. Unless these issues are handled, using GA in software testing will be of little use.

2.4.3 Discussion on GA based Testing Review

In this section, we have made a structured ordering of some of the most relevant observations of the works referred in GA based testing in table 2.7. This is done with the intent of answering the research questions (RQ1 to RQ6) related GA based testing which is given in section 2.2.3.2. Each research question (RQ) is discussed in this section. Finally, some inferences are made by referring the works on GA based software testing.

RQ1. In spite of the large volume of works in genetic algorithm based testing, why have some works considered variations of genetic algorithms?

Even when a group of researchers claim that GA is best for software testing, we can see that a lot of works have used several variations of GA for software testing. This is evident from the list of works given in table 2.8. Most of these works shows that variations of genetic algorithm perform better compared to simple GA.

Table 2.10 Observations on works using GA variation for software testing

Number of works using GA variation for test data generation	Number of works in which variation of GA is proved to be better than GA	Number of works in which GA is proved to be better	Comments
9	9	NIL	Since variation of GA is proved to be better than GA in all works, it's high time to assess which variant is best in software testing

Table 2.10 gives the observations on works using variation of GA for software testing. Form the list of observations given in table 2.10 and data collected from various works, we can see that, till now the researchers are not able to conclude which type of GA or what variation of GA is best for various software testing strategies. From the list of works which uses GA variation for software testing, we can recommend that the researchers should try to focus research in the direction which is represented in Figure 2.4.

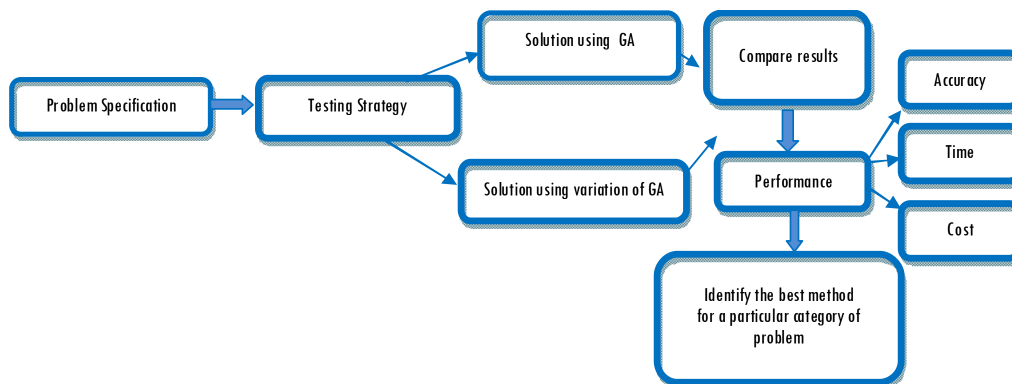


Figure 2.4 Research direction suggested from the observations on works using GA variation

From figure 2.4, we can infer that, depending on the system under test or problem, the type of GA or the variation of GA used also differs. This is one of the areas which future researchers should concentrate upon. They should clearly elucidate which type of GA or what variation of GA is universally applicable to a particular category of problems during software testing. For example, a researcher should be able to identify or get a clear idea of what type of GA or GA along with some method or any variation of GA gives the best result for a particular category of problem. Little works have shown interest in this issue till date. This may be due to wide possibilities or variations of GA which may be applied during software testing. We

recommend that, trying to perform such an in-depth study will clear the air regarding the use of GA in software testing to a great extent. The main benefits of the suggested research direction are:-

- *Programmers will be confident about the type of GA/GA combined with any other method/GA variation, that may be apt for a particular category of problem*
- *Unnecessary effort and time spent to find variants of GA in software testing may be reduced.*

RQ2. What is the effect of population representation and size in software testing?

After the discussion on RQ1, the second research question RQ2 on GA based testing is addressed in this section. We have seen different types of population representation and different population sizes and the number of generations used in software testing from the table 2.7. Some observations from table 2.7 are given below in Table 2.11.

Table 2.11 Population representation for structural testing

Number of works using different Population representation for Structural testing							
Binary	Base 10	Character	Array of bits	Integer	Real	Sbyte	Qbit
8	1	1	2	1	2	1	1

In table 2.11, the different types of population representation used in the works listed in table 2.7 are given. From the list of observations on population representation in table 2.11, we can recommend that the researchers should try to focus research in the direction which is represented in Figure 2.5.

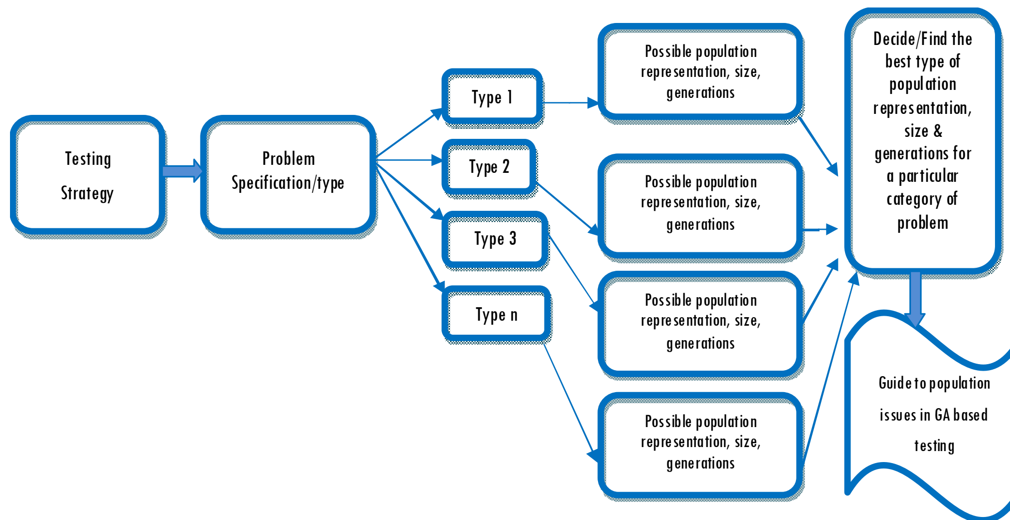


Figure 2.5 Research direction suggested from the observations in population representation in GA based software testing

From figure 2.5, it can be seen that, during GA based testing, for a given problem specification, it is suggested to find the most suitable type of population representation, size and generation. In most of the works which use GA for software testing, it has been mentioned that they have chosen a particular type of representation for population by referring previous works, whereas in some works it is mentioned that they have chosen a particular representation randomly. Only a very few works have given a clear explanation of choosing a particular type of population representation for the system under test. For example, Fraser, G.et al. have claimed that, seeding strongly influences the efficiency of search based test data generation and they have used seeding to improve the process of population initialization [47]. Similar is the case with population size and the number of generations. In such scenarios, if the researchers are able to get a picture of the different types of population representation and the approximate size of population according to the type or category of software testing, the issues related to the size and representation of the population may be solved. For example, if a study is

conducted to finalize the most suitable type of population representation for structural testing of a particular category of problem, software testers may not have any confusion in population representation. Instead of adopting a particular type of population representation, size and generations by referring previous literatures, researchers should try to find a general or the most suitable type of population representation, optimal value of size and number of generations for a particular category of problem during various testing strategies. Researchers have not explored this direction yet. Instead, they have randomly selected a particular type of population representation, size and generation and proceeded directly to the test data generation process. Testing should be done using different population settings and the best population setting should be reported. This in turn may resolve the uncertainties prevailing in population settings during GA based testing. The main benefit of following such a methodology is that:-

- *Problems related to population representation, size and generation in GA based software testing may be resolved to a great extent if general guidelines are formed for population issues.*
- *Researchers will be confident on the method selected if the base values are set according to some general guidelines*

RQ3. Is there any common method to design fitness function during software testing?

In the previous paragraph, the research question RQ2 is addressed and the some research directions are suggested for setting population. In this section, research question RQ3 is addressed. In RQ3, the issues related to fitness function design during software testing are handled.

Fitness function is one of the core aspects of GA based testing. The result of GA based testing depends on fitness function. Design of fitness function differs according to the type of testing, purpose or coverage and the method used for designing the fitness function. Before using GA for structural testing, tester should have an idea about how to design fitness and the factors to be considered for designing fitness function. This is because, many factors such as program dependency and path selection affects the process of fitness function design. After designing the fitness function, parameters are tuned to get the expected result. Response time of the system is also in turn dependent on parameter tuning. Therefore, setting up general guidelines for fitness function design in GA based testing minimizes the issues related to parameter settings in GA. From table 2.7, it can be seen that several approaches are taken for designing the fitness function according to the coverage criteria. One of the most important factors to be considered during fitness function design is the program dependency consideration. In most of the genetic algorithm based software testing, program dependency is not correctly followed [69, 70, 72, 98]. An evidence of this fact may be drawn from the table 2.9. In table 2.9, the future scope of the referred works in this review are listed and it can be noticed that most of the works have mentioned about solving dependency related issues in their future work. Solving dependency related issues in turn depends on the path identification [10]. It is also evident that, most of the works have reported path identification problem as their future research perspective. If there is no automated method to identify the potential paths during fitness function design, all the statements in the program should be analyzed to identify the relevant statements. In GA based software testing, only a very few research works have addressed the problem of potential path identification [73, 142].

Even though fitness function design varies according to the system under test, general guidelines for designing the fitness function based on the testing strategy or system under test or the factors to be considered during fitness function design may be established by the researchers. Though very few works have mentioned about such possibilities, this problem is yet to be discussed in depth. In future, if such a study is accomplished, it will go a long way in making use of GA based software testing applicable to all types of system, irrespective of whether the system is large or small. Unfortunately, little attempt is made for such a study, whereas, most of the effort is spent on designing newer fitness function for testing. Figure 2.6 given below, gives the prime reason for considering the design fitness of function as the most important factor in GA based software testing.

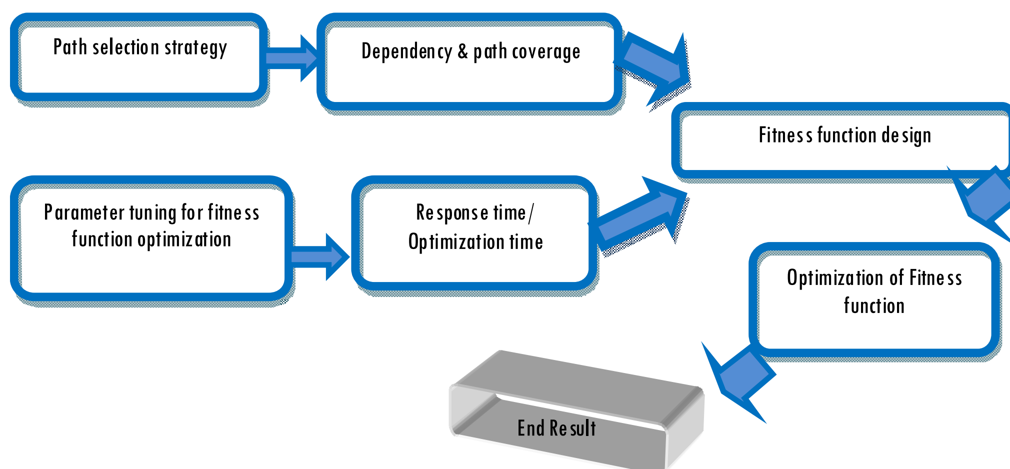


Figure 2.6 Factors affecting and affected by fitness function design

Figure 2.6 summarize the steps to be taken while designing the fitness function during GA based testing. In figure 2.6, it can be noticed that path coverage and dependency should be considered while designing the fitness

function. The end result will be dependent on the optimised value of fitness function. The value of fitness function will be dependent on some factors like parameter settings, parameter tuning, response time etc. Therefore, designing the fitness function in GA based testing needs great care. In table 2.12, some of the main issues to be considered during fitness function design are given.

Table 2.12 Fitness function design issues & suggested solution

Issues in fitness function design	Suggested solutions
Trace the potential paths/relevant paths for designing fitness function	Use methods to attain maximum coverage, identify potential paths and trace dependency while designing fitness function
Trace all possible dependencies in the program so that test data generation using GA may be applied in all practical situations.	
Try to attain maximum coverage during test data generation	

From table 2.12, it can be noticed that tracing dependency in a program, identifying relevant paths in a program and attaining maximum coverage for a program are some of the main issues to be considered during fitness function design. The suggested solution given in table 12 points that, some good strategy should be used inorder to attain maximum coverage, identify potential paths and trace dependency so that the optimal result may be obtained during program testing. Therefore, it is suggested that, the future researchers should give more emphasis on finding general methods for fitness function design according to the category of problem specification. The main steps to be followed during fitness function design in GA based software testing are given in figure 2.7.

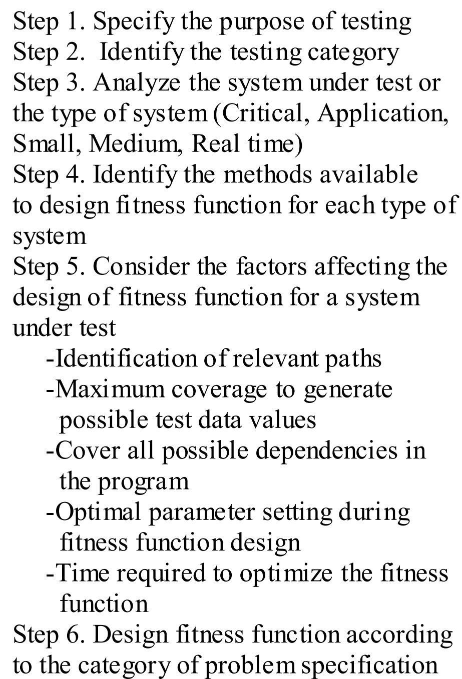


Figure 2.7 Fitness function design

In figure 2.7, the general strategy for designing fitness function for a given problem specification is listed. If such a strategy is used for designing fitness function, GA based testing may be used for testing different type of systems such as critical applications, small type applications, real time systems etc. The main advantage of using such a general strategy for fitness function design is given below.

- *One of the main issues of using GA for practical testing may be solved by alleviating the difficulties faced during the design process of fitness function*

RQ4. What is the general strategy adopted in operator selection and parameter settings during software testing?

Finding an optimal value of parameters is one of the most important issues in GA based testing which haven't received much attention till now. In table 2.7, the operators used and the parameter settings used in various works are shown. From table 2.7 it can be noticed that, even though several works which explain different types of operators and their relevance in different contexts exist, use of these operators in specific context still remains unexploited. Table 2.13 shows an evidence of this factor.

Table 2.13 Structural testing: Number of works using different types of selection in Table 2.7

Types of Selection	No: of works using different types of Selection
Tournament	4
Roulette wheel	8
Ratio	1
Binary tournament	1
Random selection	6
Rank based selection	8
Gambling Roulette wheel	1

In table 2.13, the number of works which use different type of selection is given. Similar is the case with other operators such as crossover, mutation etc. It can be noticed that in most of the works, the operators and parameters are set randomly or they have used specific operators and parameter settings by referring to some similar works in their field. Few works by researchers like Fraser, G. et al. and McMinn, P. have considered such a possibility of studying the parameter setting during GA based software testing [5, 6, 45]. Even these researchers claim that more studies are required to reach a concrete conclusion. In GA based testing, even after testing a program using the best available genetic operators and parameters, a better solution or the same solution can be obtained even if we use less competing methods of crossover, selection and

mutation for solving the same problem. This shows the uncertain nature of genetic algorithms [22]. Most of the works given in table 2. 7 have mentioned about this risk. Another issue involved in parameter settings is the time taken for optimizing the fitness function. Fitness function optimization is a heuristic process and the optimization time and effort varies according to the nature of the problem [2, 14, 58, 93, 94]. Therefore, the exact time required for testing a program cannot be accurately predicted [3]. The time varies as the parameter settings are changed.

Another factor which has a major role in parameter tuning is the ‘search budget’ [5]. As budget plays a critical role in software testing, parameter tuning should be a factor of the budget. From table 2.9 we can see that most of the works have reported operator selection and parameter tuning as their future enhancement. It is surprising that little attempts were made on how to overcome the issues in operator and parameter settings. One reason for this could be the difficulties in carrying out and designing such a study. Using GA based testing, even though a problem may be solved in less time, the field is still dormant to decide the best possible combination of parameter settings suitable for a given problem. Given a problem specification, although it may not be possible to find the exact value for all parameters, we suggest that the researchers should look into the future issues in parameter settings which are given in figure 2.8.

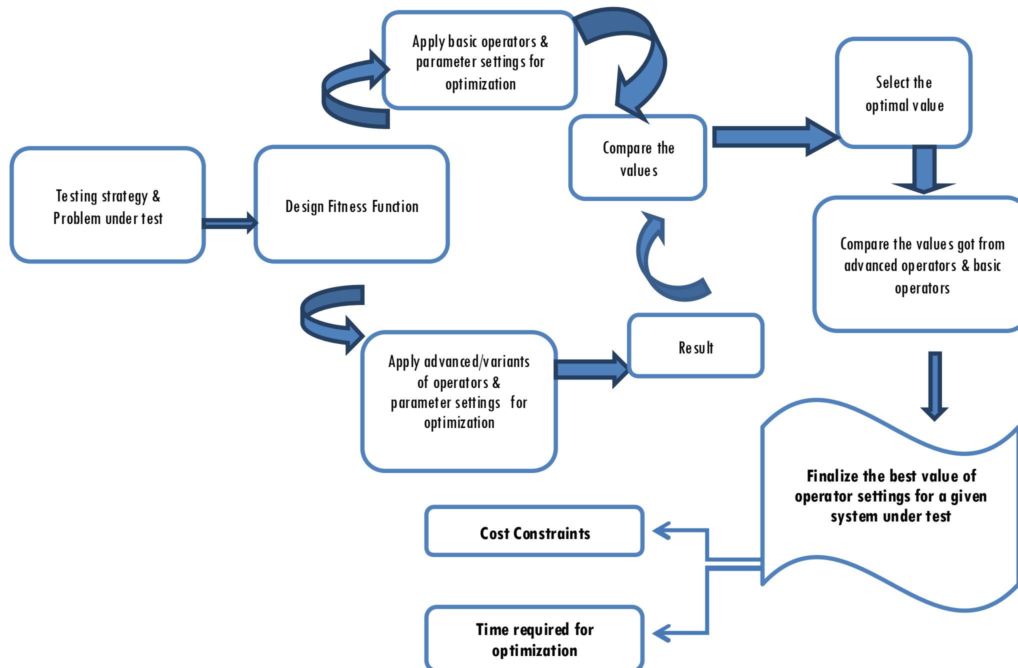


Figure 2.8 Suggested research directions in parameter settings

In figure 2.8 we can see that, after identifying the properties of the system under test, the next step is the design of fitness function. After finding the fitness function, the next step is to find the optimal value of parameters needed to optimize the function. For this, we have to analyze the outcome of applying basic and advanced operators during software testing. Finally, the best possible operator and parameter settings for a given category of problem may be found out. After finding the optimal parameter settings, the relation between fitness function optimization and time taken for optimization, as well as the relation between fitness function optimization and search budget may be identified. A very few works have even suggested such a possibility [5, 6, 24, 45, 111]. Advantages of the above mentioned suggestions are:-

- *General guidelines for setting the parameters according to the problem*
- *Time & effort spent in parameter tuning may be minimized*
- *Solves the ambiguities in setting parameters during GA based testing*

RQ5. What is the significance of computing time and convergence in GA based testing?

In RQ5, the importance of computing time and convergence in GA based testing is discussed. Finding the convergence criteria and computing time is an important step in GA based testing. Convergence defines the point at which the fitness function reaches an optimal solution [74]. In GA based software testing, the execution proceeds until the fitness function converges. Execution stops when the solution converges to a designated point. Sometimes, the fitness function may not converge and in such situations it is difficult to find test data. In such cases an optimal value of test data is considered. In GA based software testing, sometimes the solution converges in a fewer number of generations or after a large number of generations or may not converge itself. In such situations, inorder to avoid infinite number of executions and to finalize the optimal value of fitness function, stopping criterion must be predefined. Most common stopping criterion is the 'number of generations' required to find the test data during software testing. This is a trial and error procedure. Sometimes the target paths are infeasible and the search for test data to cover such paths never succeeds. In such situations the optimal value of test data obtained for a specified number of generations are considered. Generations decide the number of runs or the number of times the whole genetic cycle may be repeated to obtain the required solution. In table 2. 7, the values of generations used in various works are given. As it is not possible to repeat the whole process indefinitely, a stopping criterion should

be defined. In other words, the stopping criterion decides the cause of algorithm termination. Genetic algorithm terminates when the required result is obtained or for a specific number of generations or when there is no improvement in the fitness value which is the best in a specified time interval. Usually in GA, as the number of generation increases, the end result also improves, which in turn causes an increase in execution time of the whole process. Therefore generation is a critical factor in GA based testing.

Computing time is an important factor which influences the process of test data generation in GA based software testing [58]. Computing time is the time taken by the fitness function to reach an optimal solution or the time taken for convergence. Usually in GA based testing as time increases, the quality of solution obtained also increases. But it cannot be assured to get a better solution in all situations. The execution may continue infinitely without converging to a solution. Therefore, in GA based testing, it is better to set the convergence criteria as a certain number of generations instead of a fixed time. In figure 2.9, the suggested research directions in convergence criteria are given.

Figure 2.9 Suggested research directions in convergence criteria

In figure 2.9, it can be noticed that, if the program tester gets an idea about the number of generations in which the fitness function convergence occurs for a given class of problem, unnecessary time wastage can be reduced.

Unnecessary program execution may be avoided with the hope of getting better result in the successive generations. Advantages of suggested research focus are:-

- *Avoid unnecessary computation*
- *Save computation time*

RQ6. What is the role of coverage in GA based testing?

Coverage is one of the important aspects of GA based testing. Various coverage criteria used in different works are given in table 2.7. The test data obtained should cause the execution of statements, branches or paths in the program. During program testing, the main aim is to attain maximum coverage so that the test data generated executes all the constraints present in the program. In some scenarios, the test data generated will not cover the infeasible paths present in the program. Unnecessary effort is wasted in exploring such infeasible paths. Therefore, in some programs, a control flow coverage criterion is considered. Control flow coverage is based on the control flow of the program. Control flow coverage may not give relation between statements in a program. Attaining a complete coverage of the program will be difficult in such situation. Therefore, data flow coverage criteria is used in some works [42]. In data flow coverage, data dependence analysis is used to identify the relation between the statements and paths which lead to solution. Based on the coverage criterion, the fitness function is designed which leads to solution. Therefore coverage plays an important role in GA based software testing.

For a given type of problem, we should try to find out the best coverage criteria. For simple problems, statement or branch coverage will be sufficient. For large and complicated problems, data flow coverage may be appropriate.

Researchers should try to find out the appropriate coverage criteria according to the type of problem. After finding the best type of coverage for a particular class of problems, general guidelines to be followed for designing the fitness function are to be devised. Advantages of suggested research focus are:-

- *If the tester is having an idea of the type of coverage to be used for a particular class of problem, unnecessary effort spent in exploring the unfeasible paths may be prevented*
- *Simplifies the fitness function design process*

2.4.4 An Illustration of using GA in Software Testing

In the previous section, the issues in GA based software testing and their suggested solution is studied. After reviewing various works on GA based testing, an illustration of GA based testing using the work mentioned by Michael et al. [104] is given below. This was carried out inorder to an experience of practical difficulties related to GA based software testing. The end result of the experimental work gives supporting evidence for the literature study on GA based testing.

Consider the lines of source code given below in figure 2.10. Here, the goal is to find test data which satisfies the criterion *subject grade="A.*

```
if (total credit >10)
{
if  (subject credit>4)
subject grade="A";
}
```

Figure 2.10 Lines of code

After defining the lines of code, the test data generation using GA is carried out. Steps to be followed for generation test data are given below in figure 2.11.

Problem: – Find Test Data to satisfy subject grade ="A" using GA

1. Satisfy the condition (subject credit>4)
2. To satisfy condition (subject credit>4) we have to satisfy the criteria (total credit >10)
3. Generate the initial population
4. Calculate fitness
5. Create new population from existing
   - -Crossover
   - -Mutation
6. Check whether the new population satisfies the goal
7. Continue the process until relevant test data is found

Figure 2.11 Test data generation steps

In figure 2.11, the main steps to be followed during test data generation are given. Inorder to satisfy the condition *(subject credit>4)*, the condition *(total credit >10)* should be satisfied initially. The two paths are given in table 2.15. In test data generation, the initial population was generated randomly. After generating the initial population, the fitness of the individual was calculated based on fitness function. The fitness function was designed based on Korel's branch distance function [85, 104]. If the initial population did not satisfy the goal, new population was generated from the existing population using crossover and mutation operators. The new population generated is checked to verify whether it satisfies the required criteria. If the new population satisfies the goal, the process is stopped. Otherwise, the next individual is generated from the existing individual using genetic operations. This process is continued until the goal is found out. Table 2.14 gives the parameter settings used in GA based testing.

Table 2.14 Parameter settings used in GA based testing

GA Parameters	Value
Population size	3
Fitness function	Branch distance function
Representation	Binary, Length=4
Selection	Fitness based
Crossover	Single
Mutation	1 bit
Termination criteria	Until Condition satisfied

From table 2.14, we can see that the population size is set as 3, fitness function is Korel's branch distance function, crossover is single bit, mutation is one bit and the population is represented as binary and the length of the individual is taken as four.

Table 2.15 Paths to be handled in GA based testing

Path 1	Path II
(total credit >10)	*(subject credit>4)*
F(I)= total credit -10	F(II) = subject credit-4

Table 2.15 given above describes the two paths *(total credit >10) and (subject credit>4)* which are to be satisfied during testing. The paths are represented as expressions (total credit -10) and (subject credit-4) respectively using branch distance function.

Table 2.16 Korel's branch distance function

Branch Predicate	Branch Distance Functions
$a=b$	f(K)=-abs(a-b)
$a\neq b$	f(K)=abs(a-b)
$a>b$	f(K)=a-b
$a\geq b$	f(K)=a-b
$a<b$	f(K)=b-a
$a\leq b$	f(K)=b-a
$K1 \wedge K2$	f(K)=min(f(K1),f(K2))
$K1 \vee K2$	f(K)=f(K1)+f(K2)

The fitness function used for GA based testing is designed based on the Korel's branch distance function given in table 2.16 given above. For each type of branch predicate, the respective branch distance functions are given in table 2.16. In table 2.17 and 2.18, the test data generation steps related to the expression F(I)= total credit -10 and F(II) = subject credit-4 are given In table 2.17, the initial population is generated is 8, 4, 2. Since the initial population does not satisfy the goal, genetic operations are applied to generate the next generation of population. The next generation population satisfies the goal by generating the test data value of 13 which satisfies the path (total credit -10). Therefore the process is terminated.

Table 2.17 Test data generation steps of F(I)= total credit -10

F(I)= total credit -10	8 , 4, 2
Initial random population	8 , 4, 2
F(1)=8-10	-2
F(I)=4-10	-6
F(I)=2-10	-8
Crossover (single point)	1 0 00 (8) - 1100 0 1 00 (4) - 0000
Offspring	110 0
Mutation (one bit)	1101 (13)
Path I- (total credit >10)	13 > 10
Test Data	13

Table 2.18 Test data generation steps of F(II) = subject credit-4

F (II)= subject credit - 4	2, 3, 4
Initial random population	2, 3, 4
F(I1)=2- 4	-2
F(II)=3-4	-1
F(II)=4-4	0
Crossover (single point)	01 0 0 - 0110 00 1 1 - 0001
Offspring	0110
Mutation (one bit)	0111
Path II (subject credit >4)	6>4
Test Data	6

In table 2.18, the initial population is generated is 2, 3, 4. Since the initial population does not satisfy the goal, genetic operations are applied to generate the next generation of population. The next generation population satisfies the goal by generating the test data value of 6 which satisfies the path (subject credit-4). Therefore the process is terminated.

Practical difficulties experienced are given below:-

- How to identify relevant paths?
- Limit the functionality to only what we need
- How to check dependency?

  OR

- How a change in a statement affects other program parts?

2.4.5. Inferences from the Review on GA based Software Testing

Some inferences are made based on the GA based review conducted in the above section. From the review it can be inferred that:-

- The main shortcomings reported in GA based testing literature is that, it does not handle source code size and it is difficult to identify the relevant paths and identify dependencies during testing.
- One of the main issues of using GA in software testing is the design of fitness function. Designing a wrong or misleading fitness function will not cause convergence of the fitness function. This in turn hinders the process of test data generation.
- There should be some general guidelines for setting the parameters. Though it may not be possible to set universal guidelines for all problems in GA based testing, efforts should be made to set rules for parameter setting at least for a particular type of problem.
- Though the quality of the solution obtained improves with the increase in time, it cannot be guaranteed to obtain an optimal solution in the specified time interval.
- Implementing GA based testing after considering all the conditions mentioned above is practically difficult. In other words, solving the problems mentioned in RQ1, RQ2, RQ3, RQ4, RQ5 and RQ6 may make GA based testing as one of the strongest methods in software industry. If the researchers are able to find a solution for the future issues mentioned in RQ1, RQ2, RQ3, RQ4, RQ5 and RQ6, the uncertainties which exist in using GA based methods in practical software testing may be eliminated.

From the above inferences it can be noticed that one of the main issues in GA based testing is to identify the relevant paths. On the other side, there is no method to handle source code size during program testing. This is a major problem in structural testing, as it is impossible to test all the lines in a source code which contains a large number of LOC (Lines of Code). Again, as the statements of interest could not be handled by GA based testing, identifying dependencies was another main issue. Apart from this, GA based software testing also has many other shortcomings such as fitness function design issue, parameter setting, uncertain convergence time etc. In order to handle the issues of source code size, identifying dependencies and identifying relevant statements, we introduce the concept of using program slicing in testing.

2.5 Slicing based Approaches

In the previous section, we noticed that the main difficulty to be addressed in most of the other testing techniques is the source code size [53, 65]. GA based testing, which is one of the most widely used techniques during the last decade also faces the same issue [73]. This may be inferred from the detailed review conduced in the previous section. If the tester is able to concentrate only on the needed statements or statements of interest during program testing, then testing would have made easier and effective. The same idea was put forward by Mark Weiser in 1979 [137]. Weiser noticed that programmers concentrate only on the statements of interest during program testing and this process is known as program slicing. He stated that the whole program is divided or sliced into a number of small segments so that programmers can concentrate only on relevant statements during program testing. This makes program testing easier. As originally defined by Weiser, a "program slice consists of those statements which are (potentially) related to

the values computed at some program point or variable referred to as slicing criterion [136].

As slicing can handle the issue of source code size, the practical problems experienced during GA based testing can be easily handled by slicing based testing. Though slicing is used in many applications like fields like program comprehension, debugging, software maintenance, program cohesion, refactoring and reverse engineering [15, 16, 36, 51, 65, 81, 84, 88], works which explicitly use slicing for handling source code size during structural testing is almost nil. Taking into consideration these facts, a slicing based approach is introduced for software testing in this thesis.

We have explained some main types of slices in this section. For example, a static slice gives program statements which can influence the given variable at a particular point known as slicing criterion. The original definition of slicing given by Weiser is that of a static slice [137]. The concept of dynamic slice was given by Korel & Laski [86]. Dynamic slice consists of program statements which can influence the specified variable at a particular point for a particular execution or input. This together is known as slicing criterion. Since dynamic slices traces the changes related to a particular variable, it is very useful during debugging [87]. A program can be traversed either in the forward or back ward direction. If a program is traversed in the backward direction, the set of statements that affect the slicing criterion are identified. This type of slicing is known as backward slicing. The original slice introduced by Weiser was static backward slices [128]. Traversing a program in the forward direction gives forward slices. In forward slice, the set of statements which are influenced by the slicing criterion are identified [12]. Forward slices are very useful in identifying the statements affected by a variable & they are very useful to identify the initialization errors [64].

Chopping is another variation of slicing. It was introduced by Jackson and Rollins [76]. Chopping can be considered as a generalization of both forward and backward slicing. In chopping two sets of variables called source and sink are present. Chopping helps to identify the statements affected by the source, which in turn affect sink. They are most useful in identifying the transmit effects of variables on program statements. Another type of slice is called relevant slice [78]. This may be considered as a superset of dynamic slice. In a dynamic slice, the statements which affect the slicing criterion are identified whereas in a relevant slice, all the program statements which could have affected the slicing criterion are identified. This type of slice is useful in maintenance and debugging. Another type of slice is Hybrid slice [78, 122]. In hybrid slice, the features of both static and dynamic slice are included. Hybrid slice was introduced with an aim of increasing the precision of static slice without increasing the computational cost associated with dynamic slice. Hybrid slice is very useful for debugging.

Intra procedural slicing gives slices within a procedure [137]. It gives the statements of interest which affect the variables at a particular statement. Intra procedural slicing is used in debugging, comprehension, maintenance, differencing etc. Inter procedural slicing was put forward by Horwitz et al. for performing slicing across multi procedures [75]. A System Dependence Graph was introduced to perform intra procedural slicing. Object oriented slicing was introduced by Larsen et al. to handle the object oriented features during slicing. In such scenarios, the system dependence graph was extended to a class dependence graph [122]. Aspect oriented slicing was introduced by Zhao to slice aspect oriented programs. An aspect oriented system graph was constructed to slice such programs. The concept of Quasi- Static slicing was introduced by Venkatesh is used in applications in which a group of program

inputs are fixed and the remaining input is unknown. This is useful in debugging and program comprehension [122]. Call-mark slicing was introduced by Nishimatsu et al. This type of slice was introduced with the aim of reducing the cost associated with the construction of dynamic slice [53]. A compromise between static and dynamic slice is established. This type of slice is smaller than static and less expensive than dynamic. Dependence- Cache slicing was introduced by Takada et al. [122]. The PDG containing dynamic information was pruned by this type of slicing. The concept of database slicing was introduced by Sivagurunathan [122]. It is used for slicing databases so that the database operations are correctly accounted. Proposition-based slicing was introduced by Dwyer and Hatcliff [36]. This type of slicing was defined to reduce the finite-state transition system used in verification techniques such as model checking. Here, model checking is done with respect to linear temporal logic.

The concept of incremental slicing was introduced by Orso et al. [109]. This type of slicing allows the user to focus on a particular type of data dependence. This feature is very useful during program comprehension so that the user can ignore the weak data dependences and concentrate on strong data dependences. Almost similar idea is used in thin slicing. This was introduced by Sridharan et al. in 2007. In a thin slice, control and base pointer flow dependences are ignored. Only those statements related by producer flow dependence are included in a thin slice. This type of slicing is useful in debugging and program comprehension. The concept of decomposition slicing was introduced by Gallagher et al. [54]. This is mainly used in software maintenance to trace all the computations on a given variable. The decomposition slicing criterion consists only of a single variable. The concept of amorphous slicing was put forward by Harman et al. [132]. This is a semantic

preserving slice. Amorphous slices are constructed using some program transformation. This simplifies the program and preserves the semantics of the program with respect to the slicing criterion. This type of slicing is used in program comprehension. The concept of abstract slicing was introduced by Hong et al [122]. Abstract slicing extends the concept of static sling by incorporating predicates and constraints that are processed with an abstract interpretation and model checking based technique. Abstract slicing determines under which variable values do the statements affect or are affected by the slicing criterion. The concept of path slicing was introduced by Jhala et al. [122]. Path slicing gives the statements which can possibly influence the reachability of a given statement in a given execution path. Path slicing is used to find program statements which are not executed in a given path. This slicing is mainly used in program comprehension. The concept of conditioned slicing was introduced by Ning et al. [128]. Conditioned slices gives program statements which can influence the variables at a given statement for the initial states which satisfy the given condition. Conditioned slices are useful in debugging, software reuse, ripple effect analysis, understanding legacy code and program comprehension. The concept of backward conditioning slicing was defined by Fox et al. [52]. Backward conditioning gives the program statements which can influence the given variables at a given point when the conditions at the given statements are satisfied. The concept of pre/post conditioned slicing was put forward by Harman et al. [68]. This is a generalized form of conditioned slicing which combines forward and backward conditions. Here the forward conditions are called pre conditions and backward conditions are called post conditions. This is mainly used in program comprehension, re use and verification. Barrier slicing concept was introduced by Jens Krinke [122]. In this type of slicing, the programmer can specify the part of program that is to be

traversed during the construction of slices. Simultaneous Slicing was introduced by Hall. This can be considered as a generalization of program slicing. Here a set of slicing criteria is considered and the slices are computed with respect to a set of different points rather than the set of inputs. Interface slicing was introduced by Beck and Eichmann [122]. In Interface slicing, the module's functionality is extracted using slicing. Interface slicing is done to find the dependences between the components and global variables rather than between statements. Interface slicing is useful in reverse engineering and code reuse. Program Dicing was introduced by Lyle et al.for debugging purpose [128]. It is computed by removing those statements of a static slice of a variable that appears to be correctly computed from the static slice of an incorrectly valued variable. Stop-list slicing was introduced by Gallagher et al. [122]. In stop-list slicing, slicing is performed with respect to the variables on which the programmer is not interested. This type of slicing is mainly used in debugging.

2.5.1 Applications of Slicing

In the above section, an overview of various types of slices is given. This section explains the some of the important applications of slicing. Debugging is one of the important applications of slicing. It was Mark Weiser who suggested that program slicing may be used for debugging. Weiser suggested that during debugging, programmers ignore the parts of the program that cannot influence the bug [137]. Software maintenance is one other important application of slicing. Slicing may be used to find the changes associated with a variable during maintenance. Slicing may be also used for function extraction and restructuring. Extractable functions are identified by slicing. After extracting the functions, they can be restructured unto independent functions using slicing. These restructured functions may be used in refactoring [55]. Another important application of slicing is program differencing. Slicing may be used to find the difference

between two versions of the program. Another main application of slicing is program testing. Even though some works have mentioned the use of slicing in testing [9, 16, 62, 65, 119, 120], works that explicitly demonstrate how program slicing may be applied in software testing is extremely rare to the best of our knowledge. In these works, R. Gupta, et al., D. Binkley and Bates et al. have illustrated the use of slicing in regression testing [9, 16, 62]. Harman et al. have used the concept of robustness slice in testing [65]. In robustness slice based testing, the original program is transformed into a meaning preserving form and after that slicing is applied to the transformed program. Samuel et al. have illustrated how slicing may be applied for designing test cases from UML diagrams [119,120]. Considering these facts, we have introduced and demonstrated the significance of applying slicing in program testing to handle the one of the relevant issues of source code size.

2.5.2 Inferences from the Review of Slicing based Approaches

Based on the review of literature on program slicing, we have made some inferences which are given below.

- Slicing may be used to handle the issue of source code size during structural testing
- In source code testing, almost no works on testing using slicing

2.6 Summary of the Chapter

In this chapter, a review of literature on software testing techniques is performed. The review is carried out in three stages. In the first stage, a review of recent trends in software testing techniques was carried out. Evidence from literature showed that search based software testing techniques is one of the widely used and researched areas during the last decade. In search based techniques, GA based software testing is mostly used in software testing. Based on this inference, a review of literature on GA based techniques

was carried out in the second stage. Based on the review of GA based works in software testing, some inferences were drawn. The main issues observed during GA based testing were:-

- GA based testing was not capable of handling source code length during structural testing
- There were no general guidelines for parameter setting in GA based testing. This greatly affected the end result in GA based testing
- GA based testing could not handle program dependencies completely

In order to handle these issues, some other possibilities were researched. Finally, a program slicing based testing was proposed to handle the practical issues during program testing. A review of literature was made on program slicing techniques and the applications of program slicing. From the review it was noticed that, program slicing may be used for software testing as it can handle the difficulties faced during practical software testing. Finally, some inferences were made based on the literature review on program slicing which are given below:-

- Slicing may be used in program testing
- Works which explicitly utilize the potential of program slicing techniques in testing are rare
- Slicing identifies the statements of interest
- Identifying statements of interest is very useful in software testing, as it helps the program tester to ignore irrelevant statements during program testing

Finally, based on the inferences from literature review, it may be concluded that program slicing based testing can tackle the difficulties faced during practical software testing in a better way compared to GA based testing approaches.

…….ഇൽ…….

Chapter 3

SOFTWARE TESTING USING FORWARD SLICING

Contents

3.1 Introduction
3.2 Background
3.3 Significance of using Forward Slicing in Software Testing
3.4 Architecture of Forward Slicing based Testing (FST)
3.5 Illustration of Test Data Generation using FST
3.6 Proof of Correctness and Formalised Representation of Forward Slicing Algorithm

3.1 Introduction

This chapter introduces a novel concept of using forward slicing in software testing. The main intention of using forward slicing in testing was to handle the issue of source code size during software testing. The architecture, implementation and demonstration of forward slicing based testing (FST) discussed in this chapter help to understand the significance of using forward slicing in testing. The architecture of FST gives an overview of the components used in FST. In the implementation details, it is described how to identify forward slices form the source code. For achieving this, a linked dependency method is used. In the linked dependency method, a forward slicing algorithm is applied to identify the dependency in the source program. In addition, this chapter also discusses how to generate test data from forward slices using a Gauss Elimination approach. A formal representation of the forward slicing algorithm is also introduced in this chapter. Representation of forward slicing algorithm is used for the correctness proof of the slicing algorithm and the formal representation may be used as a foundation for developing verification methods in several other applications.

3.2 Background

"Divide each difficulty into as many parts as is feasible and necessary to resolve it"-René Descartes. We may be familiar with this divide and conquer strategy of problem solving. The same strategy can be adopted in practical software testing. By software testing what we intend to accomplish is to make sure that the actual result of the software after execution, matches its expected result without any error. Testing is a time consuming and labour intensive activity and consumes almost 50% of the development cost [11, 66, 87, 106, 119]. Among the various stages of software testing such as planning, designing and execution, the major challenge is the design of effective test cases [85]. In practical structural testing, testers have to face several challenges during the whole testing process [87]. Unrestricted size of source code is one such major issue as this affects the scalability, consistency and integrity of software systems. During such scenarios in structural testing, using the divide and conquer strategy, the given problem may be divided into manageable number of sub problems [109]. This accounts due to the concept of program slicing introduced by Weiser in 1979 in his Ph.D. thesis [137]. Although research of nearly 25 years has established program debugging, maintenance and reengineering as some of the vital applications of program slicing, the potential of using slicing in software testing has not been fully exploited till now [67, 86, 122]. Further, the works that demonstrate the use of slicing in testing are rarely reported.

To a great extent, use of slicing helps to solve the unresolved issues in software testing such as managing source code size, identifying dependency, prioritizing test cases etc. [87]. For example, programs for commercial applications usually contain several modules. Slicing provides a way to identify the relevant information, so that the programmer may not have to go through the

details of the whole source program during the testing phase and the error related to the variable of interest may be identified easily. For example, if errors are associated with a private variable which is declared in a class, the probability of existence of more errors will be greater in that particular class compared to the other classes in the program. Program statements which are directly or indirectly dependent on the defective variable may have a greater chance of carrying errors [109]. That means that, the probability of existence of additional defects in a software component is proportional to the number of defects already detected in that component [19]. Using slicing in such scenarios gives all the suspicious statements in a program, with respect to the slicing criterion specified for suspicious variables. Therefore in software testing, as slicing identifies the dependency between program statements, the root cause of errors may be identified easily. As we have introduced the concept of using forward slicing in software testing, some general terms and definitions related to program slicing is given in the next section.

3.2.1 Program Slicing

The idea of program slicing was put forward by Mark Weiser in his Ph.D. thesis in 1979 [137]. According to Weiser, “a slice consists of those program statements which are potentially related to the values computed at some program points or variable referred to as slicing criterion which is denoted as C=(S, V)”. Here ‘S’ is the program statement and ‘V’ is the variable of interest. Program slicing can be divided into various types. Based on slicing criteria, the two main types are static and dynamic slice [86], while based on direction of slicing the two main types are forward and backward slice [128]

3.2.1.1 Static Slice

In static slice, only static information is available [137]. The original definition of slicing, put forward by Weiser is static. In static slice, no specific execution sequence is considered. Static slice works for any possible input value. A static slicing criteria and it is represented as C= (x, y) where x is the statement present in a given program and y is the subset of variables present in the program. An example program is given in table 3.1, where the static slice criterion is given as <11, a>. The result will be the set of statements <4, 5, 6, 8, 9>.

Table 3.1 Static slice

Program Statements	Static slice for criterion <11, a>
1 main()	4 cin>> b;
2 {	5 a = 0;
3 inta,b;	6 while (b <= 10)
4 cin>> b;	8 a=a+b;
5 a = 0;	9 ++ b;
6 while (b <= 10)	
7 {	
8 a=a+b;	
9 ++ b;	
10}	
11 cout<< a;	
12 cout<< b;	
13 }	

3.2.1.2 Dynamic Slice

The concept of dynamic slice was put forward by Korel and Laski [86]. As static slices contain all possible executions, it may be difficult to trace the errors. In order to overcome this, the concept of dynamic slices was introduced. Dynamic

slices contain program statements that affect slicing criterion only for a particular input execution. The dynamic slicing criterion is defined as C= (x, y, i). Here ‘x’ is the statement in the program, ‘y’ is the subset of variables in the program and ‘i’ is the input value. A sample program to be sliced is given below in table 3. 2. The variable with respect to which slicing is to be done is p, slicing point is the end of the program and input given is n=0.

Table 3.2 Dynamic slice

Program Statements	Dynamic Slicing Criterion :-(10, p, n=0,)
1 scanf("%d",&n); 2 s=0; 3 p=0; 4 while (n>0) 5 { 6 s=s+n; 7 p=p*n; 8 n=n−1; 9 } 10 printf ("%d%d", p, s);	p=0

3.2.1.3 Backward Slice

During the program slicing, program traversal may be made either in the forward or backward direction [64]. In backward slicing, the program is traversed in the backward direction. Backward slice consist all the statements which affect the slicing criterion. The original definition of slicing, put forward by Weiser is of static backward slicer. Backward slicing criteria is defined as C= (x, y) [17]. Here ‘x’ is the statement number and ‘y’ is the slice variable. Consider the sample program given in table 3. 3. The backward slicing criterion is given as C= (12, i). All the program statements which affect the value of the variable ’i’ in statement number 12 is displayed.

Table 3.3 Backward slice

Program Statements	Backward slicing criterion C= (12, i)

1 main( )
2 {
3 int i, result;
4 result = 0;
5 i = 1;
6 while(i <= 10)
7 {
8 result = result + 1;
9 ++ i;
10 }
11 cout<< result;
12 cout<< i;
13 }
	i = 1;
	while(i <= 10)
	++ i;

3.2.1.4 Forward Slice

The idea of forward slicing was put forward by Bergertti& Carre [12, 64]. In forward slice, the program is traversed in the forward direction. Forward slice gives all the program statements which are affected by the slicing criterion. Forward slicing criteria are defined as C= (x, y). Here 'x' is the statement number and 'y' is the slice variable. Consider the sample program in table 3. 4. The forward slicing criteria is given as C= (3, mark1). The slice consists of all the statements in the program which is affected by declaring the variable 'mark1' in statement number 3.

Table 3.4 Forward slice

Program Statements	Forward slicing criteria C= (3, mark1)

1main ( )
2 {
3 int mark1, mark2, result;
4 result = 0;
5 mark1=40;
6 mark2=35;
7 result = mark1 + mark2;
8 cout<< result;
9 }
	mark1=40;
	result = mark1 + mark2;
	cout<< result;

3.2.2 Terms & Definitions Related to Program Slice

In this section, some terms and definitions which form the cornerstones of slicing algorithms are explained.

Directed Graph (G) - A directed graph 'G' is represented as G= (N, E) where 'N' represents the nodes and 'E' represent the edges. In a directed graph, if (a, b) represent the edges of the graph 'G', then 'a' is called the predecessor 'b' and 'b' is the successor of 'a'

Control Flow Graph (CFG) - A control flow graph of a program is represented as C= (N, E). Here 'N' represents the nodes of the graph which corresponds to the statements in the program and 'E' represents the edges connecting the nodes. In a CFG, the *Start* node represents path from start node to all other nodes in the graph and the *Exit/Stop* node- path from all the other nodes to the stop node

Dominance- In a CFG, if 'a' and 'b' are two nodes, then 'a' dominates 'b' if and only if, every path from the start node to 'b' pass through 'a'. 'b' post dominates 'a' if and only if every path from 'a' to 'stop' node pass through 'b'.

Def- Use pairs-A def-use (du) pair associates a point in a program where a value produced with a point where it is used

Definition: where a variable get a value

Use: extraction of a value from a variable

Data Dependence (DD) - In a CFG, a node 'n' is said to be data dependent on a node 'm' if the variable 'v' of the program is:-

- Defined on node 'm'
- If the node 'n' uses variable 'v'

- There exists a directed path from ‘m’ to ‘n’ and there is no intervening definition of ‘v’

Control Dependence (CD) - If ‘a’ and ‘b’ are the nodes in a CFG of a program ‘P’, then node ‘b’ is control dependent on node ‘a’ if

- ‘a’ is a test node
- There exists a directed path ‘Q’ from ‘a’ to ‘b’ such that, there is no jump dependence between the internal nodes
- ‘b’ does not post dominate ‘a’

Direct Dependence- CFG nodes which use slicing variables are directly dependent on slicing variables

Indirect Dependence- CFG nodes using slice variables which are used by other nodes are indirectly dependent on slicing variables

Having got an idea of the important terms and definitions related to program slicing, the concept of applying forward slicing in testing is explained in the next section.

3.3 Significance of using Forward Slicing in Software Testing

In this thesis, we introduce the use of forward slicing in testing. We have specifically used forward slicing due to a number of reasons. The main objective of this thesis is to handle the source code size. In addition, identifying the relevant statements, generating test data from slices and identifying dependency in the program are the other objectives of this work. Static slices give the statements affecting the slicing criterion. Therefore, using static slicing may not help to fulfill the goal of identifying relevant statements so as to generate test data from the statements. Using forward slicing in testing has several plus points compared to static and dynamic slicing.

Dynamic slicing on the other hand has some features which are in favor of software testing, but it has certain disadvantages which make it inappropriate in certain situations. Applying dynamic slicing to the whole program may result in an increase in computational overhead compared to static slicing approach [87]. Another major hurdle with the use of dynamic slicing is that, the input value or test data which is to be specified in the slicing criteria. If the programmer is totally unaware of the system, then deciding the input parameter value in dynamic slicing criterion may be a challenging task. In most of the works which use dynamic slicing, the input parameter in slicing criterion is assumed by the user. This may not be an issue in applications which uses slicing for purposes other than testing. In testing, initializing the input in dynamic slicing criterion is by itself a quasi-test data generation process which has not received much attention till now. Using backward slicing in testing will give the statements affecting the slicing criterion. Therefore using backward slicing will also cause the same problem as static slicing.

Our work identifies the affected or changed segments of the program code using forward slicing and identifies the statements in the slices which are in the form linear expression. Such expressions are finally solved using Gauss Elimination method for generating test data [27]. If some expressions are in non-linear form, random method is used for generating test data in such expressions.

Finally the work given in this chapter aims to:-

- Demonstrate how to tap the potential of forward slicing for solving practical issues in software testing
- Introduce and highlight the significance of test data generation using forward slicing based Gauss Elimination

- Present new basis for forward slicing algorithm using linked dependency method
- Introduce a formalized representation for forward slicing which may be used to design verification tools for large software systems and safety critical systems
- Discuss the merits of using slicing methods

3.4 Architecture of Forward Slicing based Testing (FST)

In this section, the architecture of FST and the various functions are described. The detailed architecture of FST is given in figure 3.1.

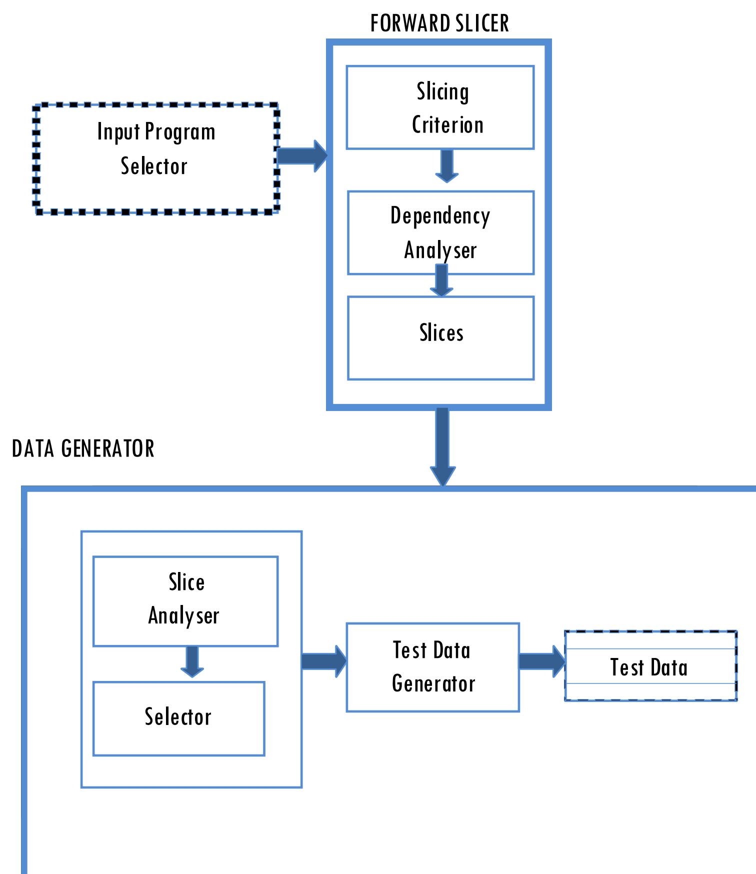


Figure 3.1 Architecture of Forward Slicing based Testing (FST)

There are mainly three units in FST. They are:-

- Input Program Selector- Selects input or program on which testing is to be performed
- Forward Slicer- Performs slicing based on the forward slicing algorithm. Sets slicing criterion and finds dependency in the program using linked dependency method
- Data Generator- Generates test data from slices using Gauss Elimination or Random method. Slice analyser present in the data generator unit checks the slices to verify whether they are in linear form or not

We have implemented a prototype tool using the FST architecture given above in figure 3.1.

3.4.1 Input Program selector

Program selector enables the selection of the program. The program selector does not perform any further operations. The selected program is given as input to the forward slicer. The slicer performs the remaining functions related to slicing.

3.4.2 Forward Slicer

Forward slicer is the most important unit of FST. Forward slicer performs some of the most important functions like deciding the slicing criterion, analysing the dependency by linked dependency method and performing forward slicing according to the slicing algorithm. Each function of the forward slicer is explained in detail as given below.

3.4.2.1 Slicing Criterion

Setting up the slicing criterion is the first function performed by forward slicer. After selecting the program by the program selector, the program is given

to the forward slicing unit. For setting up the slicing criterion, two factors are to be considered. First one is the statement number and the second one is the slicing variable. We have used forward slicing in software testing with the aim of reducing the source code size and to identify the relevant statements in the program. Therefore, inorder to identify the relevant statements in the program and to use these statements for generating test data, slicing should be performed with respect to the input variables in the program. The slicing variable is generally considered as the variables declared at the starting point of the program. We have considered the parameter statement number in the slicing criterion as the input variable declaration point in the program. The slicing criterion finally is set up as C= (n, V), where 'n' represents the statement number (initial point of variable declaration) and 'V' is the slicing variable.

3.4.2.2 Linked Dependency Method for Identifying Dependencies

After setting the slicing criterion, the first step is to identify the dependencies in the selected program. The dependency is tracked using a linked dependency method. In a linked dependency method, both control and data dependencies are followed. The main steps in a linked dependency method are given below.

3.4.2.2.1 Steps in Linked Dependency Method

Step 1: Construct the control flow graph (CFG) of the selected program

Step 2: Identify the control flow information from the control flow graph

Step 3: Identify all def-use pairs at each node

Step 4: Identify data flow information in the program using def-use pairs

Step 5: Use the def-use pair at each node to identify the data dependence

Step 6: Mark the slicing criterion C=(n, V) in the CFG
('n' is generally taken as the variable definition point)

Step 7: Identify all the nodes occurring after n which uses slicing variable (Direct dependence)

Step 8: Identify the nodes having indirect dependence

-Nodes using slice variables which are used by other nodes

Step 9: Continue the process until the end of the program is reached

Step 10: Combining all the direct and indirect dependence nodes gives the forward slice for the given slicing criterion

3.4.2.2.2 Explanation of Linked Dependency Method Steps

Step 1: Construct the control flow graph (CFG) of the selected program

This is the first step in linked dependency method. For the selected program the CFG is constructed. Each node in the CFG corresponds to the program statements of the selected program (In the successive sections, nodes of the CFG may be interchangely used with program statements)

Step 2: Identify the control flow information from the control flow graph

After constructing the CFG for the selected program, the next step is to mark the control flow information of each node in the CFG. Identifying the control flow information is very crucial as it is essential in identifying the control dependencies as well as direct and indirect dependencies of a node.

Step 3: Identify all def-use pairs at each node

After identifying the control flow information at each node, the next step is to identify the def-use pairs at each node (which corresponds to program statements). In def-use pair, the variables in each node and their usage in the successive nodes are noted.

Step 4: Identify data flow information in the program using def-use pairs

Data-flow information is identified using the def-use pair. If a variable is defined in a node and if a successive node in the program uses that

particular variable, then there exists a data flow between these nodes. Data flow information is essential in marking the data dependencies in the program.

Step 5: Use the def-use pair at each node to identify the data dependence

After identifying the def-use pair at each node in the CFG and marking the data flow information in the CFG, the next step is to identify the data dependence between the nodes in the CFG. The data flow information is used to identify the data dependence present in the program.

Step 6: Mark the slicing criterion C= (n, V) in the CFG

After identifying the control and data dependences in the program, the node which defines the slicing criterion is marked.

Step 7: Identify all the nodes occurring after n which uses slicing variable

After marking the slicing criterion node, the next step in linked dependency method is to mark all the nodes which are affected by the slicing criterion. For fulfilling this control dependence and data dependence information is utilised. All the nodes in the CFG which are directly control dependent and data dependent on the defined slicing criteria are marked. These nodes or statements in the program are having a direct dependence on the slicing variable.

Step 8: Identify the nodes having indirect dependence

After identifying the nodes which are directly dependent on the slicing variable, the nodes/statements in the program which are indirectly dependent are marked. The nodes which use the variable values in the directly dependent nodes are indirectly dependent on the slicing criterion. These indirectly dependent nodes are also affected by the slicing variable.

Step 9: Continue the process until the end of the program is reached

Marking the directly dependent and indirectly dependent nodes are continued until all the nodes in the CFG are covered

Step 10: Combining all the direct and indirect dependence nodes gives the forward slice for the given slicing criterion

All the directly and indirectly marked nodes are combined together. Finally, these nodes together form the forward slice for a given slicing criterion

3.4.2.2.3. Illustration of Identifying Dependencies using Linked Dependency Method

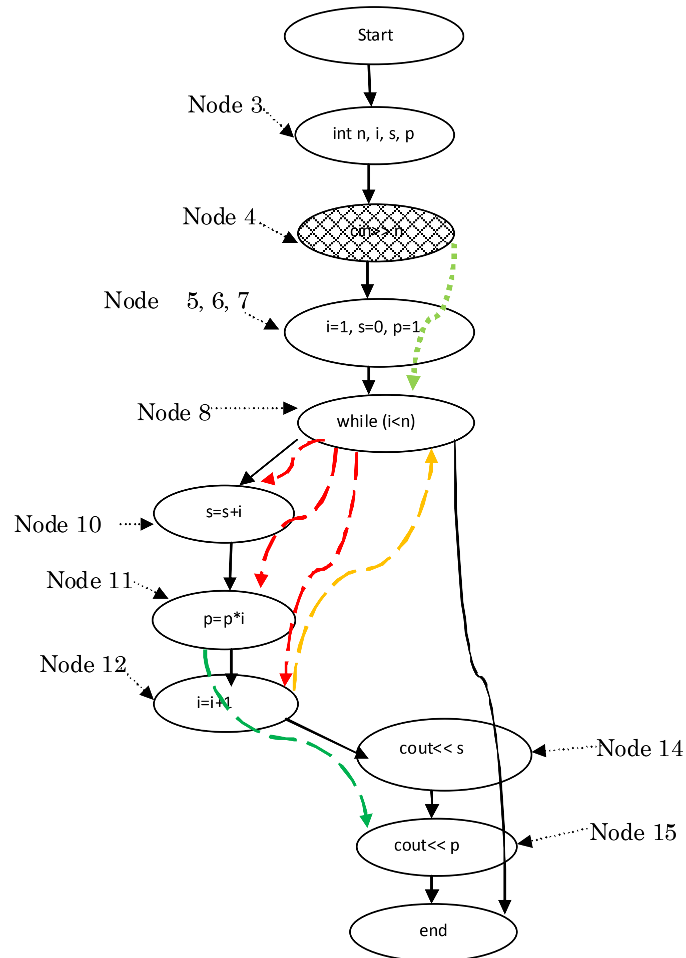
We saw the detailed steps of identifying dependencies in the above section. In this section, an illustration of identifying dependencies using linked dependency method is described.

Consider the sample segment of program code given below in figure 3.2. How dependencies are identified using the linked dependency method is explained in the successive section

```
1.  main ( )
2.  {
3.    int n, i, s, p,
4.    cin>> n;
5.    i= 1;
6.    s=0;
7.    p=1;
8.    while (i<n)
9.     {
10.      s=s+i;
11.      p=p*i;
12.      i=i+1;
13.    }
14.    cout<< s;
15.   cout<< p;
16.  }
```

Figure 3.2 Sample segment of program code

The dependency graph of sample segment of program code is given in figure 3.3 and the description of the graph is as follows:-



Represents control flow

Represent direct dependence

Represents indirect dependence

Figure 3.3 CFG of sample code in figure 3.2

In the CFG given in figure 3.3, the node shaded is set as slicing criterion. The variable ‘n’ present in the node 4 ‘cin>> n’ is the slicing variable. This means that the statements which are affected by the variable ‘n’ in node 4 are to be identified. From the figure 3.3, it is evident that node 8 ‘while (i<n)’ uses the slicing variable ‘n’. Therefore node 8 is directly dependent on node 4. Nodes 10, 11 and node 12 are control dependent on node 8. Therefore, node 10, and node 12 is also affected by the slicing variable. Node 12 affects node 8. Finally node 14 uses the variable value in node 10 and node 15 uses the variable value in node 11. Therefore, node14 and node 15 is indirectly dependent on slicing variable ‘n’. By combining all the marked nodes, the forward slice for the node is obtained. Using the forward slicing algorithm, the marked nodes are verified and finally the forward slices are recorded. The forward slice for the node ‘cin>> n’ or for the slicing criterion C=(4, n) is given below in figure 3.4.

```
8.  while (i<n)
10. s=s+i;
11. p=p*i;
12. i=i+1;
14. cout<< s;
15. cout<< p;
```

Figure 3.4 Forward slice for the slicing criterion (4, n)

3.4.2.3 Forward Slicing Algorithm

The direct and indirect dependencies are identified using the control flow and data flow information present in the program. Finally, forward slicing algorithm is applied after obtaining the control and data flow information to get the forward slices. The forward slice obtained for the node

'cin>> n' for the slicing criterion C= (4, n) in the figure is given above in figure 3.

Initially, we provide some of the terms and definitions used in forward slicing algorithm.

Definition (Forward slicing criteria)- The program statements that are affected by the value of a particular variable (V) at a particular point (n) are given by performing forward slicing with the specified slicing criterion C= (n, V).

n- Statement

L- List where the slice variables are stored

F- Forward Slice

V- Slice variable

LHS -Left side

RHS -Right side

IN- Input Statement

OUT-Output Statement

I- Initialization Statement

EXPR- Expression

COND- Conditional Statement

VAR (L) - Slice variable 'V' stored in list 'L'

RHS (EXPR) - Denotes the right side of the expression

LHS (EXPR) - Denotes the left side of the expression

VAR (RHS (EXPR)) - Denotes variables in the right side of the expression

VAR (LHS (EXPR)) - Denote the variables in the left side of the expression

FORWARD SLICING ALGORITHM
Input: - Program to be sliced (P)
Output: - Forward Slices (F)
begin
1. while p ≠ Ø, source program not empty
// Given n as the statement number and V as the slicing variable

```
2. get C= (n, V)
3. while (n ≠ 0 && n < EOP)   //EOP is the end of program
{
4. Store 'V' in 'L' // Slicing variable 'V' stored in list 'L'
5. if ( VAR (L) Є n)// Check whether slice variable 'V' stored in list 'L' is present
in statement 'n'

    {
5.1. if ( V Є (OUT))
     F= F U n //Store n, initially F will be null and include the statement n as a
slice
5.2. else if (V Є (EXPR))   )
     {
 5.2.1. if ((V) Є RHS (EXPR))
         {
           F=F U n          // Store n

           VAR (L) = VAR (L) U VAR (LHS (EXPR))
         }
5.2.2. else
do not  include the statement as a  slice
        }
5.3. else if (V Є (COND) )
     {
5.3.1. if ((V) Є LHS (COND) OR (V) Є RHS (COND))
         {
           F= F U n //Store n
           F= F U Loop body statements // Include all statements inside the
conditional loop in F
         }
5.3.2. else
do not  include the statement as a  slice
     }
5.4. else if (V Є (IN))
      F= F U n // include statement as a slice
5.5. else if (V Є RHS (D))
      F= F U n // include statement as a slice
    }
6. else
   n= n + 1
7. Repeat steps 5…6 until all the program statements are covered or till the EOP
is reached
End
```

3.4.2.3.1 Algorithm Explanation

Initially in forward slicing, after selecting the program and the list of variables, slicing is performed to identify the relevant statements in the selected program with respect to the slicing criterion. In forward slicing, analysis is done in a top-down manner. Initially, user selects the program and the slicing criterion is set. The slicing criteria can be either set by the user or set automatically for all the input variables. Slicing criterion contains the variable and statement number. Here, we have to check for the program statements that are affected by the value of a particular variable at a particular point. Slice variable ‘V’ is stored in a list ‘L’ and the program statement number is denoted by ‘n’. The process starts from the (nth) line till the end of the program is reached. In the (nth) line, it is checked whether the variable ‘V’ is present or not. If the variable ‘V’ is not present, then (n+1) th line is checked. If the variable ‘V’ is present in the (n) th line, a series of steps are to be performed. If ‘V’ is present in an expression, it is checked whether ‘V’ is present in the right side or left side of the expression. If ‘V’ is in the right side of the expression that statement is considered as a slice and all the variables in the left side of the expression are also added to the list. If ‘V’ is in the left side, then that statement is also included as a slice. While checking the next line, we have to check not only for ‘V’, but also all the dependent variables present in the list. This is because; the other variables added to the list are the dependent variables of ‘V’. Similarly, it is checked whether the slice variable is an element of conditional statement, declaration statement, input statement and output statement. If these conditions are true, the statements are considered as a slice. The statements inside the conditional body loop are also included as slice because the executions of these statements are dependent on the conditional clause. The process is repeated unit the end of the program is reached and the result will be the forward slice for the corresponding slicing criterion.

3.4.3 Data Generator

This unit of FST generated test data. There are three subunits in data generator. They are slice analyser, selector and test data generator. The detailed explanations of each unit are given below.

3.4.3.1 Slice Analyzer

The slice analyser checks the slices obtained from the forward slicing unit. Sometimes the forward slices generated by the slicer may be in the form of linear equation. In some other situations, the forward slices may be in some linear expressions which are easily convertible into linear equations. In some scenarios, the slices may not be expressed in a linear form and they remain as ordinary expressions. The slice analyser analyses the forward slices obtained and converts all the possible forward slices into linear equations, so as to enable test data generation easily. The main steps followed by slice analyser for converting the possible forward slices into linear equalities are given below in figure 3.5

Step 1. Get the slices from the forward slicer
Step 2. Analyse the slices
Step 3. Check whether the slices are already in the form of linear equations or whether they can be expressed as conditional expression in the form '(EXPRa op EXPRb)' where,
-'EXPR' is the conditional expression
-'op' is the relation operator
-(EXPRa - EXPRb) or (EXPRb op EXPRa)
Step 4. Convert the possible forward slices into (EXPRa op EXPRb)

Figure 3.5 Steps in Slice Analyser

Form the figure 3.5, it can be noticed that the slices/expression which are already in the form of linear equation are untouched. Other slices which may be expressed as linear equations are converted to linear equation the remaining slices are retained as such without any change.

3.4.3.2 Selector

After converting the possible forward slices into linear equations, the selector classifies the slices into two categories. The figure 3.6, given below explains the function of selector in detail.

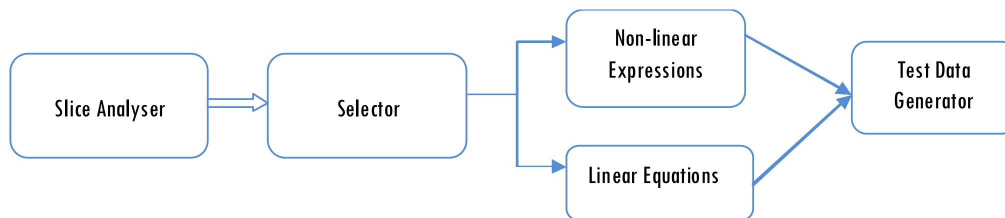


Figure 3.6 Functions of Selector

From the figure 3.6 given above, it can be noticed that the selector classifies the slices as linear equations and non-linear expressions. These expressions/forward slices are given to the test data generator. Classifying the forward slices into two categories helps to make test data generation easier.

3.4.3.3 Test Data Generator

After categorising the forward slices into two categories, the test data is generated using the test data generator. The two categories of forward slices are given to the test data generator. Test data is generated using two methods. The first method is Random method of test data generation and the second method is Gauss Elimination method of test data generation. The figure 3.7.given below shows the steps involved in test data generation.

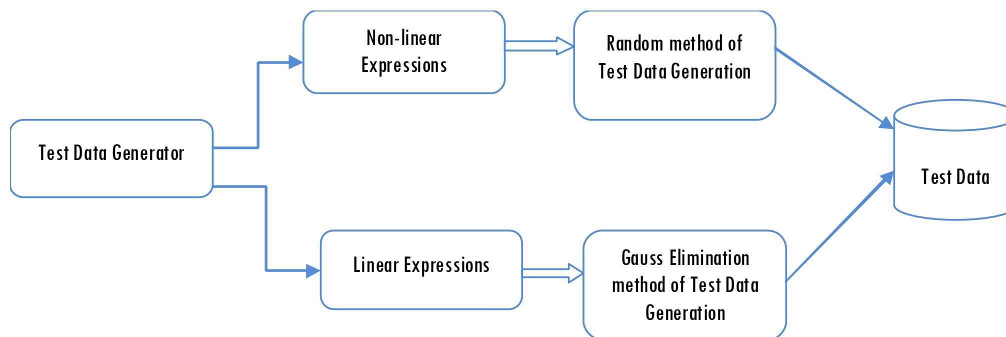


Figure 3.7 Test Data Generator

From the figure 3.7 given above, a random method is used to generate test data from non-linear equation and a Gauss elimination method is used to generate test data from linear expression.

3.4.3.3.1 Random Method of Test Data Generation

Slices which are in the form of non-linear expression are given to the Random Test Data Generators. Random values of test data are obtained using this method. Here, more than a single iteration may be required to obtain the required test data values.

3.4.3.3.2 Gauss Elimination Method of Test Data Generation

We have used a Gauss Elimination approach for test data generation for slices that are represented as linear equations due to the reasons given below:-.

- Almost all of the real world problems may be expressed as mathematical models which may contain linear equations
- Value of one variable in the slice/expression may decide the outcome of the other expressions in the program

Use of forward slicing based Gauss Elimination approach for generating test data in such large software systems can handle these core issues in a fine manner as:-

1. The changes made in the system due to modification in input condition may be easily handled using forward slicing
2. The dependency related problems during testing may be relieved using slicing
3. Finding the test data values for a number of linear equations may be solved using Gauss Elimination method.

Among the solutions available for solving linear equation, we have specifically chosen Gauss- Elimination approach due to its relative advantages over other linear equation solving methods. Though linear equation solving methods such as Graphical method, Cramer's rule, Algebraic elimination are simple, they can handle only a small number of equations. Gauss Seidel method can handle more than a thousand number of equations. Though the number of equations that can be handled and the error propagation is easy in Gauss Seidel, it takes more time for convergence in some scenarios. Gauss Elimination can handle an average number of equations and the solution may be easily derived. In Gauss Jordan method, deriving solution to linear equations is more difficult compared to Gauss elimination. From these facts, we can see that using Gauss Elimination approach, the solution may be obtained easily and it can easily handle an average number of equations. The main two stages in Gauss-Elimination method are given below.

Stage 1. Forward Elimination of Unknowns

This is the first stage in Gauss Elimination. There will be 'n' equation and 'n' unknowns initially. They are represented as follows:-

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1n}x_n = b_1$$

$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + ... + a_{2n}x_n = b_2$$

$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + ... + a_{nn}x_n = b_n$$

After getting the n equation and n unknowns, the following steps are performed.

- x_1 is eliminated from all rows below the first row.
- The first equation is selected as the pivot equation to eliminate x_1.
- To eliminate $x_1$in the second equation, one divides the first equation by (hence called the pivot element) and then multiplies it by a_{21} . This is the same as multiplying the first equation by$a_{21}/a_{11}$to give

$$a_{21}x_1 + \frac{a_{21}}{a_{11}}a_{12}x_2 + ... + \frac{a_{21}}{a_{11}}a_{1n}x_n = \frac{a_{21}}{a_{11}}b_1$$

Subtract the above equation from the second equation to get

$$\left(a_{22} - \frac{a_{21}}{a_{11}}a_{12}\right)x_2 + ... + \left(a_{2n} - \frac{a_{21}}{a_{11}}a_{1n}\right)x_n = b_2 - \frac{a_{21}}{a_{11}}b_1$$

At the end of n-1 steps of forward elimination, we get a set of equations that look like

$$\begin{aligned} a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + ... + a_{1n}x_n &= b_1 \\ a'_{22}x_2 + a'_{23}x_3 + ... + a'_{2n}x_n &= b'_2 \\ a''_{33}x_3 + ... + a''_{3n}x_n &= b''_3 \\ . \quad\quad & \quad . \\ . \quad\quad & \quad . \\ . \quad\quad & \quad . \\ a_{nn}^{(n-1)}x_n &= b_n^{(n-1)} \end{aligned}$$

After obtaining the set of equations as mentioned above, back substitution is applied.

Stage 2. Back Substitution

Back substitution hence can be represented for all equations by the formula

$$x_i = \frac{b_i^{(i-1)} - \sum_{j=i+1}^{n} a_{ij}^{(i-1)} x_j}{a_{ii}^{(i-1)}} \qquad \text{for } i = n-1, n-2, \ldots, 1$$

and

$$x_n = \frac{b_n^{(n-1)}}{a_{nn}^{(n-1)}}$$

This gives the value of unknowns in the linear equations.

3.5 Illustration of Test Data Generation using FST

For illustration of test data using FST, we have considered a brokerage application. Before testing, the programmer should try to get maximum information of the brokerage system. The partial class diagram is shown in figure 3.8 .Even though it may not be possible to get the minutest detail of each and every module's function, an overall purpose of each module may be identified by extracting the main rules using forward slicing from the system.

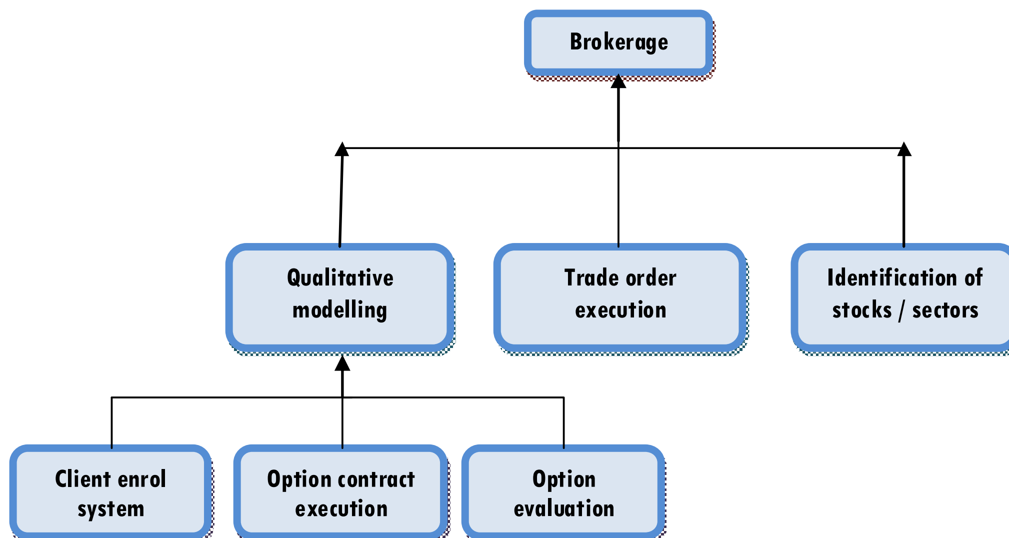


Figure 3.8 Partial Class Diagram of Brokerage System

The partial class diagram brokerage system given in figure 3.8. mainly consists of three main modules. They are as follows:-

- Qualitative modeling in financial market
- Trade order execution system
- Identification of stocks/sectors based on financial parameters

There are several subsections in each module. For illustration, we have considered only a particular subsection of the third module of the brokerage system. The third module deals with the identification of stocks. A sample segment of program code from the third module is given below. It is illustrated how to generate test data from slices from the sample code given below in figure 3.9.

```
1.  view sharevalue ( )
2.  {
3.  int company;
4.  float x1, x2, x3, amount;
5.  cin>>company;
6.  (if company==1)
7.  {
8.  x1  +   x2  + 2x3 = 8 ;
9.  -x1  -  2x2  + 3x3  = 1;
10. 3x1  - 7 x2 +  4x3 = 10;
11. }
12. amount =x1+x2+x3;
13. if (amount >50)
14. {
15. cout<< "Loss in investment";
16. }
```

Figure 3.9 Sample Code Segment

In code segment given above in figure 3.9, forward slicing based Gauss Elimination approach is applied to generated test data for the variable 'amount'. The steps followed for test data generation are given below:-

Step 1. Get the sample code segment and

Step 2. Set the forward slicing criterion C= (5, company)

Step 3. Perform forward slicing to identify all the statements affected by the forward slicing criterion C= (5, company)

Step 4. Forward slices obtained:-

```
1.  (if company==1)
2.  {
3.  x1  +   x2  + 2x3 = 8 ;
4.  -x1  -  2x2  + 3x3  = 1;
5.  3x1  - 7 x2 +  4x3 = 10;
6.  }
7.  amount =x1+x2+x3;
8.  if (amount >50)
9.  {
10. cout<< "Loss in investment";
11. }
```

Step 5.Check the forward slices to verify whether slices are present in the form of linear equations or whether the slices can be converted into linear form

The following statements in the forward slice are in the form of linear equations

```
12. x1  +   x2  + 2x3 = 8 ;
13. -x1  -  2x2  + 3x3  = 1;
14. 3x1  - 7 x2 +  4x3 = 10;
```

Step 6. Solve the forward slices in the form of linear equations by Gauss Elimination method

Step 7. Test data values obtained is: - x1 = 3, x2 = 1, x3 = 2

Step 8. Substitute the value of x1, x2 and x3 in statement 7 'amount =x1+x2+x3'.

Step 9. Test data value for amount=6

Step 10. Terminate the process

It can be noticed that the test data is obtained easily using FST. This is because, the statements of interest are easily identified from the program code by forward slicing. The forward slices (statements) are then verified and test data is generated from forward slices using Gauss Elimination method. The main point to be noted here is that, test data generation is easier as well as effective compared to other testing approaches. There is no need to worry about the hurdles of source code size, unwanted checking of program statements as well as identifying the dependency in the program. All these facts point toward the strength of our approach. A formal representation of forward slicing algorithm is given below in section 3.6. inorder to affirm the strength of our approach.

3.6 Proof of Correctness and Formalised Representation of Forward Slicing Algorithm

In this section of the thesis, we have presented a formalized representation of the forward slicing algorithm. The formalized representation of the forward slicing algorithm is presented as propositions based on First order predicate logic and Hoare Logic is used to prove these propositions [44]. The main benefit of such a representation is that, the slicing algorithm will have a strong mathematical basis and moreover, the formalized representation may be used as a foundation for developing testing methods in different type of systems [17, 134].

The program code most contains specifications where there will be a pre-condition and post-condition. A pre-condition defines the condition that must be satisfied before the specification is executed and post-condition specifies the condition satisfied after the execution of precondition [44, 68]. We have represented these conditions in our slicing algorithm as propositions

using first order predicate logic. For proving the propositions, each proposition is in turn represented in terms of Hoare Logic rules such as while, if-else etc., where the left-hand side (LHS) of the representation gives the hypothesis and right-hand side (RHS) gives the conclusion. In Hoare logic, for each statement which is supposed to get executed, there will be a pre-condition and post-condition [44]. The pre-condition is assumed to be true before the statement executes and the post condition is assumed to be true after the statement executes. Some of the terms used in the propositions and descriptions are given below:-

n- Procedure Statement/Program statement
L- List where the slice variables are stored
F- Forward Slice
V- Slice variable
LHS -Left side
RHS -Right side
EOP – represents end of the program
IN- Input Statement
OUT-Output Statement
D- Declaration Statement
I- Initialization Statement
EXPR- Expression
COND- Conditional Statement
p- Source program
C: = (n, V) – Represents the slicing criterion, where 'n' is the procedure statement and 'V' represents the slice variable
VAR (L) - Variables in list 'L'
V Є OUT- Represents that the slice variable is an element of the output statement
V Є IN- Represents that the slice variable is an element of the input statement

The propositions (1 to 6) which correspond to forward slicing algorithm are represented as follows:-

Proposition 1. $\exists p\ C\ (p) \rightarrow (\forall p\ (p \neq ø) \rightarrow \exists p\ C\ (p))$

The above proposition says that, a slicing condition may be set for all programs/procedures that are not empty.

Proof of Proposition 1:- Proposition 1 is represented in terms of Hoare logic as follows. LHS of the representation gives the hypothesis and RHS gives the conclusion.

$\{p \wedge (p \neq ø)\}$ C :=(n, V) $\{p\} \rightarrow \{p\}$ while $(p \neq ø)$ do C :=(n, V) $\{p \wedge \neg (p \neq ø)\}$

Steps:-

Step 1: $\{p\}$ - By Precondition [Invariant assumed to be true]

Step 2: $\{p \wedge (p \neq ø)\} \rightarrow$ C :=(n, V)

Step 2 is rewritten as $(p \neq ø) \rightarrow$ C :=(n, V) - By Tautology [If a procedure/program is empty, then slicing criterion cannot be determined]

Step 3: $\{p \wedge \neg (p \neq ø)\} \rightarrow$ Termination - By Post condition [Follows from Step 2, as slicing is not possible if the procedure is empty]

Thus the correctness of Proposition1 is proved.

Proposition 2. $\forall n\ ((n \neq 0) \wedge (n < EOP)) \rightarrow STORE\ (V, L)$

Proposition 2 follows from proposition 1 and may be divided into Proposition 2A and Proposition 2B for simplification

Proposition 2a. $\forall n\ (n \neq 0) \rightarrow STORE\ (V, L)$

Proposition 2B. $\forall n\ (n < EOP) \rightarrow STORE\ (V, L)$

Proof of Proposition 2a:- $\forall n\ (n \neq 0) \rightarrow STORE\ (V, L)$

Proposition 2A says, for any statement present in a program, the slice variable ‘V’ is stored in ‘L’.

Proposition 2A is represented as follows where the LHS of the representation gives the hypothesis and RHS gives the conclusion.

{L: = V ∧ (VAR (L) Є n)} Check n { n:= n+1}, { L := V ∧ ¬ (VAR(L) Є n)} Skip n { n:= n+1} →{ L := V} if (VAR(L) Є n) then Check n else Skip n { n:= n+1}

Here,

Steps:-

Step 1: {L: = V}

Step 2: {L: = V ∧ (VAR (L) Є n)} → Check n - By Tautology

Therefore, V ∧ (VAR (L) Є n

Step 3: {L: = V ∧ ¬ (VAR (L) Є n)}

Therefore, ¬ (VAR(L) Є n) → skip n - By Tautology[If the slicing variable stored in 'L' is not present in statement 'n', then that line of the program is not considered]

Step4: {n: = n+1} - By Post condition [Execution of both the 'if' and 'else' part of the logic is followed by the same post condition. Slicing proceeds to the next statement of the procedure after the execution of step 2 and step 3]

Proof of Proposition 2B:-∀n (n < EOP)] → STORE (V, L))

Proposition 2B says, for any non-null procedure with slicing a specified slicing criterion 'C', the slice variable 'V' or identifier is stored in the list 'L'

Given the precondition as slicing criterion 'C', the parameters in the slicing criteria are verified to check whether they are valid or not. If the parameters are valid, the slice variable 'V' is stored in list 'L'. The post – condition falls true when the statement specified in the slicing criterion is either zero or if the end of the procedure is reached. This is because, in forward slicing the statements affected by slicing criteria are identified and end of the

procedure/program (n= EOP) indicates no statements will be affected by slicing criteria. Proposition 2B is represented as follows:-

$\{C := (n, V) \wedge n < EOP)\}$ L: = V {C} $\rightarrow$ $\{C := (n, V)\}$ while (n < EOP) do L: = V $\{C := (n, V) \wedge \neg (n < EOP)\}$

Steps:-

Step 1: $\{C := (n, V)\}$ - By Precondition [Obtained from proposition 1 and this step is strengthened by the description given for proposition 1]

Step 2: $\{C := (n, V) \wedge n < EOP)\} \rightarrow$ L: = V - By Tautology [For a non-null program, with the statement of interest less than the end of the program, slice variable 'V' is stored in list 'L'. If the statement of interest is the end of the program, then there is no meaning in performing forward slicing]

Step 3: $\{C := (n, V) \wedge \neg (n < EOP)\} \rightarrow$ Termination - By Post condition [By tautology, follows from step 2]

Since the post-condition terminates the whole process, the correctness of proposition 2B is proved.

Proposition 3.$\forall$n [(V Є OUT) → SLICE (V, n)]

Proposition 3 may be rewritten as (V Є OUT) → $\forall$n SLICE (V, n)

Proposition 3 follows from proposition 2A. Proposition 3 means that, if a slice variable 'V' is present in any of the procedure statement and if that statement is an output statement, then the statement may be considered as a forward slice of the criterion C:= (n, V)

Proof of Proposition 3:- Proposition 3 is represented as follows:-

{(VAR (L) Є n) ∧ (V Є OUT)} F: = F ∪ n {n: = n+ 1}, {(VAR (L) Є n) ∧ ¬ (V Є OUT)} F: = F; {n: = n+ 1} → {(VAR (L) Є n)} if (V Є OUT) then F: = F ∪ n; else F: = F; {n: = n+ 1}

Steps:-

Step 1: {(VAR (L) Є n)} – By Precondition [Follows from proposition 2A and pre-strengthened by the description of proposition 2A]

Step 2: {VAR (L) Є n) ∧ (V Є OUT)} → F: = F ∪ n - By Tautology [Slice is updated if variable ‘V’ is an element of the output statement]

Step 3: {VAR (L) Є n) ∧ ¬ (V Є OUT)} → F: = F - By Tautology [Slice unaltered if the slice variable ‘V’ is not an element of output statement]

Step 4: {n: = n+ 1} - Post condition [Execution of both the ‘if’ and ‘else’ part of the logic is followed by the same post condition. Slicing proceeds or the next statement of the procedure is considered after the execution of step 3 and step 4 as per if-else rule of Hoare logic]

Therefore the correctness of the proposition 3 is proved

Proposition 4.∀EXPR (V Є RHS (EXPR)) → SLICE ((RHS (EXPR) ∧ LHS (EXPR))

Proposition 4 follows from proposition 2A. Proposition 4 says that, if a slice variable ‘V’ is present in any of the procedure statement and if that statement is an expression, then the statement may be considered as a forward slice of the criterion C:= (n, V), provided the slice variable is present in the right hand side of the expression. The slice thus includes the left hand side and right hand side of the expression.

Proof of Proposition 4:- Proposition 4 is represented as follows:-

{((VAR (L) Є n) ∧ (V Є EXPR)) ∧ (V Є RHS (EXPR))} F: =F ∪ n; VAR (L): = VAR (L) ∪ VAR (LHS (EXPR)); {n: = n+1} {((VAR (L) Є n) ∧ (V Є EXPR)) ∧ ¬ (V Є RHS (EXPR))} F: =F; {n: = n+1}

→ {(VAR (L) Є n) ∧ (V Є EXPR)} if (V Є RHS (EXPR)) F: =F ∪ n; VAR (L): = VAR (L) ∪ VAR (LHS (EXPR)); else F: =F; {n: = n+1}, {(VAR (L) Є n) ∧ ¬ (V Є EXPR)} Skip n; {n: = n+1}

Steps:-

Step1: {(VAR (L) Є n)} – By Pre-condition [Follows from proposition 2A and pre-strengthened by the description of proposition 2A]

Step 2: V Є EXPR - Mid-condition 1 [Assumed from Step 1]

Step 3: (V Є RHS (EXPR) - Mid-condition 2 [Assumed from Step 2]

Step 4: ((VAR (L) Є n) ∧ Mid-condition 1 → Mid-condition 2[Assumed from Step 3]

Step 4 may be rewritten as {((VAR (L) Є n) ∧ (V Є EXPR)) ∧ (V Є RHS (EXPR))}

{((VAR (L) Є n) ∧ (V Є EXPR)) ∧ (V Є RHS (EXPR))} → F: =F ∪ n - By Tautology [Expression which contains slice variable at the right side of the expression is considered as a slice

Step 5: {((VAR (L) Є n) ∧ (V Є EXPR)) ∧ (V Є RHS (EXPR))} → VAR (L): = VAR (L) ∪ VAR (LHS (EXPR)) - Derived from Step 4. [Variables in the left side of the expression are also added to the list as these variables are dependent on the slice variable present right side of the expression]

Step 6: {((VAR (L) Є n) ∧ (V Є EXPR)) ∧ ¬ (V Є RHS (EXPR))} → F: =F - From tautology and due to other possibilities like conditional expressions and initializations]

Step 7: {n: = n+1} - Post-condition [Execution of Step 4, Step 5 and Step 6 are followed by the same post-condition. Slicing proceeds to the next statement of the procedure if the conditions in these steps are not satisfied]

Therefore the correctness of Proposition 4 is proved.

Proposition 5.∀COND (V Є COND) → SLICE ((COND, LOOP STMTS))

Proposition 5 follows from proposition 2A. Proposition 5 says that, if a slice variable 'V' is present in any of the procedure statement and if that statement is a conditional expression, then the statement may be considered as a forward slice of the criterion C: = (n, V), provided where the slice variable is present in the conditional expression. The slice thus includes all the statements inside the conditional loop and the conditional clause, as the execution of the statements inside the conditional loop depends on the conditional clause. This is unavoidable as forward slice is a variant of static slice

Proposition 5 is derived from Proposition 5A and 5B

Proposition 5A.∀COND (V Є COND) → (∀COND ((V Є LHS (COND)) ∧ (V Є RHS (COND))

Proposition 5B. (∀COND ((V Є LHS (COND)) ∧ (V Є RHS (COND)) → SLICE (COND, LOOP STMTS)

Proof of Proposition 5:- Proposition 5 is represented as follows. LHS of the representation gives the hypothesis and RHS gives the conclusion

{(VAR (L) Є n) ∧ (V Є COND)} F: =F ∪ n ∪ LOOP STMTS; VAR (L): = VAR (L) ∪ VAR (LOOP STMTS)); {n: = n+1}, {(VAR (L) Є n) ∧ ¬ (V Є COND)} F: =F; {n: = n+1} → {(VAR (L) Є n)} if (V Є COND) then F: =F ∪ n ∪ LOOP STMTS; else F: =F; {n: = n+1}

Steps:-

Step 1: {(VAR (L) Є n)} - Pre-condition [Follows from proposition 3 and pre-strengthened by the description of proposition 3]

Step 2: {(VAR (L) Є n) ∧ (V Є COND)} → F: =F ∪ n ∪ LOOP STMTS

Step 3 : {(VAR (L) Є n) ∧ (V Є COND)} → VAR (L): = VAR (L) ∪ VAR (LOOP STMTS)) - Derived from Step 2[F: =F ∪ n ∪ LOOP STMTS means that all the statements inside the conditional clause will be affected by the conditional predicate. If any of these variables are used outside this particular conditional loop, that statement is also indirectly dependent on the conditional predicate]

Step 4: {(VAR (L) Є n) ∧ ¬ (V Є COND)} → F: =F - From tautology and due to other possibilities like proposition 4 and 8]

Step 5: {n: = n+1} - Post-condition [Execution of Step 2, Step 4 are followed by the same post-condition. Slicing proceeds to the next statement of the procedure if the conditions in these steps are not satisfied]

Therefore the correctness of Proposition 5 is proved.

Proposition 6.∀n [(V Є IN/I) → SLICE (V, n)]

Proposition 6 may be rewritten (V Є IN/I) → ∀n SLICE (V, n)

Here IN represents the input statement and I represents the initialization

Proposition 6 follows from proposition 2A. Proposition 6 says that, if a slice variable 'V' is present in any of the procedure statement and if that statement is an input statement or an initialization statement, then the statement may be considered as a forward slice of the criterion C:= (n, V)

Proof of Proposition 6:- Proposition 6 is proved in the same way as Proposition 3. It is represented as follows:-

{(VAR (L) Є n) ∧ (V Є IN)} F: = F ∪ n {n: = n+ 1}, {(VAR (L) Є n) ∧ ¬ (V Є IN} F: = F; {n: = n+ 1} → {(VAR (L) Є n)} if (V Є IN) then F: = F ∪ n; else F: = F; {n: = n+ 1}

Steps:-

Step 1: {(VAR (L) Є n)} - Precondition [Follows from proposition 3 and pre-strengthened by the description of proposition 3]

Step 2: {VAR (L) Є n) ∧ (V Є IN)} → F: = F ∪ n - By Tautology [Slice is update if variable 'V' is an element of the input statement]

Step 3: {VAR (L) Є n) ∧ ¬ (V Є IN)} → F: = F - By Tautology [Slice unaltered if the slice variable 'V' is not an element of input statement]

Step 4: {n: = n+ 1} - Post condition [Execution of both the 'if' and 'else' part of the logic is followed by the same post condition. Slicing proceeds or the next statement of the procedure is considered after the execution of step 3 and step 4]

Therefore the correctness of the proposition 6 is proved

This formalized representation of the forward slicing algorithm may be modified according to the purpose of testing. This makes it applicable in all type of systems irrespective of the application for which the system is developed.

A detailed evaluation and comparison of FST with related testing approaches is given in chapter 6.

3.7 Summary of the Chapter

Several methods are used to improve the test case generation and recently many works suggested the use of evolutionary algorithms like genetic algorithms for test case generation in spite of their inherent uncertain nature. Already the supporting evidence for this fact is drawn from literature review given in chapter 2. Therefore, we have put forth a forward slicing based Gauss Elimination method for test data generation in this work. Using slicing in our work identifies statements of interest which makes test data generation more direct. A major hurdle in most of the software testing approaches is the inability to handle voluminous code present in the program. Since our work uses forward slicing which identifies statements of interest, test data generation is made more effective by handling the size of the source code. To be more specific, we use a divide and conquer approach in testing using slicing. As test data generation is a process of generating test values for variables of interest, our method, which generates test data based on variable of interest will be more focused compared to the other approaches. Also, the FST introduced in this chapter considers dependency in the program which has a lead role in identifying effective test data. These dependent statements help to trace errors during software testing. This makes our method robust and unique. To conclude, as the fault may be located within the slice itself, debugging is also made easier with our FST method. This additional strength of forward slicing based testing makes it more suitable in practical software testing industry. To conclude, the main highlights of this chapter are:-

- Introduced forward slicing based software testing approach
- Introduced the concept of linked dependency method to identify dependencies in the program
- Presented a forward slicing algorithm based on data flow and control flow information for performing forward slicing
- Demonstrated how to generate test data from forward slices using Gauss Elimination method
- Presented a formal representation of forward slicing algorithm and proved the correctness of slicing algorithm

…….ÐÐ…….

Chapter 4

PARTITIONED FORWARD SLICES

Contents

4.1 Introduction
4.2 Motivation
4.3 Terms & Definitions Related to Partitioned Forward Slice
4.4 Partitioned Forward Slicing Algorithm & Explanation
4.5 Illustration of Partitioned Forward Slicing
4.6 Suitability of Partitioned Forward Slices
4.7 Summary of the Chapter

4.1 Introduction

In this chapter, we have introduced the concept of partitioned forward slices. Sometimes the number of statements in the forward slice will be so large that, they may not be that effective in software testing. The smaller the slice size, the better it is. Partitioned forward slices were introduced as an extension to forward slicing based testing (FST), when the size of forward slice is large. Reducing the size of forward slice is effective when the slice need to be focused closely, so as to find some critical errors. In addition to pointing out the need for partitioned forward slices, we have introduced a partitioned forward slicing algorithm and have illustrated the effectiveness of partitioned forward slicing.

4.2 Motivation

In this work, we have proposed a partitioned forward slicing. Identifying the potential impact of a variable on the source code is very essential during software testing. This allows the testers to identify the program statements by their relevance and enables the tester to identify the program statements affected by a particular variable. This helps to simplify testing during software

development [110]. We already saw in chapter 3 how forward slices are utilised to achieve the above mentioned task in software testing. This enables to handle the issue of source code size and identify statements by relevance during testing. In some scenarios, the programmer who developed the source code may not be responsible for testing the software [127]. While testing such software, lack of proper documentation of the source code makes it difficult for the programmers to understand the source code. Understanding the target program code is inevitable, as the programmer may not be able to make any modification in the program without understanding the source code [55]. This needs an in-depth understanding of the software system. Though using forward slicing in such scenarios may lessen the burden, sometimes the number of statements in the forward slice or the size of the forward slice obtained may be very large. In such situations, the slice will be of little use as the size of the slice is large and there may not be any proper documentation. The major factor which hinders the use of slicing in practical software testing is the size of the slice. Therefore, partitioned forward slices address this problem, so that they may be utilized for practical software testing when the size of forward slice is big. To summarize, the main motivation behind this work are given below:-

- Handle the large size of forward slice
- Identification of statements by relevance in a source code by identifying the statements affected by the input variable

4.3 Terms & Definitions Related to Partitioned Forward Slice

In this section some terms and definitions related to partitioned forward slicing are explained.

Definition 1- (Partitioned forward slices)

Partitioned forward slices are formed by performing forward slicing up to a specified partition point. The slices formed in this manner will be smaller than the forward slices and it will be easy to critically analyze these slices.

Definition 2- (Partitioned forward slicing criteria)

The condition with respect to which partitioned forward slicing is performed. It is denoted as C= (n, V, (P_i)). This means that the statements which are affected by the variable 'V' at statement 'n' are to be identified. The partition point (P_i) specifies the point up to which the program is to be checked or the forward slicing is be performed.

Definition 3 - (Partition point)

Partition point specifies the program statement number up to which forward slicing is to be performed.

4.3.1 Partition Point Properties

As partition point is a critical factor in partitioned forward slicing criterion, setting partition points correctly deserves prime importance. The partition point may be set be the tester and this gives them a 'dynamic' nature. Increasing the number of partition points beyond a certain limit may make the whole process meaningless. For example, while testing a program having 'n' statements, if the number of partition points is also declared as 'n', then there will be no difference in the difficulty experienced by the tester. This will be equivalent to checking the whole program line by line. Instead, if the tester is able to view the statements in the slice as a set of related statements which are executed under some condition, then it will be easy to identify the errors as well as to trace the dependent statements which may be affected by the

execution of conditional clauses in the program. Therefore, as a general standard the partition points are set up according to the following criteria.

- A program with ‘n’ statements is allowed to have not more than ‘n/2’ partition points.
- Set up partition points at the conditional loops
- Set up partition points at the class /function level

Viewing the statements in the slice as a set of conditions and their possible outcomes helps to get an idea of the program and this makes it easy for the programmer to identify the errors. The value of several variables in the program may change when the conditional loops are executed. Therefore setting the partition points at the beginning of conditional loops may help to track the errors in the output. Another option to set the partition points is to identify the classes or functions in the program and then set the point at each class or function level. This gives the possible errors associated with each class and function.

4.4 Partitioned Forward Slicing Algorithm & Explanation

In this section, an algorithm for partitioned forward slicing is introduced. Some terms and definitions related to partitioned forward slicing is given below.

n-Program Statement

L- List where the slice variables are stored

PFS- Partitioned forward slice

p_i- partition points

V- Slice variable

C: = (V, p_i) – Represents the slicing criterion, ‘V’ represents the slice variable and p_i represents partition points

LHS -Left side

RHS -Right side

IN- Input Statement

OUT-Output Statement

I- Initialization Statement

D-Declaration statement

EOP- End of program

EXPR- Expression

COND- Conditional Statement

VAR (L) - Slice variable ‘V’ stored in list ‘L’

RHS (EXPR) - Denotes the right side of the expression

LHS (EXPR) - Denotes the left side of the expression

VAR (RHS (EXPR)) - Denotes variables in the right side of the expression

VAR (LHS (EXPR)) - Denote the variables in the left side of the expression

Input: - Program to be sliced (P)
Output: - Partitioned forward slices (PFS)
begin
1. while p ≠ Ø, source program not empty
// Given V as the slicing variable
2. get C= (V, p_i*)*
3. for (pi=1; p_i<p_n*; pi++)*
{
4. Store ‘V’ in ‘L’ // Slicing variable ‘V’ stored in list ‘L’
5. if (VAR (L) Є n)// Check whether slice variable ‘V’ stored in list ‘L’ is present in statement ‘n’
{
5.1. if (V Є (OUT))
PFS_i*=* PFS_i*U n //Store n, initially PFS will be null and include the statement n as a slice*
5.2. else if (V Є (EXPR)))

{

5.2.1. if ((V) Є RHS (EXPR))

{

PFS_i=PFS_i U n // Store n

VAR (L) = VAR (L) U VAR (LHS (EXPR))

5.2.2. else

do not include the statement as a slice

}

}

5.3. else if (V Є (COND))

{

5.3.1. if ((V) Є LHS (COND) OR (V) Є RHS (COND))

{

PFS_i= PFS_i U n //Store n

PFS^i= PFS_i U Loop body statements // Include all statements inside the conditional loop in PFS_i

}

5.3.2. else

do not include the statement as a slice

}

5.4. else if (V Є (IN))

PFS_i= PFS_iU n // include statement as a slice

5.5. else if (V Є LHS (D))

PFS_i= PFS_i U n // include statement as a slice

}

6. else

n= n + 1

7. Repeat steps 3…7 until all partition points are covered or until EOP is reached

}

8. PFS=PFS_i //where PFS_i= PFS_1 U PFS_2 U PFS_3 U….PFS_n where PFS_1…PFS_n corresponds to partitioned forward slices for the specified partition points p_1, p_2 …p_n

End

In the partitioned forward slicing algorithm given above, initially, the slicing criterion is set. Slicing criterion contains the variable and partition points. By default, the variable position is taken as the initial definition point. Here, we have to check for the program statements that are affected by the slicing criterion. Initially, the slice variable ‘V’ is stored in list ‘L’. The process starts from the (n^{th}) line till the end of the program is reached. Here, (n^{th}) line represents the program statement after variable definition. In the (n^{th}) line, it is checked whether the variable ‘V’ is present or not. If the variable ‘V’ is not present, then $(n+1)^{th}$ line is checked. If the variable ‘V’ is present in the $(n)^{th}$ line, a series of steps are to be performed. If ‘V’ is present in an expression, it is checked whether ‘V’ is present on the right side or left side of the expression. If ‘V’ is on the right side of the expression that statement is considered as a slice and the entire variables in the left side of the expression are also added to the list. If ‘V’ is on the left side, then that statement is not included as a slice. While checking the next line, we have to check not only for ‘V’, but also all the dependent variables present in the list. This is because; the other variables added to the list are the dependent variables of ‘V’. Similarly, it is checked whether the slice variable is an element of conditional statement, declaration statement, input statement and output statement. If these conditions are true, the statements are considered as a slice. The statements inside the conditional body loop are also included as slice because the executions of these statements are dependent on the conditional clause. The process is repeated until the entire partition points in the slicing criteria are covered. The control and data dependent statements of the partition point are stored as a separate list. These may be considered as the partitioned forward slices for the specified criteria.

4.4.1 Extended Linked Dependency Method

The dependencies in partitioned forward slicing are found out by extending the linked dependency method which we already saw in chapter 3. The terms related to the extended linked dependency method is same as that mentioned in chapter 3. Extended linked dependency method is formed by extending the linked dependency method by inserting partition points in the linked dependency method. The main steps to be followed in extended linked dependency method are given below:-

Step 1: Select the source program and construct CFG of the program. The control flow information is noted for the program

Step 2: Identify all def-use pairs at each node

Step 3: Identify data flow information in the program using def-use pairs

Step 4: Use the def-use pair at each node to identify the data dependence

Step 5: Get slicing criterion C= (n, V, p_i) (n is the statement where the input variable is present, V is the slice variable with respect to which slicing is to performed and p_i is the partition point)

Step 6: Identify all the nodes occurring after n which are affected by the slicing variable V up to p_i

Step 7: Continue until all partition points are covered

Step 9: Combining all the marked nodes gives partitioned forward slices for the given slicing criterion

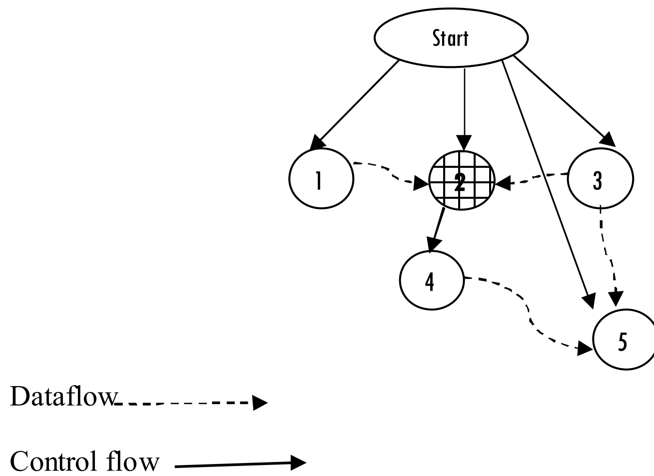


Figure 4.1 Sample CFG

Consider the CFG given in figure 4.1. If the partial slicing criterion is given as C= (5, V, (2)), given program point as 'node 2' (shaded node in the CFG), partial slice with respect to variable 'V' at statement 5 is to be obtained. We can see that node 5 is data dependent on node 3 and node 4. Node 4 is control dependent on node 2. Here, program point is marked at node 2. Therefore, all the control and data dependent nodes of node 2 are stored as a separate list. Node 2 is data dependent on node 1 and node 3, whereas node 4 is control dependent on node 2. All these together may be considered as one of the partial slices for the slicing criterion specified above

The partial slices obtained are:-

P1 – (4, 3, 2)

P2- (1, 3, 4)

Here P2 contains the control and dependent statements of node 2 which is defined as program point.

An illustration of partial slicing is given in section 4.5.

4.5 Illustration of Partitioned Forward Slicing

We have used a sample code segment of the brokerage software mentioned in chapter 3 for illustrating the working of partitioned forward slicing. We have selected the code segment from the class 'Qualitative modeling in financial market'. The sample code segment is given below in figure 4.2:-

```
1. public double BlackScholes(char CallOption, double S, double X, double T,
double r, double v)
2. {
3.      double d1, d2;
4.      d1=(Math.log(S/X)+(r+v*v/2)*T)/(v*Math.sqrt(T));
5.      d2=d1-v*Math.sqrt(T);
6.  if (CallOption=='c')
7.      {
8.      returnS*CNPD(d1)-X*Math.exp(-r*T)*CNPD(d2);
9       .}
10. else
11. {
12       return X*Math.exp(-r*T)*CNPD(-d2)-   S*CNPD(- d1);
13.}
14.}
```

Figure 4.2 Sample code segment

In the sample program segment given above in figure 4.2, the formula for calculating option value using Black Scholes formula [143] is given. Here the tester aims to separate formulas for call option and put option [143]. Therefore partitioned forward slicing is applied. The partitioned forward slicing criterion is given as (4, d1, (10, 14)). The partition point is given as (10, 14).

This means that all the statements up to partition point ‘10’ which are affected by the variable ‘d1’ in the 4th statement are identified and after that all the statements from 9 to 14 which are affected by the variable ‘d1’ in the 4th statement are identified . The result will be the set of statements (5, 8, 12) which is given below in in figure 4.3. The partitioned slices are named as P1 and P2.

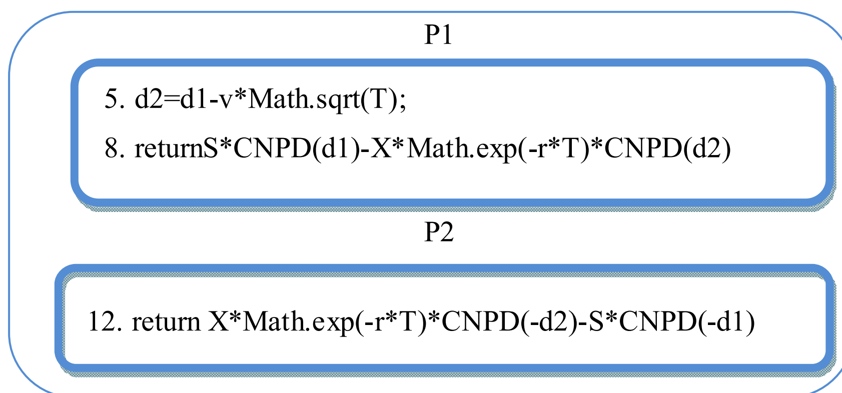


Figure 4.3 Partitioned forward slices for the criterion (4, d1, (10, 14))

The partitioned forward slices given in the above figure 4.3 may prove to be very relevant during program testing. The Partitioned forward slices P1 and P2 helps to analyze the slices closely and identifies the errors related to the execution of conditional statements. By examining P1 and P2, we can notice that statement 8 is dependent on statement 5 and statement 12 is dependent on statement 5. This means P1 affects P2 as well as P1 is self-dependent. Applying forward slicing in the above scenario will not give two separate slices. Only a single slice will be obtained. Testers should go through the statements in slice to locate errors related to conditional clause execution and such similar ones. Moreover, it will be difficult to identify dependencies in the program code when the size of the forward slice is large.

4.5.1 Proof of Correctness of Partitioned Forward Slicing Algorithm

In this section, we have given a summary of proof of correctness of partitioned forward slicing algorithm. The formalized representation of the partitioned forward slicing algorithm is presented as propositions based on First order predicate logic. Some of the terms used in the propositions and descriptions are given below:-

n-Program Statement

L- List where the slice variables are stored

PFS- Partitioned forward slice

p_i*- partition points*

V- Slice variable

C: = (V, p_i*) – Represents the slicing criterion, 'V' represents the slice variable and* p_i *represents partition points*

LHS -Left side

RHS -Right side

IN- Input Statement

OUT-Output Statement

I- Initialization Statement

D-Declaration statement

EOP- End of program

EXPR- Expression

COND- Conditional Statement

VAR (L) - Slice variable 'V' stored in list 'L'

RHS (EXPR) - Denotes the right side of the expression

LHS (EXPR) - Denotes the left side of the expression

VAR (RHS (EXPR)) - Denotes variables in the right side of the expression

VAR (LHS (EXPR)) - Denotes the variables in the left side of the expression

The propositions (1 to 6) which correspond to partitioned forward slicing algorithm are represented as follows:-

Proposition 1. $\exists p\ C(p) \rightarrow (\forall p\ (p \neq \emptyset) \rightarrow \exists p\ C(p))$

Proposition 2. $\forall n\ (p_i=1) \wedge (p_i<p_n) \rightarrow$ STORE (V, L)

Proposition 2a. $\forall n\ (p_i=1) \rightarrow$ *STORE (V, L)*

Proposition 2B. $\forall n\ (p_i<p_n) \rightarrow$ *STORE (V, L)*

Proposition 3. $\forall n$ [(V Є OUT) → PFS (V, n)]

Proposition 4. ∀EXPR (V Є RHS (EXPR)) → PFS ((RHS (EXPR) ∧ LHS (EXPR))

Proposition 5. ∀COND (V Є COND) → PFS ((COND, LOOP STMTS))

Proposition 6. $\forall n$ [(V Є IN/I) → PFS (V, n)]

4.6 Suitability of Partitioned Forward Slices

This section discusses the suggestions on the suitability of partitioned forward slices in some other applications. Getting a picture of the suitability of the slices in various other applications may help to apply the concepts introduced in this thesis to other stages of software development.

4.6.1 Suitability of Partitioned Forward Slices in Testing

We have already seen the benefits of using partitioned forward slices in testing in the previous sections. Figure 4.4 given below also describes the various steps involved in testing.

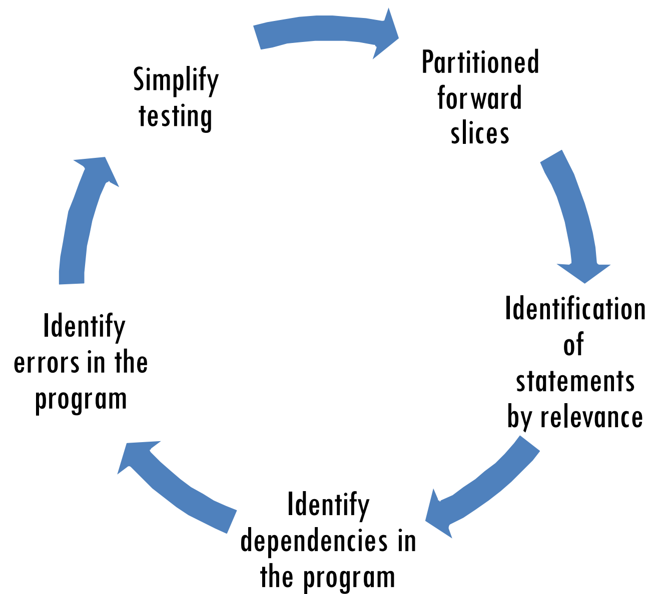


Figure 4.4 Partitioned forward slices in testing

In the figure 4.4 given above, statements of relevance may be identified using partitioned forward slices and these slices help to identify dependencies and also help to identify errors in the program. All these help to simplify testing.

4.6.2 Suitability of Partitioned Forward Slices in Maintenance

Retrieving the statements of interest using partitioned forward slicing during maintenance has several advantages. Partitioned forward slices are also used to simplify regression testing process. Regression testing consists of re-testing the program parts which are affected by the modification of an original tested program [54, 55]. The figure 4.5 given below shows the significance of partitioned forward slices in maintenance.

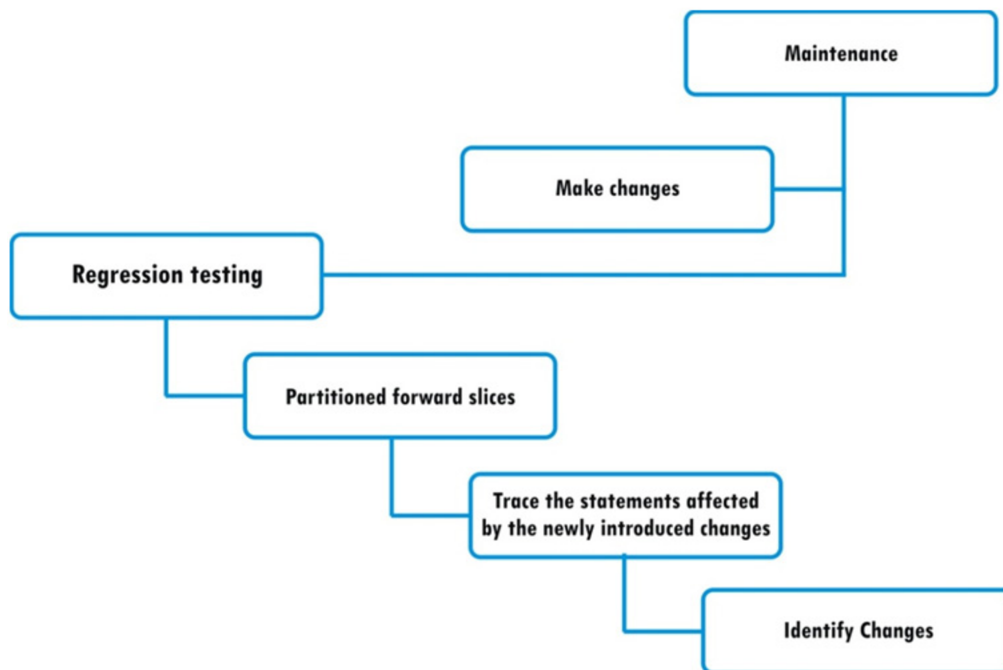


Figure 4.5 Partitioned forward slices in maintenance

From the figure 4.5 given above, we can see that the statements affected by modification are easily identified from the slices and designing test cases for the new condition becomes easier using partitioned forward slices.

4.6.3 Suitability of Partitioned Forward Slices in Program Comprehension

Partitioned forward slice may be used during program comprehension. Program comprehension is defined as the process of understanding the source code [15]. The comprehended program may be used during program maintenance, reuse, redesign etc. Unless the programmers have detailed knowledge about the program, they will not be able to make changes in the source code. A program developer who is new to the field may find it difficult to understand the purpose of the code by checking the whole source program [30, 36]. In other words, reducing the amount of source code to be verified ceases the process of program understanding. Manually creating such type of source code

segments is very tedious. In such scenarios, partitioned forward slice slicing may be used. A diagrammatic representation of how partitioned forward slicing is used for program comprehension is given below in figure 4.6.

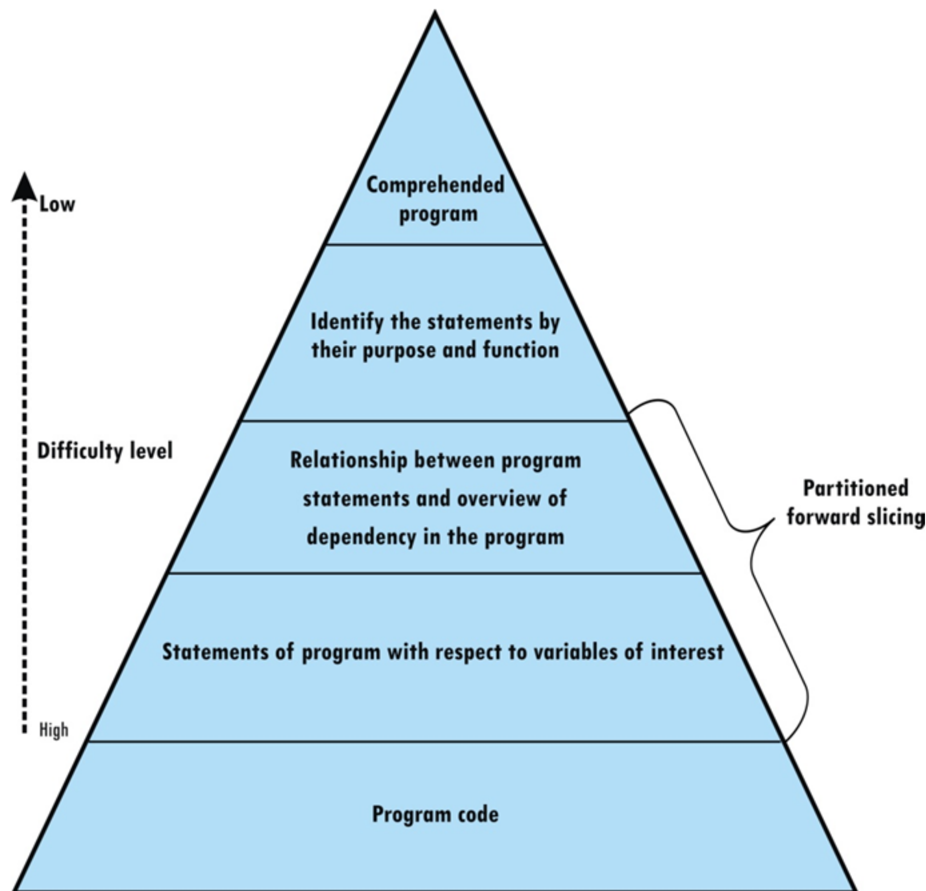


Figure 4.6 Partitioned forward slices in program comprehension

During program comprehension, there will be several questions related to the behavior of the program. These questions give some clue regarding the program execution. Sometimes the questions raised may be wrong. From the figure 4.6 given above, set of program statements as well as the static information regarding the program code may be retrieved from the partitioned forward slice slices which in turn may be used to get answer or some idea of these questions. After this, by tracing the control and data flow dependencies

of the partitioned forward slice, the programmer may gain a better idea of the programs. This simplifies the program comprehension process.

4.7 Summary of the Chapter

This chapter introduces the novel concept of partitioned forward slices. This type of slice was introduced as a remedy to handle the issue of forward slice size. When the number of statements in the forward slice is large or when there is a critical need to focus on the forward slices, partitioned forward slice may be used. We have presented an algorithm and an illustration of partitioned forward slice. We have also suggested some other applications where the concept of partitioned forward slices may be applied. To conclude, partitioned forward slices help to:-

- Handle the large number of statements in the forward slice
- Enable an in depth analysis of forward slice statements
- Make program testing easier

…….ꟸꟹ…….

Chapter 5

PARTIAL SLICES IN PROGRAM TESTING

•Contents•

5.1 Introduction
5.2 Motivation
5.3 Terms & Definitions Related to Partial Slice
5.4 Program Points Set-up
5.5 Architecture of Partial Slicer
5.6 Comparison & Performance Evaluation of Partial Slices and Static Slices
5.7 Inference from the Comparison and Evaluation of Partial Slices and Static Slices
5.8 Suitability of Partial Slices
5.9 Summary of the Chapter

5.1 Introduction

In this chapter of this thesis, we have introduced the concept of partial slice. This type of slice was introduced with the aim of identifying the changes related to output variables. In chapter three and chapter four, the concepts of forward slicing and partitioned forward slicing were used to identify the statements affected by input variable. In contrast, partial slices give the statements in a program which affect the output variable. This feature of partial slices is very helpful to find errors during testing. The need for partial slices, the architecture of partial slices and the details of partial implementation is discussed in the successive sections of this chapter. Finally, a comparison and performance evaluation of partial slices with static slices is also carried out. For performance evaluation, we have used a Mann-Whitney U test. Apart from testing, suitability of partial slices in different application is also discussed in this chapter. The chapter concludes by listing the summary of findings made in this chapter.

5.2 Motivation

The generation of different forms of slices was the result of the application of program slicing in source code analysis and manipulation [128]. We already saw that, slicing may be used in many applications like program comprehension, testing, debugging and maintenance as it helps to simplify some of the program features [55, 109, 110, 118]. When using slicing in program testing, the number of statements to be analysed in the slice is a key factor in determining the usefulness of a testing tool [65]. In several cases, it can be noticed that static slices contain a large number of program statements. Due to this increased size of the static slice, they are of little use in many practical applications especially during program testing. Therefore, the concept of partial is introduced in this chapter with the intention to reduce the difficulties faced during program testing. Unlike static slices, issues regarding lengthy source code and dependency tracing is easily handled using partial slices. In partial slices, in addition to the static slicing criterion, the user has to provide the program point. Program point specifies the program statement up to which the static slicing is to be performed. The partial slices produced in this manner combines both static and program point information. Also, the number of statements present in the partial slices will be lesser than static slice. In slicing, smaller the slice, the better it is. Partial slices facilitate an easy analysis of the slices by reducing the number of statements to be checked in the slice. This property makes them more appropriate in identifying the errors associated with unit testing, especially condition execution, predicate execution and procedures. The main highlights of this chapter are:-

- Introduced the concept of partial slice
- Partial slicing algorithm is explained
- Performed comparison and performance evaluation of partial slices with static slices using Mann-Whitney U test
- Mentioned the suitability of partial slices in some other applications

5.3 Terms & Definitions Related to Partial Slice

In this section, some terms and definitions related to partial slice are explained.

Definition 1 (Partial slice) – A partial slice is formed by performing static slicing up to a specific point mentioned in the program

Definition 2 (Partial slicing criterion) – The condition with respect to which partial slicing is performed

C= (n, V, (P_i, P_j, P_k….P_n)

- Where 'C' is the partial slicing criterion
- 'n' is the statement number
- V is the subset of variable in the program
- P_i, P_j, P_k….P_n known as program points and they specify the limit up to which static slicing is performed

The partial slicing criterion is defined as C= (n, V, P_i). This means that the statements which affect the variable 'V' at statement 'n' are to be identified. The program point 'P_i' (*where i=1, 2, 3... n*) specifies the program statement or point up to which the program is to be sliced. The process is continued until all program points are covered. Slices formed at each program point are combined together to form the partial slice of a variable.

5.4 Program Points Set-up

As the program point is a critical factor in partial slicing criterion, setting up of program points deserves prime importance. The program point may be set be the tester and this gives them a 'dynamic' nature.

Increasing the number of program points beyond a certain limit may make the whole process meaningless. For example, while testing a program

having 'n' statements, if the number of program points is also declared as 'n', then there will be no difference in the difficulty experienced by the tester. This will be equivalent to checking the whole program line by line. Instead, if the tester is able to view the statements in the slice as a set of related statements which are executed under some condition, then it will be easy to identify the errors as well as to trace the dependent statements which may be affected by the execution of conditional clauses in the program.

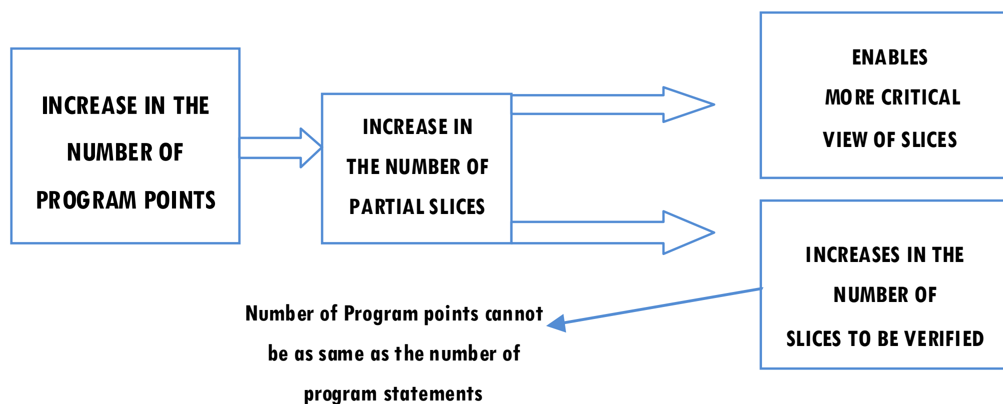


Figure 5.1 Guidelines for setting up program points

The guidelines for setting up program points are shown in figure 5.1 given above. Therefore, as a general standard, the program points are set up according to the following criteria.

- A program with 'n' statements is allowed to have not more than 'n/2' program points.
- Setup program points at the conditional loops
- Setup program points in the class /function level

Viewing the statements in the slice as a set of conditions and their possible outcomes helps to get an idea of the program and this makes it easy for the programmer to identify the errors. The value of several variables in the

program may change when the conditional loops are executed. Therefore, setting the program points at the beginning of conditional loops may help to track the errors in the output. Another option to set the program points is to identify the classes or functions in the program and then set the point at each class or function level. This gives the possible errors associated with each class and function.

5.5 Architecture of Partial Slicer

In this section the architecture of the partial slicer is explained. The architecture of partial slicer is given below in figure 5.2. The main components of the of the partial slicer are the program selector and the partial slicing unit

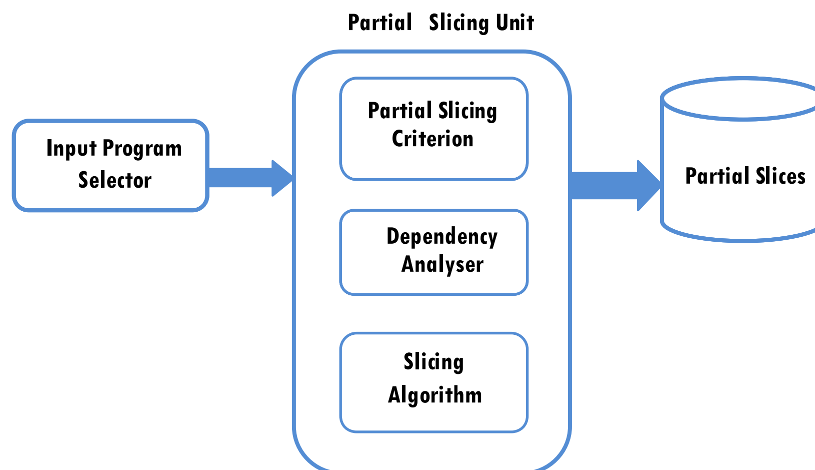


Figure 5.2 Architecture of partial slicer

The input program selector unit has the facility to select the source program. After selecting the source program, the partial slicing criterion is set by the partial slicing unit. Dependencies are identified from the program using the partial linked method. Finally, partial slicing algorithm identifies the slices from the CFG. The detailed description of partial slicing algorithm and partial linked method is given in the successive section.

5.5.1 Identifying Dependencies using Partial Linked Method

In a partial slice, we are using a partial linked method to identify the dependencies and the slicing algorithm given in section 5.5.2 is applied to the CFG. The terminologies and notations related to partial linked graph are already given in section of chapter 3. In a partial linked method, we are considering both control and data dependencies of the program. The steps followed in partial linked method are given below:-

Step 1: Construct the control flow graph of the source program and analyse the control flow in the program

Step 2: Identify all def-use pairs at each node

Step 3: Identify data flow information in the program using def-use pairs

Step 4: Use the def-use pair at each node to identify the data dependence

Step 5: Get slicing criterion C= (n, V, p_i), where 'n' is the statement where the output variable is present and 'p_i' is the program point)

Step 6: Identify all the nodes occurring before 'n', which affect slicing variable 'V' up to p_i

Step 7: Continue until all program points are covered

Step 9: Combining all the nodes marked in the CFG gives the partial slice for the specified slicing criterion

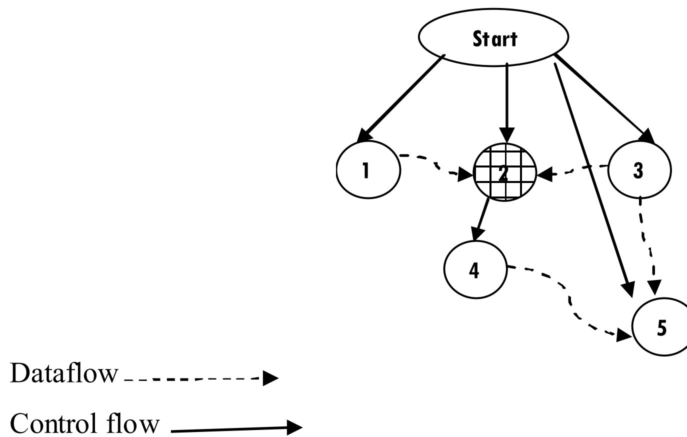


Figure 5.3 Sample CFG

Consider the CFG given in figure 5.3 If the partial slicing criterion is given as C= (5, V, (2)), given program point as 'node 2' (shaded node in the CFG), partial slice with respect to variable 'V' at statement 5 is to be obtained. We can see that node 5 is data dependent on node 3 and node 4. Node 4 is control dependent on node 2. Here, program point is marked at node 2. Therefore, all the control and data dependent nodes of node 2 are stored as a separate list. Node 2 is data dependent on node 1 and node 3, whereas node 4 is control dependent on node 2. All these together may be considered as one of the partial slices for the slicing criterion specified above.

The partial slices obtained are:-

P1- (4, 3, 2)

P2- (1, 3, 4)

Here P2 contains the control and dependent statements of node 2 which is defined as program point.

An illustration of partial slicing is given in section 5.5.3.

5.5.2 Partial Slicing Algorithm & Explanation

We saw how the dependencies are identified using partial linked method in section 5.5.1. In this section, partial slicing algorithm is explained. After identifying the dependencies, partial slicing algorithm is applied to the CFG to identify the partial slices. Some of the terms used in the algorithm are given below:-

n- Statement
L- List where the slice variables are stored
S- Partial Slice
p_i- program points
V- Slice variable
LHS -Left side
RHS -Right side
IN- Input Statement
OUT-Output Statement
I- Initialization Statement
D-Declaration statement
SOP- Start of program
EXPR- Expression
COND- Conditional Statement
VAR (L) - Slice variable ‘V’ stored in the list ‘ L’
RHS (EXPR) - Denotes the right side of the expression
LHS (EXPR) - Denotes the left side of the expression
VAR (RHS (EXPR)) - Denotes variables on the right side of the expression
VAR (LHS (EXPR)) - Denote the variables on the left side of the expression

PARTIAL SLICING ALGORITHM
Input: - Program to be sliced (P)
Output: - Partial Slices (S)
begin
1. while p ≠ Ø, source program not empty
// Given n as the statement number and V as the slicing variable
2. get C= (n, V, p_i*)*
3. for (p_i*=n;* p_i*<p1;* p_i *--)*
{
4. Store 'V' in 'L' // Slicing variable 'V' stored in the list ' L'
5. if (VAR (L) Є n)// Check whether slice variable 'V' stored in list 'L' is present in statement 'n'
{
5.1. if (V Є (OUT))
S= S_i *U n //Store n, initially S will be null and include the statement n as a slice*
5.2. else if (V Є (EXPR)))
{
5.2.1. if ((V) Є RHS (EXPR))
{
do not include the statement as a slice
5.2.2. else
S=S_i *U n // Store n*

VAR (L) = VAR (L) U VAR (RHS (EXPR))
}

}
5.3. else if (V Є (COND))
{
5.3.1. if ((V) Є LHS (COND) OR (V) Є RHS (COND))
{
S= S_i *U n //Store n*
S= S_i *U Loop body statements // Include all statements inside the conditional loop in S*
}
5.3.2. else
do not include the statement as a slice
}
5.4. else if (V Є (IN))
S= S_i *U n // include statement as a slice*
5.5. else if (V Є LHS (D))
S= S_i *U n // include statement as a slice*
}
6. else
n= n - 1
7. S=S_i *//where* S_i*=* S_1 *U* S_2 *U* S_3 *U.…* S_n *where* S_1 *...* S_n *corresponds to partial slices for the specified program points*
}
8. Repeat steps 3…7 until all program points are covered or until SOP is reached
End

In partial slicing, after selecting the program by the program selector, the slicing criterion is set by the partial slicing unit. Slicing criterion contains the variable the statement number and the program points. Here, we have to check for the program statements that affect the value of a particular slicing variable at a particular point. Consider slice variable ‘V’ stored in a list ‘L’ and the program statement number denoted by ‘n’. The process starts from the (n^{th}) line till the start of the program is reached. In the (n^{th}) line, it is checked whether the variable ‘V’ is present or not. If the variable ‘V’ is not present, then $(n-1)^{th}$ line is checked. If the variable ‘V’ is present in the $(n)^{th}$ line, a series of steps are to be performed. If ‘V’ is present in an expression, it is checked whether ‘V’ is present in the right side or left side of the expression. If ‘V’ is on the left side of the expression that statement is considered as a slice and all the variables in the right side of the expression are also added to the list. If ‘V’ is on the right side, then that statement is not included as a slice. While checking the preceding line, we have to check not only for ‘V’, but also all the dependent variables present in the list. This is because; the other variables added to the list are the dependent variables of ‘V’. Similarly, it is checked whether the slice variable is an element of conditional statement, declaration statement, input statement and output statement. If these conditions are true, the statements are considered as a slice. The statements inside the conditional body loop are also included as slice because the executions of these statements are dependent on the conditional clause. The process is repeated until the entire program points in the slicing criterion are covered. The control and data dependent statements of the program point are stored as a separate list. These may be considered as the partial slices for the specified criteria.

5.5.3 Illustration of Partial Slicing

An illustration of partial slicing is given in this section to demonstrate how the slices are identified from the program. For illustration purpose, we have applied partial slicing method to payroll processing software. The payroll software consists of several modules. Some of the modules are company details, employee information, employee salary, etc. Here we have considered the employee salary module. Given below is the partial class diagram of payroll software. The partial class diagram of payroll is given below in figure 5.4. We have illustrated how partial slicing is performed on the 'Officer' class. The 'Officer' is a subclass of 'Employee Salary'.

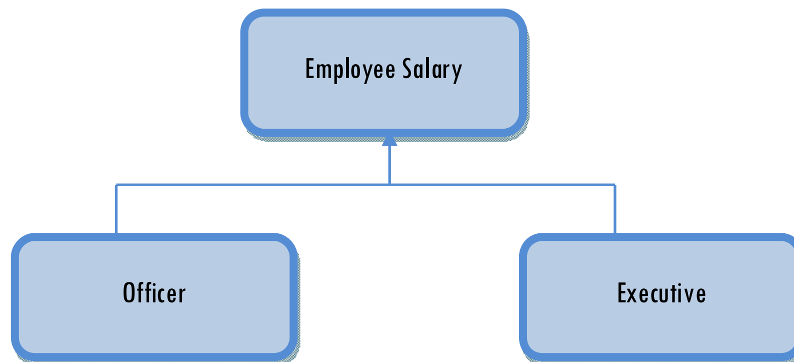


Figure 5.4 Partial class diagram of payroll software

From the 'Officer' class, we have considered a set of program statements and have illustrated how partial slicing works on the selected class. In figure 5.5, some program statements from the 'Officer' class is displayed. Let us initialize the partial slicing criterion as C= (15, total, (11, 1)) for the sample program statements in figure3.

```
1. if (basic < 20000)
2. {
3. rent=1700;
4. da = basic*93 / 100;
5.}
6. else
7. {
8. rent = basic*12 / 100;
9. da = basic*85/100;
10.}
11. total = basic+rent+da;
12. allowance= total +5000;
13. finalsalary= allowance + 1000;
14. System.out.println ("Final salary" +finalsalary);
15. System.out.println ("total salary" + total);
```

Figure 5.5 Sample program statements

The first parameter is the statement number, second one is the slice variable and the third one is the program points. Here we have declared only two program points. Initially the 15th line is checked. No element is present in 15th statement affects the value of the variable 'total' in statement 15. The next step is to decrement the statement number. Now statement number 'n' is 14. The 14th line prints the value of 'finalsalary' and it is not dependent on the statement 15. Now decrement 'n' and 'n' becomes 13. Statement 13 is 'finalsalary= allowance + 100'. Here the 'fianlsalary' value is calculated. The slice variable 'total' is not present in this line also. Therefore, this line will not affect the value of 'total' in statement 15. Again decrement 'n' and now 'n' is 12. In the 12th line it can be seen that the variable 'total' is present. The next step is to check whether the slice variable is present in the right side or left side of the expression. Here the slice variable 'total' is present in the right side of the expression. Therefore the slice variable 'total' is not affected by the 12th line

too. Now decrement the statement number. The statement number becomes 11. Here the first program point is declared. In the 11th line the slice variable ‘total’ is present. Again, check whether ‘total’ is present on the right side or on left side of the expression. Here the variable ‘total’ is present on the left side of the expression ‘total = basic + rent + da’. This indicates that the variable ‘total’ is the sum of ‘basic’, ‘rent’ and ‘da’. By analyzing closely is can be noticed that the value of the variable ‘total’ obtained in statement 11 is used in statement 15. In other words statement 15 is data dependent on statement 11. Therefore, this statement is considered as a partial slice. Since the program point is declared in the 11^{th}statement, all the control and dependent nodes of statement 11 are stored as a separate list. In statement 11, the variables on right side of the expression are ‘basic’, ‘rent’ and ‘da’. This shows that any line containing these variables will also affect the value of ‘total’ in statement 15. This is due to the dependency which exists in the program. Therefore, from the 10th statement, we have to check not only for the variable ‘total’, but also for the variables ‘basic’, ‘rent’ and ‘da’. This is also indicated in the algorithm. It is given that the List ‘L’ should be updated as VAR (L) = VAR (L) U VAR (RHS (EXPR)). This means that the list containing the slice variable is now having the variables ‘basic’, ‘rent’ and ‘da’ in addition to ‘total’. As we have obtained the first partial slice, the partial slicing process is to be resumed to obtain the other slices present in the program. From the slicing criteria, we can see that the program point is set at statement 11. Now the process starts from 10th statement. After that the statements 9 and 8 are checked. Statement 9 and 8 gives the value of the variables ‘rent’ and ‘da’ when the condition (basic<20000) is false. Statement 9 includes the variable ‘da’ which is present in the list ‘L’. Slice variable ‘total’ is dependent on ‘da’ and in statement 9 ‘da’ is present on the right side of the expression. Therefore, this is also included in the partial slice.

The left side of statement 9 contains the variable ‘basic’. This is already present in the list ‘L’. Therefore, there is no need to update the list again. Similarly, statement 8 gives the value of ‘rent’. Variable ‘rent’ is present in list ‘L’. Therefore, this statement is also considered as a partial slice. Statements 8 and 9 are control dependent on statement 1. Therefore, statement 1 is also included as a slice. The execution will stop when the next program point is encountered. The statements 3 and 4 give the value of ‘rent’ and ‘da’ when the condition (basic<20000) is true. The slice variable ‘total’ is dependent on ‘rent’ and ‘da’. Therefore, these statements will also be included in the partial slice. According to the logic of the algorithm, these variables are present in the left side of the expression and these are included in the partial slice. The process terminates when all the program points are covered. The partial slices are given in table 5.1.

Table 5.1 Partial Slices

P1	total = basic+rent+da

P2	if (basic < 20000)
	{
	rent=1700;
	da = basic*93 / 100;
	}
	else
	{
	rent = basic*12 / 100;
	da = basic*85/100;
	}

The partial slices given in the table 5.1 may prove to be very relevant during program testing. The partial slice ‘P2’, gives an idea of the conditional statements in the program code. Errors related to the execution of these conditional statements may be easily identified from the partial slice P2. By examining P1 and P2, we can notice that P1 is dependent on P2. This means that the errors related to condition execution will affect several other statements in the program.

Applying static slicing in the above scenario will not give two separate slices. Only a single slice will be obtained. Testers should go through the statements in slice to locate errors related to conditional clause execution and such similar ones. Moreover, it will be difficult to identify dependencies in the program code. This may not be a problem in a small program with small static slices. In large programs with several lines in the static slice, partial slices are very helpful.

5.5.4 Proof of Correctness of Partial Slicing Algorithm

In this section, we have given a summary of proof of correctness of partial slicing algorithm. The formalized representation of the partial slicing algorithm is presented as propositions based on First order predicate logic. Some of the terms used in the propositions and descriptions are given below:-

n- Statement

L- List where the slice variables are stored

S- Partial Slice

p_i*- program points*

V- Slice variable

C= (n, V, p_i*)- slicing criterion*

LHS -Left side

RHS -Right side

IN- Input Statement

OUT-Output Statement

I- Initialization Statement

D-Declaration statement

SOP- Start of program

EXPR- Expression

COND- Conditional Statement

VAR (L) - Slice variable 'V' stored in the list ' L'

RHS (EXPR) - Denotes the right side of the expression

LHS (EXPR) - Denotes the left side of the expression

VAR (RHS (EXPR)) - Denotes variables on the right side of the expression

VAR (LHS (EXPR)) - Denotes the variables on the left side of the expression

The propositions (1 to 6) which correspond to partial slicing algorithm are represented as follows:-

Proposition 1.$\exists p\ C(p) \rightarrow (\forall p\ (p \neq \emptyset) \rightarrow \exists p\ C(p))$

Proposition 2.$\forall n\ (p_i=n) \wedge (p_i>p_1) \rightarrow$ STORE (V, L)

Proposition 2a. $\forall n\ (p_i=n) \rightarrow$ *STORE (V, L)*

Proposition 2B. $\forall n\ (p_{i\,>}\, p_1) \rightarrow$ *STORE (V, L)*

Proposition 3.$\forall n\ [(V \in OUT) \rightarrow S(V, n)]$

Proposition 4.$\forall EXPR\ (V \in RHS(EXPR)) \rightarrow S((RHS(EXPR) \wedge LHS(EXPR))$

Proposition 5.$\forall COND\ (V \in COND) \rightarrow S((COND, LOOP\ STMTS))$

Proposition 6.$\forall n\ [(V \in IN/I) \rightarrow S(V, n)]$

5.6 Comparison & Performance Evaluation of Partial Slices and Static Slices

In this section, we have compared the performance of partial slices with static slices. A Mann-Whitney U test is used to compare the performance of the

two types of slices [33]. A research question (RQ) is framed and the comparison and performance evaluation result addressed this question. RQ is defined as follows:-

RQ. Why partial slices are considered to be more useful in testing compared to static slices?

Inorder to address RQ, a NULL hypothesis (H0) was formulated. The research question (RQ) is proved /disproved based on the outcome the null hypothesis. The null hypothesis (H0) is given below:-

H0: It does not make any difference in performing testing using static slices and testing using partial slices

For addressing H0, some test subject/programs were considered. The details of test subjects and test set-up is given in the successive sections

5.6.1 Test Subjects

Some sample programs were considered and partial slicing and static slicing was applied to the sample program. The specifications of the program are given below in table 5.2:-

Table 5.2 Test subjects (Sample programs)

Test Subject	Description of the program	LOC	Number of variables in the test subject
P1	A C program to find the value of y in the function y (x, n)	22	3
P2	A C program to calculate the commission for a sales representative	21	2
P3	A C program for deciding grades for the marks obtained	18	1
P4	A C program to calculate the bank operations	121	7
P5	A C program to find the grade of steel	35	3
P6	A C program to find the option values in online trading system	117	6
P7	A C program to generate Fibonacci series	20	4
P8	A C program to find the sum of the given series	16	5
P9	A C program to find biggest of three numbers	18	4
P10	A C program to find electricity charges	22	5

5.6.2 Test Set-up

Among the test subject programs, static slicing was applied to subject programs P1 to P5 and partial slicing for P6 to P10. The total number of significant statements to be considered during testing the subjects is recorded for each method. The result is given in table 5.3. Since there are two test conditions and two groups of test subjects and since the data is measured on an ordinal scale, a Mann-Whitney U test is used to evaluate the results obtained from the two methods [33]. Finally, the Null-hypothesis is validated based on the result got from the test.

Table 5.3 Test subject & the result of applying various testing methods on test subjects

Test Subject	Total LOC	Number of statements to be considered in static slicing based testing	Test Subject	Total LOC	Number of program points defined in partial slicing criteria	Number of statements to be considered in partial slice based testing
P1	22	10	P6	117	3	(19, 17, 10)
P2	21	10	P7	20	1	(5, 3)
P3	18	6	P8	16	1	(5, 3)
P4	121	59	P9	18	1	(6, 3)
P5	35	11	P10	22	1	(2,4)

In table 5.3 we can see that, for test subjects P1 to P5 static slicing is applied and for test subjects P6 to P10 partial slicing methods is applied. The number of program statements to be considered during program testing for each type of testing (static slicing and partial slicing) is noted. The number of program points defined in the partial slicing criteria is also given in the table. The last column in table 5 represents the total number of statements present in the partial slices. For example, for P10, the number of statements present in partial slice is (2, 4). Here two sets of program statements are obtained in the partial slice. Here the first field ‘2’denotes the number of statements in one section of the partial slice and the second field ‘4’ denotes the number of statements in the second

section of the partial slice. Though the total number of program statements obtained in static slice and partial slice may be same, partial slice enables to get a closer view of program by inserting program points in the program. This is done by reducing the number of statements to be examined in a slice.

Based on the observations from table 5.3, a questionnaire was prepared. There were 2 questions Q_1 and Q2, which is given below.

Q_1. What is the difficulty caused due the size of slice in program testing?

Q_2. What is the problem caused due to the size of slice in tracing dependency in program testing?

All the test subjects (P1 to P10) addressed this questionnaire and the outcome was recorded for subject programs P1 to P5 for static slicing and for subject programs P6 to P10 for partial slices. This means that program P1 to P5 was checked with static slicing approach and programs P6 to P10 was checked with partial slicing approach. A qualitative rating for the questions was assigned based on the evidence collected from static slicing and partial slicing. Each question had a possible response. By data coding, the outcome of the questions was rated based on a scale of 0-5. The outcome and the corresponding scale are given below in figure 5.6. Based on the testing outcome given in table 5.4, the response was noted for each type of testing.

0-Nil/No difficulty
1-Negligible
2-Low difficulty
3-Average difficulty
4-High difficulty
5- Extreme difficulty

Figure 5.6 Scale corresponding to Difficulty Levels

Table 5.4 gives the observations of using static slices in testing and the result of using partial slices in testing.

Table 5.4 Response for Q1 and Q2 (For static slicing and partial slicing)

Test Subject	Static slicing- Response for Q1	Static slicing- Response for Q2	Static slicing- Sum of the Responses for Q1& Q2	Test Subject	Partial slicing- Response for Q1	Partial slicing- Response for Q2	Partial slicing- Sum of the Responses for Q1 & Q2
P1	3	3	6	P6	0	1	1
P2	4	3	7	P7	0	1	1
P3	3	4	7	P8	0	1	1
P4	5	5	10	P9	1	1	2
P5	4	5	9	P10	0	1	1

In table 5.4 given above, the data entered in the columns 'Static slicing- Response for Q1 & Static slicing- Response for Q2' gives the response for applying static slicing on test subjects P1 to P5 for questions Q1 and Q2. The response is recorded in 0-5 scale as already mentioned. The columns 'Partial slicing- Response for Q1 & Partial slicing- Response for Q2' gives the response for applying partial slicing on test subjects P6 to P10 for questions Q1 and Q2. Finally, the sum of the 'Static slicing- Response for Q1 & Static slicing- Response for Q2 and Partial slicing- Response for Q1 & Partial slicing- Response for Q2' is found out.

Table 5.5 Ranked response for Q1 and Q2 (For static slicing and partial slicing)

Test Subject	Static slicing- Sum of the Responses for Q1& Q2	Rank of the response	Actual Rank of Static Slicing	Test Subject	Partial slicing- Sum of the Responses for Q1 & Q2	Rank of the response	Actual Rank of Partial Slicing
P1	6	6	6	P6	2	4	4.5
P2	7	7	7.5	P7	1	1	2
P3	7	8	7.5	P8	1	2	2
P4	10	10	10	P9	2	5	4.5
P5	9	9	9	P10	1	3	2
		Total	40			Total	15

In table 5.5, the sum of response of static slicing and partial slicing are noted and ranked. Ranks having tied score are given average value of ranks. Finally the actual ranks are noted. A Mann Whitney U test is applied for evaluating the ranks obtained. In Mann- Whitney U test, the U value is given as:

$U= n_1n_2 + n_x (n_x+ 1)/2 - T_x$

Here n_1 and n_2 are the participants in each group and T_x is the largest of rank total and n_x is the number of participants in the group having largest rank total

In our example, $n_1= 5$and $n_2 = 5$, $T_x=40$ and $n_x=5$

Therefore $U= 5 * 5 + 5(5+1)/2 - 40$

$U=25+15-40$

$U=0$

The next step is to find the critical value of U from the Mann-Whitney U test table of critical values. From the table at 5% significance level for a two tailed test, the critical value of U for $n_1=5$ and for $n_2=5$ is 2.

Therefore U=2 (from Mann Whitney U table at 5% significance level)

To be statically significant, the obtained U value should be less than the critical value of U.

In our case, $0 < 2$. Therefore, the null hypothesis (H0) is rejected. This indicates that there exists a significant difference in performing testing using static slices and testing using partial slices.

Inference: - Due to the ability to focus on slices, partial slices are more useful in testing compared to static slices

5.7 Inference from the Comparison and Evaluation of Partial Slices and Static Slices

In this section we have given some inferences on partial slices and static slices based on the comparison and evaluation carried out. These inferences highlight the significance of using partial slicing compared to static slicing. The inference made are listed below

- Both static slices and partial slices give all possible execution
- Static slices may be large and it may not be helpful in testing due to large slice size and difficulty in identifying dependencies between the statements in the program
- As the number of statements in the static slice is more compared to partial, error identification is difficult compared to partial slice.
- Partial slices are smaller than static slices with respect to a particular program point. This is useful in testing, as the slices are confined and dependencies in the program easy to identify compared to static slices.
- Partial slices help to identify errors related to conditional loop execution.

From the above inferences it is evident that, partial slices are more useful compared to static slices from the testing point of view.

5.8 Suitability of Partial Slices

Apart from testing, partial slicing may be used in many other applications. We have suggested the possibility of using partial slicing in two other applications other than testing.

5.8.1 Using Partial Slices for Software Reuse

Partial slices may be utilized during software reuse. Software reuse is the process of using existing software components rather than building from

the scratch [30]. Source code is one of most important reusable components of software [84]. The idea of code clones is derived from reuse idea. A code clone can be defined as a set of program statements which may be contiguous or non-contiguous and which repeat in several other parts of the same program or in different parts of the same program or in different files of the same application program [118]. Even though code reuse saves time and manual effort, some researchers claim that software code clone increases the software maintenance cost [84]. For example, if a programmer makes any slight modification in a code clone, and if the same change is not made in the other code clones present in the program, it may cause inconsistency [76]. Utilizing the positive aspects of code clones in an appropriate way can result in marked changes in the field of software testing industry. Many of our day to day computer applications take advantage of the code reuse property. The main reason behind this code reuse mentality is to make the software development process easier. For example, while making a newer version of the operating system, developers are not writing the program code from the scratch, rather they try to concentrate only on the new functions which are to be integrated into the new version. The main point to be noted here is that the code developers can concentrate on developing the new features of the software rather than putting effort on the old problems repeatedly [55].

A diagrammatic representation of how our method is used for program reuse is given below in figure 5. 7:-

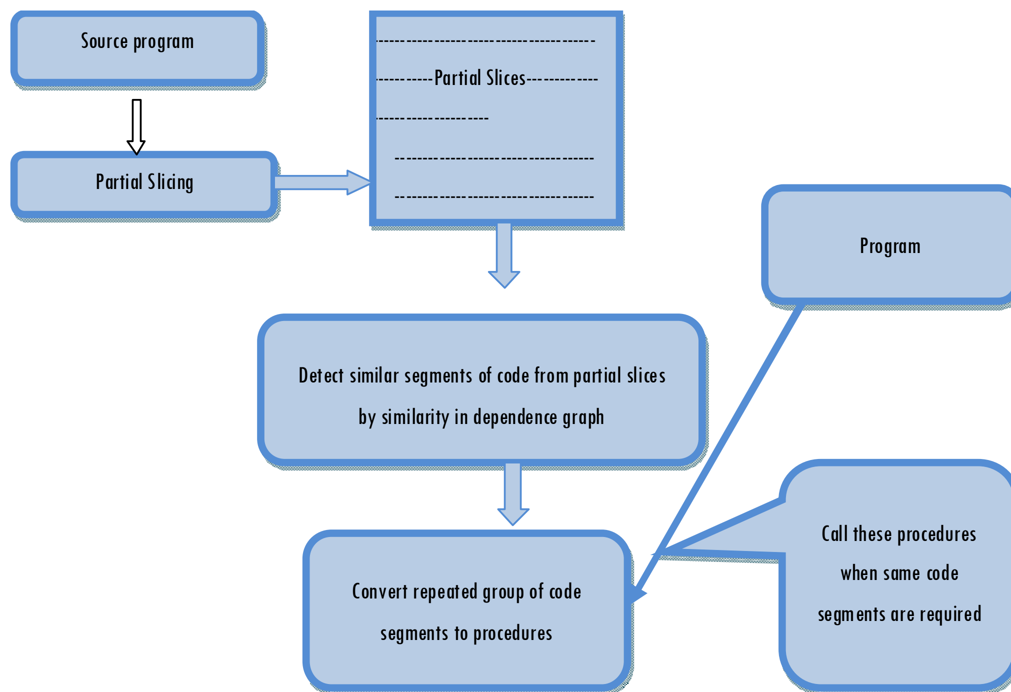


Figure 5.7 Using Partial slices for program reuse

From the figure 5.7 given above, it can be noticed that the partial slices are formed according to the slicing criteria. The partial slices may be formed for certain condition execution. These conditions, based on which partial slices are formed, may be used to search the database of reusable program statements or code clones. The same process may be performed using theorem proving. Using partial slicing is easier, compared to using theorem proving, as an empty slice is returned if no suitable reusable statements are found. In another method the control flow graphs (CFG) of the partial slices are checked for structural similarity. If the CFG of partial slices is structurally similar, then the next step is to check for node content similarity. If the node contents are also same, then it can be concluded that the partial slices present in the program code are code clones. These code clones may be used during the testing, maintenance etc.

5.8.2 Using Partial Slices for Program Comprehension

Partial slices may be used during program comprehension. Program comprehension is defined as the process of understanding the source code [15, 30]. The comprehended program may be used during program maintenance, reuse, redesign, etc. [36]. A diagrammatic representation of how our method is used for program comprehension is given below in figure 5.8. :-

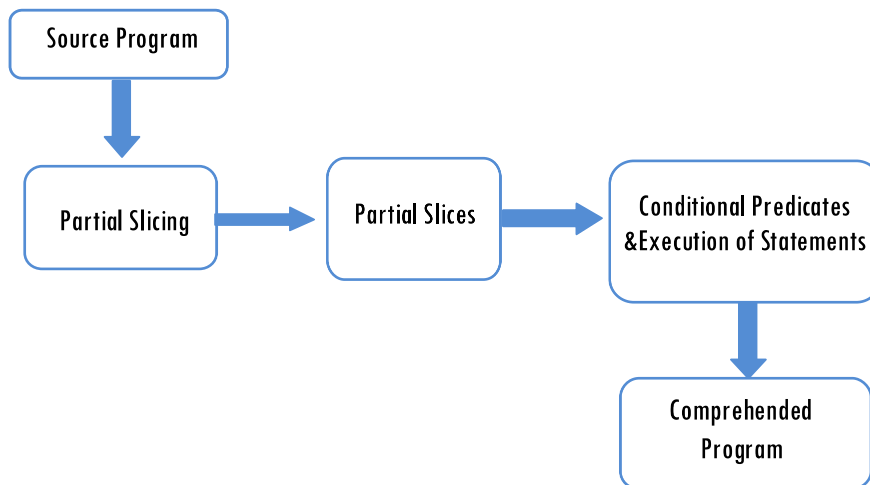


Figure 5.8 Partial slices in program comprehension

Unless the programmers have detailed knowledge about the program, they will not be able to make changes in the source code. A program developer who is new to the field may find it difficult to understand the purpose of the code by checking the whole source program [118]. In other words, reducing the amount of source code eases the process of program understanding. Manually creating such type of source code segments is very tedious. In such scenarios partial slicing may be used. During program comprehension, there will be several questions related to the behavior of the program. These questions, give some clue regarding the program execution. Sometimes the questions raised may be wrong also. From the figure 5.8 given above, a set of program statements as well as the static information regarding the program code may be retrieved from the partial slices which in turn may be used to get

answers or some idea of these questions. After this, by tracing the control and data flow dependencies of the partial slices, the programmer may gain a better idea of the programs. This simplifies the program comprehension process.

5.9 Summary of the Chapter

In this chapter we have introduced the concept of partial slices inorder to track the changes related to output. Using static slicing to track the changes related to the output gives all possible executions related to the output. This inturn helps to identify the errors in a program easily. In many situations, it can be noticed that there will be no significant reduction in the number of statements in static slice compared to the original program. In such situation, using slicing to identify changes related to output will be difficult. As a solution, we proposed a new approach named partial slicing, which can handle the issues with the lengthy source code and program dependency to identify changes related to output variable. An algorithm for performing partial slicing is also presented in this chapter. Finally a comparison and performance evaluation of static slicing and partial slicing is carried out. The evaluation result shows that, the number of statements in a partial slice is less than the corresponding static slice. This feature of partial slices makes them an excellent aid in testing and identifying changes related to output, especially in locating errors related to conditional loops and procedures. The suitability of partial slices in some other applications is also mentioned in this chapter. To conclude:

- Partial slices help to track changes related to output variables
- Partial slices help to get a focused view of the slices
- Partial slices help to identify errors related to conditional loops and procedures
- Disadvantages related to static slices may be solved up to a certain extent using partial slices

…….ꕥ…….

Chapter 6

COMPARISON AND PERFORMANCE EVALUATION

Contents

6.1 Introduction
6.2 Comparison of Forward Slicing based Testing with Related Testing Approaches
6.3 Experimental Evaluation & Comparison
6.4 Inferences from the Comparison and Evaluation of Related Methods
6.5 Summary of the Chapter

6.1 Introduction

In this chapter, we have provided a comparison with related work and performance evaluation of forward slicing based testing. Evaluation and comparison of our method with existing techniques provides an idea of the strength of our approach and stress the need for using the proposed method in software testing.

The comparison is done in two stages. Initially, we have done a qualitative comparison of forward slicing based testing with existing software testing approaches. From this comparison some intuitive inferences are made. In the next stage, we have conducted some experimental evaluations. We have used Mann-Whitney U statistical test for performance evaluation of forward slicing based testing with related testing approaches [33]. In addition, we have also evaluated our testing method using some metrics. Software testing technique metrics such as test case generation metric, user control metric and slicing metrics such as coverage and tightness are used to evaluate our method with related testing approaches [103, 110, 137].

6.2 Comparison of Forward Slicing based Testing with Related Testing Approaches

This is the first stage of comparison of forward slicing based testing with exiting testing techniques. The inferences made from the comparison highlights the significance of using slicing in software testing.

Table 6.1 Recent works on software testing

Type of Testing	Works	Total number of works in a particular testing category
Random testing	Michael, C. C. et al. [2001], Gouraud, S. D. et al.[2001], Liu, X. et al. [2005]	3
Path oriented testing	Bernard, B. et al. [2006], Mohammad, A. et al. [2006], Leonard, G. et al. [2006], Tran Sy, N. et al. [2001], Khor S. et al. [2004], Claudia, M. E. et al. [2003], Nashat, M. et al. [2004], Tran Sy, N. et al. [2003], Visvanathan, S. et al. [2002], Edvardsson, J. et al. [2001], Siqueira M. P. et al. [2000], Taylor B. J. et al. [2000], Lin, C. et al. [2001], McMinn, P. et al.[2006]	14
Goal oriented testing	Lin, C. et al. [2001], McMinn, P. et al. [2006], Diaz, E. et al. [2003], Baresel, A. et al. [2004]	4
Chaining method	McMinn, P et al. [2006], Ferguson , R. et al.[1996], McMinn, P et al. [2002]	3
Metaheuristic approach	Xue-ying et al. [2005], Xiao, J. et al. [2010], Latiu, G. I. [2012] , Ahmed, M. A et al. [2008], Pachure, A.et al. [2013], Roper, M. et al. [1995] , Jones, B. et al. [1996], Pargas, R. P. et al. [1999] , Michael, C. C. et al. [2001], Sofokleous, A. A. et al. [2008], Chen, C. et al.[2009] , Bernadt, D. J. et al. [2004] et al. [2004], Khor, S. et al. [2004], Cao, Y. et al. [2009], Malburg, J.[2011], Zhang, W. et al. [2007], Fraser, G.et al. [2012], Arcuri, A. et. al [2011], Bueno, P. M. S. et al. [2000], Wegner, J. et al. [2002], Miller, J. et al. [2006], McMinn, P. [2013], Fraser, G.et al. [2012], Gong, D. et al. [2011], Pocatilu, P. et al. [2013], Mao, C. et al.[2013], Liu, D. et al. [2013], Suresh, Y. et al.[2013], Arcuri, A. et al. [2013], Fraser, G.et al. [2013 & 2014]	30

We have considered some of the recent works (during the last decade) in software testing which are listed in table 6.1. Different categories of work such as random testing, path oriented testing, goal oriented testing, metaheuristic testing and chaining method are considered.

For comparison, we have considered mostly works on software test case generation. From the table 6.1, it is clear that a lot of research works are done

using heuristic algorithm [2, 5, 6, 14, 21, 26, 28, 29, 47, 49, 50, 59, 79, 83, 89, 95, 96, 97, 100, 104, 105, 113, 114, 116, 117, 123, 126, 136, 139, 140, 141]. We can notice that, among the works which use heuristic algorithms for software testing, a majority of the works have used evolutionary algorithms like GA or some hybrid approaches using evolutionary algorithms. The nature of each of these works shows that, in spite the inherent uncertainty in heuristic algorithms, they are widely used in test data research [95]. Another interesting fact is that, the works related to heuristic algorithm do not use any universal objective function or parameter setting during test data generation. Almost all the works have made use of their own fitness function and parameter setting for test case generation. The inferences from literature survey in chapter 2 support this fact.

In the case of using genetic algorithms for test case generation, even though using an optimal fitness function or parameter setting may alleviate the problems faced during test case generation, the real issues related to the domain size of the problem has been addressed by only a few researchers [73]. Similarly, using other methods of testing like random, path oriented and goal oriented methods have several problems and a summary of these problems are given in table 2.Using random methods for test case generation may not give successful test cases and it requires several numbers of iterations, whereas heuristic methods may give some approximate test case values. The main problem with heuristic algorithms, like GA, is that test case value varies according to the factors such as population initialization, fitness function design, crossover, selection and rate of mutation. According to the problem's nature, genetic algorithm parameters and factors need to be adjusted. This shows the inherent uncertain nature of heuristic algorithms. While trying to optimize the fitness function, sometimes the function gets modified to such an extent that, the entire solution may deviate from the expected result [61].

In path oriented approach, the problem of test data generation is considered as a ‘path’ problem. The path for which test data is to be generated may be selected automatically [101]. This path in turn leads to the destination. If the selected path doesn’t lead to the target path, then another path is considered. This process is continued until the target path is found, until the required test data is found or until the time specified for data generation is over. Symbolic execution and execution-oriented test data generation is used to generate test data in path oriented approach. In execution oriented approach, the actual execution of the program occurs. This is a goal oriented approach where the process of finding input is represented as a set of sub goals. Here the program is executed with the randomly generated input. For the generated input, the program execution flow is traced. As the program execution progresses, the search procedure decides whether the execution should proceed or whether an alternate branch is to be considered as the current path may not lead to the goal. This process is continued until the target node or goal is obtained. In path oriented approach, the main problem occurs when non-executable paths are explored which causes a loss in computation effort [72]. Goal oriented strategy may have difficulty in prioritizing the goals during testing.

Another method used for test data generation is the chaining approach [41, 101, 102]. This method may be considered as an extension of path oriented approach. In chaining approach, dataflow analysis is used for generating test data. An input value is randomly generated and the program is executed with this input. If the execution of the program with the generated input leads to a branch which does not lead to the target node, then a real valued function is associated with this node. Function minimization search algorithm is applied to find new input value. Chaining method also faces difficulties as the program size and complexity increases. Slicing based testing offers great support in such

situations and this may be used for software testing. The comparison of various testing methods is given in table 6. 2.

Table 6.2 Comparison of Different Software Testing Approaches

METHOD	ISSUES	ADVANTAGES
Random testing	-Poor code coverage -More iteration for generating test data values -Require to combine with other test data generation methods to get appropriate test data values	-Simple
Path oriented testing	-Selection of paths remains a major issue especially when the size of the program is large -No universal method suggested to handle path selection problem	-Simple(provided path selection is properly handled)
Goal oriented testing	-Priority of setting goals should be identified correctly -Dependencies may not be correctly followed during goal analysis	-No issues like path selection
Metaheuristics based testing (GENETIC ALGORITHM)	-Designing correct fitness function -Wrong fitness function may deviate the entire aim of the problem -No general rule for setting genetic parameters -Uncertainty in solution as the parameter setting varied -Uncertainty in response time prediction -Dependency may not be followed correctly in all instances	-Approximate value of test data obtained during test data generation
Chaining approach based testing	-Statements of interest not identified -Becomes complex as the size of the program increases	-Test data values based on the real execution of the program

In table 6.2, the pros and cons of various software testing approaches are given. From the table 6.2, it is evident that source code size, dependency identification and identification of statements of interest are some of the common issues related to all testing approaches.

6.3 Experimental Evaluation & Comparison

An experimental evaluation and comparison with GA based method and Random method is given in this section of this thesis. Forward slicing based testing is compared with Genetic algorithm (GA) based testing and random testing. Random method of software testing is considered for comparison with forward slicing based testing because it is considered as the benchmark for comparing testing techniques [8, 104]. GA based method for software testing is one of the most extensively used software testing techniques during the last decade [2]. Little works are done inorder to evaluate the effectiveness of GA in practical testing. Therefore, we have compared forward slicing based method with GA based testing also. A tool RAND was developed for random testing [8] for comparison with forward slicing based method in this thesis. Similarly, a GA based method for software testing was also used in this thesis [104]. The details of this method are given in Chapter 2. Finally, forward slicing based testing is compared with random method (RAND) of testing and GA based method of testing.

A research question (RQ) was formulated and this question was answered by the experimental study. The question is given below:-

RQ. How effective is forward slicing based software testing compared to random testing and GA based testing?

In order to answer RQ, comparison and evaluation are done based on the following criteria:-

- Statistical method of comparison and evaluation
- Metric based comparison and evaluation

6.3.1 Comparison and Evaluation using Statistical Method

In order to address research question RQ, the null hypothesis and alternate hypothesis were formulated as given below:-

H0: There is no difference in the effectiveness of forward slicing based testing compared to GA based testing and random testing.

$H_{A:}$ Forward slicing based testing is more effective compared to GA based testing and random testing.

6.3.1.1 Test Subjects

Some sample programs were considered. Forward slicing based testing, random testing and GA based testing were applied to the same sample program for testing the programs. The specifications of the program are given below in table 6.3:-

Table 6.3 Subject programs

Test Subject	Description of the program	LOC	Number of variable in the test subject
P1	A C program to find the value of y in the function y (x, n)	22	3
P2	A C program to calculate the commission for a sales representative	21	2
P3	A C program for deciding grades for the marks obtained	18	1
P4	A C program to calculate the bank operations	121	7
P5	A C program to find the grade of steel	35	3
P6	A C program to find the option values in online trading system	117	6
P7	A C program to generate Fibonacci series	20	4
P8	A C program to find the sum of the given series	16	5
P9	A C program to find biggest of three numbers	18	4
P10	A C program to find electricity charges	22	5

For the sample programs given in table 6. 3, the total number of lines of code (LOC) and the number of variables is noted. The details of applying testing techniques to the subject programs in explained in the next (Test Set-up) section.

6.3.1.2 Test Set-up

In order to handle the null hypothesis H0, evaluation was carried out in two stages. In the first stages, test subjects P1 to P5 was tested with GA based testing and test subjects P6 to P10 was tested with forward slicing based method and the results were noted. In the second stage, test subjects P1 to P5 was tested with random method and the test subjects P6 to P10 was tested with forward slicing based method. In stage 1 and stage 2 we can see that there are two there are two test conditions and two group of test subjects. Therefore, a Mann-Whitney U test is used to evaluate the results of evaluation [33]. The result of stage 1 and stage 2 are given below in table 6.4 and table 6.5

Table 6.4 Test subjects& the result of applying GA based testing method and forward slicing based testing on test subjects

Test Subject	Total LOC	Number of statements to be considered in GA based testing	Test Subject	Total LOC	Number of statements to be considered in forward slicing based testing
P1	22	22	P6	117	114
P2	21	21	P7	20	27
P3	18	18	P8	16	23
P4	121	121	P9	18	22
P5	35	35	P10	22	10

In table 6.4 we can see that, for test subjects P1 to P5 the total LOC in the program and the number of statements to be considered when GA based testing is used is given and for test subjects P6 to P10, the total number of LOC in the program and the number of statements to be considered when applying forward slicing based testing are listed.

Table 6.5 Test subject & the result of applying random method& forward slicing on test subjects

Test Subject	Total LOC	Number of statements to be considered in Random method of testing	Test Subject	Total LOC	Number of statements to be considered in forward slicing based testing
P1	22	22	P6	117	114
P2	21	21	P7	20	27
P3	18	18	P8	16	23
P4	121	121	P9	18	22
P5	35	35	P10	22	10

In table 6.5 we can see that, for test subjects P1 to P5 the total LOC in the program and the number of statements to be considered when random based testing is used and for test subjects P6 to P10, the total number of LOC in the program and the number of statements to be considered when applying forward slicing based testing are listed. A questionnaire was prepared based on the research problem and the research objective. The questionnaire was validated by first giving it to two subject experts and establishing face validity. Next, a pilot study was done among fellow researchers. After that the questionnaire was finalised. There were 3 questions Q_1, Q2 and Q3 in the questionnaire which is given below.

Q1. What is the difficulty faced due to the source code size in software testing?

Q2. What is the problem faced due to the difficulty in identifying relevant statements in a program during software testing?

Q3. What is the problem caused due to the difficulty in tracing dependency in software testing?

In our study, the 10 test subjects (P1 to P10) were randomly assigned to 10 independent testers. These 10 subject programs were divided into two groups in such a way that, these two groups were comparable in terms of lines of code and program difficulty level. Programs P1 toP5 were first tested using GA and Random method and programs P6 to P10 were tested using FST. The

results were analysed and compared. We then repeated the process by testing P1 to P5 with FST and P6 to P10 with GA and Random method. The results obtained were same as earlier and hence not included in the thesis.

In stage 1 of the evaluation, all the test subjects (P1 to P10) addressed this questionnaire and the outcome was recorded for subject programs P1 to P5 for GA based testing and for subject programs P6 to P10 for forward slicing based testing. This means that program P1 to P5 was checked with GA based approach and programs P6 to P10 was checked with forward slicing based testing approach. Similarly in stage 2 also all the test subjects (P1 to P10) addressed this questionnaire and the outcome was recorded for subject programs P1 to P5 for random method of testing and for subject programs P6 to P10 for forward slicing based testing. This means that program P1 to P5 was checked with random method of testing approach and programs P6 to P10 was checked with forward slicing based testing approach in the second stage.

In both the stages 1 & 2, a qualitative rating for the questions was assigned based on the evidence collected from GA based testing, forward slicing based testing and random method of testing. Each question had a possible response. By data coding, the outcome of the questions was rated based on a scale of 0-5 [60]. The outcome and the corresponding scale are given below in figure 6.1. Based on the testing outcome given in table 6.4 and table 6.5, the response was noted for each type of testing in stage 1 and stage 2.

0-Nil/No difficulty
1-Negligible
2-Low difficulty
3-Average difficulty
4-High difficulty
5- Extreme difficulty

Figure 6.1 Question outcome and corresponding scale

6.3.1.3 Stage 1- Comparing Forward Slicing based Testing and GA based Testing

In this stage, forward slicing based testing is compared with GA based testing.

Table 6.6 Response for GA based Testing & FST

Test Subject	GA based testing-Response for Q1	GA based testing-Response for Q2	GA based testing-Response for Q3	GA based testing-Total/Sum of responses (Q1+Q2+Q3)	Test Subject	Forward slicing based testing-Response for Q1	Forward slicing based testing-Response for Q2	Forward slicing based testing-Response for Q3	Forward slicing based testing-Total/Sum of responses (Q1+Q2+Q3)
P1	4	5	4	13	P6	1	1	2	4
P2	4	4	4	12	P7	1	1	1	3
P3	4	5	4	13	P8	0	1	1	2
P4	5	5	5	15	P9	0	1	1	2
P5	4	5	4	13	P10	0	1	1	2

In the table 6.6 given above, the columns 'GA based testing-Response for Q1, GA based testing-Response for Q2 and GA based testing-Response for Q3' gives the response for question Q1, Q2 and Q3 for test subjects P1 to P5 when applying GA base testing and the column 'GA based testing-Sum/Total of responses (Q1+Q2+Q3)' gives the added value of Q1, Q2 and Q3. Similarly, the columns 'Forward slicing based testing- Response for Q1, Forward slicing based testing- Response for Q2 and Forward slicing based testing- Response for Q3'gives the response for question Q1, Q2 and Q3 for test subjects P6 to P10 when applying forward slicing based testing and the column 'Forward slicing based testing- Sum/Total of responses (Q1+Q2+Q3)' gives the added value of Q1, Q2 and Q3.

To perform the evaluation of GA based testing and forward slicing based testing, a Mann-Whitney U test is used. Inorder to calculate the U statistic, the next step is to rank the sum of the responses. The table 6.7 gives rank for sum of responses for GA based testing and forward slicing based testing.

Table 6.7 Ranked response for GA based Testing & FST

Test Subject	GA based testing-Sum of responses (Q1+Q2+Q3)	Rank of the response	Actual Rank of GA based testing	Test Subject	Forward slicing- Sum of the Responses for Q1,Q2 & Q3	Rank of the response	Actual Rank of forward slicing based testing
P1	13	7	8	P6	4	5	5
P2	12	6	6	P7	3	4	4
P3	13	8	8	P8	2	1	2
P4	15	10	10	P9	2	2	2
P5	13	9	8	P10	2	3	2
		Total	40			Total	15

In table 6.7 we can see that, ranks having tied score are given average value of ranks. Finally the actual ranks are noted. A Mann Whitney U test is applied for evaluating the ranks obtained. In Mann- Whitney U test, the U value is given as:

$$U= n_1 n_2 + n_x (n_x+ 1)/2 - T_x$$

Here n_1 and n_2 are the participants in each group and T_x is the largest of rank total and n_x is the number of participants in the group having largest rank total

In our example, $n_1 = 5$ and $n_2 = 5$, $T_x=40$ and $n_x=5$

Therefore $U = 5 * 5 + 5(5+1)/2 – 40$

$U = 25+15-40$

$U = 0$

The next step is to find the critical value of U from the Mann-Whitney U test table of critical values (given in appendix). From the table at 5% significance level for a two tailed test, the critical value of U for n_1=5 and for n_2=5 is 2.

Therefore U=2 (from Mann Whitney U table at 5% significance level)

To be statically significant, the obtained U value should be less than the critical value of U.

In our case 0 < 2, which indicates that the difference we found during the testing of software with GA based method and using forward slicing based testing is unlikely to occur by chance. Testing the software with GA based method and with a forward slicing based method makes a significant difference.

Therefore stage 1 of Null Hypothesis (H0) is rejected, which proves that forward slicing based testing is more useful in the testing process

Inference: - Forward slicing framework is more useful in testing compared to GA based testing method

6.3.1.4 Stage 2- Comparing Forward Slicing based Testing and Random testing

In this stage, forward slicing based testing is compared with random method of testing.

Table 6.8 Response for Random method based Testing & FST

Test Subject	Random method based testing-Response for Q1	Random method based testing-Response for Q2	Random method based testing-Response for Q3	Random method based testing-Sum of responses (Q1+Q2+Q3)	Test Subject	Forward slicing based testing-Response for Q1	Forward slicing based testing-Response for Q2	Forward slicing based testing-Response for Q3	Forward slicing based testing- Sum of responses (Q1+Q2+Q3)
P1	4	5	5	14	P6	1	1	2	4
P2	4	5	5	14	P7	1	1	1	3
P3	4	5	5	14	P8	0	1	1	2
P4	5	5	5	15	P9	0	1	1	2
P5	4	4	4	13	P10	0	1	1	2

In the table 6.8 given above, the columns 'Random method based testing-Response for Q1, Random method based testing-Response for Q2 and Random method based testing-Response for Q3' gives the response for question

Q1, Q2 and Q3 for test subjects P1 to P5 when applying random method of testing and the column ‘Random based testing-Sum of responses (Q1+Q2+Q3)’ gives the added value of Q1, Q2 and Q3. Similarly, the columns ‘Forward slicing based testing- Response for Q1, Forward slicing based testing- Response for Q2 and Forward slicing based testing- Response for Q3’gives the response for question Q1, Q2 and Q3 for test subjects P6 to P10 when applying forward slicing based testing and the column ‘Forward slicing based testing- Sum of responses (Q1+Q2+Q3)’ gives the added value of Q1, Q2 and Q3.

In order to perform the evaluation of random method based testing and forward slicing based testing, a Mann-Whitney U test is used. Inorder to calculate the U statistic, the next step is to rank the sum of the responses. The table 6.9 gives rank for sum of responses for random method based testing and forward slicing based testing.

Table 6.9 Ranked responses for random method based testing and forward slicing based testing

Test Subject	Random method based testing-Sum of responses (Q1+Q2+Q3)	Rank of the response	Actual Rank of Random method based testing	Test Subject	Forward slicing- Sum of the Responses for Q1, Q2 & Q3	Rank of the response	Actual Rank of forward slicing based testing
P1	14	8	8.5	P6	4	5	5
P2	14	9	8.5	P7	3	4	4
P3	13	7	6.5	P8	2	1	2
P4	15	10	10	P9	2	2	2
P5	13	6	6.5	P10	2	3	2
		Total	40			Total	15

In table 6.9 we can see that, ranks having tied score are given average value of ranks. Finally the actual ranks are noted. A Mann Whitney U test is applied for evaluating the ranks obtained. In Mann- Whitney U test, the U value is given as:

$$U = n_1 n_2 + n_x (n_x + 1)/2 - T_x$$

Here n_1 and n_2 are the participants in each group and T_x is the largest of rank total and n_x is the number of participants in the group having largest rank total

In our example, $n_1 = 5$ and $n_2 = 5$, $T_x=40$ and $n_x=5$

Therefore $U = 5 * 5 + 5(5+1)/2 - 40$

$U = 25+15-40$

$U = 0$

The next step is to find the critical value of U from the Mann-Whitney U test table of critical values. From the table (given in appendix), at 5% significance level for a two tailed test, the critical value of U for $n_1=5$ and for $n_2=5$ is 2.

Therefore U=2 (from Mann Whitney U table at 5% significance level)

To be statically significant, the obtained U value should be less than the critical value of U.

In our case $0 < 2$, which indicates that the difference we found during the testing of software with random method and using forward slicing based testing is unlikely to occur by chance. Testing the software with random method and with a forward slicing based method makes a significant difference.

Therefore stage 2 of Null Hypothesis (H0) is rejected, which proves that forward slicing based testing is more useful in the testing process

Inference: - Forward slicing framework is more useful in testing compared to random method based testing

6.3.1.5 Summary from Statistical Comparison

In the previous sections, forward slicing based testing is compared with GA based testing and random method. Some subject programs were considered for comparison and these subject programs were tested with forward slicing based testing, GA based testing and random method of testing. A Mann Whitney U test was used to compare the end result. Finally, from the end result, it can be noticed that forward slicing based testing is more useful compared to GA based testing and random testing. The research question RQ is thus addressed by statistical comparison and evaluation of testing methods.

6.3.2 Metric based Comparison and Evaluation

In the previous section a statistical comparison of testing approaches is done. In this section a metric based comparison of testing approaches is performed. All the test subjects given in table 6.3 are evaluated using two categories of metrics. The first category is a software testing technique metric [103] and the second category is a slicing based metric [137]. The software testing technique metrics are noted for each test subject corresponding to forward slicing based testing, random testing and GA based testing.

6.3.2.1 Software Testing Technique Metrics

The software testing technique metrics used for evaluating forward slicing based testing, random testing and GA based testing are given below.

1. Test case generation metric
2. User control metric

1. Test Case Generation (TCG) Metric

It is defined as the ability to generate and readily modify the test cases. Test case generation is expressed as the sum of Automated test case generation (ATG) and Test case reuse functionality (TRF) [103]. Therefore, TCG= ATG+TRF

Here, ATG and TRF are defined by various scale levels. In ATG, the following scale value denotes that the following functionalities are possible in the testing tool.

10 - Generation of test cases may be automated fully

8 - Tool is provided with user friendly interface

6 - The parameters which are present in the tool are named and defined properly

4 - The possible values of various parameters are provided to the user and is easily understandable

2 - Tester provides the tool with parameter name, type and range of values

0 - Tester must generate test cases by hand

Similarly, TRF is also denoted by a scale of possible values which are given below.

10 - Ability to modify test cases and save the modified test cases in an easy and understandable manner

8 - Test cases may be modified and saved as new test cases

6 - Test cases may be modified, but cannot be saved as new test cases in a user friendly manner

4 - Test cases may be modified, but cannot be saved as new test cases

2 - Limited modification of test cases possible

0 - Test cases cannot be modified

The TCG value obtained for subject programs (P1 to P10) using forward slicing based testing, random testing and GA based testing are given below in the table 6.10.

Table 6.10 TCG values

Factors	Testing with Forward slicing (Test subjects – P1 to P10)	Testing with Random method (Test subjects – P1 to P10)	Testing with GA (Test subjects – P1 to P10)
ATG	8	2	6
TRF	10	4	4
TCG	18	6	10

From the above table 6.10, we can see that the value of TCG is 18 using forward slicing, 6 using random method and 10 using GA based method. When using a forward slicing framework we can see that the values of ATG is 8 and TRF is 10 respectively. As there is some level of manual intervention, the value of ATG is 8. Using forward slicing, the statements of interest with respect to the variables present in a program may be easily identified. This also gives us a picture of the dependency present in the program. This information may be used to design test cases for the program. As we have the dependency information, the test cases generated may be modified and reused. Therefore, TRF is 10 here. With ATG and TRF value as 8 and 10, we have TCG as 18.

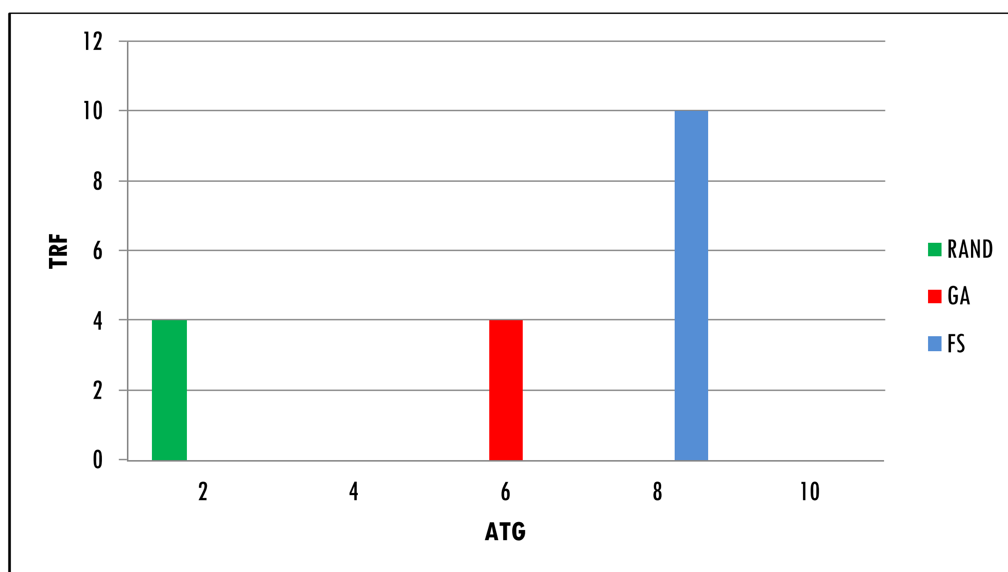


Figure 6.2 Test case generation metric

In figure 6.2, test case generation metric (TCG) is plotted. From the graph, it can be noticed that, while testing with random method, the tester generates test cases randomly and therefore the chance of success may vary which makes ATG value as 2. As the statement of interest cannot be identified, it is difficult to trace the dependency present in the module. Therefore test case reuse and design is also very difficult. The tester has to design, modify and record the test cases manually. This gives TRF value 4.

While testing with GA, parameters which are present in the tool are named and defined properly and therefore ATG value is taken as 6. As the statement of interest cannot be identified, it is difficult to trace the dependency present in the module. Therefore, test case reuse and design is also very difficult. The tester has to design, modify and record the test cases manually. This gives TRF value 4.

From the above given values of TCG in table 6.5, it is evident that using forward slicing is beneficial in testing compared to random testing and GA based testing.

2. User Control (UC) Metric

In this thesis, forward slicing framework is evaluated using UC metric too. The UC value depends on the tester [103]. If the tester is able to have an expansive control over tool operations and if the tester is able to perform testing with respect to the area of interest, then the UC value will be high. This means that, a tester can choose the program parts which need to be critically tested. This feature is very attractive when a part of source code needs to be tested alone without retesting the whole source code. We defined two factors in the experimental study based on which the value of UC is calculated which are given below:-

1. Module to be tested
2. Identify the statement of interest in the module.

Each of these factors can take either a value of 1 or 0, where 1 indicates a possibility and 0 indicates an unlikely situation. Ten independent testers who were involved in the comparison study of different methods used in our work gave values for UC metric. The averages of the value allocated by them are given in table 6.11. Here RM, GA and FST get a UC value of 1 for factor 1(as the tester has the freedom of selecting the module to be tested). For factor 2, RM and GA get a value of 0 (as the tester cannot identify the statements of interest), while FST gets a value of 1(as the tester can identify statements of interest during testing).

Table: 6.11 UC values

Factors	Testing using Random method	Testing using GA	Testing with Forward slicing
1	1	1	1
2	0	0	1
Total Value	1	1	2

From the table 6.11 given above, we can see that the sum/total value of UC factors one and two are UC=1 during testing with random method and GA and UC=2 in testing with forward slicing framework. This is because during testing with random method and GA based testing, the statements of interest cannot be identified which causes the value of factor 2 as 0 and in testing with the framework the value of factor 2 is 1, which means that the statements of interest may be found.

3. Summary from Software Testing Technique Metrics

Comparison and evaluation using software testing tool metrics TCG and UC indicate that forward slicing framework is more useful in testing process compared to random method and GA based testing.

6.3.2.2 Slicing Metrics

In this thesis, forward slicing based method is compared with random method and GA based method using some slicing metrics also. The slice based cohesion metrics namely tightness and coverage were used for evaluation and comparison of testing methods. The main reasons for using slicing metrics were:-

- Slice based metric provides a unique view of the program compared to ordinary metric.
- Slicing metrics used to quantify the changes in the source code
- Value of the slicing metrics may be used as a reference for future during program maintenance

1. Tightness Metric

The tightness of a program/module is defined as the number of statements in every slice [110, 137]. It is denoted as given below:

Tightness=$|S_{int}|$/ length (M)

Where S_{int} is the statements present in all the slices and length (M) is the length of the module present in the program. It can have a maximum value of 1. The value of tightness also indicates the level of dependency present in the module. A module with no statements in the slice will have a tightness value equal to 0. Having the tightness value as 0 is of little use during testing as it gives no information about the statements of interest. The statement of interest is helpful in finding the dependency between the statements during testing. This information will be helpful during test case generation and the test cases may be easily modified and reused. If the value of tightness is 0, it indicates that the dependency between the statements in a program is nil. In such situations the test cases designed for one condition may not be used for

the other and vice versa. Separate test cases need to be found for each and every condition. The test subject specifications and the value of slicing metrics 'tightness' noted for subject programs P7, P3 and P5 are given below in table 6.12. and table 6.13 respectively.

Table 6.12 Test subject specification

Test subject (programs)	Total LOC	Number of variables	Statements of interest with respect to variable1	Statements of interest with respect to variable 2	Statements of interest with respect to variable 3	Statements of interest with respect to variable 4	S_{int}
P7	20	4	6	8	8	5	5
P3	18	1	9	-	-	-	9
P5	35	3	16	15	13	-	13

In table 6.12, the result of applying forward slicing with respect to the variables present in the concerned programs for subject programs P7, P3 & P5 given in table 6.3 is given. The values in table 6.12 are used to calculate the tightness value of P7, P3 and P5 which is given in table 6.13.

Table 6.13 Tightness value

Test subject (programs)	Forward slicing based testing	Random method of testing	GA based testing
P7	0.37	0	0
P3	0.5	0	0
P5	0.25	0	0

From the table 6. 13 given above, it is clear that testing using random method and GA will have a tighter value of 0. Having the tightness value 0 gives no indication about the dependency present in the program. If the tester in able to get an idea regarding the dependency in the program, it will be helpful during testing, as it gives an indication of the probability of the chance of occurrence of errors in a program. Using forward slicing for testing gives the tightness value as 0.37, 0.5 and 0.25 respectively for programs P7, P3 and P5. Here, as S_{int} increases, the value of tightness increases which means that the number of dependent statements in the program increases. In such conditions

the tester should be more cautious in designing test cases as there is more chance of occurrence of errors. Therefore, using forward slicing in testing gives an indication of the level of difficulty needed during program testing.

2. Coverage Metric

Coverage (C) compares the length of the slices to the length of the entire module [137]. It is denoted as given below:

$$C = \frac{1}{V_i} \sum_{i=1}^{V_n} SL_i \,/\, \text{length}(M)$$

Here V_i is subset of V_n where V_n is the set of variables present in the module and SL_i gives the slice corresponding to V_i.

The test subject specifications and the value of slicing metrics 'coverage' noted for subject programs P7, P3 and P5 are given below in table 6.14.

Table 6.14 Coverage value

Test subject (programs)	Forward slicing based testing	Random method of testing	GA based testing
P7	0.33	0	0
P3	.5	0	0
P5	.41	0	0

From the table 6.14 given above, it is clear that testing with random method and GA will have a coverage value of zero. Having the coverage value zero gives no indication about the dependency present in the program. If the tester in able to get an idea regarding the dependency in the program, it will be helpful during testing, as it gives an indication of the probability of the chance of occurrence of errors in a program.

Using forward slicing for testing gives the coverage value as 0.33, 0.5 and 0.41 respectively for programs P7, P3 and P5. In forward slicing based testing, coverage gives the statements of interest with respect to all the variables in a module. This gives an overview of dependent statements in a module. This inturn will be helpful for the programmer, as it gives an idea of the level of difficulty associated with each module during program testing. Therefore, using forward slicing in testing is more beneficial compared to random testing and GA based testing.

3. Summary from Slicing Metric based Comparison

Evaluation and comparison of testing techniques using slicing metrics coverage and tightness indicate that, using forward slicing in testing is beneficial compared to testing using random method and GA based testing.

6.4 Inferences from the Comparison and Evaluation of Related Methods

In this section, we have mentioned the overall inferences made from the comparison of related methods

6.4.1 Main Inferences made from the Comparison of Different Testing Approaches

We have compared forward slicing based testing with some related testing techniques in section 6.2. Some inferences are made based on the comparison are already given in table 6.2. In order to handle these issues related to other testing approaches, a slicing based approach is introduced in thesis.

Slicing based testing approaches have clear advantages over other testing techniques which are highlighted in figure 6.3. Using the slicing based framework for testing minimizes the problems associated with other testing methods. The crux of using our slicing based test case generation framework is:-

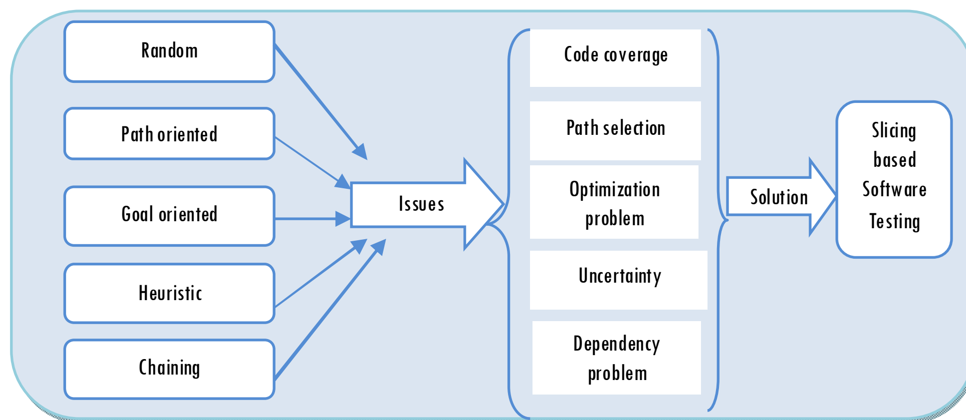


Fig 6.3 The Merits of Slicing based Test Data Generation

- Slicing identifies statements of interest which makes test data generation more effective and direct by handling the complexity of the size of the source code
- No confusion in selecting the paths as the process is based on slicing criterion, which is considerably easier to set, compared to path selection strategy
- Slicing considers dependency in the program which plays a crucial role in finding effective test cases
- No problem of uncertainty as in the case of heuristic approaches
- As test data generation itself is a process of generating test values for variables, applying slicing in such a scenario is better as our slicing also works on the basis of the variable of interest.
- Debugging is easier as we can locate the fault in the slice itself.

6.4.2. Inferences from Experimental Evaluation & Comparison

We can make some inferences based on the experimental evaluations made using statistical and metric based comparison given in the previous sections of this chapter. In the statistical evaluation using Mann-Whitney U test, we found that forward slicing based testing is more effective compared to random method of testing and GA based testing. In metric based comparison, we have used software testing metrics such as ATG and UC metric. Evaluation with both these metrics proved FST as better compared to GA based testing and random method of testing. In slicing metric based comparison, FST was compared with GA based testing approach and random method using slicing metrics 'tightness' and 'coverage'. In this evaluation also it was proved that, using forward slicing in testing is beneficial compared to testing using random method and GA based testing. Finally, based on these experimental evaluations we have made some inference on program factors like domain reduction, program dependency and number of errors which are detailed below. These inferences help to answer the research question RQ.

1. Domain Reduction: - *Effect of using GA based testing approach, random approach and forward slicing based approach on domain reduction during software testing is described below.*

During software testing using genetic algorithms and random testing, one of the main issues which affect the overall performance of the system is the domain size. Here the domain size refers to the variables present in the program [73]. For example, the variables for which the test cases are to be produced may reside inside a nested loop. In order to traverse the inner loops, the outer loops should be covered initially. Coverage of unrelated branches to reach the

destination branch involves several irrelevant paths and irrelevant variables. Finding appropriate values for such variables which are totally unrelated to the target do not make any improvement in testing, rather this causes delay in the testing process. Similarly, in GA based testing, improving or redesigning the fitness function according to the order of priority of the paths to be traversed did not make any significant improvement in the overall performance of the system. This is because redesigning the fitness function in genetic algorithm to improve the objective function value did not remove the irrelevant variables from the function. In otherwords, using GA in testing did not give statements of interest during software testing. This in turn makes testing more complicated. In random testing also, there was no reduction in the number of program statements to be considered for testing. This is evident from table 6. 4. On the other hand, forward slicing based testing concentrated on statements of interest and only variables of interest are considered. Therefore, there is no issue of considering unnecessary variables during software testing. This in turn makes the testing process more focused. The graph plotted in figure 6.4. gives an idea of the number of errors identified by forward slicing based testing, random testing and GA based testing. In the graph given in figure 6.4, it can be noticed that forward slicing based testing identifies more number of errors compared to random method and GA based testing. This is due to the fact that, for a given time duration, the number of relevant statements identified by forward slicing based testing is more compared to random method and GA based testing approach. This feature of forward slicing based testing is very much helpful during testing of large programs.

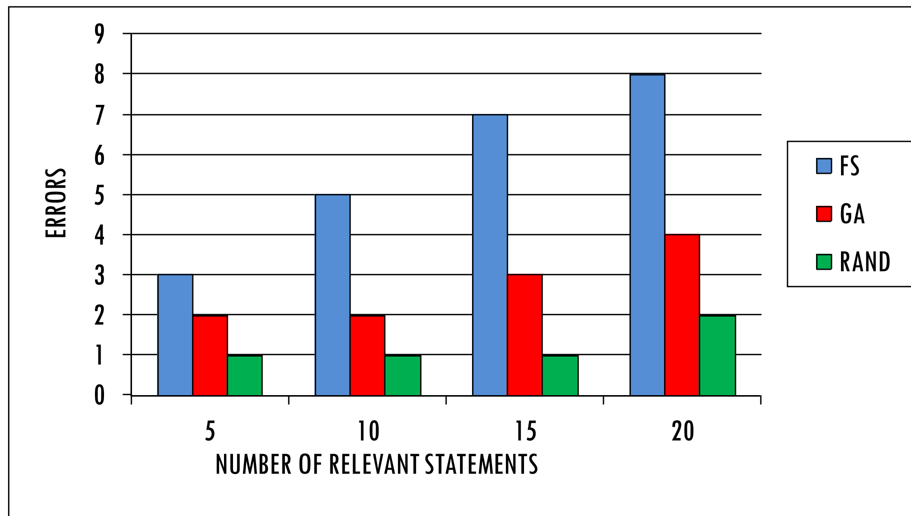


Figure 6.4 Identification of errors

2. Program Dependency: - *Effect of using GA based testing approach, random approach and forward slicing based testing approach on program dependency during software testing is as follows.*

Considering program dependencies is one of the major issues in software testing [9]. As the value of one variable may be dependent on other variables, software testing using forward slicing has a clear advantage over genetic algorithm based testing and random testing. In most of the works on genetic algorithm based software testing, program dependency is not correctly followed. Usually, in random testing and genetic algorithm based program testing, all the statements in the program are analysed initially to identify the relevant statements or else we have to get the list of statements that will have a potential role in software testing. From the testing point of view, checking the whole program line by line is an unnecessary waste of effort. Instead, if we are able to find the program statements which help in program testing, such as those that assist in finding the test case values during testing, the whole testing effort will be reduced considerably. In addition, testing can be made more methodical. Identifying the relevant statements which contribute to program testing and analysing those

statements can give the dependence relation present in the program. Using dependencies in the program helps to trace out the errors in a program. These are fulfilled in slicing based testing approach. As we already discussed in domain reduction, in GA based testing, redesigning the fitness function for traversing the target branches may cover some dependency features. This is not sufficient during software testing because the generation of test cases for a particular specification may require an exact picture of dependent variable values. This issue is not addressed in genetic algorithm based approach. Similarly in random testing also, the dependencies are not identified correctly. Using slicing based software testing approach, we can get a clear view of variable dependencies as well as the relevant variables from statements of interest, so that these variables in turn may be utilized for test case generation of some remaining branches in the program.

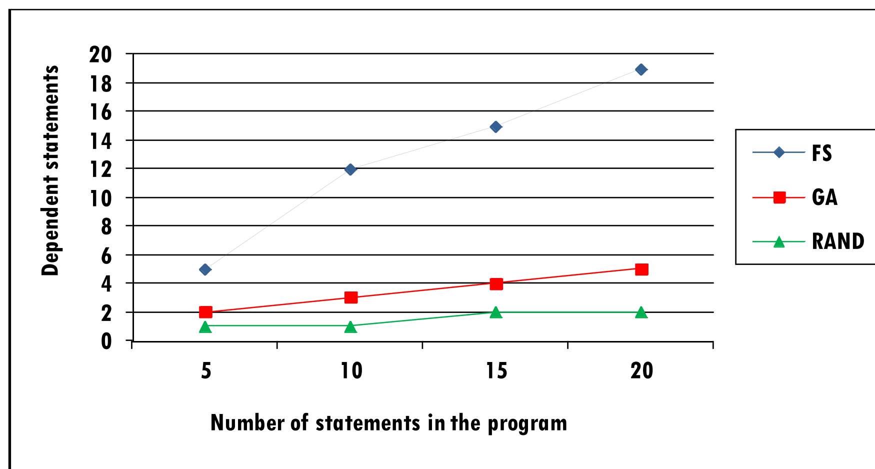


Figure 6.5 Dependency Level

The graph plotted in figure 6.5. gives an idea of the number of dependent statements identified by forward slicing based testing, random testing and GA based testing. In the graph given in figure 6.5, it can be noticed that forward slicing based testing identifies more number of dependent statements compared to random method and GA based testing. This is due to

the fact that, in a given time interval, the number of relevant statement identified by forward slicing based testing is more compared to random method and GA based testing approach. This feature of forward slicing based testing is very much helpful during testing of large programs as the dependent statements in the program gives an indication of the factors to be considered during program testing. Having a clear view of dependency will help the tester to implement changes in the source code and at the same time help to identify the dependent statements which are affected by the modification.

3. Number of errors: - *Effect of using GA based testing approach, Random approach and forward slicing based testing approach on the number of errors found during testing is as follows.*

We have done an analysis of the number of program statements which have a significant role in program testing identified by program slicing based testing, random method and GA based testing in table 6. 4. We have considered the statements in the program as a metric for analysing the testing approaches. For a given program, forward slicing covers more number of program statements with respect to the variables of interest compared to genetic algorithm in the same time span. As the probability of error distribution in a program is uniform throughout the code, an increase in the number of executable statements with respect to a particular program variable increases the chance of discovering the number of faults related to that variable [80]. This means that, rather than concentrating on a particular area for a long time to attain high coverage for that particular branch or program code, program slicing tries to analyze more number of potential statements in a given program compared to random method and GA based testing approach. Here the main principle is to identify possible program statements due to which program malfunctioning is caused, in minimal time. This re-affirms the fact that program

slicing based testing can be more effective in software testing compared to random method and GA based software testing.

An assessment of testing productivity obtained in random method, genetic algorithm and forward slicing based testing approaches is given in figure 6.6. The graph shows that, when program testing is done using forward slicing based approach, there will be high testing productivity and when program testing is implemented using genetic algorithms and random method the testing productivity will be low. Some of the terms related to the graph in figure 6.6 are given below:-

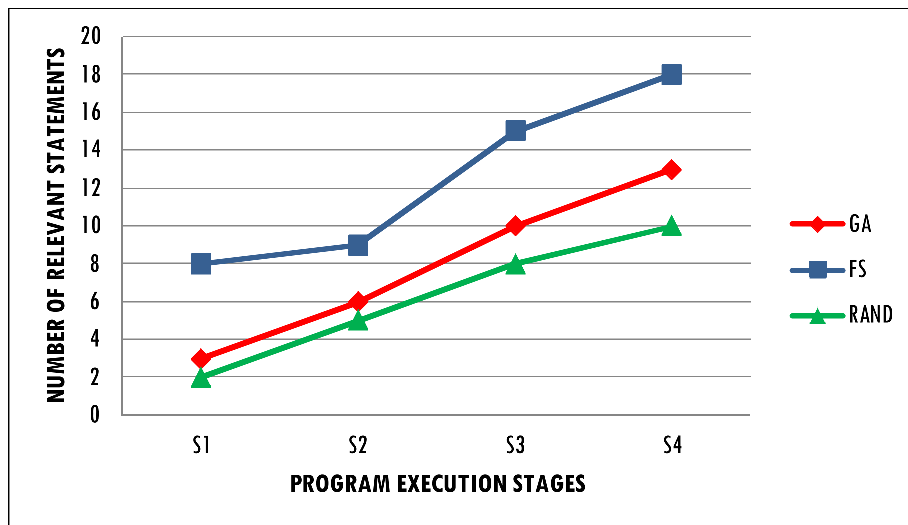


Figure 6.6 Testing productivity graph

1. Relevant branch indicates the statements of a program which may play a critical role in program testing.
2. Testing productivity indicates the measure of the number of relevant statements that can be covered in a specific time interval
3. High testing productivity means that more errors can be detected with less ‘effort’, while low testing productivity means that the number of relevant statements covered in a specific time interval will be very few

From the graph given in figure 6.6, it is evident that forward slicing based testing has high testing productivity compared to random and GA based testing. This means that as the number of relevant statements identified by forward slicing is more, the number of errors found using forward slicing based testing is more compared to random method and GA based testing. Therefore, forward slicing based testing is more beneficial that random method and GA based testing.

From the inferences made from domain reduction, program dependency and number of errors in a program, it is evident that forward slicing based software testing is more effective in software testing compared to random testing and GA based testing. Slicing based testing approaches are very essential when the source code of the program is a concern for testers.

Therefore, it can be concluded that the inferences made from the comparison of related testing techniques give satisfactory explanations for research question RQ

6.5 Summary of the Chapter

This chapter discusses the comparison and performance evaluation of forward slicing based testing with related testing approaches. In addition to a general comparison with related testing approaches, forward slicing based testing is compared with random method of testing and GA based testing. A statistical method and metric based comparison is used for comparing forward slicing based testing with random method and GA based testing. Finally, the inferences made from the comparison shows that forward slicing based testing is more effective in software testing compared to random approach of software testing and GA based testing.

…….ꙮ…….

Chapter 7

CONCLUSION & FUTURE RESEARCH DIRECTION

Contents

7.1 Introduction
7.2 Summary of Achievements
7.3 Main Contributions
7.4 Future Directions
7.5 Conclusion

7.1 Introduction

Though the software quality has gained so much importance during the last decade, the discipline of software testing, especially source code based testing still remains a challenging one. In today's world, it has become almost impractical to think of a world without software. Improper working of software may result in huge damage, loss of money or even threat to life. Thus, active research work is progressing in the area of software testing to discover efficient methods for source code based testing. In this thesis work, an effective method is introduced and demonstrated for handling the source code size during software testing. Therefore, this work addresses a relevant issue in software testing.

7.2 Summary of Achievements

As mentioned in the above section, inorder to ensure proper working of software, several software testing techniques were developed [8, 11, 80]. In most of the testing methods, the issue of source code size was not addressed [2, 3, 43, 46, 59, 67, 97, 98, 99, 100, 101, 104]. Considering these facts, the main objective of this thesis was put forward- *To develop an effective method for source code based software testing that can handle the size of source code.*

In addition, this thesis work has provided some extensions to the proposed method of source code handling. The original objectives were as follows:-

- To handle source code size during structural testing
- Identify statements of interest with respect to input variables in software testing
- Identify the dependency in the program
- Tracking changes related to output

The first objective was addressed by introducing the concept of forward slicing for software testing. The forward slicing method for software testing was introduced with this aim in chapter 3. In forward slicing, the program statements affected by a particular variable were identified by the slicing criterion. The main intention of using forward slicing is to concentrate only on statements of interest without bothering about the whole source code. Forward slicing was done based on the slicing criterion. An algorithm for performing forward slicing was introduced and the algorithm uses a linked dependency method based on data flow equations to identify dependency in the program. Finally, from linked dependent statements, the forward slices are identified. This part of the thesis helped to fulfill the first objective of handling source code size during program testing. Inorder to establish the strength of using forward slicing in software testing, the comparison and performance evaluation of forward slicing based testing with some existing software testing methods was performed. Further, we have given a formal representation of forward slicing method and proved the correctness of slicing algorithm using Hoare Logic. Forward slicing based testing was compared with genetic algorithm based testing and random testing. Comparison and experimental evaluation with these methods showed that forward slicing based software

testing outperformed random and GA based testing methods. Therefore, the descriptions related to forward slicing based testing and the supporting evidence given by the comparison of forward slicing based testing with existing software testing methods assert that the forward slicing based testing can effectively handle source code size during software testing. To the best of our knowledge, no other works reported in literature have used the concept of forward slicing to handle the issue of source code size during structural testing. Therefore, in all aspects the first objective is achieved.

The forward slices thus obtained may be used for generating test data. Generating test data from the slices is described in detail in the latter half of chapter 3. Thus the second objective of the thesis is also tackled. The forward slices identified may or may not contribute in software testing. The statements which are useful in software testing are considered as relevant. The statements of interest identified from the program code may or may not be in the form of linear expressions. Test data are generated from the statements which are in the form of linear equations using Gauss Elimination. Some relevant statements which are in non-linear form may be converted to linear equations and test data is generated using Gauss Elimination. For other relevant statements, test data are generated using Random method. Therefore, identifying the relevant statements with respect to input variable is very essential, so that test data generation can be made easier. This part of the thesis fulfilled the second objective. Formal representation for forward slicing is also provided in this work.

The literature review presented in chapter 2 of this thesis, also give supporting evidence to fulfill the first two objectives. In genetic algorithm based program testing, factors like fitness function, population initialization, response time prediction and parameter settings impact the quality of the solution obtained from testing. There is no general rule which implies the

usage of a specific type of the above mentioned factors and parameters during the software testing process. Moreover, in reported GA based software testing approaches, methods to handle source code size are not reported. When using GA based method for software testing, all the statements in the program have to be considered. As the statement of interest is not identified in reported GA based testing approaches, generating test data will be more difficult compared to slicing based testing approach. As shown in chapter 6, testing approaches that do not address the issue of source code size and dependency are highly uncertain and impractical in the software industry. From the above given facts, we can conclude that the first and second objective of this thesis is fulfilled.

A linked dependency method was introduced in this work to find the dependency in the program. Using the linked dependency method all the direct and indirect dependencies related to the variables of interest are identified. In forward slicing based testing, the linked dependency method is used to find dependent statements with respect to an input variable. The concept of partitioned forward slices was introduced in chapter 4, in order to handle the large size of slices. Here the linked dependency method is used to find the dependency in the program and also to find the dependency between different slices present in the program. Therefore, the third objective is tackled in chapter 3 and chapter 4 and these chapters also make a novel contribution to the source code based testing field.

A new concept of partial slice was introduced in chapter 5 which helps to identify the changes related to the output variables and to focus on the statements of interest during software testing. In partial slices also, the dependency between the program statements is traced using a linked dependency method. A comparison of forward slicing based testing with related methods is done in chapter 6 in order to prove the effectiveness of the

forward slicing based testing. Comparison and experimental evaluation based on statistical approach and metric based approach was used in this thesis work. In all the comparisons forward slicing based testing outperformed Random method and GA based testing. Thus, all the research objectives mentioned in this thesis work are achieved.

7.3 Main Contributions

Finally the main contributions of this thesis are:-

1. ***Developed a novel forward slicing based approach for source code based software testing that can handle the size of source code***

   Most of the testing techniques used in structural testing could not handle the issue of source code size during software testing. For programs with a large number of LOC, it is almost impractical to perform software testing by checking all the statements of the program. In such scenarios, the forward slicing based testing developed in this thesis which is capable of handling the source code size is very much helpful.

2. ***Introduced and demonstrated how to identify the statements of interest during software testing using forward slicing***

   During software testing, rather than concentrating on each and every statement and attain coverage for a particular section of source code, it is always better to identify maximum number of errors in minimum time. This is possible only if the tester is able to identify the statements of interest from the source code, so that the statements identified may be checked for correctness. The forward slicing introduced in this thesis helps to identify statements of interest so that the tester need not spend effort on checking unnecessary statements.

3. ***Demonstrated how to generate test data from forward slices using Gauss Elimination &random method***

   Generating test data occupies a main role in software testing. Using random method for generating test data from slices has the advantage of being easy and simple. Using Gauss Elimination method for generating test data from forward slices which are in the form of linear equation has the advantage of giving accurate test data values.

4. ***Introduced a novel concept named partitioned forward slices in order to handle the size of forward slices.***

   Forward slicing makes testing easier and effective by reducing the source code size and by identifying the statements of interest. Sometimes as the number of statements in the slice increases, the effect of using slicing in testing gets nullified. In such scenarios, the concept of partitioned forward slice may be used. Partitioned forward slices helps to reduce the slice size so that the tester can focus on slices more deeply during testing. This property of partitioned forward slices makes them very effective during testing.

5. ***Introduced the concept of partial slices to track the changes related to output***

   If a program output gets a wrong value, the best way to identify the cause of error is to trace the program statements which cause the error. In such scenarios, partial slices are used. In partial slices, rather than listing together all the program statements which affects the output, the programmer is able to focus separately on statements using the partial slicing criterion. This property of partial slices is very helpful during testing to identify changes related to output.

7.4 Future Directions

In this research work, we have introduced and designed a framework for software testing using forward slicing. Our main emphasis was on source code reduction and we were able to satisfy this goal with a good level of satisfaction. However, there is a scope for further research in this field and some of the future prospects are listed below:-

- **Extending the work for fault localization and debugging:** In spite of the promising results obtained by the forward slicing framework, bridging the gap between research and practical testing remains an open issue for slicing based testing tools. This may be done by extending the forward slicing framework for other fault localization issues like debugging and bug detection
- **Extending the work for different programming languages:** The work presented in this thesis is compatible with languages like C, C++. The work may be extended so that the forward slicing method for software testing may able to handle other programming languages also.
- **Extending the work for handling dynamic errors:** This works concentrates on identifying static errors associated with a program. This work can be extended to handle dynamic errors so that runtime errors may be identified easily.

7.5 Conclusion

The main goal of this research is to find effective methods to handle source code size during structural testing. From the literature study it was clear that, in the present condition using existing methods for practical software testing is very difficult, as most of the testing work does not discuss methods to

handle the size of source code. Therefore, we have proposed a forward slicing method where the tester can identify the statements of interest without concentrating on unnecessary program statements. In this thesis, it is clearly demonstrated how to generate test data from forward slices using Gauss Elimination and random method. In order to handle the large size of forward slices during testing, we have introduced a novel concept of partitioned forward slices. For tracking the changes related to output, the concept of partial slices was also introduced in this work. The work introduced and developed in this thesis is intended for intraclass structural testing. This work may be enhanced by extending the forward slicing method to incorporate interclass features like inheritance and polymorphism. The research finding finally concludes that, software testing methods should incorporate slicing concept to make testing more effective and easier.

…….ഇൽ…….

REFERENCES

1. Agarwal, H., Demillo, R. A. and Spafford, E.H. Debugging with Dynamic Slicing and Backtracking, *Software Practice and Experience,* 23, pp. 589-616, 1993

2. Ahmed, M. A and I. Hermadi, I. GA-based multiple paths test data generator, *Computer & Operations Research*, 35, pp. 3107-3127, 2008

3. Ali, S., Briand, L. C., Hemmati, H. and Panesar-Walawege, R. K. A Systematic Review of the Application and Empirical Investigation of Search-Based Test Case Generation, IEEE Transactions on Software Engineering, 99, 2009.

4. Ali, S., Iqbal, M. Z., Arcuri, A. and Briand, L. A Search-based OCL Constraint Solver for Model-based Test Data Generation, *In the Proceedings of the 11th International Conference on Quality Software,* pp. 41-50, 2011

5. Arcuri, A. and Fraser, G. On parameter tuning in search based software engineering, *In the Proceedings of* SSBSE, pp. 33-47, 2011

6. Arcuri, A. and Fraser, G. Parameter tuning or default values? An empirical investigation in search-based software engineering, *Empirical Software Engineering*, 18 (3), pp. 594-623, 2013

7. Baresel, A., Harman, M., Binkley, D. and Korel, B. Evolutionary Testing in the Presence of Loop–Assigned Flags: A Testability Transformation Approach, *In the Proceedings of the International Symposium on Software Testing and Analysis*, 29(4), pp. 108-118, 2004

8. Basili, V. and Selby, R., Comparing the effectiveness of software testing strategies, *IEEE Transactions Software Engineering*, (12), pp. 1278–1296, 1987

9. Bates, S. and Horwitz, S. Incremental Program Testing using Program Dependence Graphs, *In the Proceedings of the 20th ACM Symposium on Principles of Programming Languages*, pp.384-396, 1993

10. Beer, A. and Mohacsi, S. Efficient Test Data Generation for Variables with Complex Dependencies, *In the Proceedings of the International Conference on Software Testing*, pp. 3-11, 2008

11. Beizer, B. *Software Testing Techniques*, Second Edition, International Thomson Computer Press, ISBN 1-85032-880-3, 1990

12. Bergeretti, J and Carre, B. Information-flow and Data-flow analysis of While-programs. *ACM Transactions on Programming Language & Systems*, 7 (1), pp. 37-61, 1985

13. Bernard, B., Arnaud, G. and Claude, M. Symbolic Execution of Floating-point Computations, *Software Testing Verification and Reliability,* 16, pp. 97-121, 2006

14. Berndt, D. J., and Watkins, A., Investigating the Performance of Genetic Algorithm-Based Software Test Case Generation, *In the Proceedings of* HASE, 2004

15. T. J. Biggerstaff, Design recovery for maintenance and reuse, Computer, 22 (7), pp. 36-49, 1989

16. Binkely, D. The Application of Program Slicing to Regression Testing, *Technical Report*, Loyola College in Maryland, pp. 1-24, 1998

17. Binkely, D., Danicic, S., Gimothy, T., Harman, M. Kiss, A. and Korel, B.A Formalization of the Relationship between forms of Program Slicing, *Science of Computer Programming,*62(3), pp. 228-252, 2006

18. Binkley, D., Harman, M. and Lakhotia, K. FlagRemover: A Testability Transformation for Transforming Loop Assigned Flag, *ACM Transactions on Software Engineering and Methodology*, 2 (3), pp. 110-146, 2009.

19. Black, R. *Pragmatic Software Testing: Become an Effective & Efficient Test Professional*. John Wiley& Sons Publishers, 2007

20. Blanco, R., Tuya, J. and Diaz, A. B. Automated Test Data Generation using a Scatter Search Approach. *Information and Software Technology*, 51(4), pp. 708-720, 2009

21. Bueno, M. P. and Jino, M. Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data. *In the* P*roceedings of 15th Internationalconference on Automated Software Engineering*, pp. 209-218

22. Bueno, M. P. and Jino, S. Automatic Test Data Generation for Program Paths Using Genetic Algorithms. *International Journal of Software Engineering and Knowledge Engineering*, 12(6), pp. 691-709, 2002

23. Bueno, M. P., Jino, M. and Wong, E. Diversity Oriented Test Data Generation using Metaheuristic Search Techniques, *Information Sciences,* 2011

24. Campos, J., Arcuri, A., Fraser, G. and Abreu, R. Continuous Test Generation: Enhancing Continuous Integration with Automated Test Generation, *In the Proceedings of* Automated Software Engineering (ASE), 2014

25. Canfora, G., Cimitile, A. and De Lucia, A. Conditioned Program Slicing. *Information and Software Technology*, 40(11), pp. 595–607, 1998

26. Cao, Y., Hu, C. and Li, L. An Approach to Generate Software Test Data for a Specific Path Automatically with Genetic Algorithm, *In the Proceedings of ICRMS*, Chengdu, pp. 888-892, 2009

27. Chapra, S. and Canale, R. *Numerical methods for Engineers*, Sixth Edition, McGraw-Hill Education ,ISBN-13: 978-0073401065

28. Chen, C., X. Xu, X. , Y. Chen, Y., X. Li, X. and D. Guo, D. A New Method of Test Data Generation for Branch Coverage in Software Testing Based on EPDG and Genetic Algorithm, *In the Proceedings of* the *ASID*, pp. 307-310, 2009

29. Chen, Y. and Zhong, Y.Automatic Path-oriented Test Data Generation Using a Multi-population Genetic Algorithm, *In the Proceedings of the Fourth International Conference on Natural Computation*, pp. 566-570, 2008

30. Chikofsky, E. J and Cross II, J. H. Reverse Engineering and Design Recovery: A Taxonomy, *IEEE Software*, 7(1), 1990.

31. Christophe. M. ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution, *Software Testing, Verification and Reliability,* 11, pp. pp. 81- 96, 2001

32. Claudia, E. M. F. P. and Regina, V.S. Selection and Evaluation of Test Data Based on Genetic Programming, *Software Quality Journal,* 11, pp. 167-186, 2003

33. Corder, G.W., Foreman, D.I. *Nonparametric Statistics: A Step-by-Step Approach*, Wiley. ISBN 978-1118840313, 2014

34. Cui, H., Chen, L., Zhu, B. and Kuang, H. An Efficient Automated Test Data Generation Method, *In the Proceedings of the International Conference on Measuring Technology and Mechatronics Automation*, 1, pp. 453- 456, 2010

35. Demilli, R. A. and Offutt, A. J. Constraint-Based Automatic Test Data Generation, *IEEE Transactions on Software Engineering*, 17(9), pp. 900-910, 1991

36. De Lucia, A. 2001. Program Slicing: Methods and Applications, *In the Proceedings of the 1st IEEE Workshop on Source code Analysis and Manipulation*, pp. 142-149

37. Diaz, E., Tuya, J. and Blanco, R. Automated Software Testing Using a Metaheuristic Technique Based on Tabu Search, *In the Proceedings of the 18th IEEE International Conference on Automated Software Engineering*, pp. 310-313, 2003

38. Diaz, E., Tuya, J., Blanco, R. and Dolado, J. J. A Tabu Search Algorithm for Structural Software Testing, *Computers & Operations Research*, 35(10), pp. 3052-3072, 2008

39. Doungsa-ard, C., Daha, K., Hossai, A., and Suwannasart, T. Test Data Generation from UML State Machine Diagrams using Gas, *In Proceedings of International Conference on Software Engineering Advances, 2002*

40. Edvardsson, J and Kamkar, M. Analysis of the Constraint Solver in UNA Based Test Data Generation, *In the Proceedings of the 9th European software engineering conference held jointly with 9th ACM SIGSOFT international symposium on Foundations of software engineering*, 26(5), pp. 237-245, 2003

41. Ferguson, R. and B. Korel, B. The Chaining approach for Software Test Data Generation, *ACM Transactions on Software Engineering and Methodology*, 5(1), pp. 63-86, 1996

42. Ferrante, J., Ottenstein, K. and Warren, J. The Program Dependence Graph and its Use in Optimization, *ACM Transactions on Programming Languages & Systems,* 9(3)pp. 319-349, 1987

43. Fischer, M., Generating Test Data for Black-Box Testing using Genetic Algorithms, *In the Proceedings of 17th ETFA* , pp. 1-6, 2012

44. Floyd, R. W. Assigning Meanings to Programs. *In the Proceedings of the American Mathematical Society Symposia on Applied Mathematics*, 19, 19-31, 1967

45. Fraser, G. and Arcuri, A. It is not the length that matters, it is how you control it, *In the Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, pp. 150-159, 2011

46. Fraser, G. and Arcuri, A. EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, *In the Proceedings of ESEC/FSE*, , pp. 416-419, 2011

47. Fraser, G. and Arcuri, A., The Seed is Strong: Seeding Strategies in Search-Based Software Testing, *In the Proceedings of ICST*, pp.121-130, 2012

48. Fraser, G.and A. Arcuri, A. Whole test suite generation, *IEEE Transactions on Software Engineering*, 39(2), pp. 276 - 291, 2013

49. Fraser, G., Arcuri, A. and McMinn, P. Test Suite Generation with Memetic Algorithms, *In the Proceedings of GECCO*, 2013

50. Fraser, G., Arcuri, A. and McMinn, P. A Memetic Algorithm for whole test suite generation, *Journal of Systems and Software, DOI: 10.1016/j.jss.2014.05.032*, 2014

51. Fox, C., Danicic, S., Harman, M., and Hierons, R. M. ConSIT: a fully automated conditioned program slicer. *Software Practice and Experience,* 34, pp. 15–46, 2004

52. Fox, C., Harman, M., Hierons, R. M., and Danicic, s. Backward Conditioning: A New Program Specialisation Technique and its Application to Program Comprehension. *In the Proceedings of the 9th IEEE International Workshop on Program Comprehension,* pp. 89–97, 2001

53. Gallagher, K. B. and Binkley, D. Program Slicing, *In the Proceedings of Frontier of Software Maintenance*, pp. 58-67, 2008

54. Gallagher, K. B. and Lyle, J. R. Using Program Slicing in Software Maintenance, *IEEE Transactions on Software Engineering,* 17(8), pp. 751–761, 1991

55. Gallagher, K. B. Using Program Slicing in Software Maintenance. *Ph.D. Thesis*, University of Maryland Baltimore County, 1990

56. Galeotti, J. P., Fraser, G. and Arcuri, A. Extending a Search-Based Test Generator with Adaptive Dynamic Symbolic Execution, *In the Proceedings of ISSTA*, 2014

57. Girgis, M. R. Automatic test data generation for data flow testing using a genetic algorithm, *Journal of Universal Computer Science*, 11(5), pp. 898-915, 2005

58. Goldberg, D. E. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison Wesley, 1989

59. Gong, D., Zhang, W. And Yao, X. Evolutionary Generation of Test Data for Many Paths Coverage based on Grouping, *Journal of Systems and Software,* 84(12), pp. 2222-2233, 2011

60. Gouraud, S.D., Denise, A., Gandel, M.C. and B. Marre, B. A New Way of Automating Statistical Testing Methods, *In the Proceedings of 15th International conference on Automated Software Engineering*, pp. 5-12, 2001

61. Graham-Rowe, D. Radio Emerges from the Electronic Soup, *New Scientist*, 175, 2002

62. Gupta, R., Harrold, M.J., and Soffa, M. L. An Approach to Regression Testing using Slicing, *In the Proceedings of the IEEE Conference on Software Maintenance*, pp. 299-308, 1992

63. Harman, M. The Current State and Future of Search based Software Engineering, *In the Proceedings of FOSE*, pp. 342-357, 2007

64. Harman, M. and Binkley, D. Forward Slices are Smaller than Backward Slices, *In the Proceedings of the fifth International Workshop on Source code Analysis and Manipulation*, pp. 15-24, 2005

65. Harman, M. and Danicic, S. Using Slicing to Simplify Testing. *In the Proceedings of Eurostar*, 1994

66. Harman, M. and Danicic, S. A New Algorithm for Slicing Unstructured Programs. *Journal of Software Maintenance and Evolution*,10(6), pp. 415–441, 1998

67. Harman, M., Lakhotia, A and Binkley,D. Theory and Algorithms for Slicing Unstructured Programs, *Journal of Information and Software Technology*, 48(7), pp. 549-565, 2006.

68. Harman, M., Hierons, R. M., Fox, C., Danicic, S. and Howroyd, J. Pre/post Cconditioned Slicing, In the *Proceedings of the International Conference on Software Maintenance*, pp. 138-147, 2001

69. Harman, M., Hu, L., Zhang, X. and Munro M. Side-effect Removal Transformation. *In the Proceedings of the 9th IEEE International Workshop on Program Comprehension*, Toronto, Canada, pp. 310-319, 2001

70. Harman, M., Hu, L., Hierons, R., Baresel, A. and Sthamer, H. Improving Evolutionary Testing by Flag Removal,*In Proceedings of the Genetic and Evolutionary Computation Conference,* New York, USA, 1359-1366, 2002

71. Harman, M., Mansouri, S.A. and Zhang, Y. Search-Based Software Engineering: Trends, Techniques and Applications, *ACM Computing Surveys*, 45(1), pp. 1-66, 2012

72. Harman, M. and McMinn, P. A Theoretical and Empirical Study of Search based Testing: Local, global and hybrid search, *IEEE Transactions on Software Engineering*, 36(2), pp. 226-247, 2010

73. Harman, M., McMinn, P. and Wegener, J. The Impact of Input Domain Reduction on Search Based Test Data Generation, In the Proceedings of .ESEC/FSE, pp. 155-164, 2007

74. Holland, J., H. *Adaptation in Natural and Artificial Systems*, University of Michigan Press, Ann Arbor, 1975

75. Horwitz, S., Reps, T. and Binkley, D. Interprocedural Slicing using Dependence Graphs, *SIGPLAN Notices*, 23(7), pp. 35-46, 1988

76. Jackson, D. and Rollins, E. J. A new model for program dependences using reverse engineering, In the Proceedings of FSE, pp.2-10, 1994

77. Jayaram, B. and Govindarajan, K. Intensional Algorithmic Debugging, *Technical Reports*, Computer Science & Engineering, University of Buffalo

78. Jeffery, D. and Gupta, R. Test Case Prioritization Using Relevant Slices, *In the Proceeding of COMPSAC*, 1, pp. 411-420, 2006

79. Jones, B.F., Sthamer, H. H. and Eyres, D. E. Automatic Structural Testing Using Genetic Algorithms, *Software Engineering Research Journal*, pp. 299-306, 1996

80. Jorgensen, P.C. *Software Testing: A Craftsman's Approach*. Auerbach Publications(Taylor and Francis group), 2008

81. Juergens, E., Deissenboeck, F. and B. Hummel. B. Clone detection beyond copy & paste, *In the Proceedings of IWSC*, 2009

82. Kitchenham, B. A. Guidelines for Performing Systematic Literature Reviews in Software Engineering, *Technical Report EBSE- 2007-01*, 2007

83. Khor S. and Grogono P. Using a Genetic Algorithm and Formal Concept Analysis to Generate Branch Coverage Test Data Automatically, *In theProceedings of the 19th International Conferenceon Automated Software Engineering,* 2004

84. J. Krinke. Is cloned code more stable than non-cloned code? In the Proceedings of the Workshop on Source Code Analysis and Manipulation, pp. 57-66, 2008

85. Korel, B. Automated Software Test Data Generation, *IEEE Transactions on Software Engineering*, 16(8), pp. 870-879, 1990

86. Korel, B. and Laski, J. Dynamic Program Slicing, *Information Processing Letters,* 293, pp. 155-163, 1988

87. Korel, B. and Rilling, J. Program Slicing in Understanding of Large Programs, *In the Proceedings of the 6th International Workshop on Program Comprehension*, pp. 145-152, 1998

88. Komondoor, R. and Horwitz, S. Semantics-preserving Procedure Extraction, *In the Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages,* pp. 155-169, 2000

89. Latiu, G. I., Cret, O. A. and Vacariu, L. Automatic Test Data Generation for Software Path Testing using Evolutionary Algorithms, *In the Proceedings of the Third International Conference on Emerging Intelligent Data and Web Technologies*, pp.1-8, 2012

90. Leonard, G., Jefferson, O.A. And Anthony, C. Integration Testing of Object Oriented Components using Finite State Machines, *Software Testing, Verification andReliability,* 16, pp. 215-266, 2006

91. Li, J., Baob, W., Zhaoa, Y., Maa, Z. and Donga, H. Evolutionary generation of unique input/output sequences for class behavioral testing, *Computers and Mathematics with Applications*, 57, pp. 1800-1807, 2009

92. Lin, P., Bao, X. L., Shu, Z. Y., Wang, X. J. and Liu, J. M. Test case generation based on adaptive genetic algorithm, *In the Proceedings of the International Conference on Quality, Reliability, Risk, Maintenance, and Safety Engineering*, pp. 863-866, 2012.

93. Lin, C. and Yeh, P. Automatic Test Data Generation for Path Testing using Gas. *Information Sciences,* 131, pp. 47-64, 2001

94. Liu, X., Liu, H., Wang, B., Chen, P. and Cai, X. A Unified Fitness Function Calculation Rule for Flag Conditions to Improve Evolutionary Testing, *In the Proceedings of the 20th IEEE/ACM international Conference on Automated softwareengineering,* pp.337-341, 2005

95. Liu, D., Wang, X. and Wang, J. Automatic test case generation based on genetic algorithm, *Journal of Theoretical and Applied Information Technology*, 48(1), pp. 411-416, 2013

96. Malburg, J.and Fraser, G. Combining Search based and Constraint-based Testing, In the Proceedings of IEEE ASE, pp. 436-439, 2011

97. Mao, C. and Yu, X. Test data generation for software testing based on quantum-inspired genetic algorithm, *International Journal of Computational Intelligence and Applications*, 12(1), 2013

98. McMinn, P. Search-based Software Test Data Generation: A Survey. *Journal of Software Testing Verification and Reliability*, 14(2), pp. 105-156, 2004

99. McMinn, P. Search-Based Software Testing: Past, Present and Future, *In the Proceedings of the 4th International Conference on Software Testing, Verification and Validation Workshops*, pp. 153-163, 2011

100. McMinn, P. An identification of program factors that impact crossover performance in evolutionary test input generation for the branch coverage of C programs, *Information and Software Technology*, 55, pp. 153-172, 2013

101. McMinn, P., Harman, M., Binkley, D. and Tonella, P. The Species per Path Approach to Search-based Test Data Generation, *In the Proceedings of the International Symposium on Software Testing and Analysis (ISSTA 06)*, pp. 13-24, 2006

102. McMinn, P. and M. Holcombe. Evolutionary Testing Using an Extended Chaining Approach, *Evolutionary Computation*, 14(1), pp. 41-64, 2006.

103. Michael, J. B., Bossuyt, B. J and Snyder, B. B. Metrics for Measuring the Effectiveness of Software-Testing Tools, *In the Proceedings of the 13 th International Symposium on Software Reliability Engineering (ISSRE'02)*, 2002

104. Michael, C. C., McGraw, G. E. and Schatz M. A. Generating Software Test Data by Evolution. *IEEE Transactions on Software Engineering*, 27(12), pp. 1085-1110, 2001

105. Miller, J., Reformat, M. and Zhang, H. Automatic test data generation using genetic algorithm and program dependence graphs, *Information and Software Technology*, 48, pp. 586-605, 2006

106. Myers, G. J. *The Art of Software Testing*, Wiley, New York, 1979

107. Nashat, M. and Miran, S. Data Generation for Path Testing, *Software Quality Journal,* 12, pp. 121-136, 2004

108. Ngo, N. M. and Tan K. B. H. Heuristics-based Infeasible Path Detection for Dynamic Test Data Generation, *Information and Software Technology,* 50(8), pp. 641-655.

109. Orso, A., Sinha, S., and Harrold, M. Incremental Slicing based on Data-dependence types, *In Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, pp. 158-167, 2001

110. Ott, L. M. and Bieman, J. M. Program Slices as an Abstraction for Cohesion Measurement, *Information and Software Technology-Special issue on Program Slicing,* 40, pp.691−700, 1998

111. Outt, A.J., Jin, Z. and Pan, J. The Dynamic Domain Reduction approach to Test Data Generation, *Software Practice and Experience,* 29(2), pp. 167-193.

112. Oxman, A. D. Systematic Reviews: Checklists for Review Articles, *BMJ,* 309(6955), pp. 648-651, 1994

113. Pachauri, A. and Gursaran, Software Test Data Generation using Path Prefix Strategy and Genetic Algorithm, *In the Proceedings of the International Conference on Science and Engineering*, pp. 131-140, 2011

114. Pargas, R. P, Harrold, M. J. and Peck, R. R.Test Data Generation Using Genetic Algorithms, *Journal of Software Testing, Verifications, and Reliability*, 9, pp. 263-282, 1999

115. Pei, M., Goodman, E. D, Gao, Z. and Zhong, K.Automated Software Test Data Generation Using A Genetic Algorithm, *Technical Report* , GARAGe of Michigan State University, 1994

116. Pocatilu, P. and I. Ivan. A Genetic algorithm based system for Automatic control of Test Data Generation, *Studies in informatics and Control,* 22(2), pp. 219-226, 2013

117. Roper, M., Maclean, I., Brooks, A., Miller, J. And Wood, M.Genetic Algorithms and the Automatic Generation of Test Data. *Technical report RR/95/195[EFoCS-19-95]*, Department of Computer Science, University of Strathclyde, 1995

118. C. K. Roy and J. R. Cordy. A survey on software clone detection research, *Technical Report 541*, Queen's University at Kingston, 2007.

119. Samuel, P. And Mall, R. Slicing based Test Case Generation from UML Activity Diagrams, *ACM SIGSOFT Software Engineering notices*, 34(6), pp. 1-14, 2009

120. Samuel, P. and Mall, R. A Novel Test Case Design Technique Using Dynamic Slicing of UML Sequence Diagrams, *e-Informatica*, 2(1), pp. 71-92, 2008

121. Sharma, N., Pasala, A. and Kommineni, R.Generation of Character Test Input Data using GA for Functional Testing, *In the Proceedings of APSEC-SATA workshop*, pp. 87-94, 2012

122. J. Silva, A Vocabulary of Program Slicing-Based Techniques, *ACM Computing Surveys*, 44 (3), 2012

123. Sofokleous, A. A. and Andreou, A. S. Automatic, evolutionary test data generation for dynamic software testing, *The Journal of Systems and Software*, 81, pp. 1883-1898, 2008

124. Srinivas, M.Genetic Algorithms: A Survey, *Journal Computer*, 27, pp. 17-26, 1994.

125. Sthamer, H. H.Automatic generation of Software Test Data using Genetic Algorithms, *Ph.D. Thesis*, University of Glamorgan, Great Britain, 1996

126. Suresh, Y. and Rath, S. K. A genetic algorithm based approach for test data generation in basis path testing, *International Journal of Soft Computing and Software Engineering [JSCSE], Special Issue:In the Proceedings of the International Conference on Soft Computing and Software Engineering*, 3(3), 2013

127. Taylor, B.J. And Cukic, B. Evaluation of Regressive Methods for Automated Generation of Test Trajectories, *In the Proceedings of 11th International Symposium on Software Reliability Engineering,* pp.97-109, 2000

128. Tip, F.A Survey of Program Slicing Techniques, *Journal of Programming Languages*, 3 (3), pp. 121-189, 1995

129. Tracey, N.A Search-Based Automated Test Data Generation Framework for Safety Critical Software. *Ph. D. thesis*, University of York, 2000

130. Tran Sy, N. and Deville.Automatic Test Data Generation for Programs with Integer and Float Variables, *In the Proceedings of the 16th International Conference on AutomatedSoftware Engineering,* pp. 3-21, 2001

131. Tran Sy, N. and Deville, Y.Consistency Techniques for Interprocedural Test Data Generation, *In the Proceeding of the 9th European software engineering conference heldjointly with 11th ACM SIGSOFT international symposium on Foundations of softwareengineering,* 28(5), pp. 108-117, 2003

132. Venkatesh, G. A. The Semantic approach to Program Slicing, *In the Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* ACM Press, pp. 107-119, 1991

133. Visvanathan, S and Gupta, N.Generating Test Data for Functions with Pointer Inputs, *In theProceedings of the 17th IEEE International Conference on AutomatedSoftware Engineering,* pp. 149-160, 2002

134. Ward, M.Program Slicing via fermaT transformations, *In the Proceedings of the 26th International Computer Software and Applications Conference, Prolonging Software Life: Development and Redevelopment*,IEEE Computer Society, pp. 357-362, 2002

135. Watkins, A.A Tool for the Automatic Generation of Test Data using Genetic Algorithms, *In Proceedings of the fourth Software Quality Conference,* pp.300-309, 1995

136. Wegener, J., Baresel, A. and Sthamer, H.Evolutionary Test Environment for Automatic Structural Testing, *Journal of Information and Software Technology*, 43, pp. 841-854, 2001

137. Weiser, M. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4), pp. 352-357, 1984

138. Xanthakis, S., Ellis, C., Skourlas, C., Le Gall, A., Kastiskas, S. and Karapoulios, K. Application of Genetic Algorithms to Software Testing, *In the Proceedings of the 5th International Conference on Software Engineering and its Applications,* France, pp. 625-636, 1992

139. Xiao, J. and Afzal, W. Search-based resource scheduling for bug fixing tasks, *In the Proceedings of* 2nd SSBSE, pp. 133-14, 2010

140. Xue-ying MA Bin-kui SHENG Zhen-feng HE and Cheng-qing YE. A Genetic Algorithm for Test-Suite Reduction, *In the Proceedings of ICSMC,* pp. 133-139, 2005

141. Zhang, W., Gong, D., Yao, X. and Zhang, Y. Evolutionary generation of test data for many paths coverage, *In the Proceedings of Chinese Control and Decision Conference*, pp. 230-235, 2007

142. Zhang, Y., Harman, M. and Mansouri, A. *SBSE repository*, CREST Centre, UCL

143. Zhao, Y. *R and Data Mining -- Examples and Case Studies*, Academic Press, Elsevier, ISBN: 978-0-123-96963-7, 2012

…….ꕥ…….

LIST OF PUBLICATIONS

International Conferences

1. Anupama Surendran, Philip Samuel, Fault Localization using Forward Slicing Spectrum, *ACM conference on Research in Adaptive & Convergent Systems RACS '13, Montreal, QC, Canada*, pp. 397-398, OCT 2013

2. Anupama Surendran, Philip Samuel, Extracting Business Rules using a Partitioned Slicing Approach, *14th IEEE/ACIS International Conference on Software Engineering & Artificial Intelligence, (SNPD), Honolulu, Hawaii, USA*, pp. 336-341, JULY 2013

3. Anupama Surendran, Philip Samuel, Partial Slices in Program Testing, *35th Annual IEEE Software Engineering Workshop (SEW), Heraclion, Crete, Greece*, pp.82-89, OCT. 2012

4. Anupama Surendran, Philip Samuel, Slicing based Reverse Engineering of Business Applications, *12th IEEE International Conference on Intelligent Systems Design and Applications (ISDA), Kochi, Kerala*, pp. 673 – 679, NOV 2012

5. Anupama Surendran, Philip Samuel, Poulose Jacob, Code clones in Program Test Sequence Identification,*IEEE World Congress on Information and Communication Technologies (WICT), Mumbai*, pp. 1050 – 1055, DEC 2011

6. Philip Samuel, Anupama Surendran, Forward Slicing Algorithm based Test Data Generation, *3rd IEEE International Conference on Computer Science and Information Technology (ICCSIT), China,* pp. 270 – 274,JULY 2010

7. Anupama Surendran, Philip Samuel, An Evolutionary Multi- population approach for Test Data Generation*, IEEE World Congress on Nature & Biologically Inspired Computing, Coimbatore*, pp. 1451 – 1456, DEC 2009

International Journals

1. Anupama Surendran, Philip Samuel, Evolution or revolution: the critical need in genetic algorithm based testing, *Artificial Intelligence Review*, Springer Verlag, doi:10.1007/s10462-016-9504-8, ISSN: 0269-2821 (Print) 1573-7462 (Online),27 August 2016

2. Anupama Surendran, Philip Samuel, An Overview of Recent Trends in Software Testing, *Global Journal of Computer Science and Technology: C -Software & Data Engineering*, 14(8), Version 1.0, Global Journals Inc. (USA) Online ISSN: 0975-4172 & Print ISSN: 0975-4350, pp.7-27, 2014

3. Anupama Surendran, Philip Samuel, Poulose Jacob, Code Clones in Program Test Sequence Identification, *International Journal of Computer Information Systems and Industrial Management Applications*,5, pp. 564-570, 2013

4. Anupama Surendran, Philip Samuel, Test Data Generation Using Single Population Genetic Algorithm, *International Journal of Recent Trends in Engineering and Technology*,3(2), pp. 119-121, MAY 2010

5. Anupama Surendran, Philip Samuel,Forward Slicing based Novel Test Case Generation Framework *(under review in Journal of Software Engineering Research & Development, Springer Verlag)*

Edited Book Chapter

1. Knowledge-Based Processes in Software Development *(Chapter 4- Knowledge Based Code Clone Approach in Embedded and Real-Time Systems)*,IGI Global Publishers, JUNE 2013

…….ꕤ…….

APPENDIX

Critical Values for the Wilcoxon/Mann-Whitney Test (U)

Nondirectional α=.05 (Directional α=.025)

	n_2																			
n_1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	0	0	0	0	1	1	1	1	1	2	2	2	2
3	-	-	-	-	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8
4	-	-	-	0	1	2	3	4	4	5	6	7	8	9	10	11	11	12	13	13
5	-	-	0	1	2	3	5	6	7	8	9	11	12	13	14	15	17	18	19	20
6	-	-	1	2	3	5	6	8	10	11	13	14	16	17	19	21	22	24	25	27
7	-	-	1	3	5	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
8	-	0	2	4	6	8	10	13	15	17	19	22	24	26	29	31	34	36	38	41
9	-	0	2	4	7	10	12	15	17	21	23	26	28	31	34	37	39	42	45	48
10	-	0	3	5	8	11	14	17	20	23	26	29	33	36	39	42	45	48	52	55
11	-	0	3	6	9	13	16	19	23	26	30	33	37	40	44	47	51	55	58	62
12	-	1	4	7	11	14	18	22	26	29	33	37	41	45	49	53	57	61	65	69
13	-	1	4	8	12	16	20	24	28	33	37	41	45	50	54	59	63	67	72	76
14	-	1	5	9	13	17	22	26	31	36	40	45	50	55	59	64	67	74	78	83
15	-	1	5	10	14	19	24	29	34	39	44	49	54	59	64	70	75	80	85	90
16	-	1	6	11	15	21	26	31	37	42	47	53	59	64	70	75	81	86	92	98
17	-	2	6	11	17	22	28	34	39	45	51	57	63	67	75	81	87	93	99	105
18	-	2	7	12	18	24	30	36	42	48	55	61	67	74	80	86	93	99	106	112
19	-	2	7	13	19	25	32	38	45	52	58	65	72	78	85	92	99	106	113	119
20	-	2	8	14	20	27	34	41	48	55	62	69	76	83	90	98	105	112	119	127

Nondirectional α=.01 (Directional α=.005)

	n_2																			
n_1	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	0	0
3	-	-	-	-	-	-	-	-	0	0	0	1	1	1	2	2	2	2	3	3
4	-	-	-			0	0	1	1	2	2	3	3	4	5	5	6	6	7	8
5	-	-	-		0	1	1	2	3	4	5	6	7	7	8	9	10	11	12	13
6	-	-	-	0	1	2	3	4	5	6	7	9	10	11	12	13	15	16	17	18
7	-	-	-	0	1	3	4	6	7	9	10	12	13	15	16	18	19	21	22	24
8	-	-	-	1	2	4	6	7	9	11	13	15	17	18	20	22	24	26	28	30
9	-	-	0	1	3	5	7	9	11	13	16	18	20	22	24	27	29	31	33	36
10	-	-	0	2	4	6	9	11	13	16	18	21	24	26	29	31	34	37	39	42
11	-	-	0	2	5	7	10	13	16	18	21	24	27	30	33	36	39	42	45	46
12	-	-	1	3	6	9	12	15	18	21	24	27	31	34	37	41	44	47	51	54
13	-	-	1	3	7	10	13	17	20	24	27	31	34	38	42	45	49	53	56	60
14	-	-	1	4	7	11	15	18	22	26	30	34	38	42	46	50	54	58	63	67
15	-	-	2	5	8	12	16	20	24	29	33	37	42	46	51	55	60	64	69	73
16	-	-	2	5	9	13	18	22	27	31	36	41	45	50	55	60	65	70	74	79
17	-	-	2	6	10	15	19	24	29	34	39	44	49	54	60	65	70	75	81	86
18	-	-	2	6	11	16	21	26	31	37	42	47	53	58	64	70	75	81	87	92
19	-	0	3	7	12	17	22	28	33	39	45	51	56	63	69	74	81	87	93	99
20	-	0	3	8	13	18	24	30	36	42	46	54	60	67	73	79	86	92	99	105

N_1 \ N_2	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
5	2	3	5	6	7	8	9	11	12	13	14	15	17	18	19	20
6	3	5	6	8	10	11	13	14	16	17	19	21	22	24	25	27
7	5	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34
8	6	8	10	13	15	17	19	22	24	26	29	31	34	36	38	41
9	7	10	12	15	17	20	23	26	28	31	34	37	39	42	45	48
10	8	11	14	17	20	23	26	29	33	36	39	42	45	48	52	55
11	9	13	16	19	23	26	30	33	37	40	44	47	51	55	58	62
12	11	14	18	22	26	29	33	37	41	45	49	53	57	61	65	69
13	12	16	20	24	28	33	37	41	45	50	54	59	63	67	72	76
14	13	17	22	26	31	36	40	45	50	55	59	64	67	74	78	83
15	14	19	24	29	34	39	44	49	54	59	64	70	75	80	85	90
16	15	21	26	31	37	42	47	53	59	64	70	75	81	86	92	98
17	17	22	28	34	39	45	51	57	63	67	75	81	87	93	99	105
18	18	24	30	36	42	48	55	61	67	74	80	86	93	99	106	112
19	19	25	32	38	45	52	58	65	72	78	85	92	99	106	113	119
20	20	27	34	41	48	55	62	69	76	83	90	98	105	112	119	127

……. …….